Informationsflusstheorie der Softwareentwicklung

Von der Fakultät für Elektrotechnik und Informatik
der Gottfried Wilhelm Leibniz Universität Hannover
zur Erlangung des Grades

Doktor der Naturwissenschaften

Dr. rer. nat.

genehmigte Dissertation

von

M. Sc. Kai Stapel

geboren am 28.08.1981 in Erfurt

2012

Referent:	Prof. Dr. Kurt Schneider
Korreferent:	Asst. Prof. Dr. Samuel Fricker
Tag der Promotion:	12.04.2012

Kurzzusammenfassung

Software Engineering ist eine relativ junge Forschungsdisziplin. Bisher gibt es keine Theorie, die wesentliche und grundlegende Zusammenhänge in der Softwareentwicklung beschreibt. Eine grundlegende Theorie ist notwendig, um empirische Studien systematischer planen, interpretieren und Ergebnisse konsistenter kumulieren zu können. Zudem können die zentralen, präzise definierten Begriffe einer Theorie als Sprache zum Austausch über Softwareentwicklungsphänomene dienen. Insgesamt kann eine grundlegende Theorie helfen, die Disziplin Software Engineering voranzubringen, indem sie empirische Studien und Erkenntnisaustausch verbessert.

In dieser Arbeit wird eine grundlegende Theorie des Software Engineering entwickelt, um eine Sprache zur präziseren Kommunikation über Softwareentwicklung zu liefern und empirische Untersuchungen auf eine solidere Basis stellen zu können. Eine Software-Engineering-Theorie muss wesentliche und charakteristische Aspekte der Softwareentwicklung im Kern vereinen, um eine einheitliche theoretische Basis darzustellen. Diese Arbeit schlägt Information und deren Fluss im Projekt als wesentliches charakteristisches Merkmal der Softwareentwicklung vor. Es wird daher eine auf Informationsflüssen basierende Theorie der Softwareentwicklung vorgestellt. Zunächst werden wichtige Begriffe der Theorie wie Information, Informationsfluss, Wissen, Daten, Kommunikation und Dokumentation definiert. Im nächsten Schritt werden Theoreme aus den Definitionen hergeleitet, die sowohl neue Erkenntnisse der Softwareentwicklung darstellen als auch bereits bekannte Phänomene aus Informationsflusssicht beschreiben.

Die Qualität der Informationsflusstheorie wird geprüft, indem zunächst ihre Neuheit und externe Widerspruchsfreiheit untersucht wird. Dies geschieht durch Vergleich der Informationsflusstheorie mit verwandten Theorien. Anschließend wird die Theorie empirisch geprüft. Dazu werden die Ergebnisse empirischer Studien aus der Literatur aus Sicht der Theorie neu interpretiert und eine eigene empirische Untersuchung präsentiert. Der Vergleich mit

verwandten Theorien und die Analyse empirischer Studien aus der Literatur zeigen, dass die Informationsflusstheorie viele bekannte Softwareentwicklungsphänomene mit einem zusammenhängenden einheitlichen Modell erklären kann. Für vier der sechs in der eigenen empirischen Studie geprüften Hypothesen werden statistisch signifikante Belege präsentiert. Insgesamt lässt sich feststellen, dass die Informationsflusstheorie zur kohärenten Beschreibung wesentlicher Softwareentwicklungsphänome und somit als theoretische Grundlage für die Softwareentwicklung geeignet ist.

Abschließend wird auf Basis der Informationsflusstheorie eine Methode zur Analyse und Verbesserung von Softwareentwicklungsprojekten hergeleitet. Die FLOW-Methode ist ein Framework zur Durchführung und Erstellung von Techniken, die gezielt bestimmte Informationsflussprobleme lösen. Exemplarisch werden drei FLOW-Techniken vorgestellt. Für zwei Techniken werden empirische Belege ihrer Nützlichkeit geliefert. Dies zeigt, dass die Informationsflusstheorie auch genutzt werden kann, um Softwareentwicklung in der Praxis zu verbessern.

Abstract

Software Engineering is a relatively young academic discipline. To date, no theory exists that describes essential and fundamental properties of software development and the relationships between them. A fundamental theory is needed to systematically plan, interpret and to consistently cumulate results of empirical studies. In addition, the precisely defined concepts of such a theory can be used as a terminology to communicate about software development phenomena. In summary, a fundamental theory can help to bring forward the discipline of Software Engineering by improving empirical studies and knowledge exchange.

In this thesis a fundamental Software Engineering theory is developed to provide a terminology for precise communication about software development and to put empirical investigations on a solid basis. A Software Engineering theory needs to incorporate essential and characteristic properties of software development in order to provide a coherent theoretical basis. This thesis proposes information and its flow in a project as the essential characteristic aspect of software development. Hence, an information flow based theory of software development is proposed. First, important concepts of the theory like information, information flow, knowledge, data, communication, and documentation are defined. Then, theorems describing new insights as well as common Software Engineering knowledge are derived from the definitions.

Novelty and external consistency of the Information Flow Theory are shown by comparing it with related theories. Empirical validation is performed through re-interpretation of empirical study results from Software Engineering literature and by presenting an own empirical study. The comparison to related theories and analysis of studies from literature shows that the Information Flow Theory is suitable to explain many known software development phenomena using a consistent model. The own empirical investigation provides statistically significant evidence for four out of the six analyzed hypotheses.

Overall, it is shown that the Information Flow Theory is suitable to coherently describe fundamental software development phenomena. Therefore the Information Flow Theory is a suitable theoretical basis for software development.

Finally, a method for analysis and improvement of software development projects is derived from the theory. The FLOW Method is a framework for executing and creating techniques that solve specific information flow problems. As an example, three FLOW Techniques are introduced. Empirical evidence for the usefulness of two techniques is presented. This shows that the Information Flow Theory can also be used to improve software development in practice.

Inhaltsverzeichnis

1 Einleitung

Spätestens mit der NATO-Konferenz im Jahr 1968 [NATO 1968] wurde der Bedarf einer systematischen Herangehensweise an das Problem der Softwareentwicklung postuliert und der Name Software Engineering geprägt. Betrachtet man diese Konferenz als Startpunkt des Software Engineering als eigenständige Forschungsdisziplin, dann ist sie heute knapp über 40 Jahre alt. Seit 1968 wurden viele Fortschritte im Bereich der Softwareentwicklung gemacht. Es wurden die unterschiedlichsten Methoden entwickelt, um Software systematischer entwickeln zu können. Dokumenten-zentrierte Vorgehensmodelle wie das V-Modell [VModell 2009] oder der Rational Unified Process [RUP 2003] und agile Methoden wie eXtreme Programming [Beck 2000] oder SCRUM [Schwaber 2002] sind nur einige Beispiele. Viele Methoden wurden von Praktikern basierend auf Erfahrungen in eigenen Softwareprojekten entwickelt. Eine umfassende theoretische Grundlage, die eine einheitliche Sprache zur Beschreibung von Softwareentwicklung liefert und die verschiedenen Methoden vergleichbar macht, fehlt im Software Engineering bisher.

Dieses Kapitel legt zunächst die noch heute aktuellen Probleme des Software Engineering dar und motiviert den Bedarf für eine Theorie dieses Forschungsgebiets. Anschließend wird beschrieben, warum sich Informationsflüsse als zentraler Bestandteil für eine Softwareentwicklungstheorie eignen. Abschließend wird die Forschungsmethodik und der Aufbau der vorliegenden Arbeit präsentiert.

1.1 Probleme im Software Engineering

Software Engineering ist im Vergleich zu anderen Ingenieursdisziplinen, wie der Architektur oder dem Maschinenbau, eine junge Disziplin. Daher ist es nicht verwunderlich, dass viele Softwareentwicklungsprobleme noch nicht gelöst sind. In den folgenden beiden Unterkapiteln wird auf die heute aktuellen Probleme im Software Engineering eingegangen, die für die Motivation

dieser Arbeit relevant sind. Zunächst wird kurz erläutert, dass die Entwicklung von Software durch ihre Immaterialität und die häufig große Anzahl von Menschen, die an einem Softwareentwicklungsprojekt beteiligt sind und sich abstimmen müssen, ein inhärent schwieriges Problem darstellt. Anschließend wird das Fehlen einer grundlegenden Theorie des Software Engineering begründet.

1.1.1 Softwareentwicklung ist ein schwieriges Problem

In diesem Abschnitt wird erläutert, warum die Erstellung von Software ein inhärent schwieriges Problem ist. Im Wesentlichen gibt es dafür zwei Gründe. Erstens ist Software ein immaterielles Gut und dadurch für den Menschen nur schwer handhabbar, da er auf natürliche Weise nur an den Umgang mit materiellen Dingen gewöhnt ist [Ludewig 2010, S. 35]. Zweitens sind je nach Größe und Komplexität des zu lösenden Problems viele Personen an der Entwicklung von Software beteiligt, teilweise aus vielen unterschiedlichen Disziplinen. Dadurch entsteht ein großer Abstimmungsaufwand, der durch Phänomene zwischenmenschlicher Kommunikation noch erschwert wird, wie der Symmetry of Ignorance [Rittel 1984]. Bei den heute immer häufiger anzutreffenden global verteilten Softwareentwicklungsprojekten kommen zusätzlich noch Probleme u.a. durch die eingeschränkte Verfügbarkeit von Kommunikationsmedien hinzu. Die im lokalen Fall sehr effektive Kommunikation von Angesicht zu Angesicht ist in verteilten Projekten nicht möglich oder zu teuer.

Die Schwierigkeit der Softwareentwicklung drückt sich schließlich in der relativ großen Anzahl von fehlgeschlagenen, verspäteten oder zu teuren Softwareprojekten aus. Armour hat die Ergebnisse einer Studie zusammengefasst [Armour 2006], die Projektdaten von 564 IT Projekten aus 31 Unternehmen in 16 Branchen aus 16 Ländern ausgewertet hat. Die Daten wurden seit den späten 1970ern gesammelt. Sie zeigen damit einen Querschnitt über die letzten 30 Jahre. Ein Ergebnis der Studie ist, dass kleine Projekte sehr viel effizienter sind als große Projekte. Große Teams (29 Personen) produzieren ca. sechsmal so viele Fehler wie kleine Teams (3 Personen). Eine weitere Analyse dieser Projektdaten von 2009, beschränkt auf Projekte des vergangenen Jahrzehnts, hat gezeigt, dass große Projekte, definiert als Projekte mit mehr als 50.000 ESLOC[1], nur eine Chance von 19 % haben, den Zeitplan einzuhalten und mit

[1] ESLOC – Effective Source Lines of Code, LOC-Metrik aus der Werkzeugsammlung der Firma QSM (http://www.qsm.com), wahrscheinlich LOC ohne Leerzeilen, Kommentare und

einer Wahrscheinlichkeit von 30 % das Budget einhalten [Armel 2009, Armel 2011]. Das heißt, dass 81 % der großen Projekte länger brauchen als geplant und 71 % der großen Projekte teurer sind als geplant. Weiterhin haben die Daten gezeigt, dass Softwareentwicklung in den vergangenen 15 Jahren nicht produktiver[2] geworden ist [Armel 2011a, Armel 2011b].

Die Auswertung zweier Studien aus den Jahren 2005 und 2007 hat gezeigt, dass zwischen 26 % und 34 % aller Softwareprojekte fehlschlagen [Emam 2008]. Als fehlgeschlagen gelten dabei Projekte, die entweder abgebrochen wurden oder erfolglos[3] waren.

Insgesamt lässt sich feststellen, dass insbesondere große Projekte, d.h. Projekte mit vielen Entwicklern und mit viel Quellcode, heute immernoch oft fehlschlagen.

Was sind die Ursachen für die Fehlschläge? Nach Boehm [Boehm 2002], der wiederum die CHAOS-Reports der Standish Group[4] zitiert, in denen über 8000 IT-Projekte analysiert wurden, sind die sechs gravierendsten Gründe für Projektfehlschläge in der Softwareentwicklung:

- Unvollständige Anforderungen
- Nutzer werden nicht genug einbezogen
- Unrealistische Erwartungen
- Fehlende Unterstützung durch das Management
- Sich ändernde Anforderungen und Spezifikationen
- Nicht genug Ressourcen

Die ersten fünf dieser sechs Gründe haben etwas mit Kommunikation zwischen Softwareentwicklern und Stakeholdern (Nutzer oder Kunden) zu tun. Auch Curtis [Curtis 1988] hat in einer Studie über 17 große Softwareentwicklungsprojekte die folgenden Hauptursachen für Probleme festgestellt:

---

einfache Blockkonstrukte wie Klammern

[2] Produktivität gemessen als Function Points pro Personenmonat (FP/PM)

[3] Als erfolglos wurden diejenigen Projekte eingestuft, die von den Studienteilnehmern bei mindestens 4 der 5 Erfolgskriterien (Benutzerzufriedenheit, Budget, Zeitplan, Qualität, Produktivität) als ausreichend oder schlecht (4-Punkt Likert-Skala: sehr gut, gut, ausreichend, schlecht) bewertet wurden.

[4] http://www.standishgroup.com

- Geringe Verbreitung von Wissen über die Anwendungsdomäne
- Sich ändernde und in Konflikt stehende Anforderungen
- Kommunikations- und Koordinationsstörungen

Auch diese drei Ursachen haben etwas mit Kommunikation zu tun bzw. könnten durch geeignete Kommunikation überwunden werden. Zudem haben Kraut und Streeter [Kraut 1995] festegestellt, dass nicht nur formale Kommunikation über Dokumente, sondern insbesondere auch informelle, ungeplante und direkte Kommunikation wichtig für den Erfolg in der Softwareentwicklung ist.

Eine andere, aber verwandte Ursache für Probleme in Softwareprojekten könnten die in großen Projekten häufig eingesetzten Dokumenten-zentrierten Prozesse sein, die eine große Anzahl an zu erstellenden Dokumenten fordern (z.B. zw. 60 und 165 je Projekt [Stapel 2007]). Ludewig und Lichter stellen fest, dass *"in der Praxis die Dokumentation meist zu den Problemzonen des Software Engineerings gehört."* [Ludewig 2010, S. 266].

Auch global verteilte Softwareentwicklung führt immer wieder zu Problemen. Meist lassen sich diese Probleme auf Kommunikations- und Dokumentationsprobleme zurückführen, die sich wiederum auf kulturelle und Wissensunterschiede zurückführen lassen. Nach Carmel stellen die folgenden Dinge eine Herausforderung an das global verteilte Softwareentwicklungsteam dar [Carmel 1999].

- Verlust von Kommunikationsreichhaltigkeit
- Koordinationsstörungen
- Geografische Zerstreuung
- Kulturunterschiede

In einer systematischen Literaturauswertung haben da Silva et al. [Silva 2010] Kommunikation als die am meisten auftretende Schwierigkeit bei der verteilten Softwareentwicklung identifiziert. Ähnlich zu dem Ergebnis von Carmel sind die fünf häufigsten Schwierigkeiten:

- Kommunikation
- Kulturunterschiede
- Koordination

- Zeitzonenunterschiede
- Vertrauen

Auch Mohapatra et al. [Mohapatra 2010] haben Kommunikation als das Konstrukt mit dem größten Einfluss auf die Koordinationseffizienz in global verteilter Softwareentwicklung identifiziert.

Zusammenfassend lässt sich feststellen, dass Softwareentwicklung ein schwieriges Problem ist. Insbesondere große Projekte, an denen viele Personen beteiligt sind, sind heute immer noch oft nicht erfolgreich. Abstimmungsaufwände in Form von Kommunikation und Dokumentation machen dabei einen großen Teil der Schwierigkeit aus. Bei global verteilter Softwareentwicklung werden Dokumentations- und Kommunikationsschwierigkeiten nochmal verstärkt.

1.1.2 Im Software Engineering fehlt eine grundlegende Theorie

In den letzten Jahren haben sich die Stimmen etablierter erfahrener Software Ingenieure gemehrt, die nach Theorien im Software Engineering verlangen, um Software Engineering als Disziplin voranzubringen. So sagen z.B. Herbsleb und Mockus folgendes zu Theorien für das Software Engineering:

> *What is relatively rare in software engineering, however, is the sort of theory-driven investigation that dominates the* [other] *sciences.* [Herbsleb 2003]

Auch Ludewig und Lichter sind der Meinung, dass die Disziplin Software Engineering von neuen Theorien profitieren kann:

> *Im Software Engineering steckt die Theorie-Bildung noch in den Kinderschuhen, wir müssen uns gerade darum mit ihr befassen.* [Ludewig 2010, S. 5]

Jacobson und Spence haben einen Artikel verfasst, indem es ausschließlich darum geht, warum das Software Engineering eine grundlegende Theorie benötigt. Aus dem Artikel mit dem Namen "Why We Need a Theory for Software Engineering" [Jacobson 2009a] stammen folgende Zitate:

> *Do we really know how to develop great software? The answer for many people is clearly yes. But do we know how to communicate and continuously improve the way that we develop software? Do we really understand the best*

> *way to communicate and share our knowledge? The answer, as we saw in the previous article* [Jacobson 2009] *is clearly no!*
>
> [...]
>
> *Our understanding of software engineering lacks a basic theory.* [Jacobson 2009a]

Eine Fachsprache des Software Engineering ist demnach notwendig, um Softwareentwicklung kontinuierlich verbessern zu können und das Wissen darüber, wie man gute Software entwickelt, weitergeben zu können. Eine präzise Terminologie sollte nach Ansicht von Broy [Broy 2011] und Easterbrook et al. [Easterbrook 2008] also immer Teil einer Software-Engineering-Theorie sein:

> *Software engineering also deals with numerous abstract concepts that are often hard to understand because of the imprecise terms used to describe them* [...]. *This is why theory matters — it helps determine and evaluate the concepts that provide the basis for identifying terminology and developing engineering methods.* [Broy 2011]
>
> *A good theory precisely defines the theoretical terms, so that a community of scientists can observe and measure them. A good theory also explains why certain relationships occur.* [Easterbrook 2008]

Auch nach Ludewig und Lichter [Ludewig 2010] ist es im Software Engineering wichtig, die verwendeten Begriffe zu präzisieren:

> *Der präzise Umgang mit den Begriffen ist im Software Engineering besonders wichtig. Denn da Software immateriell ist* [...], *können wir nicht wie der Konstrukteur einer Maschine darauf ausweichen, den Gegenstand selbst zu zeigen oder zu betrachten. Hier ist der Begriff der Gegenstand.* [Ludewig 2010, S. 40]

Nach Kruchten ist eine von vier Schlüsseleigenschaften, die Software Engineering von anderen Ingenieurdisziplinen wie Bauingenieurwesen, Maschinenbau oder Elektrotechnik unterscheiden, *"das Fehlen einer grundlegenden Theorie"* [Kruchten 2002]. Kruchten meint damit zwar eine Theorie über die Eigenschaften von Software selbst, aus der sich wie in den anderen Disziplinen bestimmte Methoden begründen lassen, spricht aber im selben Artikel u.a. auch von iterativer Entwicklung als ein Beispiel einer erfolgreichen Methode des Software Engineering, die in anderen Ingenieursdisziplinen nicht eingesetzt

wird. Eine Theorie, die Vorteile bestimmter Entwicklungsmethoden erklären kann, ist demnach auch wertvoll.

Seit 2009 gibt es eine von Jacobson, Meyer und Soley begründete Initiative [Jacobson 2009b] mit dem Namen Software Engineering Method and Theory (SEMAT), die sich mit der Suche nach einer Theorie des Software Engineering beschäftigt. SEMAT hat in kurzer Zeit eine breite Unterstützung durch die Software-Engineering-Gemeinschaft erhalten. Dies zeigt sich vor allem durch die 1650 eingetragenen Befürworter der Initiative (Stand 04.12.2011).

Nicht nur Forscher, sondern auch Software-Engineering-Praktiker, sind an einer Theorie als wertvollem Beitrag für die Softwareentwicklung interessiert. So sagt z.B. Cockburn folgendes zur Suche nach einer Theorie in SEMAT:

> *I accepted to join SEMAT in order to join the search for such a thing* [theory], *because that topic interests me. I do think there is a possible theory and that it can be useful both in industry and in education;* [Cockburn 2010]

Ein weiterer Bereich, in dem eine Theorie das Software Engineering voran bringen könnte, ist die empirische Forschung.

> [...] *we believe software engineering must evolve its own tradition of empirical theorizing and hypothesis testing.* [Herbsleb 2003]

> *Software engineering needs theory to evaluate the efficiency and effectiveness of development techniques, which would ultimately result in justified and empirically verified methodological knowledge.* [Broy 2011]

Nach Juristo und Moreno [Juristo 2001] sind Studien, die auf einer Theorie basieren, wertvoller für die Forschungsgemeinschaft:

> *A mechanistic model* [or theoretical model], *supposedly backed by the nature of the system under study and verified by means of experimentation, is a much stronger position than a model obtained empirically and not backed by the theory of the phenomenon.* [Juristo 2001]

Wenn Hypothesen auf einer Theorie basieren, fällt es leichter Studienergebnisse zu interpretieren. Nach Wohlin [Wohlin 2000, S. 32] ist eine Theorie Ausgangspunkt zur Erstellung sinnvoller Hypothesen in einem Experiment und später Referenz für Interpretation und Generalisierung der Ergebnisse. Zudem können Ergebnisse unterschiedlicher Studien besser zusammengeführt werden, wenn sie auf einer einheitlichen Theorie basieren.

> [...] *we believe that an increased emphasis on theory can have a major impact on empirical investigations by providing a mechanism whereby results become more cumulative.* [Herbsleb 2003]
>
> *Theories also play a role in connecting research to the relevant literature. By defining the key terms, the results of empirical studies can be compared. Furthermore, theories support the process of empirical induction because an individual study can never offer conclusive results. Each study adds more evidence for or against the propositions of the theory. Without the theory, we have no way of making sense of the accumulation of empirical results.* [Easterbrook 2008]

Eine grundlegende Theorie mit präziser Terminologie ist also wichtig für das Software Engineering. Im folgenden Abschnitt werden die Probleme des Software Engineering nocheinmal zusammengefasst dargestellt.

1.1.3 Zusammenfassung der Probleme im Software Engineering

Software Engineering als Forschungsdisziplin hat weiterhin viele Probleme, die nicht gelöst sind. Da die Disziplin erst relativ jung ist, im Vergleich zu anderen Ingenieursdisziplinen wie Architektur oder Maschinenbau, ist das zwar nicht verwunderlich aber dennoch Ansporn, weiter zu forschen. Die für diese Arbeit relevanten Probleme im Software Engineering sind:

- Softwareentwicklung ist ein *schwieriges* Problem.
  - Softwareentwicklung ist in den vergangenen 15 Jahren nicht produktiver geworden [Armel 2011a, Armel 2011b].
  - Zwischen 26 % und 34 % aller Softwareprojekte schlagen fehl [Emam 2008].
  - Große Projekte sind ineffizienter, brauchen mit hoher Wahrscheinlichkeit länger als geplant, brauchen mit hoher Wahrscheinlichkeit ein größeres Budget als geplant und produzieren relativ mehr Fehler als kleine Projekte [Armour 2006, Armel 2009].
  - Global verteilte Softwareentwicklung verstärkt diese Probleme noch durch erschwerte Kommunikationsbedingungen [Carmel 1999, Silva 2010].

- Im Software Engineering fehlt eine grundlegende *Theorie* [Wohlin 2000, Juristo 2001, Kruchten 2002, Herbsleb 2003, Easterbrook 2008, Jacobson 2009a, Broy 2011].
  - Allgemein, um Software Engineering als Disziplin voranzubringen [Kruchten 2002, Jacobson 2009, Cockburn 2010, Broy 2011].
  - Ohne grundlegende Theorie können Studien nur schwer geplant und interpretiert werden [Wohlin 2000, Herbsleb 2003, Easterbrook 2008].
  - Ohne grundlegende Theorie sind Studienergebnisse nur schwer vergleichbar [Wohlin 2000, Herbsleb 2003, Easterbrook 2008].
  - Durch die Immaterialität von Software sind präzise Definitionen der im Software Engineering verwendeten Begriffe wichtig [Easterbrook 2008, Ludewig 2010, Broy 2011].

1.2 Lösungsansatz – Informationsflusstheorie der Softwareentwicklung

Um die oben genannten Probleme anzugehen, soll in dieser Arbeit eine Theorie entwickelt werden. Die grundlegenden Definitionen dieser Theorie sollen eine zusammenhängende Taxonomie bilden, die als Ausgangspunkt für die Entwicklung einer übergreifenden Fachsprache für das Software Engineering genutzt werden kann. Die Theorie soll die Planung und Auswertung von Studien vereinfachen. Zudem soll die Theorie die Entwicklung neuer Methoden ermöglichen, die die Softwareentwicklung verbessern können.

1.2.1 Warum eine neue Theorie?

Wenn das Ziel die Entwicklung einer neuen Theorie für das Software Engineering ist, sollte zunächst die Frage geklärt werden, warum die bisher vorhandenen Theorien zur Lösung der in Kapitel 1.1.3 genannten Probleme nicht ausreichen.

In Kapitel 5.1 werden zehn existierende Theorien des Software Engineering vorgestellt und gegen die in dieser Arbeit entwickelte Informationsflusstheorie abgegrenzt. Zum Beispiel sind das die Theory-W [Boehm 1989], die Theorie der Socio-Technical Congruence [Cataldo 2008] oder The Cooperative Game [Cockburn 2007]. An dieser Stelle sollen zunächst nur die wesentlichen Unterschiede genannt werden, um zu motivieren, warum es sinnvoll ist, eine neue Theorie zu entwickeln. Die bisherigen Theorien sind:

- zu speziell, d.h. sie beschreiben nur einen kleinen Teil von dem, was Softwareentwicklung ausmacht und können daher nicht genutzt werden, um einen großen Teil der noch heute aktuellen Softwareentwicklungsprobleme zu beschreiben,
- zu allgemein, d.h. sie beschreiben Bereiche, die unabhängig von der Softwareentwicklung sind, z.B. Managementtheorien,
- nicht formal genug, d.h. empirische Studien, insbesondere Metriken, können nur schwer daraus abgeleitet werden,
- nicht praktikabel, d.h. sie können nicht genutzt werden, um reale Softwareprojekte zu verbessern.

Es scheint also noch keine grundlegende Software Engineering Theorie zu geben. Das zeigt sich z.B. auch darin, dass in der Software-Engineering-Ausbildung nur ein kleiner Teil von dem gelehrt wird, was in der Praxis gebraucht wird [Boehm 2002]:

> *The theory that most current students get covers maybe 15 percent of the activities they encounter in practice.* [Boehm 2002]

Ein weiterer Grund, der für die Entwicklung einer neuen Theorie spricht, ist ein Forschungsergebnis aus der Kognitionswissenschaft. Demnach hat die Problemrepräsentation großen Einfluss auf die Lösungsfindung [Anderson 2001, S. 265]. Eine neue Repräsentation von Softwareentwicklungsphänomenen kann also auch helfen neuartige Lösungen für die heute noch vorhandenen Probleme der Softwareentwicklung zu finden.

Im Folgenden wird erläutert, warum es sinnvoll ist, Informationsflüsse zur Beschreibung von Softwareentwicklungsphänomenen zu verwenden.

1.2.2 Warum Informationsflüsse als Theoriekern?

Als wesentlichen charakteristischen Aspekt der Softwareentwicklung schlägt die hier vorgestellte Theorie Informationsflüsse vor. Mit Hilfe der Informationsflusssicht kann ein Großteil der relevanten Softwareentwicklungsphänomene beschrieben werden. In diesem Abschnitt wird die Frage geklärt, warum es sinnvoll ist Informationsflüsse in den Mittelpunkt der Betrachtung zu stellen.

Software ist im IEEE Standard Glossary of Software Engineering Terminology [IEEE 1990] wie folgt definiert:

> ***Software.*** *Computer programs, procedures and possibly associated documentation and data pertaining to the operation of a computer system.* [IEEE 1990]

Software sind demnach Computerprogramme und evtl. zugehörige Daten und Dokumentation, die zum Betrieb eines Computersystems notwendig sind. Ein Computersystem umfasst dabei sowohl den Personal-Computer (PC) als auch eingebettete Systeme wie Mobiltelefone oder Automobile. Software erfüllt vordefinierte Aufgaben. Software unterstützt den Benutzer bei der Arbeit oder

sie unterhält ihn. Die Bedingungen oder Eigenschaften, die eine Software erfüllen soll, um ein Problem eines Nutzers zu lösen, heißen Anforderungen [IEEE 1990].

Bei der Erstellung von Software geht es darum, ein Problem der Nutzer mit Hilfe von Software zu lösen. Dazu müssen die Anforderungen der Nutzer bekannt sein und in Software umgesetzt werden. Im einfachsten Fall gibt es genau einen Nutzer einer Software und dieser kennt die Anforderungen, die er an die Software hat. Der Fall, dass ein späterer Nutzer die Software selbst erstellt, ist aus verschiedenen Gründen nicht immer sinnvoll oder möglich. Die meisten Nutzer können Software nicht selbst erstellen. Selbst wenn sie es technisch könnten, gibt es viele Aufgaben, die so komplex sind, dass sie eine Person nicht in akzeptabler Zeit alleine lösen kann. Ab einer bestimmten Größe des zu lösenden Problems werden daher die meisten Softwareentwicklungsvorhaben von ausgebildeten Softwareentwicklern in Softwareprojekten durchgeführt. Ein Projekt ist nach dem Project Management Body of Knowledge wie folgt definiert.

> *A **project** is a temporary endeavor undertaken to create a unique product or service.* [PMBOK 1996]

Ein Projekt ist zeitlich eingeschränkt. Es gibt einen Start- und einen Endzeitpunkt. Ein Projekt verfolgt ein Ziel, nämlich die Schaffung eines neuen Produktes, das es so vorher noch nicht gab. In diesem Punkt unterscheidet sich ein Projekt von der Produktion, bei der immer wieder das selbe Produkt erstellt wird. Zur Erreichung des Projektziels werden Ressourcen benötigt. Diese kosten Geld. Softwareprojekte haben die Erstellung von Software als Ziel. Ihre wichtigste Ressource sind die Softwareentwickler. Meist übernehmen die Nutzer der zu erstellenden Software oder die ihnen übergeordnete Organisation die dabei entstehenden Kosten. Der Nutzer wird zum Kunden. Die Benutzer sind insbesondere bei größeren Projekten nicht mehr die einzige Quelle für Anforderungen. Andere Stakeholder, wie dritte Geldgeber, Lieferanten oder das Management, haben auch Anforderungen. Das können zum Beispiel Budget- oder technische Einschränkungen, sowie gelebte Geschäftsprozesse sein. Darüber hinaus können Anforderungen zu Projektstart auch schon in dokumentierter Form vorliegen. Das können zum Beispiel schriftliche Verträge, gesetzliche Vorgaben oder Dokumentationen über zu ersetzende Altsysteme sein. Die Schwierigkeit der Softwareentwicklung besteht also zum Teil darin, alle Quellen von Anforderungen zu kennen und in die Entwicklung einzubeziehen. Z.B. sagt Armour [Armour 2000]:

> *The hard part of building* [software] *systems is not building them, it's knowing what to build — it's in acquiring the necessary knowledge.* [Armour 2000]

Das heißt, dass zu Beginn eines Softwareprojekts alle wichtigen Informationen entweder in den Köpfen der beteiligten und betroffenen Personen oder in Dokumenten vorhanden sind. Im Laufe eines Projekts werden diese Informationen in je nach Projektgröße und Vorgehensmodell unterschiedlich vielen Zwischenschritten in weiteren Köpfen und Dokumenten weiterentwickelt, bis sie schließlich im finalen Produkt, der Software, zusammen fließen. Auch das Wissen darüber, *wie* Software entwickelt wird und wie ein Projekt durchzuführen ist, ist entweder dokumentiert oder in den Köpfen der Entwickler vorhanden. Auch Softwareentwicklungs- und Prozesswissen muss im Projekt zwischen den relevanten Personen fließen, damit das Projekt erfolgreich sein kann.

Insgesamt spielen Informationen und Informationsflüsse in vielen grundlegenden Aktivitäten der Softwareentwicklung eine wichtige Rolle. Informationsflüsse sind daher auch als das Kernelement einer grundlegenden Software-Engineering-Theorie geeignet.

1.2.3 Zusammenfassung des Lösungsansatzes

Die folgenden beiden Punkte sind die Grundidee der vorliegenden Arbeit.

- In dieser Arbeit soll eine *Theorie* entwickelt werden, um das Problem einer fehlenden einheitlichen theoretischen Basis der Softwareentwicklung anzugehen. [Kruchten 2002, Herbsleb 2003, Jacobson 2009, Jacobson 2009a, Jacobson 2009b, Cockburn 2010, Broy 2011]
- Diese Theorie soll *Informationsflüsse* als zentrales Element enthalten (siehe oben).

Damit die Theorie möglichst hilfreich bei der Lösung der in Abschnitt 1.1.3 genannten Probleme ist, werden in dieser Arbeit auch die folgenden Ziele verfolgt.

- Die Theorie soll – soweit möglich und sinnvoll – auf bestehende Erkenntnisse des Software Engineering und anderer Wissenschaften aufbauen.

- Die Theorie soll nicht im Widerspruch mit etablierten Erkenntnissen der Softwareentwicklung stehen.
- Die Theorie soll *Forschern* im Bereich der Softwareentwicklung helfen
  - Probleme zu erklären,
  - in einem gegebenen Softwareentwicklungsprojekt Probleme vorherzusagen,
  - Verbesserungsmethoden zu entwickeln und
  - sich mit einer einheitlichen Sprache über Softwareentwicklung auszutauschen.
- Die Theorie soll beispielhaft als Grundlage für die Erstellung einer Methode genutzt werden, die ein *Praktiker* im Bereich der Softwareentwicklung einsetzen kann, um Softwareentwicklungsprojekte verstehen und verbessern zu können.

1.3 Einordnung und Abgrenzung

Nachdem das Ziel dieser Arbeit erörtert wurde, wird an dieser Stelle abgegrenzt, was in dieser Arbeit nicht betrachtet werden soll. Mit Hilfe der Informationsflusstheorie soll eine Grundlage geschaffen werden, mit der Softwareentwicklung besser verstanden, besser erforscht und praktisch verbessert werden kann. Im Rahmen dieser Arbeit wird nicht darauf eingegangen, was Ingenieurwesen im Kontext der Softwareentwicklung ist. Weiterhin wird nicht die Frage betrachtet, ob Softwareentwicklung nach Ingenieursprinzipien der beste Weg ist, Software zu entwickeln. Der Begriff Software Engineering wird als Name der Forschungsdisziplin als gegeben angenommen und entsprechend benutzt.

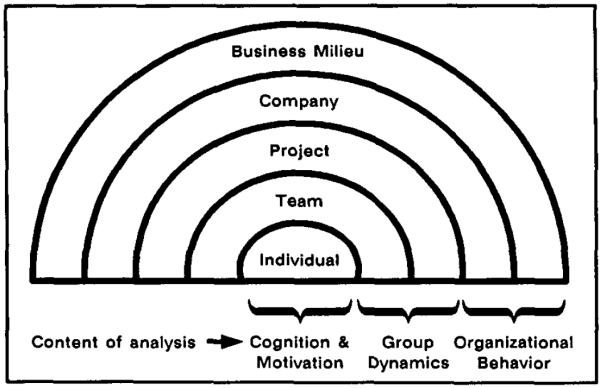


Abb. 1.1: Verhaltensorientiertes Schichtenmodell der Softwareentwicklung (aus [Curtis 1988])

Die Reichweite der Informationsflusstheorie wird mit Hilfe des verhaltensorientierten Schichtenmodells der Softwareentwicklung [Curtis 1988] aus Abb. 1.1 eingegrenzt. Curtis et al. sind der Meinung, dass *"Softwareentwicklung auf verschiedenen Verhaltensebenen untersucht werden sollte"* [Curtis 1988]. Diese Verhaltensebenen eignen sich gut für eine Eingrenzung:

Individual: Auf der individuellen Ebene spielen Dinge wie Kognition, Problemlösen, Wissen und Externalisierung von Wissen eine Rolle. Motivationsaspekte werden nicht betrachtet.

Group (Team, Project): Die Team- und Projektebene ist der Hauptbetrachtungsgegenstand der Informationsflusstheorie. Informationsfluss, Kommunikation, Kommunikationsmedien, Dokumentation etc. sind hier relevant.

Organizational (Company, Business): Auf der Organisationsebene werden nur Informationsfluss-relevante Dinge betrachtet, z.B. projektübergreifende Informationsflüsse. Es werden insbesondere *nicht* politische Dinge und die Machtdimension von Entscheidungen betrachtet.

Die eingangs genannten Probleme der Softwareentwicklung (vgl. Kapitel 1.1.1) werden meist durch den Prozess der Softwareentwicklung verursacht, z.B. wenn relevante Informationen nicht kommuniziert werden. Daher ist die Informationsflustheorie auf den Vorgang der Softwareentwicklung fokussiert und nicht auf das Wesen von Software selbst.

1.4 Forschungsmethode und Aufbau der Arbeit

An dieser Stelle wird die Forschungsmethodik erörtert, die der vorliegenden Arbeit zu Grunde liegt. Anschließend wird die sich daraus ergebende Struktur der Arbeit vorgestellt.

Nach Easterbrook et al. ist Forschungsdesign der Prozess der Auswahl einer Forschungsmethode, die zu einem bestimmten Forschungsproblem passt [Easterbrook 2008]:

> *Research Design is the process of selecting a method for a particular research problem, tapping into its strengths, while mitigating its weaknesses. The validity of the results depends on how well the research design compensates for the weaknesses of the methods.* [Easterbrook 2008]

Die Validität des Ergebnisses der Forschung hängt dann davon ab, wie gut die Vorteile der Methode zum Forschungsproblem passen und wie gut die Nachteile der Methode gemindert wurden.

Da das in dieser Arbeit behandelte Forschungsproblem die Erstellung einer grundlegenden Theorie für die Softwareentwicklung ist (vgl. Kapitel 1.2.3), die nicht im Widerspruch zu etablierten Software-Engineering-Erkenntnissen steht, kann keine Methode wie Grounded Theory verwendet werden, die das Vorgehen zwar systematisch beschreibt, aber die Theorie ausschließlich aus empirischen Daten entwickelt (vgl. [Easterbrook 2008]). Empirische Daten der letzten 40 Jahre Software-Engineering-Forschung liegen nicht vor. Eine Wiederholung einer ausreichend großen Menge an empirischen Studien ist nicht praktikabel. Daher wird für die Entwicklung der Informationsflusstheorie vor allem auf Wissen aus der Software-Engineering-Literatur und angrenzender Gebiete zurückgegriffen (vgl. Kapitel 2.1.1). Die Theoriebildung geschieht über die üblichen induktiven Methoden (vgl. Kapitel 2.1.2). Der Vorteil dieses Vorgehens ist, dass alle veröffentlichten Erkenntnisse in die Theoriebildung einbezogen werden können. Die forscherische Leistung dieses Vorgehens besteht darin, aus der Vielzahl an Erkenntnissen eine Theorie zu abstrahieren, die grundlegend genug ist, um möglichst viele problematische Softwareentwicklungsphänomene beschreiben, und dennoch situationsspezifische Vorhersagen machen zu können (vgl. [Boehm 1989]).

Die so entwickelte Theorie soll nicht nur zu fundiertem Software-Engineering-Wissen passen (vgl. Kapitel 5.1), sondern auch zu empirischen Beobachtungen

(vgl. Kapitel 5.2 und 5.3). Für die empirische Prüfung der Informationsflusstheorie wird ein Quasi-Experiment durchgeführt. Durch die gezielte Kontrolle bestimmter Variablen und durch natürliche Belegung anderer Variablen werden eine große Anzahl von Datenpunkten und gleichzeitig realistischere Ergebnisse ermöglicht (vgl. Kapitel 5.2.2):

- Indem relevante Parameter so kontrolliert werden, dass die Ergebnisse vergleichbar bleiben, können viele Datenpunkte in der Studie erhoben werden. So können signifikante und besser generalisierbare Ergebnisse erzielt werden [Wohlin 2000, Basili 2007].
- Indem andere Parameter nicht kontrolliert werden, können Daten in einer typischen und realistischen Umgebung erhoben werden, was mit dem Ziel einer Fallstudie vergleichbar ist [Kitchenham 2007, Wohlin 2000, S. 12].

Ein weiteres Ziel dieser Arbeit ist die praxistaugliche Nutzbarmachung der Informationsflusstheorie. Dazu wird eine Methode entwickelt, deren Techniken genutzt werden können, um Softwareentwicklungsprojekte zu verbessern. Zwei dieser Techniken wurden empirisch evaluiert. Die FLOW-Mapping-Technik wurde mit einer Fallstudie eines studentischen verteilten agilen Softwareprojekts evaluiert (vgl. Kapitel 6.3.1.5). Die Verfestigung-als-Nebenprodukt-Technik wurde mit einem Fragebogen-basierten Experiment empirisch geprüft (vgl. Kapitel 6.3.2.1).

Insgesamt ergibt sich folgende Struktur für die Arbeit:

2. Grundlagen: In diesem Kapitel werden der für die hier vorgestellte Informationsflusstheorie verwendete Theoriebegriff und andere Grundlagen, die zum Verständnis der Arbeit notwendig sind, eingeführt.

3. Grundbegriffe der Theorie: In Kapitel 3 werden die Grundbegriffe der Informationsflusstheorie definiert. Dabei werden Erkenntnisse aus dem Software Engineering und anderer Wissensgebiete zu einer grundlegenden Softwareentwicklungsterminologie kombiniert.

4. Theoreme der Theorie: In diesem Kapitel werden die Sätze und Hypothesen der Informationsflusstheorie aus den Grundbegriffen hergeleitet.

5. Prüfung der Theorie: In Kapitel 5 wird die Prüfung der Informationsflusstheorie vorgestellt. Die Theorie wird mit verwandten Theorien des Software Engineering verglichen und empirisch evaluiert.

6. Der FLOW-Ansatz: Mit Hilfe des FLOW-Ansatzes wird die praktische Anwendbarkeit der Informationsflusstheorie gezeigt. Auf Basis der Informationsflusstheorie wird eine Methode zur Analyse und Verbesserung von Softwareentwicklung hergeleitet. Ausgewählte Techniken werden vorgestellt und evaluiert.

7. Zusammenfassung: Abschließend werden die Ergebnisse der Arbeit in Bezug auf ihre wissenschaftliche und praktische Relevanz diskutiert und ein Ausblick auf mögliche zukünftige Weiterentwicklungen der Informationsflusstheorie und der FLOW-Methode gegeben.

2 Grundlagen

In diesem Kapitel werden die für die Entwicklung der Informationsflusstheorie notwendigen Grundlagen gelegt. Zunächst wird der Begriff Theorie selbst näher untersucht und die bei der Entstehung einer Theorie relevanten Begriffe wie Definition, Axiom, Satz, Hypothese und Theorem erläutert. Anschließend werden Grundlagen aus der Informatik angrenzenden Fachgebieten vorgestellt, die später in Kapitel 3 und 4 für die Herleitung der Informationsflusstheorie benötigt werden. Es werden Grundlagen für die Themen Wissen, Gedächtnis, Daten, Information, Kommunikation und Abstraktion gelegt. Andere Theorien und Ansichten darüber, was Softwareentwicklung ist, sind aus Sicht dieser Arbeit nicht Grundlage, sondern verwandt, und werden daher als Teil der Prüfung in Kapitel 5 vorgestellt.

2.1 Theorien

In diesem Abschnitt wird zunächst geklärt, was im Rahmen dieser Arbeit unter Theorie verstanden wird. Nach dem Handlexikon zur Wissenschaftstheorie [Seiffert 1989, S. 368] ist Theorie wie folgt definiert.

Definition 2.1: **Theorie [Seiffert 1989]**

"Theorie im Sinne des logischen Empirismus: ein gesichertes Wissen, das aus dem Zusammenwirken von Erfahrung und Denken – und zwar nach ganz bestimmten, in der Theorie bzw. Methodologie der induktiven Wissenschaften beschriebenen Methoden – entsteht." [Seiffert 1989]

Eine Theorie besteht demnach aus einer Sammlung von Wissen, welches entsprechend der induktiven Methode aus dem Zusammenwirken von Erfahrung und Denken entstanden ist. Zudem muss dieses Wissen in irgendeiner

Form gesichert sein. Was als gesichert gilt, wird von der Definition nicht abgedeckt.

2.1.1 Grundbegriffe von Theorien

In diesem Abschnitt werden einige Grundbegriffe erläutert, die im Zusammenhang mit Theorien und Theoriebildung eine wichtige Rolle spielen. Zunächst wird der in der obigen Definition verwendete Begriff der Induktion und ihr Gegenteil, die Deduktion, definiert.

> ***Induktion*** [ist die] *Ableitung des Allgemeinen aus dem Besonderen.*
> [Seiffert 1989, S. 22]

> ***Deduktion*** *ist die Ableitung des Besonderen aus dem Allgemeinen.*
> [Seiffert 1989, S. 22]

Weiterhin spielen die Begriffe Axiom, Satz, Theorem und Hypothese eine wichtige Rolle für die Informationsflusstheorie. In dieser Arbeit werden sie wie folgt beschrieben verwendet.

Definition: Eine Definition ist die Festsetzung der Bedeutung einer zu erklärenden Sache, z.B. eines Wortes.

Axiom: Ein Axiom ist ein nicht deduktiv abgeleiteter Grundsatz einer Theorie, von dem innerhalb der Theorie angenommen wird, dass er wahr ist. Ein Axiom kann z.B. durch Induktion hergeleitet oder einfach nur eine vernünftig erscheinende zu Grunde gelegte Aussage sein.

Satz: Ein Satz ist eine Aussage, die aus Axiomen oder anderen Theoremen logisch deduktiv abgeleitet oder anders formal bewiesen wurde.

Lemma: Ein Lemma ist ein wesentliches Zwischenergebnis bei der Herleitung eines Satzes, welches potenziell für andere Herleitungen wiederverwendet werden kann.

Korollar: Ein Korollar ist eine Aussage, die sich ohne großen Beweisaufwand aus einem bereits bewiesenen Satz, dem Beweis eines Satzes oder aus einer Definition bzw. einem Axiom ergibt.

Hypothese: Eine Hypothese ist eine Aussage, von der angenommen wird, dass sie wahr ist, deren Gültigkeit aber noch zu prüfen ist, z.B. durch eine logische Herleitung aus den Axiomen und Theoremen einer Theorie oder durch empirische Beobachtung. Die in dieser Arbeit formulierten Hypothesen sind meist für eine empirische Evaluation vorgesehen.

Theorem: Theorem wird in dieser Arbeit als Oberbegriff für Satz, Korollar, Lemma und Hypothese verwendet.

Da Definitionen in dieser Arbeit eine wichtige Rolle spielen, wird der Begriff noch etwas genauer untersucht. Definition ist im weitesten Sinne die Festsetzung einer Begriffsverwendung [Mittelstrass 1980, S. 439]. Der zu definierende Begriff heißt Definiendum. Der Teil der Definition, der die Verwendung des Definiendums festsetzt, heißt Definiens. In dieser Arbeit werden die folgenden beiden Definitionsarten benutzt:

Festsetzende Abkürzungsdefinition: Bei der Abkürzungsdefinition wird ein Definiendum für ein Definiens festgelegt, weil der Gebrauch des Definiens wegen dessen Länge unpraktisch geworden ist [Seiffert 1989, S. 27]. Definiendum und Definiens haben die gleiche Bedeutung und können ohne Änderung der Aussage eines Satzes gegeneinander ausgetauscht werden (in dem Kontext, in dem die Definition gilt) [Seiffert 1989, S. 29]. Zudem sind *"festsetzende Definitionen keine Aussagen und können daher auch nicht wahr oder falsch sein"* [Mittelstrass 1980, S. 439].

Explikation: Bei der Explikation wird ein Begriff (Definiendum oder Explikandum) aus *"der vorwissenschaftlichen Alltagssprache oder der noch nicht hinreichend präzisen Wissenschaftssprache"* [Mittelstrass 1980, S. 625] mit Hilfe des Explikats präzisiert. Anschließend kann das Explikandum im Sinne des Explikats genutzt werden. Nach Carnap soll eine Explikation folgenden Adäquatheitsforderungen genügen [Mittelstrass 1980, S. 625, zitiert nach]:

- Ähnlichkeit von Explikat und alltagssprachlichem Begriff
- Exaktheit im Sinne der Aufnahme des Explikats in eine wissenschaftliche Terminologie
- Fruchtbarkeit des Explikats für die Aufstellung neuer Sätze
- Einfachheit des Explikats (den oberen drei nachgeordnete Forderung)

Explikation ist im Zusammenhang mit der hier vorgestellten Informationsflusstheorie ein wichtiges Konzept, da in der Theorie viele Begriffe der Alltagssprache oder anderen Wissenschaften benutzt werden (z.B. Information, Kommunikation), deren Bedeutungen im Rahmen der Theorie präzise festgelegt werden müssen. Zudem stellt nach Radnitzky [Seiffert 1989, S. 73, zitiert nach] eine Explikation einen Teilaspekt einer Theorieentwicklung dar. Eine Explikation ist nach Radnitzky *"der Versuch, einen vorhandenen Begriff* [...] *für die Arbeit an bestimmten theoretischen Problemen durch eine verbesserte Version* (Anm.: im Sinne der Adäquatheitsforderungen) [...] *zu ersetzen."* [Seiffert 1989, S. 75].

Im folgenden Abschnitt werden die induktiven Methoden genauer untersucht.

2.1.2 Induktive Methoden

Der in Definition 2.1 zitierte logische Empirismus ist eine wissenschaftstheoretische Strömung, die fordert, dass sich allgemeine Erkenntnis logisch aus dem in der Realität Erlebten ableiten lässt [Seiffert 1989, S. 268]. Demnach muss eine Theorie aus empirischen Beobachtungen entstanden sein. Empirische Beobachtungen sind immer etwas Konkretes. In diesem Zusammenhang beschreiben induktive Methoden, wie man aus den konkreten Beobachtungen allgemeines Wissen ableitet, d.h. wie man vom Konkreten auf das Allgemeine schließt. Die folgenden beiden induktiven Methoden wurden zur Ableitung der Informationsflusstheorie verwendet [Seiffert 1989, S. 151].

Induktion durch einfache Aufzählung: Bei der Induktion durch einfache Aufzählung wird eine allgemeine Gesetzmäßigkeit bzw. eine Hypothese aus einer Anzahl von Beobachtungen abgeleitet. Zum Beispiel die Hypothese, dass alle Schwäne weiß sind, abgeleitet aus vielen Beobachtungen weißer Schwäne. So hergeleitete Hypothesen sind leicht durch gegenteilige Beobachtungen wiederlegbar, z.B. durch Beobachtung eines schwarzen Schwans.

Ausscheidende Induktion: Bei der ausscheidenden Induktion werden aus der Gesamtzahl der möglichen Hypothesen alle ausgeschaltet, die nicht in Frage kommen. Nach [Mill 1868, S. 453 - 478] werden fünf Fälle der ausscheidenden Induktion unterschieden:

1. **Methode der Übereinstimmung:** Haben zwei oder mehr Beobachtungen einer Sache nur einen Umstand gemeinsam, so wird davon

ausgegangen, dass dieser Umstand die Ursache für die Erscheinung ist.

2. **Methode des Unterschiedes:** Unterscheiden sich zwei oder mehr Beobachtungen nur in einem Umstand, so wird davon ausgegangen, dass dieser Umstand die Ursache für den Unterschied der Erscheinung ist. Mit der Methode des Unterschiedes gewonnene Hypothesen entsprechen mit größerer Wahrscheinlichkeit der Wirklichkeit als Hypothesen, die mit der Methode der Ähnlichkeit gewonnen wurden, weil zum Beispiel in einem Experiment möglichst viele Umstände, bis auf den zu untersuchenden, gleich gewählt werden können.

3. **Vereinigte Methode von Ähnlichkeit und Unterschied:** Haben zwei oder mehr Beobachtungen einer Sache nur einen Umstand gemeinsam, während zwei oder mehr weitere Beobachtungen, die die Sache nicht enthalten, diesen Umstand genau *nicht* haben, so wird davon ausgegangen, dass dieser Umstand die Ursache für die Beobachtung der Sache der ersten Beobachtungsreihe ist.

4. **Methode der Reste:** Zieht man von einer Beobachtung die durch frühere Induktionen bekannten Dinge ab, so wird davon ausgegangen, dass der Rest der Beobachtung die Ursache für den Rest der Erscheinung ist.

5. **Methode der begleitenden Veränderung:** Gibt es bei der Beobachtung von Änderungen einer Sache eine auf eine bestimmte Weise begleitende Änderung einer anderen Sache, so wird davon ausgegangen, dass die beiden Sachen durch einen bestimmten kausalen Zusammenhang verbunden sind.

Die in dieser Arbeit vorgestellte Theorie basiert auf folgenden Beobachtungen:

- Erfahrungen anderer bei der Softwareentwicklung
  - Aus Lehrmaterial (Bücher, Vorlesung)
  - Aus Veröffentlichungen (vgl. Literaturverzeichnis)
  - Aus Erzählungen
  - Durch Beobachtung Anderer

- Eigene Erfahrungen bei der Softwareentwicklung

Zur Herleitung der Definitionen und Axiome der Informationsflusstheorie in Kapitel 3 wurden teilweise Erkenntnisse aus der Informatik angrenzenden Wissensgebieten einbezogen. Dabei wurden größtenteils die Grundlagen aus Lehrbüchern entnommen, vereinzelt aber auch aktuelle Publikationen berücksichtigt. Auf Basis dieser externen Erkenntnisse, von Software-Engineering-Fachwissen und eigenen Beobachtungen wurden dann unter Anwendung der oben beschriebenen induktiven Methoden die Definitionen und Axiome der Informationsflusstheorie festgelegt (vgl. Kapitel 3). In den folgenden Abschnitten werden wichtige Konzepte aus Sicht anderer Wissensbereiche als Grundlage für den Rest der Arbeit vorgestellt.

2.2 Wissen und Gedächtnis

Zentraler Faktor der Softwareentwicklung ist der Mensch. Intelligenz, Kreativität und Problemlösungsfähigkeit des Menschen sind in seinem Gedächtnis verankert. Sie bilden sein Wissen.

2.2.1 Wissen

In diesem Abschnitt werden zunächst verschiedene Ansichten darüber vorgestellt, was Wissen ist. Die Auswahl der Ansichten ist dabei auf betriebswirtschaftliche und die damit verwandten Wissensmanagementansichten beschränkt, weil es vor allem aus betriebswirtschaftlicher Sicht wichtig ist, Softwareentwicklung zu verstehen und zu verbessern. Im Folgenden werden Definitionen von Wissen zitiert, die alle einen ähnlichen Kern haben.

> *Knowledge – the insights, understandings, and practical know-how that we all possess – is the fundamental resource that allows us to function intelligently.* [Wiig 1996]

> *Wissen bezeichnet die Gesamtheit der Kenntnisse und Fähigkeiten, die Individuen zur Lösung von Problemen einsetzen.* [Probst 1997]

> [...] *knowledge* [is] *the human expertise stored in a person's mind, gained through experience, and interaction with the person's environment.* [Sunassee 2002]

> *Representation of a problem-solution in a human-mind is knowledge.* [Rao 2006, indische Sichtweise]

Allen Definitionen ist gemeinsam, dass Wissen unsere Intelligenz ausmacht bzw. wir damit Probleme lösen können, und dass Wissen an eine Person gebunden ist, genauer an deren Verstand.

Wissen wird oft noch in weitere Wissensarten unterteilt. Die im betriebswirtschaftlichen Umfeld wohl bekannteste ist die Unterteilung in explizites und implizites (tacit) Wissen.

2.2.1.1 Explizites und implizites Wissen

Nonaka unterscheidet zwei Arten von Wissen, da ihre unterschiedlichen Charakteristika für das Wissensmanagement im Unternehmen wichtig sind [Nonaka 1991].

> [...] *two very different types of knowledge.* [...]
> *Explicit knowledge is formal and systematic. For this reason, it can be easily communicated and shared, in product specifications or a scientific formula or a computer program.* [...]
> *Tacit knowledge is highly personal. It is hard to formalize and, therefore, difficult to communicate to others. Or, in the words of the philosopher Michael Polanyi, "We can know more than we can tell."* [Nonaka 1991, Nonaka 2007]

Explizites Wissen ist Wissen, welches systematisch und formal ist und daher leicht kommuniziert und weitergegeben werden kann. Im Gegensatz dazu ist implizites Wissen (eng.: tacit knowledge) Wissen, welches höchst personenspezifisch, schwer zu formalisieren und daher auch schwer zu kommunizieren ist [Nonaka 1991, Nonaka 2007]. Eine Art impliziten Wissens sind technische Fähigkeiten, die man erst nach jahrelanger Erfahrungssammlung erhält, und dann trotzdem nicht in der Lage ist, die zu Grunde liegenden wissenschaftlichen und technischen Prinzipien erklären zu können.

Nonaka hat den Begriff tacit von Polanyi übernommen, der eine noch einfachere Beschreibung für implizites Wissen liefert:

> *There are things that we know but cannot tell. This is strikingly true for knowledge of skills.* [Polanyi 1962] oder noch kürzer: *We can know more than we can tell.* [Polanyi 1966]

Polanyi setzt zwar nicht implizites Wissen mit all dem, was man weiß, aber nicht wiedergeben kann, gleich, dennoch ist diese nicht Artikulierbarkeit ein zentraler Bestandteil des impliziten Wissensbegriffs, auch nach Polanyi [Polanyi 1962].

Nonaka leitet aus der Unterscheidung zwischen explizitem und implizitem Wissen vier Möglichkeiten ab, wie ein Unternehmen neues Wissen erzeugen kann [Nonaka 1991, Nonaka 2007].

Sozialisation (implizit zu implizit): Direkte Weitergabe von implizitem Wissen zwischen zwei Personen. Implizite Fähigkeiten werden dabei durch

Beobachtung, Nachahmung und Übung erlernt. Ein typisches Beispiel für Wissensweitergabe als Sozialisation ist das Meister-Lehrling-Lernen. Der Lehrling kann zwar die Fähigkeiten des Meisters erlernen, aber weder Lehrling noch Meister erhalten durch Sozialisation systematische Erkenntnisse über das Wissen ihres Fachs. Für das Unternehmen als Ganzes ist diese Art der Wissenserzeugung schwer zu nutzen, da das Wissen nie explizit wird.

Kombination (explizit zu explizit): Ein Individuum kann durch Verbindung separaten expliziten Wissens neues explizites Wissen erzeugen. Diese Kombination von vorhandenem expliziten Wissen erweitert aber meist nicht die Wissensbasis eines Unternehmens.

Externalisierung (implizit zu explizit): Die Artikulation der Grundlagen impliziten Wissens führt zu explizitem Wissen, und damit zu Wissen, das im Unternehmen weitergegeben und genutzt werden kann.

Internalisierung (explizit zu implizit): Internalisierung ist die Nutzung von explizitem Wissen zur Erweiterung, zum Ausbau und zur Umdeutung des eigenen impliziten Wissens. Individuen einer Organisation können sich durch Internalisierung Wissen anderer aneignen und so ihre Arbeit verbessern.

Eine weitere, auch für die Softwareentwicklung wichtige, Art des Wissens ist das prozedurale Wissen.

2.2.1.2 Prozedurales Wissen

Nach Anderson hat prozedurales Wissen seinen Ursprung im Problemlösen (im gedanklichen Sinne) [Anderson 2000]. Wenn einmal eine Lösung ausgedacht wurde, wird sie durch mehrmaliges Anwenden in das prozedurale Gedächtnis (im impliziten Gedächtnis, siehe unten) überführt. Mit prozeduralem Wissen im impliziten Gedächtnis kann man eine Tätigkeit ausführen, ohne bewusst darüber nachdenken zu müssen (z.B. Fahrrad fahren, Schwimmen).

Es gibt aber auch prozedurales Wissen im expliziten Gedächtnis. Das sind z.B. Fakten darüber, was wie in welcher Reihenfolge gemacht werden muss, um ein bestimmtes Ziel zu erreichen. Dieses Wissen befähigt einen aber nicht direkt. auch das Problem lösen zu können. Die meisten Fahranfänger bekommen die

Kupplung-Gas-Koordination trotz theoretischem Wissen darüber, wie es zu machen ist, erst nach mehrmaligen Üben hin.

Umgekehrt heißt das aber auch, dass implizites prozedurales Wissen aus zwei Gründen nicht direkt an eine andere Person weitergegeben werden kann. Erstens fällt es der Person, die die Fähigkeit bereits beherrscht, eventuell schwer explizit auszudrücken, was sie genau macht. Zweitens muss der Lernende das ihm explizit beigebrachte prozedurale Wissen erst verinnerlichen, also ins implizite Gedächtnis überführen.

Zusammenfassend lässt sich festhalten, das Wissen an den Verstand des Menschen gebunden ist. Eine Untersuchung der Funktionsweise unseres Gedächtnisses kann daher Aufschluss geben, warum es unterschiedliche Arten von Wissen gibt und warum diese unterschiedlich schwierig zu managen sind, d.h. aus dem Gedächtnis des Menschen extrahiert werden können, um sie dauerhaft und wiederholt für das Unternehmen nutzbar zu machen.

2.2.2 Gedächtnis

Die hier vorgestellten Grundlagen für die Funktionsweise des menschlichen Gedächtnisses werden hauptsächlich den Kognitionswissenschaften entnommen. Nach Ansicht einiger Software-Engineering-Forscher bietet die Kognitionswissenschaft viel Potential zur Verbesserung der Softwareentwicklung. Zum Beispiel sagt Curtis dazu [Curtis 1984]:

> *Cognitive science is a paradigm which offers the best opportunity to study and gain control over the largest source of influence of* [software development] *project performance.* [Curtis 1984]

Die im Folgenden aufgeführten Erkenntnisse über das menschliche Gedächtnis stammen hauptsächlich aus Lehrbüchern des Kognitionspsychologen Anderson [Anderson 2000, Anderson 2001]. Andere Quellen und direkte Seitenverweise werden gesondert ausgewiesen.

Zunächst wird eine Unterscheidung der Funktion des Gedächtnisses nach Speicherdauer des Wissens vorgenommen. Dabei wird auch erklärt, warum die lange verbreitete Unterscheidung zwischen Kurzzeit- und Langzeitgedächtnis heute nicht mehr aufrecht erhalten werden kann.

2.2.2.1 Flüchtiges und permanentes Gedächtnis

Prinzipiell lässt sich zwischen einem flüchtigen und einem permanenten Teil des Gedächtnisses unterscheiden. Die lange verbreitete Unterscheidung zwischen Kurzzeit- und Langzeitgedächtnis kann neueren Erkenntnissen zufolge nicht aufrecht erhalten werden [Anderson 2000], da das Vergessen wahrscheinlich keine direkte Funktion der Zeit ist, sondern andere Faktoren einen größeren Einfluss haben, wie zum Beispiel das Speichern neuer interferierender Informationen [Oberauer 2008]. Die Zeit, wie lange eine Information im Gedächtnis gespeichert ist und abgerufen werden kann, ist eine indirekte Größe. Sie ist davon abhängig, wie gut eine neue Informationen beim Abspeichern mit bereits vorhandenem Wissen verknüpft wird. Akustische und visuelle Reize werden schnell wieder vergessen, wenn sie nicht in einen semantischen Zusammenhang mit bereits Bekanntem gebracht werden.

Die etwas neutralere Unterscheidung zwischen flüchtigem und permanentem Gedächtnis trifft demnach die Realität besser. Das flüchtige Gedächtnis umfasst die Funktionen des Gedächtnisses, bei denen Informationen nur vorübergehend verfügbar sind.

Das sensorische Gedächtnis und das Arbeitsgedächtnis erfüllen diese Eigenschaft. Im *sensorischen Gedächtnis* werden die eingehenden Informationen aus den Sinnesorganen, wie zum Beispiel die visuellen Reize über die Augen oder akustische Signale über die Ohren, so lange gehalten, bis das Gehirn feststellen kann, was wahrgenommen wurde, und daraus eine permanente Repräsentation erstellen kann. Im Arbeitsgedächtnis werden Informationen für den aktuellen Denkprozess vorgehalten. Zum Beispiel die Zahlen und Operationen, die zur Lösung einer Kopfrechenaufgabe benötigt werden. Das Arbeitsgedächtnis hat eine limitierte Kapazität. So können zum Beispiel etwa 1,5 bis 2,0 Sekunden akustische Informationen [Anderson 2000, S. 179] oder um die vier visuelle Objekte [Rensink 2000] zur kognitiven Verarbeitung vorgehalten werden.

Alle dauerhaft bzw. längerfristig verfügbaren Informationen sind im permanenten Gedächtnis gespeichert.

Das menschliche Gehirn scheint also ähnlich wie heutige Personal Computer zu funktionieren. Es gibt einen flüchtigen Arbeitsspeicher (vgl. RAM), der aktuell benötigte Informationen zur Weiterverarbeitung hält und eine begrenzte Kapazität hat, und einen dauerhaften Speicher, der Informationen langfristig

speichern kann (vgl. Festplatten, optische Datenträger). Bei beiden können Informationen verloren gehen, wenn sie überschrieben werden oder Zugriffsverknüpfungen fehlen (vgl. defekte Zuordnungstabellen eines Dateisystems). Zudem gibt es keinen direkten Zusammenhang zwischen Zeit und Informationsverlust.

Zimbardo beschreibt Gedächtnis in seinem Lehrbuch zur Psychologie als die Fähigkeit des Menschen, Informationen aufzunehmen, zu speichern (aufzubewahren) und bei Bedarf wieder abzurufen [Zimbardo 2003, S. 234]. Auch in der Kognitionswissenschaft werden drei Phasen des Gedächtnisprozesses unterschieden, um die Funktionsweise des permanenten Gedächtnisses besser analysieren und verstehen zu können. Nach Anderson sind dies die folgenden drei Phasen [Anderson 2000, S. 185] [Anderson 2001, S. 174]:

1. Enkodierung und Speicherung
2. Behalten versus Vergessen
3. Abruf

Es wird also unterschieden, wie Informationen in das Gedächtnis gelangen, wie sie dort gehalten bzw. warum sie vergessen werden und wie sie wieder abgerufen werden. Empirisch ist es nicht möglich die drei Phasen unabhängig voneinander zu betrachten, da immer mindestens zwei Phasen durchlaufen werden müssen. Um festzustellen, ob Informationen gespeichert wurden, müssen sie abgerufen werden. Um Informationen abrufen zu können, müssen sie vorher im Gedächtnis enkodiert werden. Der Fokus kann aber gezielt auf eine Phase gelegt werden.

Für die Softwareentwicklung relevante Funktionen des Gedächtnisses sind:

Problemlösen: Das Problemlösen ist die Hauptaufgabe eines Softwareentwicklers. Durch Nachdenken versucht er das Problem zu verstehen, Lösungsmöglichkeiten zu erarbeiten und gegeneinander abzuwägen, und schließlich eine geeignete Lösung soweit zu detaillieren, bis sie umgesetzt werden kann.

Wissen abrufen: Für das Problemlösen muss ein Entwickler unterschiedliches Wissen abrufen. Er muss sich an die Wünsche des Kunden erinnern, die Problemdomäne kennen, technische und organisatorische Rahmenbedingungen kennen, und er muss Wissen darüber abrufen, wie Computer programmiert werden. Dieses Wissen kann nur so lange aus dem

Gedächtnis abgerufen werden, wie es nicht vergessen wurde. Daher ist es auch sinnvoll zu betrachten, wie lange Wissen im Gedächtnis gespeichert wird und wie das Vergessen funktioniert.

Wissen aufnehmen: Typischerweise muss ein Softwareentwickler in jedem neuen Projekt neues Wissen aufnehmen. Er muss die Anforderungen des Kunden und die Problemdomäne verstehen lernen und evtl. neue Technologien für die Umsetzung lernen. Zudem muss er in größeren Projekten mit mehreren Beteiligten deren Entscheidungen und Zwischenergebnisse kennen, d.h. in sein Wissen aufnehmen.

Im Folgenden werden die Funktionsweise und die Eigenschaften des menschlichen Gedächtnisses aus Sicht der drei Phasen (1) Wissen abspeichern, (2) Wissen behalten bzw. vergessen und (3) Wissen abrufen und näher betrachtet. Als wichtig werden dabei diejenigen Eigenschaften angesehen, die in der Softwareentwicklung eine Rolle spielen können. Anschließend werden noch zwei Unterscheidungen von Gedächtnisteilen vorgestellt, deren Eigenschaften in der Softwareentwicklung ausgenutzt werden können (explizit vs. implizit, vgl. Kapitel 2.2.2.5, und episodisch vs. semantisch, vgl. Kapitel 2.2.2.6).

2.2.2.2 Enkodierung und Speicherung

In der Enkodierungsphase des Gedächtnisprozesses geht es darum, wie Informationen aus der Umwelt des Menschen in sein permanentes Gedächtnis gelangen und welche Mechanismen dafür sorgen, dass sie dort besonders stark – d.h. langfristig abrufbar – gespeichert werden. Die Aufnahme von Informationen und Speicherung im Gedächtnis wird auch als Lernen bezeichnet. Folgende Aktivitäten unterstützen das Lernen und sorgen dafür, dass das gelernte Wissen permanent im Gedächtnis enkodiert wird:

1. Wiederholtes Üben (auch bekannt als "Übung macht den Meister")
2. Inhaltliche (semantische) Auseinandersetzung mit dem zu Lernenden
3. Selbsterzeugung von Wissen

Der Lernerfolg ist abhängig von der Lernmethode. Bloßes Lesen und Merken des zu Lernenden führt zu wenig dauerhaftem Wissen. Tieferes Lernen, bei dem das Neue aktiv mit bereits Bekanntem semantisch in Verbindung gebracht wird, führt zu einem besseren Lernerfolg. So konnte zum Beispiel

gezeigt werden, dass Probanden synonyme Wortpaare (semantischer Zusammenhang) mit einer höheren Wahrscheinlichkeit wiedererkannt haben als sich reimende Wörter (keine semantische Verbindung)[Anderson 2000]. Lernen ist nach [Anderson 2000] dann am effektivsten, wenn Wissen selbst erzeugt wird. Im oben genannten Beispiel wären das z.B. selbst gebildete Reime. Die positiven Effekte können auch kombiniert werden. Die Kombination aus Semantik und selbst erzeugtem Wissen, z.B. selbst gebildete Synonymwortbeziehungen, wie *Meer-Ozean*, führen zu einem noch besseren Lernerfolg. Ein weiterer positiver Effekt auf den Lernerfolg kann erzielt werden, wenn die Lernenden Zugang zu konkreten Beispielen und der abstrakten allgemeinen Lösung haben [Anderson 2001, S. 248].

Bemerkenswert ist, dass die Lernabsicht keine Auswirkungen auf den Lernerfolg hat. In einer Studie wurden Probanden gebeten, Wörter nach bestimmten Kriterien zu bewerten. Eine Hälfte wurde informiert, dass sie nach der Bewertung befragt werden, welche Wörter sie bearbeitet haben. Die andere Hälfte wurde nicht gewarnt. Beide Gruppen konnten sich gleich gut an die Wörter erinnern, obwohl eine Gruppe eher die Absicht hatte, sich die zu bewerteten Wörter auch zu merken [Anderson 2000].

An visuelles Material kann man sich besonders gut erinnern. Insbesondere, wenn mehrere abgebildete Objekte interagieren. Dabei werden Bilder meist nicht photographisch gespeichert, sondern deren Bedeutung.

Emotionen haben einen großen Einfluss auf die Enkodierung. Negative Erfahrungen werden schlecht gelernt, da der Mensch negative Situationen vermeidet und sich somit auch nicht intensiv mit einer solchen beschäftigt. Bei positiven Erfahrungen wird zwar auch weniger gelernt, aber das, was gelernt wird, wird stärker aufrechterhalten. Der Mensch konzentriert sich auf das Positive und blendet andere Dinge aus. Das führt dazu, dass die positive Erfahrung viel stärker semantisch verknüpft und damit länger gespeichert wird.

2.2.2.3 Behalten versus Vergessen

Bei dieser Sichtweise auf die Funktion des Gedächtnisses geht es darum, wie Wissen besonders lange im Gedächtnis behalten werden kann. Oder anders ausgedrückt, welche Faktoren dafür sorgen, dass Wissen wieder vergessen wird. Es gibt im Wesentlichen drei Faktoren, von denen angenommen wird, dass sie Einfluss auf das Vergessen haben:

1. Zeit
2. Interferenzen: Störung durch anderes Wissen (z.B. Überschreiben)
3. Weniger Assoziationen: Fehlende Zugriffsverknüpfungen

Das Vergessen folgt im Allgemeinen über die Zeit einer Potenzfunktion mit negativem Exponenten. Das heißt, dass man anfänglich sehr schnell und über längere Zeiträume immer weniger vergisst. Mittlerweile wird aber davon ausgegangen, dass das Vergessen keine direkte Funktion der Zeit ist, sondern von anderen Faktoren indirekt beeinflusst wird.

Einer dieser Faktoren ist die so genannte Interferenz. Lernt man zum Beispiel zwischen dem Erlernen und Abruf von Wortpaaren (Liste 1) eine andere Liste mit Wörtern (Liste 2), die mit der ersten Liste semantisch interferieren, so hat das negativen Einfluss auf den Abruf der ersten Liste [Anderson 2000]. Semantisch ähnliche Inhalte werden an denselben Stellen im Gedächtnis gespeichert und überschreiben dabei eventuell vorhandenes Wissen. Das Vergessen hängt in dieser Theorie indirekt von der Zeit ab. Je mehr Zeit vergeht, desto größer ist die Wahrscheinlichkeit, dass Interferenzen auftreten, d.h. dass Zugriffspfade auf neues Wissen alte Zugriffspfade auf semantisch ähnliches Wissen überschreiben.

Ein anderer Faktor ist die Anzahl der Zugriffsverknüpfungen. Die Theorie besagt, dass umso mehr Verknüpfungen zu einem Gedächtnisinhalt bestehen, um so größer ist die Wahrscheinlichkeit, dass man eine zum gesuchten Wissen führende Verknüpfung aktivieren und damit das Wissen abrufen kann. Vergessen kann nun über den Abbau dieser Verknüpfungen erklärt werden. Bestehende Verknüpfungen werden über die Zeit auf andere Gedächtnisinhalte umgebogen. Eine Kombination aus Interferenz- und Zugriffsverknüpfungstheorie ist auch vorstellbar. Es ist durchaus plausibel, dass interferierende Inhalte dazu führen, dass Verknüpfungen von bestehenden Inhalten auf die semantisch ähnlichen neuen Inhalte umgebogen werden.

2.2.2.4 Abruf

Um Wissen aus dem Gedächtnis abrufen zu können, benötigt man Hinweise, die die richtigen Zugriffspfade aktivieren. Dabei gibt es einen Unterschied zwischen Erkennen von bereits Erlerntem und dem Abruf dieses Wissens. Bloßes Erkennen fällt den meisten leichter. Zum Beispiel geben viele Studenten an,

dass Multiple-Choice-Fragen einfacher sind als Lückentexte. Dennoch konnte gezeigt werden, dass Erkennen nicht immer besser funktioniert als Abrufen. Bei entsprechend guten Hinweisen können Probanden Wissen abrufen, das sie ohne Hinweise nicht als im Gedächtnis vorhanden erkannt hätten.

Es gibt Techniken, die das Erinnern unterstützen. Diese nutzen z.B. die hohe Effektivität der Selbsterzeugung von Wissen bzw. Verknüpfungen und von visuellem Wissen aus. Eine Technik ist die Loci-Methode. Dabei werden neu zu lernende Informationen über visuelle Bilder mit bereits Bekanntem aktiv verknüpft. Will man sich zum Beispiel eine Einkaufsliste merken, so versucht man die Elemente der Liste über eine möglichst lebhafte Geschichte mit z.B. markanten Wegpunkten des eigenen Arbeitswegs zu verknüpfen. Das führt dazu, dass man sich an so gespeicherte Informationen nahezu perfekt erinnern kann. Die Technik wird daher auch von Gedächtnissportlern genutzt.

Der Kontext (die Umgebung), in dem etwas gelernt wurde, hat großen Einfluss auf den späteren Abruf dieses Wissens. Beim Lernen werden Kontextinformationen zu Hinweisen, die das spätere Abrufen verbessern können. So ist es zum Beispiel sinnvoll in dem Raum für eine Klausur zu lernen, in dem später die Prüfung stattfindet. Lernkontext ist aber nicht nur die Umgebung sondern auch der innere Zustand der Person. So haben Aufregung, Glück oder gar Hunger Einfluss auf das Erinnern, je nachdem in welchem Zustand man sich beim Erlernen befand.

2.2.2.5 Explizites und implizites Gedächtnis

Das explizite Gedächtnis ist ein Teil des permanenten Gedächtnisses, aus dem Wissen bewusst abrufen werden kann. Das Gegenstück ist das implizite Gedächtnis. Wissen kann aus dem impliziten Gedächtnis nicht bewusst abgerufen werden.

> *For memory, the main contrast* [between explicit and implicit memory] *involves the presence or absence of deliberate or conscious recollection.* [Kirsner 1998, S. 14]

Der Hauptunterschied zwischen explizitem und implizitem Gedächtnis ist also das Vorhandensein bzw. die Abwesenheit bewusster Erinnerung. Folgendes Beispiel soll den Unterschied zwischen explizitem und implizitem Gedächtnis verdeutlichen. Ein geübter Zehnfingerschreiber weiß offensichtlich, wo jeder

Buchstabe auf der Tastatur ist. Dennoch fällt es den meisten schwer, auf Anfrage die genaue Position eines bestimmten Buchstaben auf der Tastatur zu bestimmen. Erst wenn ein Wort, das diesen Buchstaben enthält, getippt wird (im Geiste, oder genauer, im Arbeitsgedächtnis) und man die Finger verfolgt, kommt man auf die genaue Position.

Das implizite Gedächtnis scheint unabhängig vom expliziten Gedächtnis zu sein. So konnte z.B. gezeigt werden, dass Amnesiepatienten, die kein explizites Wissen mehr erlernen können, trotzdem noch implizite Fähigkeiten wie eine neue Sprache oder einen mathematischen Algorithmus lernen und ausführen können, obwohl sie behaupten dies noch nie gesehen bzw. gelernt zu haben [Anderson 2000].

2.2.2.6 Episodisches und semantisches Gedächtnis

Das semantische Gedächtnis speichert unser aus der Welt gewonnenes Wissen über die Welt. Das episodische Gedächtnis ist der Teil des Gedächtnisses, der für die bewusste Erinnerung an persönliche Ereignisse und Situationen verantwortlich ist [Tulving 2009]. Der Unterschied zwischen semantischem und episodischem Gedächtnis exisitert in der Realität nicht, d.h. es gibt keine unterschiedlichen neurophysiologischen Vorgänge. Im Kontext der Softwareentwicklung ist es aber sinnvoll diese Unterscheidung zu treffen, da Speicherung über autobiografische Vorgänge sehr viel langfristiger ist als Speicherung durch einfaches Lernen (vgl. z.B. Erfahrungen in Kapitel 6.1.2).

Das episodische Gedächtnis ermöglicht dem Menschen die mentale Zeitreise in die Vergangenheit oder sogar in die Zukunft. Drei wesentliche Komponenten des episodischen Gedächtnisses unterscheiden es von dem semantischen Gedächtnis und ermöglichen damit die mentale Zeitreise [Tulving 2009]:

Subjektives Zeitempfinden. Das bewusste mentale Nacherleben vergangener oder die Vorstellung zukünftiger Ereignisse.

Das Selbst. Die bewusste Vorstellung von sich selbst als eine vom Rest der Welt separierten Einheit. Das episodische Gedächtnis ist egozentrisch, ichbezogen.

Autonoetisches Bewusstsein. Die Kombination aus subjektivem Zeitempfinden und dem Selbst ergibt ein autonoetisches Bewusstsein, dass es dem Menschen ermöglicht, sich bewusst an autobiografische Ereignisse der

Vergangenheit zu erinnern oder sich in zukünftigen Situationen zu sehen.

Im episodischen Gedächtnis gespeicherte Informationen beinhalten also immer einen zeitlichen ich-bezogenen Aspekt. Das Selbsterlebte wird mit anderen Informationen verknüpft. Diese zusätzlichen Verknüpfungen der Zeit und des Selbst schaffen im Gedächtnis neue Zugriffspfade, die es ermöglichen, wichtige Informationen umfangreicher und schneller abzurufen. Dieser Vorteil kann auch gezielt in der Softwareentwicklung ausgenutzt werden. So haben zum Beispiel Kersten und Gail zeigen können, dass aufgabenzentrierte Entwicklungsumgebungen (Eclipse + Mylyn) die Produktivität von Programmierern signifikant steigern können [Kersten 2006, Kersten 2007]. Daher ist es sinnvoll diesen Gedächtnistyp separat in der Informationsflusstheorie zu betrachten.

2.3 Daten und Dokumente

Daten und Dokumente sind Konzepte, die eine wichtige Rolle bei der Softwareentwicklung spielen. Daten sind nach IEEE-Definition von Software (vgl. Def. 1.2.2) Teil der Software und für dessen Ausführung notwendig [IEEE 1990]. Dokumente sind zentraler Bestandteil vieler Vorgehensmodelle zur Softwareentwicklung, z.B. dem Rational Unified Process (RUP) [RUP 2003] oder dem V-Modell XT [VModell 2009].

2.3.1 Daten

> *A datum is a putative fact regarding some difference or lack of uniformity within some context.* [Floridi 2006]

Nach [Floridi 2006] ist ein Datum ein Fakt, der einen Unterschied oder das Fehlen von Gleichförmigkeit in einem Kontext darstellt. Der Begriff Datum ist vom griechischen Wort "diaphora" (Betonung des Unterschieds zweier Dinge – aus der Rhetorik) abgeleitet. Floridi unterscheidet dabei zwischen Unterschieden in der realen Welt ("diaphora de re"), zwischen physischen Zuständen ("diaphora de signo") und zwischen Zeichen ("diaphora de dicto"). Dabei lässt sich jeweils das letztere auf das vorhergehende zurückführen. So sind Zeichen (dicto) erst durch Signale (signo) möglich. Ein Buchstabe (dicto) ist zum Beispiel im Computer durch eine Folge von Bits (signo) repräsentiert. Signale wiederum begründen sich auf Unterschiede der realen Welt. So wird der Wert eines Bits (signo) zum Beispiel durch unterschiedliche physikalische Spannungen (re) repräsentiert [Floridi 2006].

2.3.2 Dokumente

Dokumente spielen in der Softwareentwicklung eine wichtige Rolle. Spätestens seit dem Wasserfallmodell sind sie zentrales Zwischenprodukt im Softwareentwicklungsprozess [Ludewig 2010, S. 157]. Zudem sind sie Teil der fertigen Software (vgl. Def. 1.2.2). Viele Vorgehensmodelle haben Dokumente als zentralen Bestandteil (u.a. [VModell 2009, RUP 2003], vgl. z.B. [Stapel 2006]), auch wenn sie teilweise mit Synonymen wie Artefakt [UMLStruct 2011] oder Produkt [VModell 2009] bezeichnet werden. Im IEEE Standard 610.12-1990

[IEEE 1990] ist Dokument im Kontext des Software Engineerings wie folgt definiert.

> *A medium, and the information recorded on it, that generally has permanence and can be read by a person or a machine.* [IEEE 1990]

Demnach ist ein Dokument ein Speichermedium für Informationen. Eine typische Eigenschaft von Dokumenten ist, dass sie beständig sind, d.h. dass man über einen längeren Zeitraum auf die enthaltenen Informationen zugreifen kann. Eine ähnliche Definition wird in der UML-Superstructure-Spezifikation [UMLStruct 2011] für Artefakt gegeben.

> *An artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system. Examples of artifacts include model files, source files, scripts, and binary executable files, a table in a database system, a development deliverable, or a word-processing document, a mail message.*
>
> [...]
>
> *An Artifact defined by the user represents a concrete element in the physical world.*
> [UMLStruct 2011, S. 204]

Hier wird eine weitere Eigenschaft von Dokumenten hervorgehoben, nämlich das physische Vorhandensein in der realen Welt. Diese Eigenschaft bringt den Begriff Dokument wieder in Zusammenhang mit dem Begriff Daten. Nach Floridi [Floridi 2006] (oben) sind Daten physisch unterscheidbare Dinge. Somit könnte man ein Dokument auch als Datenspeicher ansehen, wobei die Daten eine Repräsentation der Informationen sind (vgl. Kapitel 2.4 und später Kapitel 3.4). Eine weitere wichtige Eigenschaft von Dokumenten ist die Struktur der darin enthaltenen Daten. So stellt Buckland [Buckland 1997] folgendes als wichtigen Unterschied zwischen einer unstrukturierten Ansammlung von Daten und einem Dokument fest:

> [...] *that it is indexicality — the quality of having been placed in an organized, meaningful relationship with other evidence* [data] *— that gives an object its documentary status.* [Buckland 1997]
>
> [...]
>
> *Suzanne Briet equated "document" with organized physical evidence.* [Briet 1951], zitiert nach [Buckland 1997]

Ein Dokument sind demnach organisierte Daten, sofern man "physical evidence"[1] mit Daten gleich setzt.

Im Fokus der Softwareentwicklung steht aber weniger das Dokument selbst, sondern der Prozess des Erstellens eines Dokuments, d.h. die Dokumentation, weil *"die Dokumentation als ewiges Sorgenkind der Software-Entwicklung."* [Ludewig 2010, S. 259] gilt. Ludewig und Lichter nennen zwei Gründe, warum das so ist [Ludewig 2010, S. 267]:

- Software-Entwickler können auf Grund mangelnder Ausbildung nicht gut dokumentieren.
- Dokumentation fällt dem hohen Zeitdruck der meisten Projekte zum Opfer.

Folgen fehlender oder schlechter Dokumentation sind dann [Ludewig 2010, S. 267]:

- In der Wartung fehlen wichtige Überlegungen und Entscheidungen der Entwicklung der Software.
- *"Grundlagen für Handbücher fehlen."* [Ludewig 2010, S. 267]
- Aus einem Projekt ohne Dokumentation kann eine Organisation nicht für zukünftige Projekte lernen.

Insgesamt haben Dokumente in der Softwareentwicklung folgende Funktion [Ludewig 2010, S. 260 ff]:

- Know-how-Transfer innerhalb des Entwicklungsteams.
- Als Mittel zur Kommunikation zwischen Auftraggeber und Auftragnehmer.
- Sie bewahren das Wissen über die entwickelte Software.
- Sie sind die Grundlage für systematische Prüfungen. Was jemand "im Kopf" hat, kann nicht so leicht geprüft werden.
- Dokumente erleichtern die Feststellung des Projektfortschritts, weil sie materiell sind und damit leichter "gemessen" werden können.

[1] "Physical evidence" ist die englische Übersetzung von Buckland [Buckland 1997] des französischen Originals "indice concrete ou symbolique" von Briet [Briet 1951], welches man auch direkt ins Deutsche als "sachliches oder symbolisches Zeichen/Indiz" übersetzen könnte.

- Gute Dokumentation hilft, Fehler zu vermeiden oder zumindest zu finden.
- Gute Dokumentation erleichtert die Erweiterung und Wiederverwendung der Software.

In der Softwareentwicklung lassen sich Dokumente nach Ludewig und Lichter [Ludewig 2010, S. 262] in eine der folgenden vier Kategorien einordnen. Mit Hilfe der Kategorien kann leichter bestimmt werden, welchem Zweck ein Dokument dient und für welche Zielgruppe es verfasst ist.

Systemdokumentation: Alle Dokumente, die für Konstruktion, Betrieb und Wartung der Software benötigt werden. Die Software selbst (Quellcode) zählt auch zur Systemdokumentation. Beispiele sind Anforderungsspezifikation, Entwurf (Konstruktion, Wartung), Handbücher oder Programmcode (Betrieb).

Projektdokumentation: Alle Dokumente für die Planung und Durchführung des Softwareentwicklungsprojekts. Beispiele sind Projektpläne, Verträge oder Statusberichte.

Qualitätsdokumentation: Alle Dokumente, die die Maßnahmen der Qualitätssicherung festhalten, z.B. Testberichte oder Reviewprotokolle.

Prozessdokumentation: Alle Dokumente, die den Entwicklungsprozess und seine Umsetzung im Projekt beschreiben. Beispiele sind Standards, Richtlinien oder Dokumententemplates.

2.4 Information

Nachdem die Grundlagen zu Wissen, Gedächtnis, Daten und Dokumenten gelegt wurden, die alle etwas mit Information zu tun haben, bleibt die folgende Frage zu klären: Was ist Information? Diese Frage kann nicht mit einer allgemein anerkannten und fachbereichsübergreifend gültigen Definition beantwortet werden. Die Beschäftigung mit dieser Frage hat eine lange Tradition [Capurro 2000, Kap. 4] und hat ganze Wissenschaftszweige hervorgebracht. So beschäftigen sich zum Beispiel die Informationstheorie, die Informationswissenschaft sowie die Semiotik mit Information. Die folgende Auseinandersetzung mit dem Begriff Information ist zu großen Teilen aus [Stapel 2006] entnommen. Die Originalquellen werden weiterhin zitiert.

Im Folgenden wird auf verschiedene Ansätze zur Definition von Information in den Wissenschaftsbereichen Informatik, Betriebswirtschaftslehre und Philosophie eingegangen.

2.4.1 Information als Verringerung von Ungewissheit

Shannon definiert einen quantitativen Informationsbegriff, der es ermöglicht, die Menge der über einen Kanal transportierten Information anzugeben [Shannon 1948]. Die Bedeutung, die eine Information hat, spielt für seine Betrachtungen keine Rolle und wird deshalb außen vor gelassen. Nach Shannon werden bei der Kommunikation Entscheidungen eines Auswahlprozesses aus einer Menge von Zeichen über einen Kanal geschickt. Der Auswahlprozess entscheidet, welches Zeichen aus einer festgelegten Menge gemeint ist. Bei zwei Zeichen bedarf es einer Ja-/Nein-Entscheidung. Bei vier Zeichen hingegen bedarf es mindestens zwei solcher Entscheidungen, usw. Da sich alle Entscheidungsprozesse auf eine Folge von binären Entscheidungen reduzieren lassen, kann man diese auch binär kodieren und die einzelnen Bits verschicken. Der Empfänger wertet diese aus, bis er mit Sicherheit sagen kann, welches Zeichen ihm geschickt wurde.

Angenommen, die Zeichenmenge sei $\{A, B, C, D\}$ kodiert als $\{00, 01, 10, 11\}$, dann ist der Empfänger – solange noch keine Entscheidung geschickt wurde – ungewiss, ob A, B, C oder D gemeint ist. Wird ihm nun das erste Bit 0 mitgeteilt, nimmt seine Ungewissheit ab. Die Auswahl beschränkt sich jetzt nur noch auf A oder B. Nach Empfang des 2. Bits 0 ist seine Ungewissheit

beseitigt. Er hat die Entscheidungen in Form der beiden Bits 00 und damit die Information *A* erhalten. Information ist demnach die Verringerung von Ungewissheit.

Ein Maß für die Ungewissheit – genauer die Menge der Information – ist der Logarithmus Dualis ($ld()$) über die Anzahl verschiedener Zeichen. Dieser Wert entspricht aufgerundet der Anzahl zu übertragender Ja-/Nein-Entscheidungen (Bits) und bietet den weiteren Vorteil, dass die übertragene Informationsmenge eher dem intuitiven Verständnis entspricht: Werden zwei Zeichen aus der oben genannten 4-elementigen Menge hintereinander über einen Kanal übertragen, gibt es $4 \cdot 4 = 16$ verschiedene Kombinationsmöglichkeiten. Intuitiv würde sich der Informationsgehalt bei der Übertragung zweier aufeinander folgender Zeichen, im Vergleich zur Übertragung eines Zeichens, jedoch höchstens verdoppeln. Dies ist bei dem logarithmischen Maß gegeben: $2 \cdot ld(4) = ld(16) = 4$.

Diese ursprünglich Hartley [Hartley 1928] beschriebene Sicht auf Information wurde von Shannon aufgegriffen und erweitert. Hartley ging davon aus, dass alle Zeichen gleich häufig auftreten. Shannon hingegen machte sich die Tatsache zunutze, dass bestimmte Zeichen in einer Sprache häufiger benutzt werden als andere, um die zu übertragenden Entscheidungen noch weiter zu verringern. So konnte er die maximale Kapazität eines gegebenen Kommunikationskanals weiter erhöhen. Schließlich beachtete Shannon noch die Möglichkeit von gestörten Kanälen, die seine Theorie praxisrelevant und damit so erfolgreich gemacht hat.

Bereits Shannon stellte fest, dass es kaum vorstellbar ist, dass ein einzelnes Informationskonzept zufrieden stellend für die zahlreichen möglichen Anwendungen dieses allgemeinen Gebiets beiträgt.

> *It is hardly to be expected that a single concept of information would satisfactorily account for the numerous possible applications of this general field.*
> [Shannon 1953]

Da bei der Informationsübertragung in der Softwareentwicklung eher Verständnisprobleme als Kapazitätsprobleme auftreten, wird im Folgenden auf weitere Konzepte des Begriffs Information eingegangen, die weniger den Aspekt der enthaltenen Informationsmenge, sondern die Semantik von Information in den Vordergrund stellen.

2.4.2 Information als Vorstufe zur Ressource Wissen

Im betriebswirtschaftlichen Umfeld spielt die Informationsmenge nicht die entscheidende Rolle, sondern vielmehr der Wert und Nutzen von Information. Dieser ist eng verbunden mit der Semantik, also mit dem eigentlich nützlichen Inhalt. Viele Konzepte zum Informationsbegriff betrachten diesen im Zusammenhang mit den Begriffen Daten (vgl. Kap. 2.3) und Wissen (vgl. Kap. 2.2). Häufig bauen diese aufeinander auf. So zum Beispiel auch bei Probst et al. [Probst 1997], bei denen der Wissensbegriff im Mittelpunkt steht (vgl. Kap. 2.2). Sie stellen den Zusammenhang zwischen den Ebenen Daten, Information und Wissen als einen Anreicherungsprozess dar. So werden Zeichen durch Syntaxregeln zu Daten, die in einem Kontext interpretierbar sind und damit für einen Empfänger Information darstellen. Eine Vernetzung von Informationen macht daraus Wissen. Dieses ist im Gegensatz zu Daten und Informationen an Personen gebunden und kann von ihnen zur Lösung von Problemen genutzt werden. Information ist so betrachtet die Nahtstelle zwischen Daten und Wissen. Die drei Begriffe lassen sich zusammenfassend wie folgt voneinander abgrenzen:

Daten: Zeichen bzw. Symbole, die entsprechend bestimmter Syntaxregeln angeordnet sind.

Information: Daten, in einem bestimmten Kontext interpretiert.

Wissen: Im menschlichen Geist zur Problemlösung vernetzte Information.

Informationen sind also Daten im Sinne von syntaktisch strukturierten Zeichen, die von einem Empfänger in einem bestimmten Kontext interpretiert werden. Der Kontext ist demnach wichtig, um aus den an sich bedeutungslosen Daten Information zu machen. Kontext ist notwendig für die Extraktion der Bedeutung einer Information. Im folgenden Abschnitt wird der Zusammenhang zwischen Information und dem zur Interpretation notwendigen Kontext näher betrachtet.

2.4.3 Information und Kontext

Dey definiert Kontex für den Bereich der Computer-Benutzerinteraktion wie folgt [Dey 2001]:

> *Context is any information that can be used to characterise the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between the user and the application, including the user and the applications themselves.* [Dey 2001]

Demnach ist Kontext Information, die die Situation einer Person, eines Ortes oder eines Objektes beschreiben kann. Die Definition von Dey ist eng an den Anwedungsbereich der Computer-Benutzerinteraktion gebunden. Zimmermann et al. erweitern die Definition von Dey um formale und operationale Elemente [Zimmermann 2007]:

> [Allgemeiner Teil:] *Context is any information that can be used to characterize the situation of an entity* [Dey 2001].
>
> [Formaler Teil:] *Elements for the description of this context information fall into five categories: individuality, activity, location, time, and relations.*
>
> [Operationaler Teil:] *The activity predominantly determines the relevancy of context elements in specific situations, and the location and time primarily drive the creation of relations between entities and enable the exchange of context information among entities.* [Zimmermann 2007]

Durch diese 3-stufige Definition soll ein gemeinsames Verständnis des Kontextbegriffes auf Benutzer- und Entwicklerseite von kontextsensitiven Systemen erreicht werden [Zimmermann 2007]. Zudem kann Kontext etwas sein, was in einer Gruppe gemeinsames Verständnis schaffen kann:

> *A shared context emerges, when the contexts of two entities overlap and parts of the context information become similar and shared.* [...] *Additionally, a group of entities sharing certain context parts share knowledge of how things are done and understood in this group.* [Zimmermann 2007]

Eine von der Anwendungsdomäne unabhängigere Betrachtung des Kontextbegriffes liefern Bazire und Brézillon. Sie haben 66 Definitionen von Kontext analysiert und sind zu folgendem Schluss gekommen [Bazire 2005]:

> *Finally we can say that context occurs like what is lacking in a given object for a user to construct a correct representation.* [Bazire 2005]

Auf abstrakter Ebene, nach Entfernung der Spezifika der verschiedenen Wissenschaftsbereiche, aus denen die einzelnen Definitionen stammen, ist Kontext also etwas, das bei einem gegebenen Objekt fehlt, um einem Nutzer des

Kontextes zu ermöglichen, eine korrekte Repräsentation des Objekts zu konstruieren. Am Beispiel von Information als Daten plus Bedeutung könnte man Kontext demnach als notwendiges Vorwissen zur korrekten Bedeutungszuweisung gegebener Daten definieren. Wobei Information das Objekt, eine Person der Nutzer, das Vorwissen das was fehlt (also der Kontext) und die intendierte Bedeutung der Information die korrekte Repräsentation ist.

Auch Brézillon definiert Kontext als Wissen, genauer als die Summe von drei Wissensarten: (1) Externes Wissen, (2) kontextuelles Wissen und (3) prozeduraler Kontext [Brezillon 2005]. Der Kontext ist dabei nötig, um eine Entscheidung bei einer gerade durchgeführten Tätigkeit treffen zu können. Externes Wissen ist der Teil des Kontextes, der für die Entscheidungsfindung im aktuellen Schritt der Tätigkeit nicht notwendig ist. Kontextuelles Wissen ist das gerade benötigte Wissen. Prozeduraler Kontext ist eine Untermenge des kontextuellen Wissens, die dem gegebenen Fokus entsprechend aktiviert, strukturiert und angeordnet und allen am Entscheidungsprozess beteiligten Personen gemeinsam ist. Brézillon kommt zu dem Schluss, dass Kontext nur im Zusammenhang mit Wissen und Schlussfolgerungen sinnvoll betrachtet werden kann [Brezillon 2005].

Der Information Technology Metadata Standard ISO/IEC 11179-4 von 1995 [ISO 1995] beschreibt den Zusammenhang zwischen Daten und dem geteilten gemeinsamen Wissen zur korrekten Interpretation folgendermaßen:

> *When two or more parties exchange data, it is essential that all are in explicit agreement on the meaning of that data. One of the primary vehicles for carrying the data's meaning is the data element definition. Therefore, it is mandatory that every data element have a well-formed definition; one that is clearly understood by every user. Poorly formulated data element definitions foster misunderstandings and ambiguities and often inhibit successful communication.*

Demnach ist es essenziell, dass sich alle Beteiligten einer Kommunikation über die Bedeutung von Daten einig sind. Andernfalls kann es zu Missverständnissen oder gar zum Fehlschlagen der Kommunikation kommen.

Im Folgenden wird das Wort Information als Daten plus Bedeutung definiert, was einer ähnlichen Sichtweise wie der bisher beschriebenen entspricht.

2.4.4 Information als Daten plus Bedeutung

Floridi betrachtet Information mit Fokus auf den semantischen Inhalt. Er beschränkt sich dabei auf deklarative objektive semantische (DOS) Information [Floridi 2005], also solche, die für gewöhnlich in Datenbankmanagementsystemen gespeichert und bearbeitet wird. DOS-Information ist nach Floridi und einigen anderen Daten plus Bedeutung [Probst 1997, North 1998, Floridi 2005].

Aufbauend auf dem Datenbegriff (vgl. Kap. 2.3) konkretisiert Floridi in der Standard-Definition von Information (SDI), was genau mit Daten plus Bedeutung gemeint ist.

SDI: α ist eine Informationseinheit im Sinne von DOS-Information, genau dann wenn:

1. α aus mindestens einem Datum bzw. Daten besteht,
2. die Daten wohlgeformt sind und
3. die wohlgeformten Daten eine Bedeutung haben. [Floridi 2005]

SDI 1. impliziert, dass es keine Information ohne Daten geben kann. Es ist mindestens ein Datum erforderlich, damit Information entstehen kann. SDI 2. stellt sicher, dass die Daten einer festgelegten Syntax folgen. SDI 3. repräsentiert schließlich den Kern der Definition, nachdem wohlgeformte Daten nur zur DOS-Information werden, wenn sie eine Bedeutung haben. Diese ist unabhängig von der informierten Person. Die Daten sind also zumindest potenziell bedeutsam. So erschließt sich zum Beispiel durch die Lektüre einer chinesischen Tageszeitung für einen des Chinesischen nicht mächtigen Leser keine Bedeutung der darin enthaltenen Zeichen. Man würde aber trotzdem sagen, dass diese Zeitung Information enthält, da ein Chinese den Zeichen (Daten) die intendierte Bedeutung entnehmen könnte. Ähnlich ist es bei ägyptischen Schrifttafeln. Die Bedeutung erschließt sich erst nach einer Interpretation der Hieroglyphen. Die Information war aber schon vor Kenntnis einer Interpretation der Zeichen in den Tafeln enthalten [Floridi 2006]. Es ist heute leider nicht mehr möglich, die Korrektheit der Übersetzung einer Hieroglyphenschrift zu überprüfen. Die Bedeutung der Zeichen kann mehr oder weniger nur aus ihrer Nutzung in den verschiedenen Texten „erraten“ werden. Es ist denkbar, dass zwar eine Interpretation gefunden wurde, die die damit übersetzten Texte sinnvoll erscheinen lässt, aber nicht der ursprünglichen ägyptischen Bedeutung

entspricht. Die Informationen, die aus den möglicherweise falsch übersetzten Schrifttafeln abgeleitet wurden, könnten also falsch sein. Es stellt sich die Frage, ob diese Falsch-Informationen überhaupt noch Informationen sind. Diese Frage wird im Folgenden näher betrachtet.

2.4.5 Information als nicht personengebundenes Wissen

Nach den Ausführungen in Kapitel 2.2.1 ist Wissen etwas, was an das Gedächtnis einer Person gebunden ist und daher nicht direkt weitergegeben werden kann. Wie bereits in [Stapel 2006, S. 23] erläutert, kann Information als potenzielles nicht an eine Person gebundenes Wissen verstanden werden und stellt somit "transportfähiges" Wissen dar. Eine Information kann das Wissen mehrerer Personen erweitern und damit mehrere Personen dazu in die Lage versetzen ein neues Problem zu lösen. Voraussetzung dafür ist, dass die Personen das zur korrekten Interpretation der Information notwendige Vorwissen besitzen, den so genannten Kontext (vgl. Kapitel 2.4.3 oben). So kann zum Beispiel eine Wegbeschreibung (die Information) mehrere Personen unabhängig voneinander dazu in die Lage versetzen (das Wissen), ein bestimmtes Ziel ohne Umwege zu erreichen (das Problem), wenn die Personen wissen, wie die Wegbeschreibung zu interpretieren ist (der Kontext).

Zusammenfassend lässt sich feststellen, dass Information etwas ist, dass:

- im Gegensatz zu Daten Bedeutung hat,
- Bedeutung mit Hilfe von Kontext erhält und
- personenunabhängiges potenzielles Wissen ist.

2.5 Kommunikation

Kommunikation ist ein Begriff, der ähnlich wie Information in vielen Wissenschaften eine wichtige Rolle spielt. Daher gibt es viele Definitionen und Interpretationen darüber, was genau Kommunikation ist. Im Kontext der hier vorgestellten Theorie ist Kommunikation ein Informationsfluss zwischen Personen. Quelle und Ziel einer Kommunikation sind daher immer Menschen. Mit Hilfe von Kommunikation kann also Wissen einer Person an eine andere Person weitergegeben werden. Um das hier benutzte Kommunikationsmodell besser abgrenzen zu können, werden im Folgenden einige grundlegende Kommunikationsmodelle vorgestellt.

2.5.1 Allgemeines Kommunikationsmodell nach Shannon

Ein häufig anzutreffendes und so genanntes allgemeines Kommunikationsmodell (auch Sender-Empfänger-Modell) ist das von Shannon [Shannon 1948]. Es ist in den Kommunikationswissenschaften weit verbreitet, obwohl es ursprünglich zur Modellierung von technischen Kommunikationssystemen erstellt wurde. Wahrscheinlich hat es sich als grundlegendes Modell durchsetzen können, weil es Kommunikation auf einer abstrakten Ebene mit leicht verständlichen Begriffen erklärt. Abbildung 2.1 zeigt ein Kommunikationssystem nach [Shannon 1948]. Im Folgenden werden die wesentlichen Teile des allgemeinen Kommunikationsmodells kurz erläutert (in Anlehnung an [Shannon 1948]):

Nachricht: Die zu kommunizierende Information, z.B. eine Menge von Buchstaben.

Informationsquelle: Der Ursprung der Nachricht oder einer Menge von Nachrichten, z.B. eine Person, die einen Text (d.h. eine Nachricht) schreibt.

Sender: Erstellt aus der Nachricht ein Signal, dass über einen Kanal übertragen werden kann, z.B. die Umwandlung von Schallwellen in elektrische Spannungen beim Telefon.

Kanal: Das Medium, dass ein Signal von Sender zu Empfänger übertragen kann, z.B. ein Kabel oder ein Funkfrequenzband.

Störung: Eine Störung ist eine Veränderung des Signals auf dessen Weg vom Sender zum Empfänger, z.B. durch interferierende Funksignale bei der Funkübertragung (Störsignal im selben Frequenzband).

Empfänger: Typischerweise die Umkehroperation des Senders. Der Empfänger rekonstruiert die Nachricht aus dem empfangenen Signal.

Informationsziel: Die Person (oder das Ding), für die die Nachricht bestimmt ist.

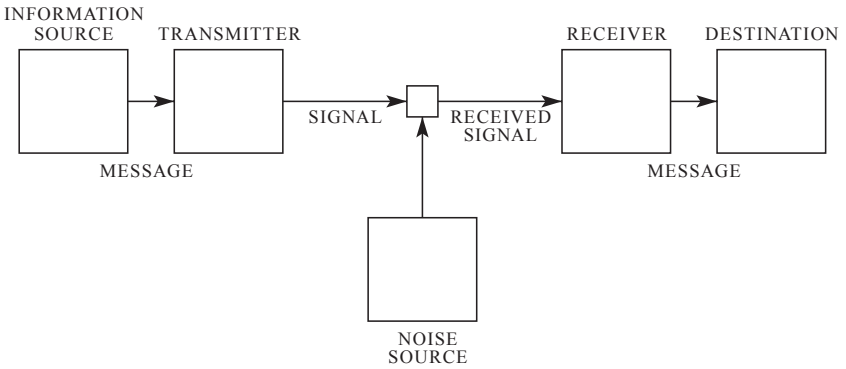


Abb. 2.1: Allgemeines Kommunikationsmodell nach Shannon [Shannon 1948]

2.5.2 Kommunikationstheorie nach Watzlawick

Watzlawick et al. haben eine Kommunikationstheorie entwickelt, die das Verhalten der Kommunikationsteilnehmer beschreibt [Watzlawick 1969]. Grundlage der Theorie ist ein Sender-Empfänger-Modell (siehe oben), bei dem Sender und Empfänger Personen sind. Während einer so genannten Interaktion wechseln Sender und Empfänger ihre Rollen (teils mehrfach). Kern der Theorie sind fünf Axiome, die die Wirkung zwischenmenschlicher Kommunikation über den bloßen Informationsaustausch hinaus beschreiben. Watzlawick bezeichnet den Teil der Kommunikation, der über inhaltliche Kommunikation hinaus geht, als Metakommunikation und formuliert die folgenden fünf metakommunikativen Axiome:

1. Man kann nicht nicht kommunizieren. Nachrichten bestehen nicht nur aus Worten. Auch Tonfall, Schnelligkeit, Pausen, Lachen, Seufzen, Körperhaltung, Mimik, Gestik usw. können Teil der Nachricht sein. All das ist das Verhalten des Senders. Da es kein Gegenteil zu Verhalten gibt – man kann sich nicht *nicht* verhalten – und jedes Verhalten einen Nachrichtencharakter besitzt, kann man nicht *nicht* kommunizieren. Kommunikation findet also auch dann statt, wenn sie nicht absichtlich, bewusst oder erfolgreich ist.

2. Jede Kommunikation hat einen Inhalts- und einen Beziehungsaspekt, wobei letzterer den ersten bestimmt. Neben dem Inhalt der Nachricht, der zu übermittelnden Information, möchte der Sender auch transportieren, wie der Empfänger die Nachricht verstehen soll. Dies bezeichnet Watzlawick als die persönliche Stellungnahme zum Anderen. Dabei wird selten bewusst direkt über die Beziehung zueinander gesprochen. Bei einer gesunden Beziehung zwischen den Gesprächspartnern rückt diese in den Hintergrund. Bei einer kranken Beziehung kann es passieren, dass der eigentliche Inhalt einer Nachricht keine Rolle mehr spielt und nur noch der Beziehungsaspekt kommuniziert wird.

3. Die Natur einer Beziehung ist durch die Interpunktion der Kommunikationsabläufe seitens der Partner bestimmt. Interpunktion sind die von jedem Kommunikationsteilnehmer subjektiv empfundenen Startpunkte einer Interaktion, die vom jeweiligen Sender als Begründung einer neuen Nachricht genutzt werden. Kommunikation kann nur dann erfolgreich sein, wenn beide Kommunikationspartner vom selben Startpunkt ausgehen. Bei unterschiedlichen Meinungen über die Interpunktion, kann es zu Beziehungskonflikten und zu endlosen kreisförmigen Ursache-Wirkungs-Abhängigkeiten kommen. Zum Beispiel, wenn sich der Ehemann zurückzieht, weil die Ehefrau kritisiert und die Ehefrau kritisiert, weil sich der Mann zurückzieht.

4. Menschliche Kommunikation bedient sich digitaler und analoger Modalitäten. Digitale Kommunikation hat eine komplexe logische Syntax (z.B. Sprache), hat aber auf der Beziehungsebene keine Semantik. Umgekehrt hat die analoge Kommunikation eine Semantik auf der Beziehungsebene (z.B. Mimik), aber keine Syntax, um eine Beziehung eindeutig definieren zu können. Mit analogen Modalitäten wird häufig Beziehungsinformation ausgetauscht, mit digitalen Modalitäten Inhalte. Die Kommunikation gelingt, wenn digitale und analoge Nachrichten übereinstimmen. Probleme kann es geben, wenn beide Nachrichten nicht übereinstimmen.

5. Zwischenmenschliche Kommunikationsabläufe sind entweder symmetrisch oder komplementär. Symmetrische Kommunikation findet zwischen gleichen Kommunikationspartnern statt (z.B. gleiche soziale Stellung). Sie versuchen Ungleichheiten zu minimieren. Komplementäre Kommunikation findet zwischen ungleichen Partnern statt (z.B. ungleiche soziale Stellung). Die unterschiedlichen Verhaltensweisen ergänzen sich.

2.5.3 Kommunikationstheorie nach Schulz von Thun

Auch die Kommunikationstheorie nach Schulz von Thun basiert auf einem einfachen Sender-Empfänger-Modell, bei dem eine Nachricht verschickt wird. Jede Nachricht hat dabei vier Seiten, von denen aus sie gleichsam betrachtet werden kann – das so genannte Kommunikationsquadrat [Thun 1981]:

Sachinhalt: Der Sachinhalt ist die eigentliche Information, die vermittelt werden soll.

Beziehung: Hier teilt der Sender mit, was er vom Empfänger hält (Du-Botschaft) und wie beide zueinander stehen (Wir-Botschaft). Non-verbale Sprachelemente, wie Tonfall oder Blicke, können dabei eine große Rolle spielen.

Selbstoffenbarung: Bei jeder Nachricht schickt der Sender auch Information über sich selbst mit. Dies kann sowohl bewusst (Ich-Botschaft) als auch unbewusst (z.B. Erröten bei Scham) geschehen.

Appell: Hier geht es um die auf die gesendete Information folgende erwartete Handlung. Der Sender teilt dem Empfänger mit etwas zu tun oder zu unterlassen. Dies kann direkt und offen oder verdeckt, manipulierend geschehen.

Der Sender schickt also immer vier Botschaften. Der Empfänger erhält diese Botschaften, kann sie aber anders verstehen, als vom Sender intendiert. Das ist eine Ursache für Probleme bei der Kommunikation. Das Vier-Ohren-Modell stellt dar, wie die vier Botschaften einer Nachricht empfangen werden können:

Das Sach-Ohr: Ein ausgeprägtes Sach-Ohr hört nur auf die Information einer Nachricht und überhört die anderen Aspekte. Zwischenmenschliche Beziehungen können dabei nur schwer (weiter-)entwickelt werden.

Das Beziehungs-Ohr: Ein ausgeprägtes Beziehungs-Ohr neigt dazu alle Botschaften auf sich persönlich zu beziehen. Der eigentliche Inhalt kann dabei leicht überhört werden.

Das Selbstoffenbarungs-Ohr: Ein ausgeprägtes Selbstoffenbarungs-Ohr hört verstärkt auf die Ich-Botschaften des Senders und kann so zum Beispiel leichter indirekte Botschaften erkennen.

Das Appell-Ohr: Ein ausgeprägtes Appell-Ohr sieht in jeder Nachricht eine Handlungsaufforderung. Es fragt sich: "Was soll ich mit dieser Nachricht machen?"

2.5.4 Kommunikationsmedien und Kommunikationskanäle

Ein Kommunikationsmedium – auch Kommunikationsmittel – ist ein Hilfsmittel, das der Kommunikation dient [Duden 1996]. Nach [WikiCommMedia] können Kommunikationsmittel nach unterschiedlichen Kriterien weiter unterteilt werden.

Empfänger: Individualkommunikationsmittel, Gruppenkommunikationsmittel und Massenkommunikationsmittel

Technik: Natürliche und technische Kommunikationsmittel. Eine heute immer wichtiger werdende Spezialform technischer Kommunikationsmittel ist die computervermittelte Kommunikation.

Sprache: Verbale und nonverbale Kommunikationsmittel

Kommunikationsphase: Speichermedien, Bearbeitungsmedien und Übertragungsmedien

Da in der Softwareentwicklung Sender und Empfänger einer Kommunikation üblicherweise bekannt sind und ihre Anzahl meist zumindest abgeschätzt werden kann, sollen Massenkommunikationsmittel von der weiteren Betrachtung ausgeschlossen werden. Zudem soll Kommunikationsmedium hier nicht als Informationsträger im Sinne eines Speichermediums und nicht als Bearbeitungsmedium, sondern als Informations*über*träger im Sinne eines Übertragungsmediums verstanden werden. Alle weiteren Unterteilungen sind auch mit dem hier verwendeten Begriff von Kommunikationsmedien weiter möglich.

Ein Kommunikationskanal ist ein technisches Medium zur Übertragung von Signalen von einem Sender zu einem Empfänger [Shannon 1948]. Ein Kommunikationskanal kann also zur Datenübertragung benutzt werden und stellt damit ein Hilfsmittel für die Kommunikation dar. Demnach ist ein Kommunikationskanal ein Kommunikationsmedium nach obiger Definition. Der Begriff Kommunikationskanal wird daher auch häufig synonym mit dem Begriff Kommunikationsmedium verwendet [Kritzenberger 2004, S. 168].

Es werden unterschiedliche Eigenschaften von Kanälen und Medien betrachtet, um Effektivität und Effizienz bewerten und vergleichen zu können. Typische technische Eigenschaften eines Kanals sind maximale Bandbreite, Latenz, Synchronität (synchron vs. asynchron) oder Richtungsabhängigkeit (einseitig vs. beidseitig). Wobei sich einige dieser Eigenschaften gegenseitig beeinflussen. Zum Beispiel hat die Richtungsabhängigkeit eines Kanals Einfluss auf dessen Synchronität, da ein einseitiger Kanal (z.B.: Radio) keine synchrone Kommunikation erlaubt. Andere, nicht technische Eigenschaften, beziehen den Menschen als Start und Ziel einer Kommunikation mit ein, z.B. Sichtbarkeit oder Kopräsenz.

Welche Faktoren Einfluss auf die Eigenschaften und damit die Wahl von Medien haben wurde bereits in vielen unterschiedlichen Forschungsbereichen betrachtet. Eine frühe Erkenntnis ist, dass Bandbreite keinen direkten Einfluss auf Effektivität der Kommunikation hat. So haben zum Beispiel Chapanis et al. gezeigt, dass ein zusätzlicher visueller Kanal zur Sprache keinen Effekt auf die Kommunikationseffektivität bei problemlösenden Aufgaben hat [Chapanis 1972, Chapanis 1977]. Dennoch ist Bandbreite auch heute noch ein zentraler Aspekt, wenn es darum geht, unterschiedliche Medien miteinander zu vergleichen. So ordnet zum Beispiel Cockburn verschiedene Medien anhand ihrer Reichhaltigkeit [Cockburn 2007, S. 149], eine Eigenschaft, die sich aus der Bandbreite ableitet.

Social Presence Theory Eine der frühen Theorien zur Medienwahl ist die Social Presence Theory (SPT) nach Short et al. [Short 1976]. Der Fokus der Theorie liegt auf sozialen Hinweisen[2], die meist visuell übermittelt werden, z.B. über Mimik und Gestik. Die Hauptaussage der Theorie ist, dass das Fehlen von visuellen Informationen den emotionalen Teil der Kommunikation beeinflusst und somit auch höhere Kommunikationsziele wie Verhandlungen oder

[2]Engl.: social cueing

Entscheidungsfindungen beeinträchtigen kann [Whittaker 2003]. Nach STC unterscheiden sich Medien in der Fähigkeit, Ziele, Einstellungen und Motive der anderen Kommunikationsteilnehmer zu transportieren. Direkte Kommunikation von Angesicht zu Angesicht und Videokonferenzen sind Medien mit hoher sozialer Präsenz, weil sie zwischenmenschliche Informationen visuell übertragen können. Schriftliche Kommunikation hingegen hat eine geringe soziale Präsenz [Whittaker 2003].

Group Task Circumplex Die Wahl eines Kommunikationsmediums ist auch von der zu lösenden Aufgabe abhängig. Nach McGrath lassen sich die meisten Gruppenaufgaben in vier grundlegende Kategorien klassifizieren [McGrath 1984, Straus 1994]: erzeugende, wählende, verhandelnde und ausführende Tätigkeiten (vgl. Abb. 2.2, generation, choice, negotiation, execution). Die vier Aufgabentypen stehen in einem zweidimensionalen Raum in Verbindung zueinander, im so genannten Group Task Cirmcuplex aus Abbildung 2.2. Die horizontale Dimension beschreibt, welche Anforderungen die Aufgabe an die kognitive bzw. ausführende Leistungsfähigkeit der Gruppenmitglieder hat. Sie reicht von rein intellektuellen Tätigkeiten, z.B. der Wahl aus einer gegebenen Menge von Alternativen, bis zu rein ausführenden Tätigkeiten, z.B. das Befolgen einer Anleitung. Die vertikale Dimension beschreibt die Anforderungen auf Grund der emotionalen Wechselbeziehungen zwischen den Gruppenmitgliedern in drei Stufen: Kollaboration, Koordination und Konfliktlösung [Straus 1994]. Die vertikale Dimension hat großen Einfluss auf den Bedarf an emotionalen zwischenmenschlichen Informationen bei der Kommunikation und damit auch auf die Wahl eines geeigneten Mediums.

Media Richness Theory Die Idee der Anordnung von Medien nach ihrer Reichhaltigkeit stammt ursprünglich aus der Media Richness Theory von Daft und Lengel [Daft 1984, Daft 1986, Daft 1987, Lengel 1989]. Im Wesentlichen besagt die Media Richness Theory, dass Kommunikationsmedien nach der kontinuierlichen Größe Informationsreichhaltigkeit geordnet werden können. Informationsreichhaltigkeit ist dabei das Vermögen von Information, Verständnis innerhalb eines Zeitintervalls zu ändern[3]. Beispiele für Kommunikationsmedien in absteigender Reihenfolge ihrer Reichhaltigkeit sind (1) von

[3]Engl. Originalzitat: "Information richness is defined as the ability of information to change understanding within a time interval." [Daft 1986]

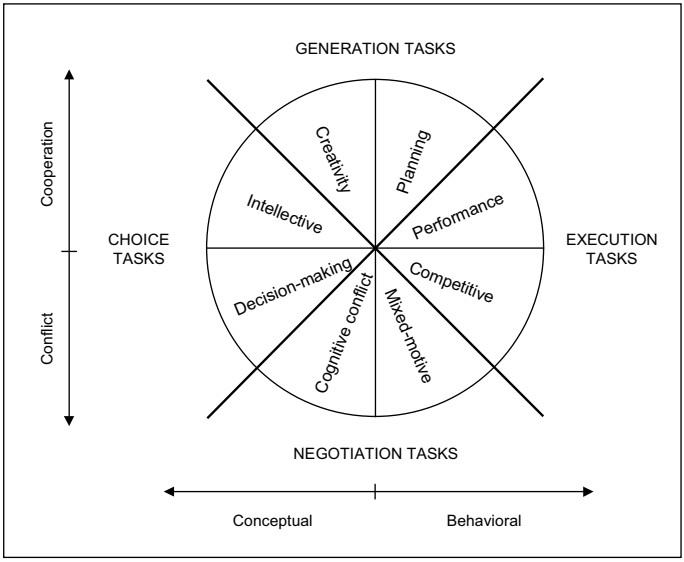


Abb. 2.2: Group Task Circumplex von McGrath [McGrath 1984, S. 61], Abbildung aus [Wilson 2001]

Angesicht zu Angesicht, (2) Telefon, (3) persönliche Dokumente oder (4) unpersönliche Dokumente. Die Reichhaltigkeit eines Mediums hängt von den genutzten Kanälen (z.B. natürliche Sprache, visuell), der Möglichkeit sofortigen Feedbacks, sozialer Hinweise (z.B. Körpersprache) und zur Personalisierung der Nachrichten ab.

Des weiteren besagt die Theorie, dass je unklarer die Informationen sind, die für die Bewältigung einer Aufgabe benötigt werden, desto höher sind die Anforderungen an die Reichhaltigkeit des Kommunikationskanals, der für die Aufgabe genutzt wird. Die Wahl eines Mediums, das nicht reichhaltig genug für die geplante Aufgabe ist, führt zu einer Minderung der Qualität des Resultats der Aufgabe. Umgekehrt sind weniger reichhaltige Medien besser für Aufgaben geeignet, bei denen Informationen einfach fehlen, ihre Interpretation bei Vorhandensein aber klar wäre. Die Reichhaltigkeit hat also direkten Einfluss auf den Kommunikationserfolg.

Grounding Clark und Brennan nennen acht Eigenschaften von Medien, die einen Einfluss auf das so genannte Grounding haben. Grounding ist der Prozess der Sicherstellung, dass das Gesagte einer Konversation vom Kommunikationspartner auch verstanden wurde [Clark 1991].

Kopräsenz: Die Kommunikationspartner teilen sich die gleiche physikalische Umgebung. Es gibt nur ein Medium, was dies ermöglicht, nämlich von Angesicht zu Angesicht (face-to-face).

Sichtbarkeit: Die Kommunikationspartner können sich gegenseitig sehen.

Hörbarkeit: Die Kommunikationspartner können sich gegenseitig hören.

Synchronität: Der Empfänger bekommt Nachricht sofort oder kurz nachdem der Sender sie abgeschickt hat.

Simultanität: Die Kommunikationspartner können gleichzeitig Nachrichten senden und empfangen. (auch Gleichzeitigkeit oder Richtungsabhängigkeit).

Sequenzierbarkeit: Die zusammenhängende Reihenfolge von Einzelbeiträgen einer Konversation. Bei direkter Kommunikation von Angesicht zu Angesicht werden die Kommunikationspartner üblicherweise nicht von Beiträgen aus anderen Kommunikationen mit anderen Personen unterbrochen. Bei E-Mail- oder Brief-basierter Kommunikation können eine Nachricht und ihre Antwort von beliebig vielen anderen irrelevanten Nachrichten und Aktivitäten unterbrochen werden.

Überprüfbarkeit: Alte Nachrichten können noch einmal angesehen werden. Bei verbaler Kommunikation sind vergangene Nachrichten nicht erneut einseh- bzw. abhörbar. Bei der Nutzung textueller Medien oder wenn verbale Kommunikation aufgezeichnet wird, bleiben Artefakte zurück, die noch einmal zur Überprüfung des bisherigen Kommunikationsverlaufs vom Sender, Empfänger oder dritten Parteien, eingesehen werden können.

Korrigierbarkeit: Nachrichten können vor dem Senden korrigiert werden. Zum Beispiel können E-Mails vor dem Abschicken korrekturgelesen und ggf. verbessert werden. Bei direkter verbaler Kommunikation ist das nicht möglich. "Gesagt ist gesagt".

Das Vorhandensein dieser Eigenschaften in einem Medium hat Einfluss auf das Grounding. Fehlt eine Eigenschaft, so führt das zu höheren Grounding-Kosten, was wiederum zu einer längeren oder weniger erfolgreichen Kommunikation führen kann.

Media Synchronicity Theory Eine weitere Theorie, die die geeignete Wahl eines Kommunikationsmediums für eine gegebene Kommunikationssituation ermöglichen soll und so den Kommunikationserfolg planbar bzw. vorhersehbar machen soll, ist die Media Synchronicity Theory (MST) [Dennis 1999, Dennis 2008]. Die Theorie schlägt vor, dass sich alle Gruppenkommunikationsprozesse aus Kombination der zwei Grundkommunikationsprozesse *Übermittlung* und *Fokussierung* zusammenstellen lassen, und das, unabhängig von der zu lösenden Aufgabe. Übermittlung ist der Austausch von Informationen gefolgt von Überlegungen über ihre Bedeutung. Fokussierung ist die Einigung aller Kommunikationspartner auf ein gemeinsames Verständnis der ausgetauschten Informationen. Einigung auf ein gemeinsames Verständnis ist dabei die Einigung auf die Bedeutung der Information und die Einigung, dass man sich darüber geeinigt hat oder eine Einigung nicht möglich ist.

Des Weiteren haben Medien nach der Media Synchronicity Theorie fünf Eigenschaften, die eine wichtige Rolle bei der Durchführung der Grundkommunikationsprozesse spielen. Kommunikationseffizienz wird verbessert, wenn diese Eigenschaften mit den Anforderungen der Kommunikationsprozesse abgestimmt werden. Die folgenden fünf Eigenschaften von Medien können Einfluss auf die Kommunikation haben:

Unmittelbarkeit von Feedback: Die Möglichkeit eines Mediums schnelle bidirektionale Kommunikation zu unterstützen, d.h. wie unmittelbar der Empfänger einer Nachricht auf diese antworten kann.

Symbolvielfalt: Die Anzahl unterschiedlicher Möglichkeiten, in denen Information kommuniziert werden kann. Sie schließt Daft und Lengel's [Daft 1986] Reichhaltigkeit (Sprachvielfalt, nutzbare Kanäle, Möglichkeit sozialer Hinweise) ein. Das Wesentliche von Kommunikation und Sprache sind Symbole [Littlejohn 1983]. Es kann zwischen natürlichen (visuell, verbal, physisch) und künstlichen (geschrieben) Symbolmengen unterschieden werden [Dennis 2008]. Es gibt mindestens vier Arten, wie Symbolvielfalt Kommunikation und das Verständnis von Nachrichten beeinflusst:

1. Manche Information kann in einem Format leichter ausgedrückt werden, als in einem anderen. Symbolmengen, die auf den Nachrichteninhalt angepasst sind, führen zu effektiverer und effizienterer Kodierung [Dennis 2008].

2. Verbale und nonverbale Symbole ermöglichen dem Sender Information zu schicken, die über die puren Wörter hinaus geht.

3. Die Art und Weise, wie ein Sender eine Nachricht erstellt oder ein Empfänger eine Nachricht versteht, kann je nach verwendeter Symbolmenge durch Produktionskosten [Clark 1991] bei der Erstellung einer Nachricht und durch Verzögerungskosten [Reinsch 1990] beim Empfangen einer Nachricht verändert werden.

4. Das Fehlen verbaler und nonverbaler Symbole kann einen signifikanten Einfluss auf die soziale Wahrnehmung haben [Williams 1977].

Parallelität: Die Anzahl möglicher simultaner Konversationen eines Mediums. Traditionelle Medien wie das Telefon ermöglichen nur eine Konversation zu einer Zeit, wohingegen viele elektronische Medien mehrere gleichzeitige Konversationen erlauben (z.B. Instant-Messaging-Systeme).

Verifizierbarkeit: Verifizierbarkeit ist der Grad eines Mediums, der es dem Sender einer Nachricht ermöglicht, diese vor dem Verschicken noch einmal kontrollieren und verbessern zu können.

Wiederverarbeitbarkeit: Wiederverarbeitbarkeit ist der Grad eines Mediums, der es im Kontext eines Kommunikationsvorgangs ermöglicht Nachrichten noch einmal verarbeiten zu können (d.h. noch einmal zu lesen oder zu hören).

Schließlich definieren Dennis und Valacich Mediensynchronität als den Grad zu dem Individuen zur selben Zeit an der selben Aufgabe arbeiten, d.h. einen gemeinsamen Fokus haben[4] [Dennis 1999]. Allgemein wird niedrige Mediensynchronität für den Grundkommunikationsprozess der Übermittlung bevorzugt und hohe Mediensynchronität für den Grundkommunikationsprozess der Fokussierung.

[4]Engl. Originalzitat: "Media synchronicity is the extent to which individuals work together on the same activity at the same time; i.e., have a shared focus." [Dennis 1999]

Weiterhin beschreibt die MST drei Faktoren des Kontextes in dem die Kommunikation passiert, die Einfluss auf die relative Verteilung von Übermittlung und Fokussierung eines Kommunikationsprozesses haben. Die drei Faktoren sind Vertrautheit der Kommunikationsteilnehmer miteinander, mit der Aufgabe und mit dem genutzten Medium [Dennis 2008].

Dennis et al. fassen die Aussage der MST wie folgt zusammen:

> *The 'best' medium is that which best provides the set of capabilities needed by the situation: the individuals, the communication processes, and the social context within which they interact.* [Dennis 2008]

Dabei kann das "beste Medium" je nach Situation auch eine Kombination von Medien sein [Karim 2005, Watson 2007, Dennis 2008].

Media Naturalness Theory Kock schlägt eine Ergänzung bzw. eine Alternative zur Media Richness Theory vor, die Media Naturalness Theory [Kock 2005]. Nach dieser Theorie ist evolutionsbedingt das natürlichste Kommunikationsmedium die direkte Kommunikation von Angesicht zu Angesicht (face-to-face). Natürliche Sprache, Kopräsenz und Synchronität zeichnen dieses Medium aus. Medien, die von diesem natürlichsten Medium abweichen, beeinflussen zunächst nicht direkt die Effizienz oder Qualität der Kommunikation, sondern die kognitive Anstrengung, die Mehrdeutigkeit der Kommunikation und das physiologische Arousal[5]. Wie dies wiederum Einfluss auf Effizienz und Qualität hat, wird von der Media Naturalness Theory nicht beantwortet. DeLuca et al. haben herausgefunden, dass durch häufige Nutzung unnatürlicher Medien ein Lerneffekt eintritt, sodass die kognitive Anstrengung bei häufiger Nutzung sinken kann [DeLuca 2006a].

Zusammenfassung Tabelle 2.1 fasst die vorgestellten Theorien und ihre Kernaussagen zusammen. Tabelle 2.2 gibt eine Übersicht der Faktoren, die nach diesen Theorien Einfluss auf die Medienwahl haben.

[5] Grad der Aktivierung des zentralen Nervensystems, wie Aufmerksamkeit, Wachheit, Reaktionsbereitschaft

Tabelle 2.1: Zusammenfassung von Theorien zur Medienwahl

Theorie	**Faktoren mit Einfluss auf Medienwahl**	**Vorhersagen**
Social Presence Theory [Short 1976]	Soziale Präsenz	Visuelle Kanäle haben Einfluss auf emotionale zwischenmenschliche Kommunikation
Group Task Circumplex [McGrath 1984]	Aufgabentyp	Wechselbeziehungen der Gruppenmitglieder (kollaborativ, koordiniert, Konflikt) korrelieren mit Bedarf an sozialer Präsenz
Media Richness Theory [Daft 1984]	Reichhaltigkeit	Je unklarer die Informationen, desto reichhaltiger sollte das Medium sein
Grounding [Clark 1991]	Grounding-Kosten	Hohe Kosten führen zu weniger erfolgreicher Kommunikation
Media Synchronicity Theory [Dennis 2008]	Synchronität	Übermittlung bedarf niedriger Synchronität, Fokussierung bedarf hoher Synchronität
Media Naturalness Theory [Kock 2005]	Natürlichkeit	Natürliche Medien sind weniger kognitiv anstrengend und ermöglichen eindeutige Kommunikation

Tabelle 2.2: Zusammenfassung von Faktoren aus der Literatur mit Einfluss auf die Medienwahl

Faktor	Kurzbeschreibung	Beispiel
Kopräsenz	örtliche Nähe	Angesicht zu Angesicht vs. mediierte Kommunikation
Sichtbarkeit	visuelles Medium	Videokonferenz
Hörbarkeit	auditives Medium	Telefon
Feedback, Antwortgeschwindigkeit, Synchronität	Latenz des Mediums	Telefon (direkt, synchron) vs. E-Mail (verzögert, asynchron)
Simultanität	Möglichkeit des Mediums, dass alle Beteiligten gleichzeitig kommunizieren	Simplex vs. Vollduplex
Sequenzierbarkeit, Parallelität	Möglichkeit des Mediums eine Kommunikationsaktivität zu unterbrechen bzw. die Anzahl gleichzeitig stattfindender unabhängiger Kommunikationsaktivitäten	Ein Meeting vs. mehrere parallel stattfindende Text-Chats
Überprüfbarkeit, Wiederverarbeitbarkeit	Kommunikationsverlauf ist wiederabrufbar	alte E-Mails können immer wieder gelesen werden, alte Telefonate sind nicht mehr verfügbar
Korrigierbarkeit, Verifizierbarkeit	Möglichkeit des Mediums Nachrichten vor dem Senden noch einmal korrigieren zu können	E-Mail kann vor dem Senden korrigiert werden
Symbolvielfalt	Anzahl unterschiedlicher Möglichkeiten Information zu repräsentieren	Sprache und Mimik
Zusammengesetzte Faktoren		
Reichhaltigkeit: Kombination aus Kanalvielfalt (Sprache, visuell), Feedback, sozialer Hinweise und Personalisierung von Nachrichten	Vermögen eines Mediums Verständnis pro Zeitintervall zu ändern	Angesicht zu Angesicht > Telefon > persönliche Dokumente > unpersönliche Dokumente
Synchronität: Kombination aus Feedback, Symbolvielfalt, Parallelität, Verifizierbarkeit und Wiederverarbeitbarkeit	Grad zu dem Individuen zur selben Zeit an der selben Aufgabe arbeiten	Angesicht zu Angesicht > Telefon > E-Mail > Dokumente
Natürlichkeit: Kombination aus natürlicher Sprache, Synchronität und Kopräsenz (inkl. Hörbarkeit, Sichtbarkeit, Simultanität)	Natürlichstes Medium ist direkte Kommunikation von Angesicht zu Angesicht	Persönliches Gespräch vs. Sofortnachrichten (IM)

2.6 Abstraktion

Jeder Mensch kann abstrahieren, da es eine wesentliche Fähigkeit des menschlichen Denkens ist.

> *Lebenswichtig für uns sind die Modelle, die wir als Begriffe kennen und verwenden, um uns ein Bild (d.h. ein Modell) der Realität zu machen. Ohne die Begriffe wäre für uns jeder Gegenstand ganz neu; weil wir aber zur Abstraktion fähig sind, können wir die Identität eines Gegenstands, seinen Ort, seinen Zustand und u.U. viele andere individuelle Merkmale ausblenden, um ihn einer Klasse von Gegenständen, eben dem Begriff, zuzuordnen.* [Ludewig 2010, S. 3]

Beim Abstrahieren fassen wir gleichartige Dinge der Realität zusammen und geben dieser Zusammenfassung einen Namen (Begriff, Modell oder Klasse) [Mill 1868b, S. 205 ff]. Dabei werden die Eigenschaften der realen Dinge, die nicht gleichartig sind, weggelassen. Umgekehrt trifft dann eine Abstraktion auf eine große (potenziell größer als die zur Erstellung der Abstraktion genutzte) Menge von realen Dingen zu:

> *An abstraction A is abstraction not just of one object X, but also of other objects Y, … which are different from X in those properties which A ignores.*
>
> *An abstraction A thus defines (specifies) an entire class of objects, the class of objects of which it is an abstraction.* [Schuenemann 2004]

Bei der Abstraktion wird also auf das Wesentliche einer Menge realer Dinge reduziert. Durch diese Reduktion wird die Menge der betrachteten Dinge vom Menschen einfacher handhabbar. Daher sind Abstraktion und Modellierung wichtige Hilfsmittel zur Beherrschung von Komplexität. Das Gegenteil von Abstraktion ist Konkretisierung, also das Hinzufügen von Eigenschaften zu einem abstrakten Begriff. Beim Konkretisieren nähert man sich durch fortlaufendes Hinzufügen von Eigenschaften wieder dem realen Ding der wirklichen Welt. Ein konkreter Begriff beschreibt also eine kleinere Gruppe von realen Dingen. So ist die Menge der Gruppe Java-Programmierer kleiner als die Menge aller Programmierer, da sie weniger Individuen enthält.

Ein abstrakter Begriff ist also einfach, weil er nur wenige Eigenschaften umfasst, beschreibt aber viele konkrete Dinge. Ein konkreter Begriff ist im Vergleich schwieriger zu erfassen, weil er eine größere Menge von Eigenschaften umfasst, beschreibt aber weniger konkrete Dinge.

In der Softwareentwicklung ist die Fähigkeit zur Abstraktion schon allein deswegen notwendig, weil Software selbst nicht real greifbar ist (vgl. u.a. [Brooks 1987] oder [Ludewig 2010, S. 35]), man also nur mit Hilfe von Abstraktionen (z.B. Modellen) über Software reden kann. *"Abstraktionen sind ein zentrales Mittel für das Erstellen und Verstehen von Modellen"* [Glinz 2009]. Glinz [Glinz 2009] definiert Abstraktion wie folgt:

> *Abstraktion* [ist] *das Ableiten oder Herausheben des unter einem bestimmten Gesichtspunkt Wesentlichen/Charakteristischen/Gesetzmäßigen aus einer Menge von Individuen (Dingen, Beobachtungen, …).* [Glinz 2009]

Damit ist Abstraktion ein wichtiges Hilfsmittel, um die essentielle Komplexität der Softwareentwicklung (vgl. [Brooks 1987]) handhabbar zu machen. Zudem ist der Prozess der Entwicklung von Software die Umkehrung von Abstraktion, d.h. Konkretisierung.

> […] *the overall process of software development resembles more abstraction's inverse, concretization: An initial, unspecific model is refined upon by filling-in what precisely is required (analysis), how to solve the requirements in the abstract (design), and how to make a computer actually carry out that solution for us (implementation).* [Schuenemann 2005, S. 16]

Auch Pohl beschreibt Softwareentwicklung als die Konkretisierung von Anforderungen auf hohem Abstraktionsniveau zu implementierter Software auf niedrigem Abstraktionsiveau, wobei in jeder Phase des Entwicklungsprozesses der Lösungsraum sukzessive verringert wird [Pohl 2008, S. 20-22]. Kruchten sieht einen Unterschied in der Abstraktheit von Informationen zwischen Vision (Intent) und Software (Product). Damit ist auch für Kruchten eine wesentliche Aufgabe der Softwareentwicklung die Konkretisierung von Informationen [Kruchten 2007].

> *Intent precedes Product: Intent is an abstraction, a virtuality that sketches the reality that the project is set to achieve.* [Kruchten 2007]

Zusammenfassend haben Abstraktion und ihre Umkehrung, die Konkretisierung, zwei wichtige Funktionen bei der Entwicklung von Software [Schuenemann 2005, S. 16]:

1. Abstraktion ist ein wichtiges Hilfsmittel zur Beherrschung der Komplexität von Software.

2. Softwareentwicklung ist die Konkretisierung von sehr abstrakten Anforderungen über einen relativ abstrakten Entwurf zur konkreten Implementierung.

3 Grundbegriffe der Informationsflusstheorie

3.1 Überblick

Den Kern der Informationsflusstheorie bilden die in diesem Kapitel vorgestellten Definitionen zentraler Begriffe. Die Definitionen werden auf Basis der Grundlagen des vorigen Kapitels erstellt. Sie sind später der Ausgangspunkt zur Herleitung der Theoreme in Kapitel 4. Zudem bilden die Definitionen eine Fachsprache, mit der relevante Aspekte von Informationsflüssen in der Softwareentwicklung beschrieben und analysiert werden können.

Die wichtigsten und daher namensgebenden Begriffe der Theorie sind Information und Informationsfluss. Sie wurden als zentraler Bestandteil der Theorie gewählt, da mit ihrer Hilfe viele Softwareentwicklungsphänomene aus einem einheitlichen Blickwinkel beschrieben werden können. Information ist entweder als Wissen im Gedächtnis von Personen oder in Form von Daten in Dokumenten gespeichert. Durch Softwareentwicklungs-Aktivitäten fließen Informationen zwischen diesen so genannten Informationsspeichern. Kommunikation, Lernen und Dokumentation lassen sich als Informationsfluss charakterisieren. Somit können auch Kommunikations- und Dokumentationsprobleme mit der Informationsflusstheorie betrachtet werden, welche einen Großteil der heute noch aktuellen Probleme des Software Engineering ausmachen (vgl. Kapitel 1.1.3). Den Kern der Theorie bilden daher die Begriffe der folgenden Auflistung.

- Information und Informationsfluss
- Wissen und Gedächtnis
- Daten und Dokumente
- Kommunikation, Dokumentation und Lernen

Diese Begriffe werden in den folgenden Abschnitten näher betrachtet und für die Informationsflusstheorie definiert. Wenn es möglich und sinnvoll ist, werden dabei Erkenntnisse aus anderen Disziplinen (vgl. Kapitel 2) wie der Kognitionspsychologie oder den Kommunikationswissenschaften berücksichtigt und einbezogen. Abbildung 3.1 fasst die Grundbegriffe der Theorie zusammen und zeigt in einem Domänenmodell, wie sie miteinander in Verbindung stehen. Die Zahlen in der Abbildung stehen für die Kapitel, in denen der jeweilige Begriff näher betrachtet wird und schließlich für die Theorie definiert wird.

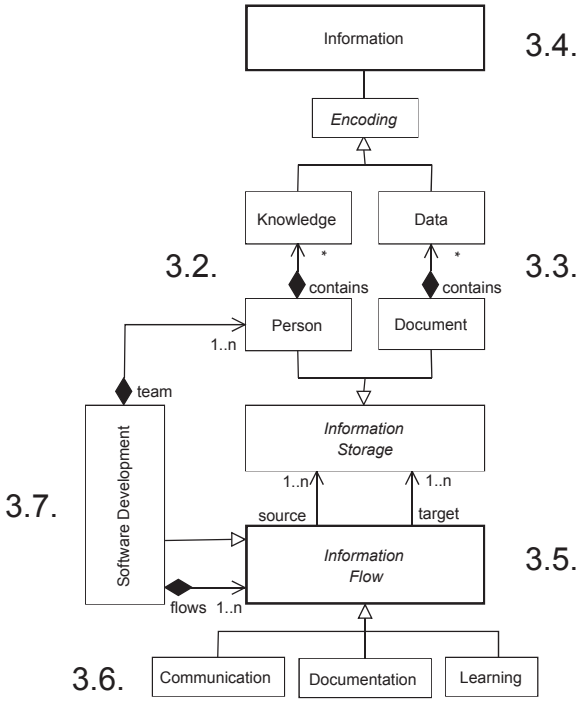


Abb. 3.1: Domänenmodell: Informationsflusstheorie

Das zentrale Element in Abbildung 3.1 (oben) ist Information. Jede Information hat eine Kodierung, d.h. sie liegt entweder in Form von Daten oder als Wissen vor. Mehrere Daten können in Dokumenten gespeichert werden. Wissen wird im Gedächtnis des Menschen gespeichert. Dokumente und das Ge-

dächtnis des Menschen sind somit Informationsspeicher. Informationen können zwischen Informationsspeichern fließen. Dieser Vorgang heißt Informationsfluss. Lernen, Dokumentation und Kommunikation sind spezielle Informationsflüsse. Softwareentwicklung ist ein Informationsfluss, der aus einem Team und weiteren Informationsflüssen besteht.

In den folgenden Abschnitten werden die Definitionen der Informationsflusstheorie hergeleitet.

3.2 Wissen und Gedächtnis

Wie in Kapitel 2.2.2 beschrieben hat der Mensch ein flüchtiges und ein permanentes Gedächtnis. Die Funktion des flüchtigen Gedächtnisses spielt für die weitere Betrachtung keine große Rolle und wird daher nicht in die Theorie aufgenommen. Das Wissen eines Menschen ist im permanenten Gedächtnis enthalten bzw. gespeichert.

> Definition 3.1: **Gedächtnis**
> Das Gedächtnis ist der Wissensspeicher des Menschen.

Der Mensch kann sein Wissen nutzen, um Probleme zu lösen. So kann ein Entwickler sein Wissen über Programmierung einsetzen, um eine Software zu erstellen. Die Software kann wiederum das Problem eines Nutzers lösen, wenn der Entwickler Wissen über die Anwendungsdomäne des Nutzers besessen und in die Software integriert hat.

Wissen und Problemlösen sind im Kontext der Softwareentwicklung eng miteinander verbunden. Die in Kapitel 2.2.1 vorgestellten Definitionen von Wissen führen den Wissensbegriff auf Begriffe wie Fähigkeiten, Kenntnisse, Erfahrung oder Expertise zurück. Diese Definitionen werden hier nicht übernommen, weil diese Begriffe teilweise selbst mit Hilfe von Wissen definiert werden. Dies würde zu einer Zirkeldefinition führen. Die Definition von Rao [Rao 2006], nach der Wissen die Repräsentation einer Problemlösung im menschlichen Verstand ist, kommt der Sicht, nach der Wissen etwas ist, was den Menschen dazu befähigt Probleme zu lösen, am nächsten und mindert zugleich die Gefahr einer Zirkeldefinition. Da "menschlicher Verstand" auch mehrdeutig interpretiert werden kann, wird er für die hier verwendete Definition durch das menschliche Nervensystem ersetzt. Wie das Nervensystem, im speziellen das Gehirn, des Menschen mit seiner Fähigkeit zu denken in Verbindung steht, soll hier nicht weiter betrachtet werden. Die Verbindung wird als gegeben angenommen. Es ergibt sich also folgende Definition für Wissen:

> Definition 3.2: **Wissen**
> Wissen ist der Zustand des Nervensystems, der den Menschen befähigt, Probleme zu lösen.

Es kann zwischen explizitem und implizitem Wissen unterschieden werden (vgl. Kapitel 2.2.1.1). Explizites Wissen kann bewusst zur Lösung eines Problems genutzt werden. Explizites Wissen kann bewusst abgerufen und daher ohne Weiteres kommuniziert werden. So kann zum Beispiel ein Java-Programmierer eine for-Schleife nutzen, um die Elemente eines Arrays zu durchlaufen (das Problem) und er könnte problemlos das Wissen über die Nutzung der for-Schleife einem anderen Programmierer erklären.

Definition 3.3: **Explizites Wissen**

Explizites Wissen ist Wissen, das *bewusst* zur Problemlösung eingesetzt werden kann.

Explizites Wissen kann nur durch sehr häufige Nutzung – zum Beispiel durch wiederholtes Üben – in implizites Wissen überführt werden (d.h. vom expliziten in das implizite Gedächtnis überführt werden, vgl. Kapitel 2.2.2.5). Implizites Wissen kann dann unbewusst zur Lösung von Problemen eingesetzt werden, aber nicht mehr ohne weiteres kommuniziert werden. So kann zum Beispiel ein geübter Zehnfingerschreiber unbewusst Wörter tippen, obwohl er die Position einzelner Buchstaben auf der Tastatur nicht auf Anhieb bestimmen und daher auch nicht kommunizieren könnte.

Definition 3.4: **Implizites Wissen**

Implizites Wissen ist Wissen, das *unbewusst* zur Problemlösung eingesetzt werden kann.

Um implizites Wissen kommunizieren zu können, muss es zunächst wieder in explizites Wissen transformiert werden. Dies kann entweder durch Erinnerung an das ursprünglich gelernte theoretische Wissen oder durch intensive Analyse der Ausführung einer Tätigkeit geschehen. Am Beispiel des Zehnfingerschreibers wäre dies entweder die Erinnerung an die Position der Tasten, wie sie beim Lernen des Zehnfingersystems vermittelt wurde, oder die Bestimmung der Tastenposition durch Beobachtung des eigentlichen Tippvorgangs. Im Gegensatz zur Sicht von Nonaka [Nonaka 1991] (vgl. Kapitel 2.2.1.1) ist der wesentliche Unterschied zwischen implizitem und explizitem Wissen ausschließlich der bewusste bzw. unbewusste Abruf. Formalität, Systematik und Artikulierbarkeit des Wissens spielen bei der Unterscheidung keine direkte

Rolle. Ein weiterer wesentlicher Unterschied ist, dass Wissen nach der obigen Definition *immer* an eine Person gebunden ist. Ein Dokument kann also explizites Wissen nicht direkt enthalten, sondern nur eine Repräsentation dieses Wissens (vgl. Kapitel 3.4). Nonakas und die hier festgelegte Unterscheidung zwischen implizitem und explizitem Wissen haben aber gemeinsam, dass explizites Wissen leichter artikuliert und damit auch leichter kommuniziert werden kann. Nach den hier vorgestellten Definitionen resultiert dies aber nur indirekt aus der Eigenschaft expliziten Wissens bewusst, also leichter, abgerufen werden zu können. Um die Gefahr der Verwechslung dieser beiden Sichtweisen zu minimieren, wird der Begriff "tacit" in der Informationsflusstheorie nicht benutzt.

Explizites Wissen kann weiter in propositionales und persönliches Wissen unterteilt werden. Diese Unterscheidung ergibt sich aus der Unterscheidung zwischen semantischem (propositional) und episodischem (persönlich) Gedächtnis (vgl. Kapitel 2.2.2.6). Da Wissen im episodischen Gedächtnis durch den persönlichen Bezug umfangreicher und schneller abgerufen werden und dieser Vorteil in der Softwareentwicklung gezielt genutzt werden kann, wird diese Unterscheidung in die Informationsflusstheorie übernommen. Propositionales Wissen sind alle Fakten, Konzepte, Regeln und Abläufe, deren Kenntnisse zur Lösung von Problemen notwendig sind. Propositionales Wissen ist im semantischen Gedächtnis gespeichert.

Definition 3.5: **Propositionales Wissen**

Propositionales Wissen ist explizites Wissen, das aus Fakten, Konzepten, Regeln und Prozeduren besteht, die zur Lösung von Problemen eingesetzt werden können.

Persönliches Wissen sind Kenntnisse, die durch eigene Erfahrungen erlernt wurden. Persönliches Wissen verknüpft propositionales Wissen mit einer persönlichen Komponente. Die persönliche Komponente führt zu mehr und stärkeren Zugriffspfaden, sodass persönliches Wissen typischerweise längerfristig und schneller abgerufen werden kann (vgl. Kapitel 2.2.2.6). Persönliches Wissen ist im episodischen Gedächtnis gespeichert.

Definition 3.6: **Persönliches Wissen**

Persönliches Wissen ist explizites Wissen, das durch persönliche Erfahrung entstanden ist.

In Abbildung 3.2 sind die gerade beschriebenen Zusammenhänge noch einmal zusammenfassend dargestellt. Dabei werden die unterschiedlichen Gedächtnisteile nicht direkt modelliert. Der Name des Gedächtnisteils, den man referenzieren möchte, ergibt sich immer aus der Wissensart, die darin gespeichert ist. Propositionales und persönliches Wissen wurde auf derselben Vererbungsebene modelliert, obwohl das episodische Gedächtnis eigentlich eine Spezialisierung des semantischen Gedächtnisses ist (damit ist auch persönliches eine Spezialisierung propositionalen Wissens), da das episodische Gedächtnis zusätzlich zu den zeit- und selbstbezogenen auch alle semantischen Eigenschaften besitzt [Tulving 2009]. Hier wird das semantische Gedächtnis (propositionales Wissen) aber als nicht-episodisches Gedächtnis modelliert (gleiche Vererbungsebene), um die in der Softwareentwicklung häufiger auftretenden semantischen Gedächtnisvorgänge klarer von den seltener auftretenden aber dennoch wichtigen (Stichwort Erfahrungswissen und Best Practices) episodischen Erlebnissen trennen zu können. Tabelle 3.1 zeigt einen Überblick über Wissensarten, zugehörige Gedächtnistypen und Beispiele für eine darin gespeicherte Wissensart aus der Software-Engineering-Domäne.

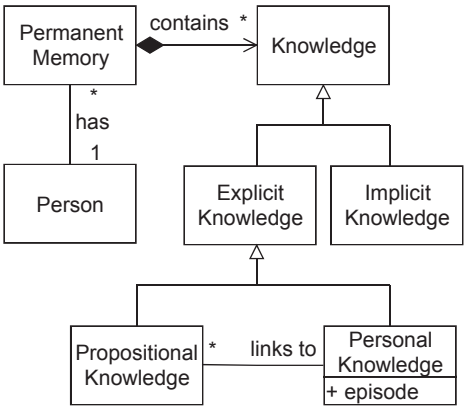


Abb. 3.2: Domänenmodell: Gedächtnis und Wissen

Wenn ein Entwickler durch jahrelange Praxis Test-First-Programmierung so verinnerlicht hat, dass er die einzelnen Schritte und deren Reihenfolge unbewusst ausführen kann, dann kann er dieses implizite Wissen für seine Arbeit nutzen, aber evtl. einem neuen Kollegen nicht mehr auf Anhieb genau erklä-

Tabelle 3.1: Wissensarten mit zugehörigen Gedächtnisnamen und Beispielen aus der Software-Engineering-Domäne

Wissensart	**Gedächtnistyp**	**SE-Beispiel**
Implizites Wissen	Implizites Gedächtnis	Eine *verinnerlichte* Best Practice, z.B. Test-First-Programmierung nach jahrelangem praktischen Einsatz
Explizites Wissen	Explizites Gedächtnis	Eine *gelernte* Best Practice, z.B. Test-First-Programmierung aus Lehrbuch gelernt
Propositionales Wissen	Semantisches Gedächtnis	Fakt: Java ist eine Programmiersprache Konzept: Qualitätsanforderungen sind nichtfunktionale Anforderungen Regel: Wenn die Spezifikation mangelhaft ist, dann wird die Wartung schwierig Prozedur: Erst Anforderungen erheben, dann entwerfen, dann implementieren
Persönliches Wissen	Episodisches Gedächtnis	Selbst erfahren haben, dass es Vorteile hat, zunächst einen Testfall zu spezifizieren und erst danach die eigentliche Funktionalität zu implementieren

ren. Andererseits kann dieser neue Kollege sich das Wissen über Test-First-Programmierung zwar explizit anlesen, ist danach aber nicht so effizient in der Umsetzung, da ihm die praktische Übung fehlt. Beispiele für propositionales explizites Wissen sind Fakten, Konzepte, Regeln und Prozeduren, wie in Tabelle 3.1 gezeigt.

3.3 Daten und Dokumente

Daten Daten sind wichtiger Bestandteil der Softwareentwicklung. Nach IEEE-Definition von Software (siehe Def. 1.2.2) sind sie Teil der Software, d.h. Teil des Endprodukts der Softwareentwicklung. Während der Softwareentwicklung spielen sie auch eine große Rolle. Es gibt Testdaten. Ergebnisse von Prozess- und Produktmetriken liegen als Daten vor. Im weitesten Sinne kann man auch den Quellcode einer Software als Daten ansehen. Es ist also sinnvoll den Begriff Daten in die Informationsflusstheorie aufzunehmen und seine Bedeutung festzulegen. Nach Floridi [Floridi 2006] (vgl. Kapitel 2.3.1) ist ein Datum ein Unterschied zweier Dinge der realen Welt. Diese abstrakte Sicht wird hier übernommen, da damit ein Großteil der für die Softwareentwicklung relevanten Daten abgedeckt wird.

Definition 3.7: **Daten**
Daten sind unterscheidbare physische Zustände.

Diese Definition schließt sowohl vom Menschen geschaffenes, wie Sprache, sowie nicht vom Menschen geschaffenes, wie physikalische Größen ein. Zustände müssen nur unterscheidbar sein. D.h. der Unterschied muss entweder direkt vom Menschen wahrnehmbar sein (Bsp. unterschiedliche Frequenzen von Schallwellen bei Sprache) oder anders ermittelt werden können, z.B. durch Messen (Bsp. unterschiedliche elektrische Spannungen bei der Signalverarbeitung).

In der Softwareentwicklung wichtige Daten sind Sprache (natürliche Sprache: Schallwellen oder geschriebener Text, künstliche Sprache: Quellcode), Bilder, Abbildungen (z.B. Modellrepräsentationen), Formeln etc. sowie deren beliebige Kombination untereinander. Um die vielen verschiedenen Daten im weiteren Verlauf der Theorie besser unterscheiden zu können (z.B. für die Herleitung von Theoremen für die Medienwahl in Kapitel 4.3), wird eine Klassifizierung für Daten eingeführt. Die Klassifizierung ordnet Daten Datentypen zu.

Definition 3.8: **Datentyp**
Ein Datentyp D ist eine Menge von Daten $d \in D$. Alle Daten eines Datentyps

- haben die gleiche Dimension,
- können mit dem gleichen Sinn eines Menschen wahrgenommen werden und
- sind dem Ursprung nach entweder *künstlich* oder *natürlich*.

Eine *Dimension* ist eine zusätzliche physikalische Größe oder Eigenschaft, die zur Unterscheidung von Daten genutzt werden und damit Daten weitere Bedeutung geben kann (zur Bedeutung von Daten siehe Kapitel 2.4). Nach Definition 3.7 haben Daten immer mindestens eine solche unterscheidbare Eigenschaft. Da die hier relevanten Daten meist eine Kombination aus mindestens zwei solcher Eigenschaften sind (z.B. Zeichen plus räumliche Anordnung bei geschriebener Sprache oder Silben plus zeitliche Anordnung bei gesprochener Sprache), entspricht eine unterscheidbare Eigenschaft der 0-ten Dimension, zwei Eigenschaften der 1-ten Dimension, drei Eigenschaften der 2-ten Dimension usw. Häufig auftretende vom Menschen wahrnehmbare Unterscheidungsmerkmale sind:

- Zeitliche Anordnung, z.B. von Silben bei gesprochener natürlicher Sprache oder von Bildern bei Videos
- Räumliche Anordnung (1D, 2D, 3D), z.B. von Buchstaben bei Text (1D), Linien und Text bei Diagrammen (2D) oder realen Objekten (3D, Bsp. Modellauto).

Tabelle 3.2 zeigt einige Beispiele verschiedendimensionaler Daten.

Das zweite Klassifizierungsmerkmal eines Datentyps ist der *Sinn*, mit dem das Datum vom Menschen wahrgenommen werden kann. In der Softwareentwicklung spielen im Wesentlichen nur die beiden Sinne der auditiven und visuellen Wahrnehmung eine Rolle. Alle weiteren Sinne werden daher hier nicht weiter betrachtet.

Beim Ursprung der Daten wird zwischen künstlichen und natürlichen Daten unterschieden, wobei zu natürlichen Daten auch natürliche Sprachen, wie die

Tabelle 3.2: Beispiele verschiedendimensionaler Daten

	Beispiel	Dimensionen
1D	Geschriebener Text	1-dimensional räumlich angeordnete Zeichen
	Gesprochener Text	zeitliche Anordnung von Schallfrequenzänderungen
2D	Diagramme, Abbildungen, Formeln	2-dimensional räumlich angeordnete Zeichen
3D	Animierte 2D Daten (z.B. Video)	zeitliche Anordnung von 2-dimensionalen Daten (z.B. zeitliche Anordnung von Bildern)

deutsche oder englische Sprache, zählen, obwohl sie vom Menschen geschaffen wurden. Einerseits ist dies sinnvoll, damit die Bedeutung des natürlichen Sprachbegriffs nicht im Gegensatz zu dessen umgangssprachlichen Nutzung steht (vgl. Adäquatheitsforderung von Explikationen in Kapitel 2.1.1). Andererseits ist es bei der Herleitung der Theoreme wichtig die Grenze später zu ziehen, weil künstliche Sprachen meist formaler sind und daher anders, meist einfacher, bearbeitet werden können. Künstliche Sprachen sind Sprachen, die vom Menschen für einen bestimmten Zweck (außer der alltäglichen Kommunikation) geschaffen wurden. Meist sind diese formaler als natürliche Sprachen. Beispiele sind mathematische Formalismen oder Programmiersprachen.

Die folgenden sechs Datentypen decken einen Großteil der in der Softwareentwicklung anzutreffenden Datentypen ab:

1D auditiv natürlich: Gesprochene natürliche Sprache, wie Deutsch oder Englisch. Dieser Datentyp kommt u.a. in jedem Softwareprojekt vor, bei dem Entwickler miteinander reden.

1D visuell natürlich: Geschriebene natürliche Sprache, wie deutsche oder englische Texte. Dieser Datentyp spielt bei der Dokumentation (Prozess- oder Produktdokumentation) und bei der Kommunikation (z.B. E-Mails) in der Softwareentwicklung eine wichtige Rolle.

1D visuell künstlich: Geschriebene künstliche Sprache, wie Quellcode. Alle visuell wahrnehmbaren Sprachen, die einer nicht-natürlichsprachigen Syntax und/oder Grammatik folgen. Alle textuellen Programmiersprachen und textuelle mathematische Formalismen gehören zu diesem Datentyp.

2D visuell natürlich: Natürliches Bild, wie Fotos. Hierzu zählen insbesondere auch Bilder von Menschen In der Softwareentwicklung können z.B. Fotos von entfernten Kollegen oder von der Umgebung, in der ein neues System eingesetzt werden soll, hilfreich sein.

2D visuell künstlich: Künstliches Bild, wie Diagramme oder spezielle grafische Modelle. Hierzu zählen alle künstlich geschaffenen grafischen Notationen (2D, visuell). Für die Softwareentwicklung wichtige grafische Notationen sind u.a. die Diagrammtypen der UML oder ER-Diagramme.

3D visuell natürlich: Bewegte natürliche Bilder, wie von Angesicht zu Angesicht oder Videos. Die zusätzliche zeitliche Dimension kann den 2D visuellen natürlichen Daten weitere Bedeutung hinzufügen. Mimik, Gestik, Kopfschütteln, Schulterzucken, etc. können so übertragen werden. Videokonferenzen und Gespräche von Angesicht zu Angesicht sind Beispiele für die Nutzung des 3D visuellen natürlichen Datentyps, die häufig in der Softwareentwicklung anzutreffen sind.

Weiterhin wird der Begriff der Datenmenge benötigt, um später Begriffe wie Bandbreite definieren (vgl. Kapitel 3.5.3) und in Kapitel 4 Theoreme herleiten zu können. Datenmenge wird wie folgt definiert:

Definition 3.9: **Datenmenge**
Sei $||:(D,+)\rightarrow(\mathbb{R}^+,+)$ ein Homomorphismus[1]für einen Datentyp D, der es ermöglicht, beliebige Daten $d \in D$ zu messen. Man bezeichnet $|d|$ als Datenmenge von d.

Zum Beispiel kann die Datenmenge eines natürlichsprachigen Textes über die Anzahl der Bytes bestimmt werden, die die Repräsentation des Textes in ASCII-Kodierung einnimmt.

[1] Es seien $(G,\cdot)$ und $(H,*)$ zwei Mengen mit Verknüpfung. Eine Abbildung $\psi: G \rightarrow H$ heißt Homomorphismus, falls für alle $x,y \in G$ gilt $\psi(x\cdot y)=\psi(x)*\psi(y)$. Geschrieben $\psi:(G,\cdot)\rightarrow(H,*)$.

Dokumente In der Softwareentwicklung gibt es eine Vielzahl von Daten, die es aus unterschiedlichen Gründen zu speichern und weiterzuleiten gilt (vgl. Kapitel 2.3.2 oder [Ludewig 2010, S. 259 ff]). So ist es z.B. hilfreich, wenn Diagramme über den Entwurf der Software in der Wartung noch verfügbar sind, oder wenn die Daten, die für die Ausführung der Software notwendig sind, mit der Software an den Nutzer ausgeliefert werden. Daten können in Dokumenten dauerhaft gespeichert werden. Dokumente können dann als Mittel zur Kommunikation (vgl. Kapitel 3.6 später) genutzt werden. Neben der dauerhaften Speicherung ist die Struktur der Daten für ein sinnvolles Dokument wichtig (vgl. Kapitel 2.3.2). Eine gute Struktur erleichtert oder ermöglicht erst die korrekte Interpretation der in einem Dokument enthaltenen Daten. So ist z.B. eine Anforderungsspezifikation, die die Anforderungen kategorisiert, priorisiert und miteinander in Verbindung setzt hilfreicher für die Softwareentwicklung, als eine Spezifikation, die die Anforderungen als eine unsortierte lose Aneinanderreihung von unabhängigen Einzelanforderungen enthält.

Es wird die bereits in [Stapel 2006, S. 27] eingeführte Definition von Dokument übernommen.

Definition 3.10: **Dokument**

Ein Dokument ist ein Datenspeicher, der Daten beliebiger Typen strukturiert zusammenfasst.

Die in Kapitel 2.3.2 genannten wesentlichen Eigenschaften von Dokumenten werden durch diese Definition abgedeckt:

- Informationen werden in Form von Daten gespeichert.
- Informationen sind über einen längeren Zeitraum zugreifbar, weil sie im Dokument *physisch* in Form von Daten repräsentiert sind.
- Informationen können in Dokumenten transportiert werden, weil sie in physischer Form an einen anderen Ort gebracht werden können.
- Die Daten sind organisiert bzw. strukturiert.

In Tabelle 3.3 sind einige Beispiele von in der Softwareentwicklung häufig genutzten Dokumenten und die darin typischerweise gespeicherten Datentypen gegeben. Dabei wird nicht gezielt zwischen materiellen und elektronischen

Tabelle 3.3: Beispiele von Dokumenten im SE

Dokument	**Typische enthaltene Datentypen**
Buch, Zeitung, Artikel, Spezifikation, Protokoll, Präsentationen, Internetseiten, E-Mails etc.	Geschriebene natürliche Sprache, teilweise auch künstliche oder natürliche 2-dimensionale Abbildungen.
Audioaufzeichnungen	gesprochene natürliche Sprache
Videoaufzeichnungen	bewegtes natürliches Bild und gesprochene natürliche Sprache
Quellcode, Kompilat, Logdateien	künstliche Sprache
Grafische Modelle	künstliche Abbildungen / Diagramme

Formen unterschieden, da ein Dokumententyp meist in beiden Varianten vorliegen kann (z.B. ein Buch). Der Unterschied ist lediglich, dass bei elektronischen Dokumenten die Daten in einem anderen als für die Repräsentation vorgesehenen Format für die physische Speicherung umkodiert werden (z.B. die Buchstaben des Alphabets als Bitfolgen im Computer) Dies spielt aber für die weitere Betrachtung von Dokumenten in der Informationsflusstheorie keine Rolle.

3.4 Information

Wie bereits in der Einleitung erläutert, ist Information die zentrale erfolgskritische Ressource der Softwareentwicklung. Die professionelle Softwareentwicklung ist heute von zwei gegensätzlichen Entwicklungsmethoden geprägt. Einerseits liegt in klassischen Prozessen der Fokus auf Dokumentation. Andererseits konzentrieren sich agile Methoden auf die intensive Kommunikation der Entwickler und Stakeholder. Dabei lassen sie sich von allgemeinen Prinzipien und spezielle Praktiken leiten. Es sind also sowohl *Daten*, die es im Laufe des Projekts bis zur finalen Software über Dokumente immer weiter zu verfeinern gilt, als auch *Wissen*, das zu den relevanten Personen kommuniziert werden muss, um schließlich im Dokument Software manifestiert zu werden, zentrale Betrachtungsgegenstände, d.h. zentrale Ressource, der Softwareentwicklung.

Um diese beiden wichtigen Ressourcen, bei denen je nach Vorgehensweise der Fokus stärker auf der einen oder anderen liegt (plangetrieben vs. agil), aus einem einheitlichen Blickwinkel betrachten zu können, bietet sich der Begriff Information als zentrales relationales Element an, da Information sowohl als Daten plus Bedeutung (vgl. Kapitel 2.4.4) als auch als potenzielles, nicht personengebundenes Wissen (vgl. Kapitel 2.4.5) aufgefasst werden kann.

Abbildung 3.3 zeigt die Zusammenhänge zwischen Wissen, Information und Daten. Das Modell wurde aus [Stapel 2006, S. 22] übernommen und leicht angepasst. Wissen ist im Gedächtnis des Menschen gespeichert und an eine Person gebunden. Aus Sicht des Menschen ist Wissen etwas Internes. Daten, d.h. physisch unterscheidbare Zustände, sind aus Sicht des Menschen extern. Die Schnittstelle für den Übergang von internem Wissen zu externen Daten (auch Externalisierung) bzw. den Übergang von externen Daten zu internem Wissen (auch Internalisierung) bildet Information. Externalisierung und Internalisierung werden hier als Begriffe für die Übergänge zwischen Wissen und Daten genutzt und sind nicht mit der von Nonaka geprägten Nutzung als Übergänge zwischen explizitem und implizitem Wissen (vgl. Kapitel 2.2.1.1) zu verwechseln. Externalisierung und Internalisierung bestehen je aus zwei Schritten mit dem Zwischenergebnis Information. Im Folgenden werden die vier Schritte, die an den Übergängen zwischen Wissen, Information und Daten beteiligt sind, näher erläutert.

① **Nachricht und Kontext trennen:** Beim Übergang von Wissen zu Information werden Nachricht und Kontext getrennt. Die Nachricht kann

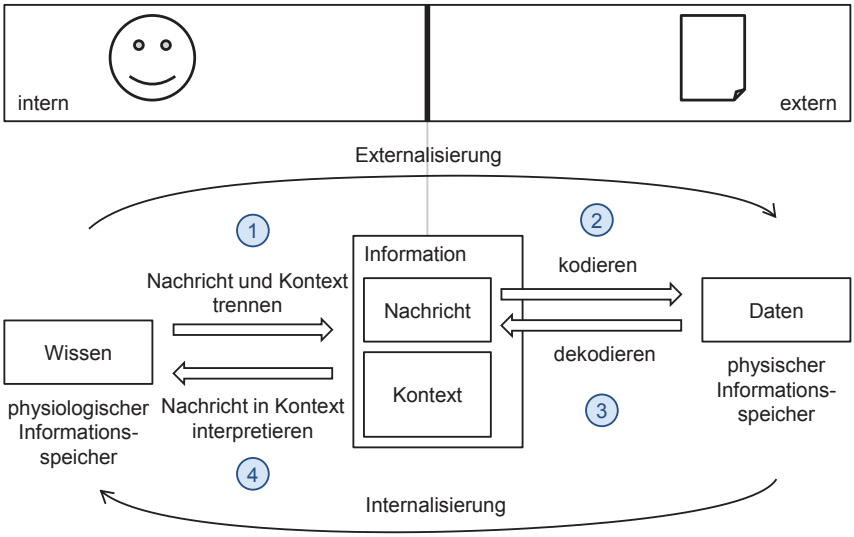


Abb. 3.3: Information in Relation zu Wissen und Daten (angepasst aus [Stapel 2006, S. 22])

zu einer anderen Person übertragen und dort mit Hilfe des Kontextes wieder in Wissen überführt werden. Der Kontext wird aber nicht übertragen. Die Nachricht kann also nur in das Wissen der Zielperson aufgenommen werden, wenn der Kontext zur korrekten Interpretation dort bereits vorhanden ist (vgl. Kapitel 2.4.3). Beim Trennen von Kontext und Nachricht muss die Person also überlegen, welcher Teil seines Wissens als Nachricht vorgesehen und welcher Teil als Vorwissen vorausgesetzt werden soll. Zum Beispiel müßte ein Softwareentwickler, je nachdem ob sein Kommunikationspartner ein anderer Softwareentwickler oder ein fachfremder Kunde ist, überlegen, wie er sein Wissen darüber, dass das zu lösende Problem ein NP-vollständiges Problem ist, in Nachricht und Kontext aufteilen soll. Bei seinem Softwareentwicklerkollegen kann er mehr Wissen über NP-Vollständigkeit voraussetzen und dementsprechend eine kleinere Nachricht wählen. Der Kunde hingegen wird eine umfangreichere Nachricht benötigen.

② **Kodieren:** Beim Übergang von Information zu Daten wird nur der Nachrichtenteil der Information in Daten überführt. Die Daten repräsentieren dann die Information in der physischen Welt. Information, als Daten repräsentiert, kann transportiert, langfristig aufbewahrt, vervielfältigt und anderweitig verändert werden. Die Nachricht für den Softwareentwicklerkollegen aus dem obigen Beispiel könnte als Daten kodiert wie folgt aussehen: *"Wir können für unser Problem höchstens eine approximative Lösung finden, da es NP-vollständig ist"*.

③ **Dekodieren:** Der Übergang von Daten zu Information geschieht durch die Zusammenführung der Nachricht mit dem Kontext. D.h. den Daten, z.B. Schallwellen oder Signalen, wird eine Bedeutung zugewiesen. Diese Zuweisung ist kein realer Akt, sondern nur der virtuelle, gedachte Unterschied zwischen Daten und Information. Ein Kontext kann Daten eine Bedeutung geben. Im obigen Beispiel würde der Softwareentwickler die empfangene Nachricht lesen, aber noch nicht interpretieren. Die Daten haben im Kontext eine Bedeutung. Damit sind sie wieder Information.

④ **Nachricht in Kontext interpretieren:** Der Übergang von Information zu Wissen geschieht durch Interpretation der Nachricht im zugehörigen Kontext. Dies ist nur möglich, wenn der zur Interpretation notwendige Kontext bei der interpretierenden Person als Wissen bereits vorhanden ist. Das zur Interpretation notwendige Wissen heißt Kontextwissen. Kontextwissen umfasst spezielles Wissen, z.B. den aktuellen Gesprächsverlauf (vgl. [Clark 1993]) und allgemeines Wissen, z.B. Fachwissen der Informatik, Wissen der deutschen Sprache oder Wissen über die westeuropäische Kultur (vgl. Kapitel 2.4.3). Wenn der Softwareentwickler aus dem obigen Beispiel die Nachricht seines Kollegen empfangen, gelesen und – das nötige Fachwissen vorausgesetzt – verstanden hat, wurde die Information, dass es im besten Fall eine approximative Lösung für das gegebene Problem gibt, in sein Wissen aufgenommen. Mit diesem Wissen kann er nun beginnen, eine approximative Lösung zu suchen, anstatt vergebens nach einer perfekten und gleichzeitig effizienten Lösung zu suchen.

Damit ist Information, wie bereits in [Stapel 2006, S. 25] beschrieben, ein relationaler Begriff, der die komplexen Beziehungen zwischen Daten und Wissen abstrahiert. Aus Sicht von Daten und Wissen kann Information wie folgt charakterisiert werden:

Datensicht: Information ist Daten plus *Bedeutung*. Die Bedeutung ergibt sich durch Interpretation der Nachricht in einem Kontext.

Wissenssicht: Information ist *potenzielles, nicht personengebundenes* Wissen. Durch Interpretation der Nachricht in einem Kontext kann Information wieder an eine Person gebunden werden, d.h. in das Wissen dieser Person aufgenommen werden.

Im Gegensatz zu Daten und Wissen ist Information ein gedankliches Konstrukt, das kein direktes Gegenstück in der realen Welt hat. Wissen ist im Gedächtnis des Menschen gespeichert. Daten sind unterscheidbare physische Zustände. Information hingegen kann beides sein, grenzt sich aber durch zusätzliche virtuelle Eigenschaften von Wissen (nicht personengebunden) und Daten (Bedeutung) ab. Dies macht es eventuell schwierig, sich vorzustellen, was Information ist. Dennoch wird Information hier als zentraler Begriff der Theorie über die Entwicklung von Software eingeführt, da er es ermöglicht, die beiden für die Softwareentwicklung wichtigen Konstrukte Daten und Wissen einheitlich zu betrachten. Zudem ist das umgangssprachliche Verständnis über die Bedeutung von Information ähnlich zu der hier vorgestellten Definition (vgl. Adäquatheitsforderung von Explikationen in Kapitel 2.1.1). Dies ermöglicht es auch mit Personen über Softwareentwicklung aus Informationssicht zu sprechen, die die Informationsflusstheorie nicht kennen.

Die vorangegangenen Betrachtungen zu Information werden in den folgenden Definitionen zusammengefasst:

Definition 3.11: **Information**

Eine Information i besteht aus einer Nachricht m und zugehörigem Kontext c. Man schreibt:

$$i = (m, c)$$

Definition 3.12: **Nachricht**

Die Nachricht m ist der Teil der Information i, der

- im Falle des physischen Informationsspeichers Daten als Daten repräsentiert, oder
- im Falle des physiologischen Informationsspeichers Wissen für die Repräsentation als Daten vorgesehen ist.

Definition 3.13: **Kontext**

Kontext c ist Information, die einer Nachricht m Bedeutung gibt.

Axiom 3.1: **Grundinformation**

Sei $\pi(m)$ die Grundinformation der Nachricht m. Dann ist die Information i die Grundinformation der Nachricht $\pi(m)$ plus Kontext c. Man schreibt:

$$i = \pi(m) + c$$

Die Grundinformation ist die Bedeutung, die ausschließlich in der Nachricht steckt. Zum Beispiel ist die Grundinformation der Nachricht "Java ist eine Programmiersprache" lediglich die Zuordnung zwischen den beiden Konzepten Java und Programmiersprache. Damit diese Nachricht richtig interpretiert werden kann ist weiterhin Kontext notwendig, also z.B. das Wissen darüber, was eine Programmiersprache ist. Für die Definition von Kommunikationerfolg (vgl. Def. 3.38) wird noch ein Axiom benötigt, welches den Zusammenhang zwischen Informationsgehalt und Datenmenge einer Nachricht beschreibt.

Axiom 3.2: **Informationsgehalt**

Sei $|| : (\mathbb{I}, \cup) \rightarrow (\mathbb{R}^+, +)$ ein Homomorphismus für die Menge aller Informationen $\mathbb{I}$, der es ermöglicht, beliebige Informationen $i \in \mathbb{I}$ zu messen. Man bezeichnet $|i|$ als Informationsgehalt von i. Im Falle des physischen Informationsspeichers Daten korreliert der Informationsgehalt der Nachricht $|\pi(m)|$ positiv mit der Datenmenge $|m|$. Man schreibt:

$$|\pi(m)| \overset{+}{\approx} |m|^2$$

Abbildung 3.4 veranschaulicht die sich aus den Definitionen ergebenden Zusammenhänge zwischen Information, Nachricht und Kontext. Kontext gibt der Nachricht eine Bedeutung. D.h. umgekehrt, dass die Bedeutung der Nachricht abhängig vom Kontext ist.

[2]Positive Korrelation: $x \overset{+}{\approx} f(x) \Leftrightarrow (x_1 \leq x_2 \Rightarrow f(x_1) \leq f(x_2))$

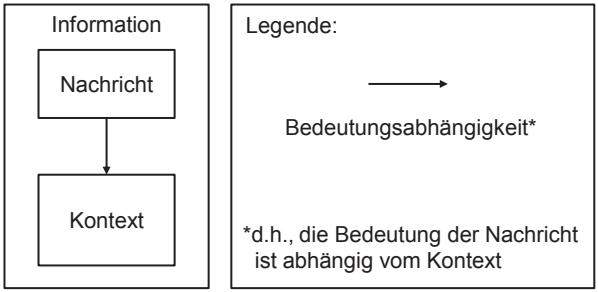


Abb. 3.4: Information, Nachricht und Kontext

Da Kontext selbst Information ist, besteht dieser auch wieder aus einer Nachricht und weiterem Kontext. Abbildung 3.5 zeigt ein Beispiel aus der Softwareentwicklung, welches diesen rekursiven Zusammenhang verdeutlicht. Die rekursive Definition von Kontext als Information ermöglicht es, dass Kontext die Bedeutung einer Nachricht nicht nur über Kontextwissen einer Person zuweisen kann, sondern dass bis zu einem gewissen Grad Kontext auch als Daten repräsentiert und damit dokumentiert und kommuniziert werden kann. Kontextdaten sind zum Beispiel schriftliche Erläuterungen zu einer Nachricht (vgl. Kontext in Abb. 3.5), Glossareinträge oder Lexika zu einer Fachsprache. Kontextwissen hingegen ist das Vorwissen einer Person, welches ihr ermöglicht eine Nachricht zu verstehen, z.B wenn die Person eine bestimmte Fachsprache bereits beherrscht. Die Rekursion von Kontext als Information *endet* immer mit Kontextwissen. Ab einem gewissen Punkt ist es nicht mehr praktisch sinnvoll (aus Sicht des Software Engineering) Kontext in Form von Daten zu formulieren. Zum Beispiel ist es in den meisten für die Softwareentwicklung relevanten Fällen nicht sinnvoll den Kontext der Bedeutung von Wörtern einer natürlichen Sprache wie Deutsch oder Englisch als Glossar in einer Spezifikation auszuformulieren. Das Wissen über die natürliche Sprache, die zum Verständnis der Spezifikation notwendig ist, wird als gegeben vorausgesetzt. Damit endet die Rekursion.

Die Nachricht einer Information kann je nach vorausgesetztem Kontext unterschiedlich groß sein. So kann z.B. die Information, dass in einem Softwareprojekt die Programmiersprache Java und die unterstützenden Werkzeuge Eclipse, Subversion, Trac und Ant eingesetzt werden sollen, entweder durch die Nachricht *"In diesem Projekt setzen wir die Programmiersprache Java und zur*

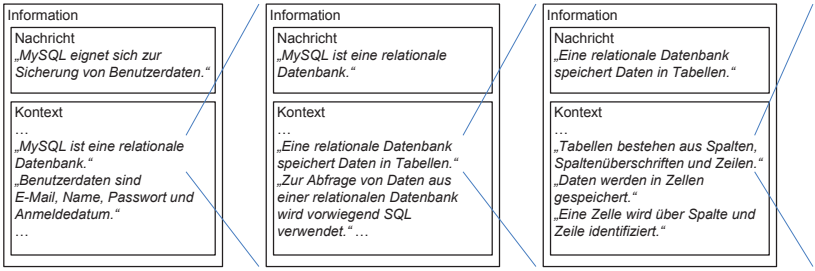


Abb. 3.5: Beispiel des rekursiven Charakters von Kontext

Unterstützung die Werkzeuge Eclipse, Subversion, Trac und Ant ein." oder durch die kürzere Nachricht *"In diesem Projekt benutzen wir die gleichen Werkzeuge wie immer."* kommuniziert werden. Wobei die zweite Nachricht einen größeren Kontext, d.h. mehr Vorwissen oder mehr zusätzliche Daten, voraussetzt. Abbildung 3.6 verdeutlicht diesen Zusammenhang.

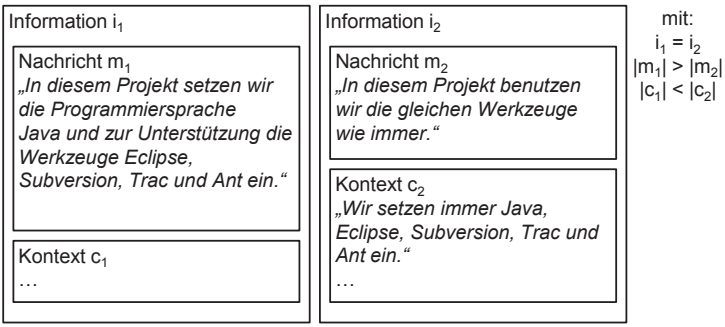


Abb. 3.6: Vergleich von Information mit unterschiedlich großen Nachrichten

3.4.1 Informationskomplexität

Durch den rekursiven Charakter von Informationen können vielfältige Abhängigkeiten der Bedeutungszuweisung existieren. Komplexe Informationen haben viele solche Bedeutungsabhängigkeiten. Das Verständnis einer Nachricht einer komplexen Information bedarf viele voneinander abhängige Kon-

textinformationen. Einfache Informationen hingegen haben wenige voneinander abhängige Kontext-Informationen. Z.B. kann der Kontext für das Verständnis einer Anforderung an die Steuerungssoftware eines Space-Shuttles viel komplexer sein, als für eine Anforderung an die Steuerungssoftware eines Taschenrechners. Folgende Definition fasst das zusammen.

Definition 3.14: **Informationskomplexität**

Informationskomplexität ist die Anzahl der Bedeutungsabhängigkeiten zwischen den Teilinformationen des zum Verständnis der Nachricht notwendigen Kontextes einer Information.

Abbildung 3.7 veranschaulicht den Unterschied zwischen komplexen und einfachen Informationen. Es werden zwei unterschiedlich komplexe Informationsmengen gegenüber gestellt. Dabei ist zu beachten, dass ein Vergleich der Komplexität zweier Informationen nur dann sinnvoll ist, wenn sich die Bedeutung der verschiedenen Nachrichten irgendwann in der Rekursion auf den gleichen Kontext zurückführen lässt.

Weiterhin ist zu beachten, dass die hier definierte Informationskomplexität nicht vergleichbar mit der Informationsmenge der Informationstheorie nach Shannon [Shannon 1948] ist. Hier ist Informationskomplexität die Anzahl der Abhängigkeiten zwischen Informationen und nicht die Menge der Informationen. Es kann auch eine große Menge an Informationen geben (d.h. großer Informationsgehalt), die nicht komplex sind. Dieser Unterschied spielt für die Softwareentwicklung eine wichtige Rolle, da komplexe Probleme nur schlecht parallel und unabhängig voneinander bearbeitet werden können, was zu einem erhöhten Abstimmungsaufwand führt. Ein großes, aber wenig komplexes Problem hingegen, kann gut von vielen Entwicklern gleichzeitig und unabhängig voneinander bearbeitet werden (vgl. Kapitel 3.7.1 und 4.4). Ein besser passenderer Vergleich ist der mit der Komplexitätstheorie der Theoretischen Informatik. Dort ist Komplexität der Ressourcenverbrauch (Zeit, Speicher) bei der Berechnung von Algorithmen. Entsprechend den Komplexitätsklassen der Theoretischen Informatik, die Kategorien für unterschiedlich "schwierige" algorithmische Probleme bilden, sind unterschiedlich komplexe Softwareentwicklungsprobleme vorstellbar. Z.B. Probleme, deren Lösung von vielen Entwicklern ohne viel Abstimmungsaufwand parallel bearbeitet werden kann und Probleme, die nur durch geschickte Abstraktion der Lösung eine effiziente Arbeitsteilung erlauben.

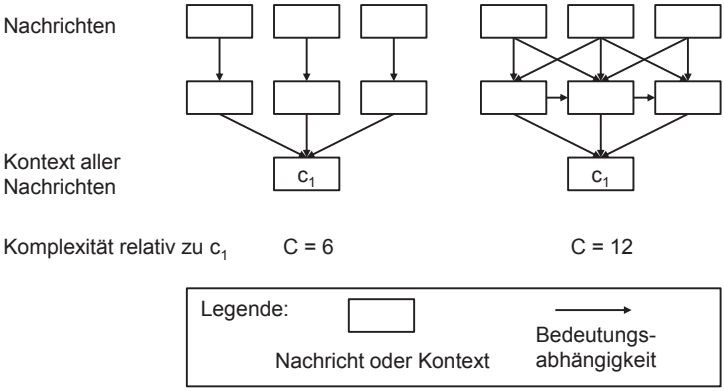


Abb. 3.7: Veranschaulichung von Informationskomplexität

3.4.2 Informationsdomäne

Während der Softwareentwicklung müssen unterschiedliche Probleme gelöst werden. Die Anforderungen des Kunden müssen erhoben werden. Ein zu den Anforderungen passender Entwurf muss gefunden werden. Der Entwurf muss implementiert und getestet werden. Die Entwickler müssen sich untereinander abstimmen. Der Fortschritt des Projekts muss überwacht werden. Indem eine Software für den Kunden entwickelt wurde, wird letztendlich auch ein Problem des Kunden gelöst. Damit in der Informationsflusstheorie Informationen leichter den unterschiedlichen Arten dieser Probleme zugeordnet werden können, wird der Begriff der Informationsdomäne (kurz Domäne) eingeführt. Eine Domäne fasst alle Informationen zusammen, die für eine bestimmte Art von Problemen relevant sind. In der Softwareentwicklung spielen die folgenden Domänen eine wichtige Rolle.

Software-Domäne: Die Software-Domäne umfasst alle Informationen, die direkt für die Softwareentwicklung relevant und auch nur dafür sinnvoll nutzbar sind. Sie schließt Informationen über den Softwareentwicklungsprozess ein, zum Beispiel Techniken zur Anforderungserhebung, UML zur Modellierung einer Softwarearchitektur, Programmiersprachen für die Implementierung und Testmethoden. Informationen, die zur Software-Domäne gehören, beschreiben also *wie* Software entwickelt wird.

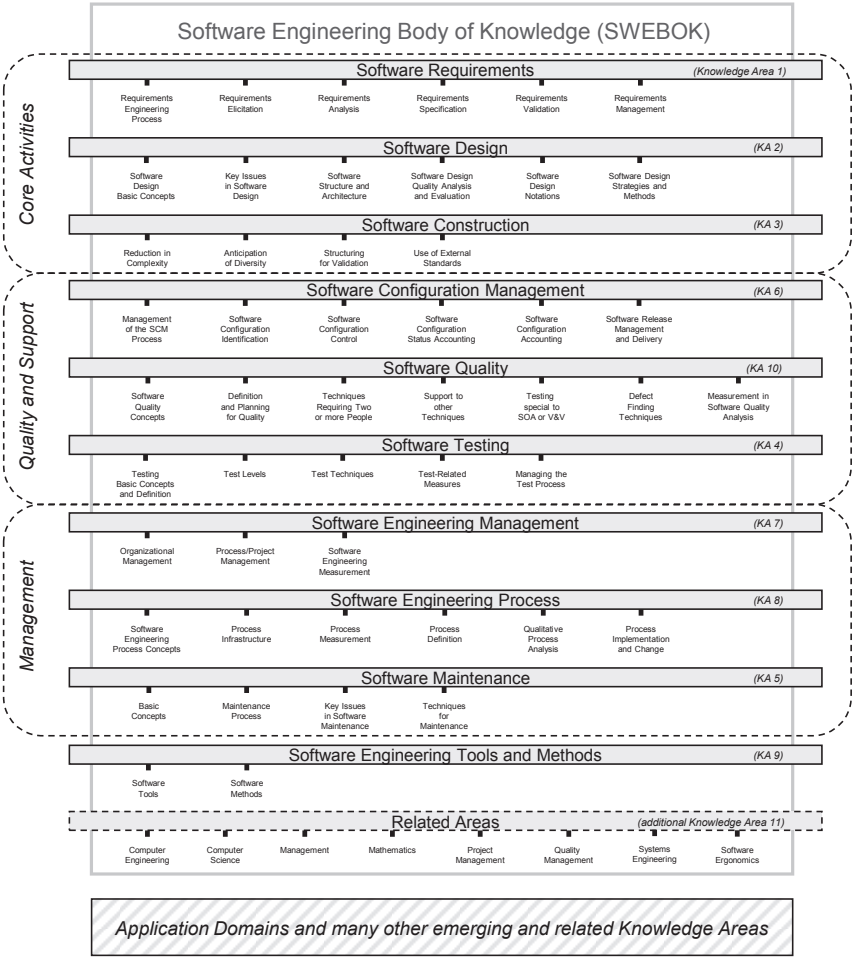


Abb. 3.8: Wissensgebiete des Software Engineering, angrenzende Domänen und Anwendungsdomänen aus [Schneider 2009, S. 57]

Anwendungs-Domäne: Die Anwendungsdomäne (auch Problem-Domäne) umfasst Informationen, die aus dem Bereich des mit Software zu lösenden Problems stammen. Typische Anwendungsdomänen der Softwareentwicklung sind die Medizin, der Automobilbau, der Finanzsektor, die Telekommunikationsbranche und viele mehr. Jede Anwendungsdomäne hat ihre eigenen Fachbegriffe und Besonderheiten, die es bei der Softwareentwicklung zu berücksichtigen gilt. Fachliche Anforderungen sind ebenfalls Informationen aus der Anwendungsdomäne. Informationen, die zur Anwendungsdomäne gehören, beschreiben also entweder *was* für ein Problem die Software lösen soll, oder *wie* das Problem aus fachlicher Sicht zu lösen ist.

Andere Problemlösungs-Domänen: In diese Domäne fallen alle Informationen, die in der Softwareentwicklung relevant sind, aber keiner der beiden anderen Domänen zugeordnet werden können. So ist zum Beispiel das Wissen über Abstraktion und Modellierung eine wichtige Fähigkeit bei der Entwicklung von Software (vgl. z.B. [Devlin 2003] oder [Kramer 2007]) und damit auch Information über dieses Wissen. Weitere wichtige Domänen sind u.a. Projektmanagement (Wie werden Projekte durchgeführt?) und Kommunikation (Wie kommuniziert man erfolgreich?).

Diese drei für die Softwareentwicklung wichtigen Domänen lassen sich auch im Software Engineering Body of Knowledge (SWEBOK) [SWEBOK 2004] wiederfinden. Abbildung 3.8 zeigt die 10 Wissensgebiete des SWEBOK, angrenzende Domänen und Anwendungsdomänen nach [Schneider 2009, S. 57].

Die Namensgebung dieser drei Domänen wurde aus Softwareentwicklungssicht gewählt. Die Software-Domäne ist eine spezielle Problemlösungsdomäne, da Informationen, die in ihr enthalten sind, beschreiben, wie Probleme mit Hilfe von Software gelöst werden können. Die Anwendungs-Domäne ist eine spezielle Problemlösungsdomäne, in der die mit Software zu lösenden Probleme oder Problemlösungen in der jeweiligen Fachsprache beschrieben werden. Alle anderen Problemlösungsdomänen, wie das Projektmanagement, werden unter "andere" Domänen zusammengefasst. Zusammenfassend wird Informationsdomäne wie folgt definiert:

Definition 3.15: **Informationsdomäne**
Eine Informationsdomäne ist der Gesamtkontext eines Wissensgebiets.

Beispiele für Wissensgebiete sind die Medizin, der Automobilbau, die Bankenbranche, die Mathematik, das Projektmanagement oder die Softwareentwicklung, also Bereiche, die durch ähnliche wirtschaftliche oder forscherische Probleme zusammenhängen. Eine Informationsdomäne enthält auch immer eine Fachsprache, da sie die Fachbegriffe eines Wissensgebietes, deren Bedeutung und ihren Zusammenhang enthält.

Nach Curtis [Curtis 1984, Curtis 1988] kennen sich Softwareentwickler in verschiedenen Wissensgebieten, d.h. Informationsdomänen, unterschiedlich gut aus:

> *Expertise is specific to different knowledge domains. A programmer can be expert in one domain and a novice in another.* [Curtis 1984]

Eine gegebene Information kann immer mindestens einer Informationsdomäne zugeordnet werden. Tabelle 3.4 zeigt einige Beispiele von Informationen in verschiedenen Domänen.

3.4.3 Informationsabstraktheit

Eine weitere Eigenschaft von Information, die in die Theorie aufgenommen werden soll, ist die Abstraktheit von Information (vgl. Kapitel 2.6). Sie ist wichtig, weil mit ihr eine charakteristische Eigenschaft der Softwareentwicklung beschrieben werden kann: die fortlaufende Konkretisierung von abstrakten Anforderungen zur konkreten Software (vgl. Kapitel 3.7.3).

Abstraktion ist die Reduktion auf die wesentlichen Eigenschaften einer betrachteten Menge von Dingen (vgl. Kapitel 2.6). Diese Dinge sind entweder Dinge der realen Welt oder selbst Abstraktionen (vgl. u.a. [Schuenemann 2005, S. 16]). Die Umkehrung der Abstraktion ist die Konkretisierung, d.h. das Wiederhinzufügen von vormals reduzierten Eigenschaften. Durch Konkretisierung nähert man sich wieder den realen Dingen. Man reduziert die Menge der Dinge, die von der Abstraktion noch abgedeckt werden, bis man durch Hinzufügen aller ursprünglichen Eigenschaften schließlich wieder bei einem konkreten Ding der realen Welt ist.

Welcher Zusammenhang lässt sich zwischen Abstraktion und Information herstellen? Information ist personenunabhängiges Wissen. Wissen ist Fähigkeit zum Problemlösen. Also beschreiben Informationen Problemlösungen. Eine

Tabelle 3.4: Beispiele für Informationen aus verschiedenen Informationsdomänen

Domäne	**Beispielinformation***
Software	Ein tabellarischer UseCase sollte mind. einen Namen, einen Hauptakteur und das Hauptszenario enthalten
Software	Eine Model-View-Controller-Architektur erleichter die Wartung von Web-Anwendungen
Software	Testen zeigt die Anwesenheit, nicht die Abwesenheit von Fehlern [NATO 1969, S. 16]
Anwendung (Bankenbranche)	Nach § 6 Preisangabenverordnung sind bei Krediten die Gesamtkosten als effektiver Jahreszinssatz auszuweisen
Anwendung (Bankenbranche)	Der durchschnittliche effektive Jahreszins i_{eff} wird wie folgt berechnet: $$i_{\text{eff}} = \sqrt[n]{1+\frac{K}{D}} - 1$$ mit K Kreditgesamtkosten, D Darlehensbetrag und n Laufzeit in Jahren.
Software	`double iEff = Math.pow(1+(K/D), 1.0/n) - 1;`
Andere (Deutsche Grammatik)	Verwendung des Hilfsverbs "werden" und Verb im Partizip Perfekt (Information darüber, wie man Passivsätze erkennen kann)
Andere	Die Fähigkeit zur Abstraktion hilft bei der Modellbildung (meist implizites Wissen)
Andere (Projektmanagement)	Risikomanagement hilft unerwartete Ereignisse in die Projektplanung einzubeziehen und somit Kosten zu senken, indem frühzeitig Gegenmaßnahmen getroffen werden können.

* Die dargestellten Aussagen repräsentieren Informationen, welche in der angegebenen Domäne zur Problemlösung eingesetzt werden können. Es werden nur die Nachrichten gezeigt. Sie sind schriftlich in deutscher Sprache verfasst. Zur korrekten Bedeutungszuweisung (vgl. Def. 3.13) ist Vorwissen (d.h. Kontextwissen) der deutschen Sprache und ggf. bestimmter Fachsprachen (z.B. Mathematik oder Software Engineering) notwendig.

abstrakte Information beschreibt dann eine größere Menge konkreter (d.h. realer) Problemlösungen, als eine konkrete Information. Abstraktheit von Information wird wie folgt definiert:

Definition 3.16: **Abstraktheit von Information**
Abstraktheit von Information ist ein Maß für die Anzahl möglicher Lösungen, die eine Information beschreibt.

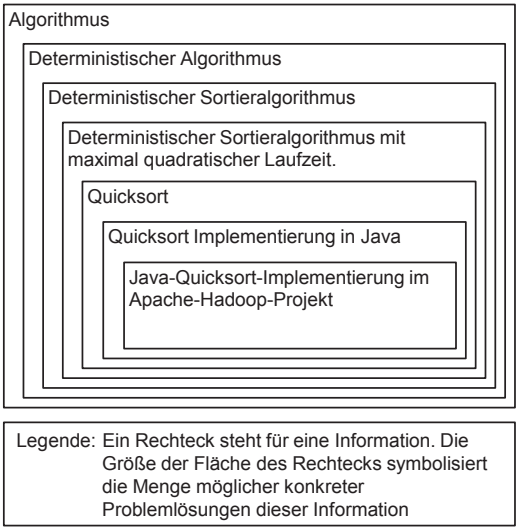


Abb. 3.9: Beispiel unterschiedlich abstrakter Informationen

Zum Beispiel ist die Information *"Liste sortieren"* abstrakter als die Information *"Liste mit maximal quadratischer Laufzeit sortieren"*, die wiederum abstrakter als die Information *"Liste mit Quicksort sortieren"* ist. Jede dieser Konkretisierungen schränkt den Lösungsraum weiter ein. Ähnlich verhält es sich mit Informationen aus der Anwendungsdomäne. Häufig starten Softwareprojekte mit einer eher abstrakten Idee oder Vision des zu lösenden Problems. Im Verlauf der Anforderungserhebung muss die abstrakte Vision in konkrete Anforderungen überführt werden. Das in Abbildung 3.9 gezeigte, an [Hofstadter 1999, S. 376] angelehnte Beispiel zeigt, dass Informationen auf sehr vielen unterschiedlichen Abstraktionsebenen sein können, d.h. unterschiedlich große

potenzielle Lösungsmengen beschreiben können (absteigend von abstrakter zu konkreter).

3.4.4 Zusammenfassung

In Abbildung 3.10 sind die gerade beschriebenen Eigenschaften von Information und ihre Zusammenhänge noch einmal zusammenfassend dargestellt.

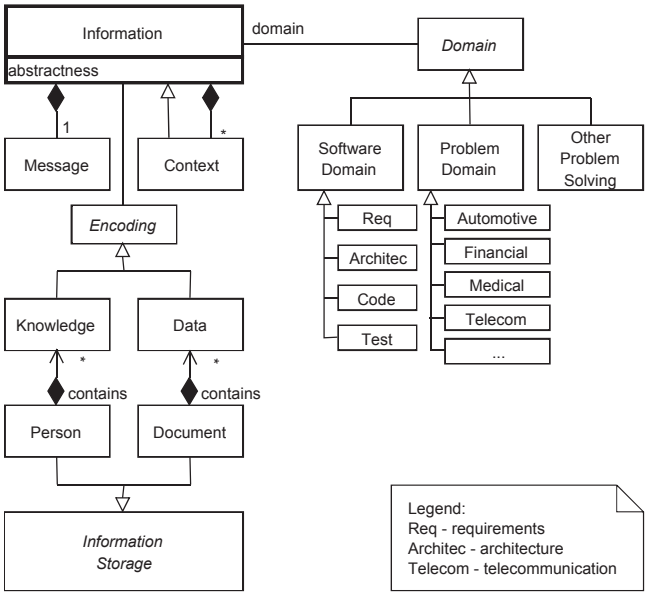


Abb. 3.10: Domänenmodell: Information

Im Zentrum der Abbildung ist Information. Information besteht aus einer Nachricht und zugehörigem Kontext. Der Kontext ist selbst auch Information. Weiterhin hat Information eine Abstraktheit und ist einer Domäne zugeordnet. Wichtige Domänen sind die Problem- bzw. Anwendungs-Domänen wie Medizin, Telekommunikation oder Automobilbau, die Software-Domäne sowie andere Domänen wie die Projektmanagementdomäne.

3.5 Informationsfluss

Die Informationsflusstheorie schlägt Informationsfluss als zentralen Betrachtungsgegenstand zur Analyse und Verbesserung von Softwareentwicklungsprojekten vor. Daher ist Informationsfluss der namensgebende Aspekt der Theorie. In diesem Unterkapitel soll geklärt werden, was Informationsfluss ist und warum sich Informationsfluss als zentraler Betrachtungsgegenstand für die Theorie eignet.

Information ist entweder in Form von Daten in Dokumenten oder in Form von Wissen im menschlichen Gedächtnis gespeichert (vgl. die drei vorangegangenen Kapitel). Ein Dokument wird daher auch als physischer Informationsspeicher und der Mensch als physiologischer Informationsspeicher bezeichnet (vgl. Abb. 3.3 oben oder Abb. 3.11 unten).

Informationsfluss ist der Übergang zwischen Informationsspeichern. Dabei kann die Information unverändert übertragen werden, was einem Kopiervorgang entspricht, oder bei der Übertragung verändert werden, z.B. durch gezieltes Weglassen oder kreative Weiterentwicklung von Information im menschlichen Geist. Informationsfluss wird wie folgt definiert.

Definition 3.17: **Informationsfluss**

Ein Informationsfluss ist eine Aktivität, bei der Informationen i aus Quellinformationsspeichern in der Zeit $t > 0$ in Informationen i' in Zielinformationsspeicher überführt werden. Die Überführung kann aus Kopieren, Kombinieren und Weglassen von Informationen bestehen, sowie einer beliebigen Kombination dieser Operationen. Man schreibt:

$$i \xrightarrow{t} i'$$

Informationsfluss wird hier als Aktivität definiert, weil er immer durch einen aktiven Prozess geschieht oder dieser Prozess zumindest aktiv angestoßen wird. Eine Person schreibt aktiv etwas in ein Dokument (Wissen → Daten). Eine Person liest aktiv ein Dokument (Daten → Wissen). Ein Kopiervorgang zwischen zwei Dokumenten muss aktiv gestartet werden (Daten → Daten). Zwei Personen kommunizieren aktiv miteinander (Wissen → Wissen). Auch wenn

eine Person ihr Wissen nicht aktiv preisgibt, so kann ihr Wissen an eine andere Person übertragen werden, indem diese z.B. Tätigkeiten der ersten Person aktiv beobachtet.

3.5.1 Elementare Informationsflüsse

Insgesamt lassen sich vier grundlegende Informationsspeicherübergänge unterscheiden. Diese werden als elementare Informationsflüsse bezeichnet.

> Definition 3.18: **Elementarer Informationsfluss**
> Ein elementarer Informationsfluss ist ein Informationsfluss, bei dem Information ohne Zwischenspeicher von genau einem Quell- in genau einen Zielinformationsspeicher überführt wird.

"Ohne Zwischenspeicher" meint, dass am Informationsfluss kein Informationsspeicher beteiligt ist, der weder Quell- noch Zielinformationsspeicher ist.

Da Information in zwei verschiedenen Formen gespeichert werden kann, nämlich als Daten in Dokumenten und als Wissen im menschlichen Gedächtnis (vgl. Abbildung 3.10), gibt es insgesamt vier Kombinationsmöglichkeiten eines Informationsspeicherwechsels und damit vier Typen von elementaren Informationsflüssen (vgl. Abb. 3.11): ① Externalisierung, ② Internalisierung, ③ Kombination und ④ Sozialisation.

① **Externalisierung (Wissen → Daten):** Externalisierung ist der Übergang von Information gespeichert als Wissen im physiologischen Informationsspeicher Mensch in Information gespeichert als Daten im physischen Informationsspeicher Dokument. Beim Übergang von Wissen in Daten wird die Nachricht vom Kontext getrennt. Nur die Nachricht wird als Daten kodiert. Der Kontext wird nicht mit im Dokument gespeichert. Der Kontext wird für einen potenziellen Leser des Dokuments als vorhanden vorausgesetzt. Externalisierung kommt bei der Softwareentwicklung häufig vor. Es werden u.a. Fachkonzepte, Anforderungsspezifikationen, Entwürfe, Testpläne und schließlich die Implementierung erstellt. Da das Endergebnis von Softwareentwicklung immer Software ist, die nach Definition 3.10 auch ein Dokument ist, beinhaltet Softwareentwicklung immer wenigstens eine Externalisierung, spätestens bei der Erstellung des Quelltextes.

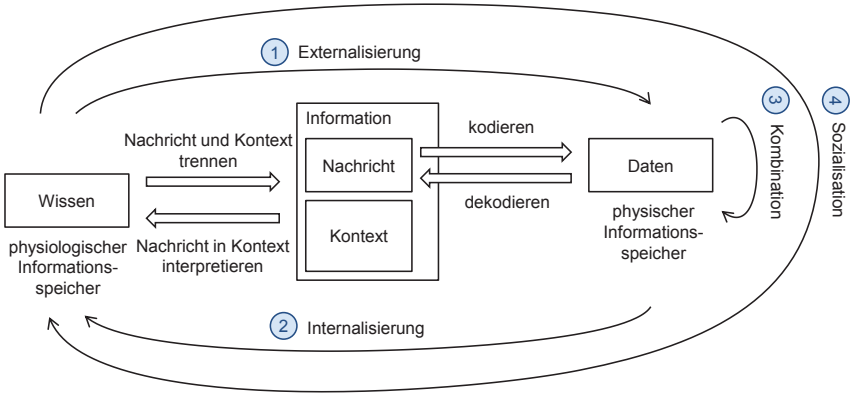


Abb. 3.11: Elementare Informationsflüsse

② **Internalisierung (Daten → Wissen):** Internalisierung ist der Übergang von Information gespeichert als Daten im physischen Informationsspeicher Dokument in Information gespeichert als Wissen im physiologischen Informationsspeicher Mensch. Beim Übergang von Daten in Wissen wird die Nachricht dekodiert und im zugehörigen Kontext interpretiert. Der Kontext wird als vorhanden vorausgesetzt. Auch Internalisierung kommt häufig in der Softwareentwicklung vor. So müssen Entwickler zum Beispiel Spezifikationen lesen und verstehen, d.h. die gewünschte Funktion der zu entwickelnden Software in ihr Wissen aufnehmen, oder ihre Kenntnisse einer bestimmten Technologie erweitern, indem sie ein Fachbuch lesen oder ein Online-Tutorial durcharbeiten. Probleme bei der Internalisierung entstehen meist, weil der für die Interpretation notwendige Kontext als vorhanden vorausgesetzt wird. Ist das nicht der Fall, können die Daten nicht internalisiert werden (z.B. versteht der Entwickler eine Anforderung nicht) oder sie werden falsch interpretiert (z.B. glaubt der Entwickler die Anforderung verstanden zu haben, der Kunde meinte aber etwas völlig anderes, vgl. Symmetry of Ignorance [Rittel 1984]).

③ **Kombination (Daten → Daten):** Kombination ist der Übergang von Information zwischen zwei physischen Informationsspeichern, d.h. Datenverarbeitung zwischen Dokumenten. Dabei können Daten transfor-

miert, kombiniert oder weggelassen werden. Auch Kombinationen sind in heutigen Softwareentwicklungsprojekten immer vorhanden. Zum Beispiel ist der Kompiliervorgang, der aus dem Quelltext das binäre Computerprogramm erzeugt, eine Kombination. Andere in der Softwareentwicklung häufig auftretende Kombinationen sind u.a. die Generierungen aus Modellen, Datenmigrationen oder automatische Zusammenfassungen von Daten aus verschiedenen Quellen zu einem Report, z.B. beim Controlling von Projekten.

④ **Sozialisation (Wissen $\overset{\text{Daten}}{\longrightarrow}$ Wissen):** Sozialisation ist der Übergang von Information zwischen zwei physiologischen Informationsspeichern, d.h. die Weitergabe von Wissen zwischen Personen. Wissen kann nicht direkt von einer Person an eine andere weitergegeben werden. Wissen kann nur über den Umweg Daten transportiert werden. Sozialisation ist also vergleichbar mit einem aus Externalisierung und Internalisierung zusammengesetzten Informationsfluss, bei dem die externalisierten Daten nicht in einem Dokument zwischengespeichert werden (vgl. ④ in Abb. 3.11). In der Softwareentwicklung ist Sozialisation häufig nötig, z.B. muss ein Entwickler wissen, was der Kunde möchte, oder in größeren Teams müssen Entwickler wissen, welche Aufgaben und Zwischenergebnisse ihre Kollegen haben. In beiden Fällen können sie dieses Wissen durch Sozialisation erlangen.

Es werden die gleichen Begriffe benutzt, wie sie Nonaka [Nonaka 1991] zur Beschreibung der vier Wissenserzeugungsarten eines Unternehmens nutzt (vgl. Kapitel 2.2.1.1). Dies bietet sich an, weil die Bedeutung ähnlich ist. Sozialisation hat etwas mit dem Zusammenwirken mehrerer Personen zu tun. Bei Externalisierung und Internalisierung extrahiert der Mensch Wissen bzw. nimmt dieses in sich auf. Unterschiede ergeben sich daraus, dass Nonake nicht zwischen Wissen, Information und Daten unterscheidet. Insbesondere ist Kombination bei Nonake etwas, bei dem ein Mensch intern aus vorhandenem Wissen neues Wissen erzeugt, wohingegen es hier etwas externes ist, bei dem Daten neu zusammengesetzt werden. Es entstehen dabei aber keine neuen Informationen. Die Neuerzeugung von Wissen wird hier als Denken bezeichnet. Denken ist etwas, was ein Mensch nur intern tun kann, und ist daher kein Informationsfluss. Sozialisation, Externalisierung, Internalisierung und Kombination werden für die Informationsflusstheorie wie folgt definiert:

Definition 3.19: **Sozialisation**
Sozialisation ist ein elementarer Informationsfluss, bei dem Quell- und Zielinformationsspeicher Personen sind.

Sozialisation ist die Weitergabe von Wissen, da Wissen im Gedächtnis von Personen gespeichert ist.

Definition 3.20: **Externalisierung**
Externalisierung ist ein elementarer Informationsfluss, bei dem der Quellinformationsspeicher eine Person und der Zielinformationsspeicher ein Dokument ist.

Externalisierung ist die Umwandlung von Wissen in Daten, z.B. durch Artikulation oder Niederschrift.

Definition 3.21: **Internalisierung**
Internalisierung ist ein elementarer Informationsfluss, bei dem der Quellinformationsspeicher ein Dokument und der Zielinformationsspeicher eine Person ist.

Internalisierung ist die Erzeugung von Wissen aus Daten.

Definition 3.22: **Kombination**
Kombination ist ein elementarer Informationsfluss, bei dem Quell- und Zielinformationsspeicher Dokumente sind.

Durch Zusammenführen, gezieltes Weglassen oder Transformation in ein anderes Format werden Daten zu neuen Daten kombiniert. Kombination ist Datenverarbeitung.

3.5.2 Informationsflussaktivität

Aus den elementaren Informationsflüssen lassen sich komplexere Informationsflüsse zusammensetzen. Zusammengesetzte Informationsflüsse sind notwendig, um Softwareentwicklungsaktivitäten auf einer passenden Abstraktionsebene analysieren zu können. So besteht zum Beispiel eine typische Programmieraktivität nicht nur aus der Erstellung des Quellcodes (Externalisierung), sondern schließt üblicherweise auch Internalisierung (z.B. Anforderungen lesen, unbekannte Funktionen von Programmierwerkzeugen lernen), Sozialisation (z.B. Anforderungen erfragen, Abstimmung mit Kollegen) und Kombination (z.B. Kompilierung, Generierung) ein.

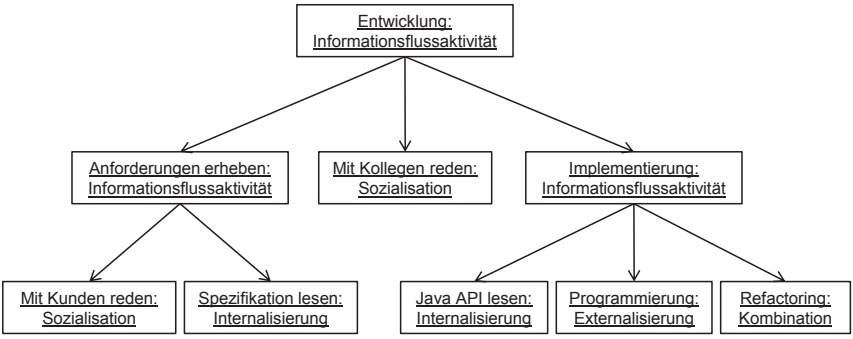


Abb. 3.12: Beispiel für einen hierarchischen Zusammenhang von Informationsflüssen

Definition 3.23: **Informationsflussaktivität**

Eine Informationsflussaktivität ist ein Informationsfluss, der sich aus einer Menge von Informationsflüssen zusammensetzt.

Diese rekursive Definition ermöglicht es, dass sich hierarchisch strukturierte Informationsflüsse modellieren lassen. Der Zusammenhang zwischen Informationsfluss, Informationsflussaktivität und elementarem Informationsfluss entspricht einem Kompositum-Entwurfsmuster (vgl. [Gamma 1995], Abb. 3.13 Mitte). Ein Beispiel für eine Informationsflusshierarchie ist in Abbildung 3.12

gezeigt[3]. Die Blätter des Baumes sind elementare Informationsflüsse, die anderen Knoten sind Informationsflussaktivitäten auf unterschiedlichen Abstraktionsebenen. Das Rekursionsende der rekursiven Definition von Informationsflussaktivität ist durch die elementaren Informationsflussaktivitäten möglich.

Im Gegensatz zu elementaren Informationsflüssen kann eine Informationsflussaktivität Zwischenspeicher enthalten. Das heißt, die Informationen aus den Quellinformationsspeichern können über Zwischeninformationsspeicher fließen, bevor sie in die Zielinformationsspeicher überführt werden. Zum Beispiel kann die Informationsflussaktivität Anforderungserhebung durch ein direktes Gespräch des Analysten mit dem Kunden durchgeführt werden (elementarer Fluss) oder der Kunde schreibt zunächst die Anforderungen in ein Lastenheft, welches der Analyst liest. Aus Sicht der Informationsflussaktivität ist das Lastenheft dann ein Zwischenspeicher.

Wie in der Einleitung beschrieben, führen Kommunikations- und Dokumentationsprobleme häufig zu Softwareentwicklungsproblemen. Sowohl Kommunikation als auch Dokumentation können als Informationsfluss beschrieben werden. Da sie für die Softwareentwicklung besonders relevant sind, werden sie als spezielle Informationsflüsse definiert und so in die Informationsflusstheorie aufgenommen.

Definition 3.24: **Kommunikation**

Kommunikation ist eine Informationsflussaktivität, bei der alle Quell- und Zielinformationsspeicher Personen und, falls vorhanden, alle Zwischeninformationsspeicher Dokumente sind.

Definition 3.25: **Dokumentation**

Dokumentation ist eine Informationsflussaktivität, bei der alle Zielinformationsspeicher Dokumente und, falls vorhanden, alle Zwischeninformationsspeicher Dokumente sind.

[3] Diese Art der Darstellung soll den hierarchischen Zusammenhang der Informationsflüsse veranschaulichen, sie ist nicht geeignet, Reihenfolgen und inhaltliche Abhängigkeiten zwischen den an den Informationsflüssen beteiligten Informationsspeichern abzubilden. Dafür kann z.B. die FLOW-Notation (vgl. Kapitel 6.1.3) benutzt werden.

Definition 3.26: **Lernen**

Lernen ist eine Informationsflussaktivität, bei der alle Zielinformationsspeicher Personen und, falls vorhanden, alle Zwischeninformationsspeicher Dokumente sind.

Kommunikation ist also die Weitergabe (Sozialisation) von Wissen zwischen Personen, Dokumentation die Externalisierung und Kombination von Wissen und Daten in Dokumente und Lernen ist der Neuerwerb von Wissen durch Internalisierung von Daten oder aus dem Wissen anderer (Sozialisation).

Allen drei Definitionen ist gemein, dass Zwischenspeicher, sofern es welche gibt, nur Dokumente sein dürfen. Diese Einschränkung wurde gemacht, damit die hier definierten Begriffe Kommunikation, Dokumentation und Lernen nicht im Widerspruch zum umgangssprachlichen Verständnis dieser Begriffe stehen (vgl. Adäquatheitsforderung von Explikationen in Kapitel 2.1.1). Wären Personen als Zwischenspeicher erlaubt, so könnte man bei einem Informationsfluss von Person A zu Person B und einem anschließenden Informationsfluss von Person B zu Person C von einer Kommunikation zwischen Person A und Person C sprechen. Dies soll aber ausgeschlossen werden. Kommunikation über Dokumente widerspricht hingegen nicht dem allgemeinen Verständnis von Kommunikation und ist daher erlaubt. Ähnliches gilt für diese Einschränkung bei Dokumentation und Lernen.

Da Kommunikation eine Informationsflussaktivität ist, die in der Softwareentwicklung eine wichtige Rolle spielt, wird sie in Kapitel 3.6 gesondert betrachtet. Dokumentation und Lernen spielen in dem Zusammenhang zwar auch eine wichtige Rolle (vgl. z.B. Kapitel 2.3.2), da sie aber zum Teil durch die Überlegungen über Kommunikation mit abgedeckt werden, werden sie nicht noch einmal in einem eigenen Kapitel ausführlicher behandelt. Zunächst werden aber noch wichtige Begriffe für die Informationsübertragung definiert.

3.5.3 Übertragungsmedium

Informationen können nur in Form von Daten fließen, da Wissen immer an ein Individuum gebunden ist (vgl. Def. 3.2). Das heißt aber nicht, dass an einem Informationsfluss immer ein Dokument beteiligt ist. Zum Beispiel fließen bei einem direkten verbalen Gespräch zwischen zwei Personen Informationen in Form von Schallwellen, die aber nicht in einem Dokument gespei-

chert werden. Die Daten sind direkt nach dem Gespräch nicht mehr verfügbar. Die Daten sind in diesem Fall Übertragungsmedium des Informationsflusses und nicht Speichermedium. Im Rahmen der Informationsflusstheorie werden Übertragungsmedium und Medium synonym benutzt, da das Speichermedium bereits durch den Begriff Informationsspeicher abgedeckt ist.

In der Softwareentwicklung werden viele unterschiedliche Übertragungsmedien benutzt. Dies sind z.B. technische Medien wie E-Mail, Instant Messaging oder Videokonferenzsysteme. Aber auch das natürliche Medium von Angesicht zu Angesicht wird in der Softwareentwicklung häufig genutzt, z.B. bei Abstimmungsmeetings oder im direkten Kundengespräch. Da einige Medien wie Angesicht zu Angesicht tatsächlich eine Kombination mehrerer Medien sind (Sprache und Bild), wird in der Informationsflusstheorie noch einmal zwischen Medium und Kanal unterschieden. Das elementare Hilfsmittel zur Informationsübertragung wird mit Kanal bezeichnet. Ein Medium gibt einem Kanal oder einer Menge von Kanälen einen Namen.

Definition 3.27: **Kanal**

Ein Kanal chn für einen Datentyp D ist ein Hilfsmittel zur Datenübertragung, der die Daten $d \in D$ in der Zeit $t > 0$ überträgt. Man schreibt:

$$chn(d) = t$$

Definition 3.28: **Medium**

Ein Medium ist eine Menge von Kanälen.

Definition 3.29: **Hintereinanderschaltung von Kanälen**

Für Kanäle chn_1 und chn_2 ist die Hintereinanderschaltung $chn = chn_1 \circ chn_2$ ein Kanal mit

$$chn(d) = (chn_1 \circ chn_2)(d) = chn_1(d) + chn_2(d).$$

Ein Kanal ermöglicht also den Transport von Daten. Ein Kanal ist immer notwendig für den elementaren Informationsfluss Sozialisation, weil Wissen nur über Daten weitergegeben werden kann und diese Daten vom Sender zum

Empfänger transportiert werden müssen. Damit benötigen auch alle Informationsflussaktivitäten, die mindestens eine Sozialisation enthalten, einen Übertragungskanal, d.h. insbesondere auch jede Kommunikation. Ein Beispiel für eine Hintereinanderschaltung von drei Kanälen ist ein Telefongespräch, bei dem zunächst die Schallwellen des Sprechers über den Kanal Luft zum Mikrofon des Telefonhörers, anschließend die elektrischen Signale über die Telefonleitung und abschließend die Schallwellen vom Lautsprecher zum Empfänger transportiert werden.

Ein Kanal ist immer für einen bestimmten Datentyp besonders gut geeignet. Das kann zum einen auf natürliche Weise so sein, z.B. Luft für Schallwellen, oder weil der Kanal speziell für einen bestimmten Datentyp entwickelt wurde, z.B. elektronische Verbindungen für digitale Signale. Besonders gut geeignet meint dabei, dass die Daten für den Transport nicht noch einmal aufbereitet, z.B. in ein anderes Format umgewandelt, werden müssen. Luft ist direkt für die Übertragung von gesprochener natürlicher Sprache geeignet. Für die Übertragung der natürlichen Sprache über Telefonleitungen müssen die Schallwellen erst in elektrische Signale umgewandelt werden.

Ein Kanal hat neben der Beschaffenheit für einen bestimmten Datentyp noch weitere Eigenschaften. Für die Informationsflusstheorie sind die folgenden drei Eigenschaften besonders relevant:

1. Er ist besonders gut geeignet für einen bestimmten Datentyp.

2. Er hat eine bestimmte Latenz.

3. Er hat eine bestimmte Bandbreite.

Für die Informationsflusstheorie werden die Datentypen, für die ein Kanal ausgelegt ist, nicht auf der technischen Ebene von Signalen und Schallwellen betrachtet, sondern auf den abstrakteren in Kapitel 3.3 beschriebenen für die Softwareentwicklung relevanten Datentypen mit der Klassifikation nach Dimension, Sinn und Ursprung (vgl. Def. 3.8).

Bandbreite und Latenz werden allgemein für einen Kanal definiert:

Definition 3.30: **Bandbreite**
Für einen Kanal chn und Daten $d \in D$ ist $b_{chn}(d)$ die Bandbreite, mit

$$b_{chn}(d) = \frac{|d|}{chn(d)}.$$

Definition 3.31: **Latenz**
Für einen Kanal chn ist die Latenz l_{chn} die kürzeste Zeit, in der ein Antwort-Datum auf ein gesendetes Datum wieder beim Absender sein kann. Es gilt:

$$l_{chn} = 2 \cdot \min(\{chn(d) | d \in D\})$$

Latenz bezeichnet also die kürzest mögliche Antwortzeit.

Für die Bandbreite von hintereinandergeschalteten Kanälen gilt folgendes:

Satz 3.1: **Kanalbandbreite**
Für einen hintereinander geschalteten Kanal $chn = chn_1 \circ chn_2$ für den Datentyp D und Daten $d \in D$ gilt:

$$b_{chn}(d) < \min(b_{chn_1}(d), b_{chn_2}(d))$$

Herleitung.

$$\begin{aligned}
b_{chn}(d) &= \frac{|d|}{chn(d)} && \text{nach Def. 3.30} \\
&= \frac{|d|}{chn_1(d)+chn_2(d)} && \text{nach Def. 3.29} \\
&= \frac{1}{\frac{chn_1(d)+chn_2(d)}{|d|}} \\
&= \frac{1}{\frac{chn_1(d)}{|d|}+\frac{chn_2(d)}{|d|}} \\
&= \frac{1}{\frac{1}{b_{chn_1}(d)}+\frac{1}{b_{chn_2}(d)}} && \text{nach Def. 3.30}
\end{aligned}$$

Für $b_{chn_2}(d) > 0$ gilt:

$$b_{chn}(d) = \frac{1}{\frac{1}{b_{chn_1}(d)} + \frac{1}{b_{chn_2}(d)}} < b_{chn_1}(d)$$

Für $b_{chn_1}(d) > 0$ gilt:

$$b_{chn}(d) = \frac{1}{\frac{1}{b_{chn_1}(d)} + \frac{1}{b_{chn_2}(d)}} < b_{chn_2}(d)$$

Daraus folgt:

$$b_{chn}(d) = \frac{1}{\frac{1}{b_{chn_1}(d)} + \frac{1}{b_{chn_2}(d)}} < \min(b_{chn_1}(d), b_{chn_2}(d))$$

□

Für die Latenz von hintereinandergeschalteten Kanälen gilt folgendes:

Satz 3.2: **Kanallatenz**
Für einen hintereinander geschalteten Kanal $chn = chn_1 \circ chn_2$ gilt:

$$l_{chn} = 2 \cdot \min(\{chn_1(d) + chn_2(d) | d \in D\})$$

Herleitung. Dies folgt direkt aus der Definition von Latenz (Def. 3.31) und der Hintereinanderschaltung von Kanälen (Def. 3.29). □

In Tabelle 3.5 sind einige häufig in der Softwareentwicklung anzutreffende Übertragungsmedien aufgelistet, inklusive der Kanäle, aus denen sie bestehen.

Da in der Softwareentwicklung Bandbreite und Latenz aus Informationsflusssicht nicht nur durch die technischen Übertragungskanäle begrenzt sind, sondern oft auch durch die limitierten Fähigkeiten des Menschen, werden im Folgenden Kanäle auch auf der Ebene der elementaren Informationsflüsse definiert.

Tabelle 3.5: Typische Medien im SE und zugehörige Kanäle

SE Beispiel	Medien	Kanäle
Pair Programming	Angesicht zu Angesicht, Quellcode	Gesprochene natürliche Sprache, bewegtes natürliches Bild, künstliche Sprache (Quellcode)
Besprechung am Whiteboard	Angesicht zu Angesicht, Whiteboard	Gesprochene natürliche Sprache, bewegtes natürliches Bild, künstliches Bild (Whiteboard)
Lokale Besprechung	Angesicht zu Angesicht	Gesprochene natürliche Sprache, bewegtes natürliches Bild
Verteilte Besprechung	Videokonferenz mit geteiltem Desktop	Gesprochene natürliche Sprache, bewegtes natürliches Bild, künstliches Bild (Computer-Desktop)
Telefon-konferenz	Telefon	Gesprochene natürliche Sprache
Text-Chat im Team	Sofort-Nachricht (Instant Messaging – IM)	Geschriebene natürliche Sprache
Statusbenach-richtigungen	E-Mail	Geschriebene natürliche Sprache
Unterneh-mensintranet	Wiki	Geschriebene natürliche Sprache, künstliche Sprache (optional), künstliches Bild (optional)
Anforderungs-spezifikation	Papier-Dokument	Geschriebene natürliche Sprache, künstliche Sprache (optional), künstliches Bild (optional)
Quellcode	elektron. Dokument	künstliche Sprache

3.5.4 Informationsflusseigenschaften

Heute sind technische Übertragungskanäle sehr leistungsfähig. Netzwerke können 1GBit/s und mehr übertragen. Das ist für die meisten Datentypen mehr als der Mensch verarbeiten kann. Daher werden hier weitere Kanäle eingeführt, die den Menschen und seine Fähigkeiten berücksichtigen, und die später u.a. für Theoreme über Kommunikation genutzt werden können (vgl. Kapitel 4.2).

Definition 3.32: **Externalisierungskanal**

Ein Externalisierungskanal ext ist ein Kanal für einen Datentyp D, der beschreibt, in welcher Zeit $t > 0$ ein Mensch die Daten $d \in D$ externalisieren kann. Man schreibt:

$$ext(d) = t$$

Definition 3.33: **Internalisierungskanal**

Ein Internalisierungskanal int ist ein Kanal für einen Datentyp D, der beschreibt, in welcher Zeit $t > 0$ ein Mensch die Daten $d \in D$ internalisieren kann. Man schreibt:

$$int(d) = t$$

Ein Externalisierungskanal ext kann vor einen Kanal chn geschaltet werden. Ein Internalisierungskanal int kann nach einen Kanal chn geschaltet werden.

Definition 3.34: **Sozialisationskanal**

Ein Sozialisationskanal soc ist die Hintereinanderschaltung von Externalisierungskanal ext, beliebigen Kanälen chn und Internalisierungskanal int. Man schreibt:

$$soc = ext \circ chn \circ int$$

Für Daten $d \in D$ gilt $soc(d) = ext(d) + chn(d) + int(d)$. Aus Satz 3.1 folgt dann für die Sozialisationsbandbreite:

Korollar 3.1: **Sozialisationsbandbreite**
Die Sozialisationsbandbreite $b_{soc}(d)$ ist kleiner als das Minimum aus

- Externalisierungsbandbreite $b_{ext}(d)$,
- Kanalbandbreite $b_{chn}(d)$ und
- Internalisierungsbandbreite $b_{int}(d)$.

Es gilt:

$$b_{soc}(d) < \min(b_{ext}(d), b_{chn}(d), b_{int}(d))$$

Die Kanalbandbreite bestimmt nicht immer die Sozialisationsbandbreite. So ist zum Beispiel bei einer Instant-Messaging-Anwendung die Tippgeschwindigkeit des Senders (Repräsentation der Daten) der limitierende und damit bestimmende Faktor für die Sozialisationsbandbreite.

Typische Externalisierungs- und Internalisierungsbandbreiten je Datentyp sind in Tabelle 3.6 gezeigt.

Für die Bestimmung der Latenz einer Sozialisation, d.h. der Externalisierung von Wissen in Daten und der Internalisierung der Daten in Wissen, spielen folgende Zeiten eine wichtige Rolle:

- Die Zeit, die der Mensch benötigt, um Nachricht und Kontext zu trennen. Diese Zeit wird vom Denkprozess bestimmt.
- Die Zeit, die der Mensch benötigt, um die Nachricht in Daten zu repräsentieren. Diese Zeit ist von der menschlichen Fähigkeit zur Repräsentation von Daten abhängig. Z.B. kann ein Mensch nur ca. 210 Wörter pro Minute sprechen (vgl. [Tauroza 1990]) oder ca. 65 Wörter pro Minute tippen (vgl. [Roeber 2003]).
- Die Zeit, die der Mensch benötigt, um Daten wahrzunehmen.
- Die Zeit, die der Mensch benötigt, um Daten zu interpretieren. Diese Zeit ist wieder vom Denkprozess bestimmt, da der Mensch die wahrgenommenen Daten mit seinem vorhandenen Wissen in Verbindung bringen muss und den Daten so Bedeutung gibt. Z.B. kann der Mensch zwischen 180 und 200 Wörter pro Minute lesen und verstehen (vgl. [Ziefle 1998]).

Tabelle 3.6: Typische Bandbreiten für Externalisierung und Internalisierung je Datentyp

Datentyp D	**Externalisierungs-bandbreite** $b_{ext}(d)$	**Internalisierungs-bandbreite** $b_{int}(d)$
Gesprochene natürliche Sprache	210 Wörter (260 Silben) pro Minute[1]	250 Wörter pro Minute[2]
Geschriebene natürliche Sprache	Handschrift: bis zu 38 Wörter pro Minute[3] Tippgeschwindigkeit: 65 Wörter pro Minute[4]	Papier: 200 Wörter pro Minute[5] Monitor: 180 Wörter pro Minute[5]
Geschriebene künstliche Sprache	Notations-abhängig, mit Übung ähnlich natürlicher Sprache, Bsp.: ca. 25 LOC pro Stunde[6]	Notations-abhängig, Bsp.: ca. 120 LOC pro Stunde[7]
Statisches natürliches Bild	–	begrenzt durch den menschlichen Sehapparat, techn. Bsp.: Full HD Auflösung (1920x1080)[8]
Statisches künstliches Bild	Stark Notations-abhängig	Stark Notations-abhängig
Bewegtes natürliches Bild	–	begrenzt durch den menschlichen Sehapparat, techn. Bsp.: Full HD Auflösung (1920x1080)[8] mit ca. 30fps

[1] Durchschnittlich 263,3 Silben pro Minute, Standardabweichung ±31,26, bei Konversation in britischem Englisch [Tauroza 1990]
[2] Hörverständnistest in Englisch ohne signifikante Verständnisprobleme [Foulke 1968]
[3] Durchschnittlich 37,83 Wörter pro Minute, Standardabweichung ±4,71, bei Schnellschreibtest in Englisch [Summers 2003]
[4] Durchschnittlich 64,83 Wörter pro Minute, Standardabweichung ±17,3, bei Tippgeschwindigkeitstest in Englisch [Roeber 2003]
[5] Korrekturlesen englischer Texte [Ziefle 1998]
[6] Produktivität von Studenten in Java, C++ oder C [Prechelt 2000]
[7] Code-Inspektion von C-Quellcode [Barnard 1994]
[8] Bei 50 Zoll Bildschirm, 2 m Bildabstand und einer Sehschärfe von Visus = 1,0 ausreichend.
[3,4] Enthalten ausschließlich Repräsentationskosten. Bei diesen Tests auf maximale Geschwindigkeit sind keine Kontexttrennungskosten enthalten, da vordefinierte Texte geschrieben werden müssen.

Korollar 3.2: **Sozialisationslatenz**
Die Sozialisationslatenz l_{soc} zwischen zwei Personen p_1 und p_2 ist:

$$\begin{aligned} l_{soc} \quad = \quad & \min(\{ext_{p_1}(d) + chn(d) + int_{p_2}(d) | d \in D\}) + \\ & \min(\{ext_{p_2}(d) + chn(d) + int_{p_1}(d) | d \in D\}) \end{aligned}$$

Die Sozialisationslatenz ist also ungefähr das Doppelte der minimalen Zeit, in der Wissen von einer Person an eine andere Person weitergegeben werden kann. Zum Beispiel ist die Sozialisationslatenz bei einem Gespräch von Angesicht zu Angesicht die Zeit, die der Sender braucht, eine Nachricht an den Empfänger zu übertragen, plus die Zeit, die der Empfänger braucht, den Empfang der Nachricht zu bestätigen.

3.5.5 Zusammenfassung

In Abbildung 3.13 sind die gerade beschriebenen Eigenschaften von Informationsflüssen und ihre Zusammenhänge noch einmal zusammenfassend dargestellt.

Im Zentrum der Abbildung befindet sich Informationsfluss. Ein Informationsfluss hat einen oder mehr Quell- und einen oder mehr Zielinformationsspeicher. Informationsspeicher sind entweder Personen (Wissen) oder Dokumente (Daten). Ein Informationsfluss ist entweder ein elementarer Informationsfluss oder eine Informationsflussaktivität. Elementare Informationsflüsse haben genau einen Quell- und einen Zielinformationsspeicher und enthalten keine Zwischenspeicher. Informationsflussaktivitäten bestehen aus mehreren Informationsflüssen, die wiederum Informationsflussaktivitäten oder elementare Informationsflüsse sein können. Dies entspricht dem Kompositum-Entwurfsmuster (vgl. [Gamma 1995]) und ermöglicht hierarchische Modellierung von Informationsflüssen.

Die Quellinformationsspeicher einer Informationsflussaktivität ergeben sich aus den Quellinformationsspeichern der darin enthaltenen Informationsflüsse, die in keinem enthaltenen Informationsfluss Zielinformationsspeicher sind. Analog ergeben sich die Zielinformationsspeicher einer Informationsflussaktivität aus den Zielinformationsspeichern der darin enthaltenen Informationsspeicher, die in keinem enthaltenen Informationsfluss Quellinformationsspei-

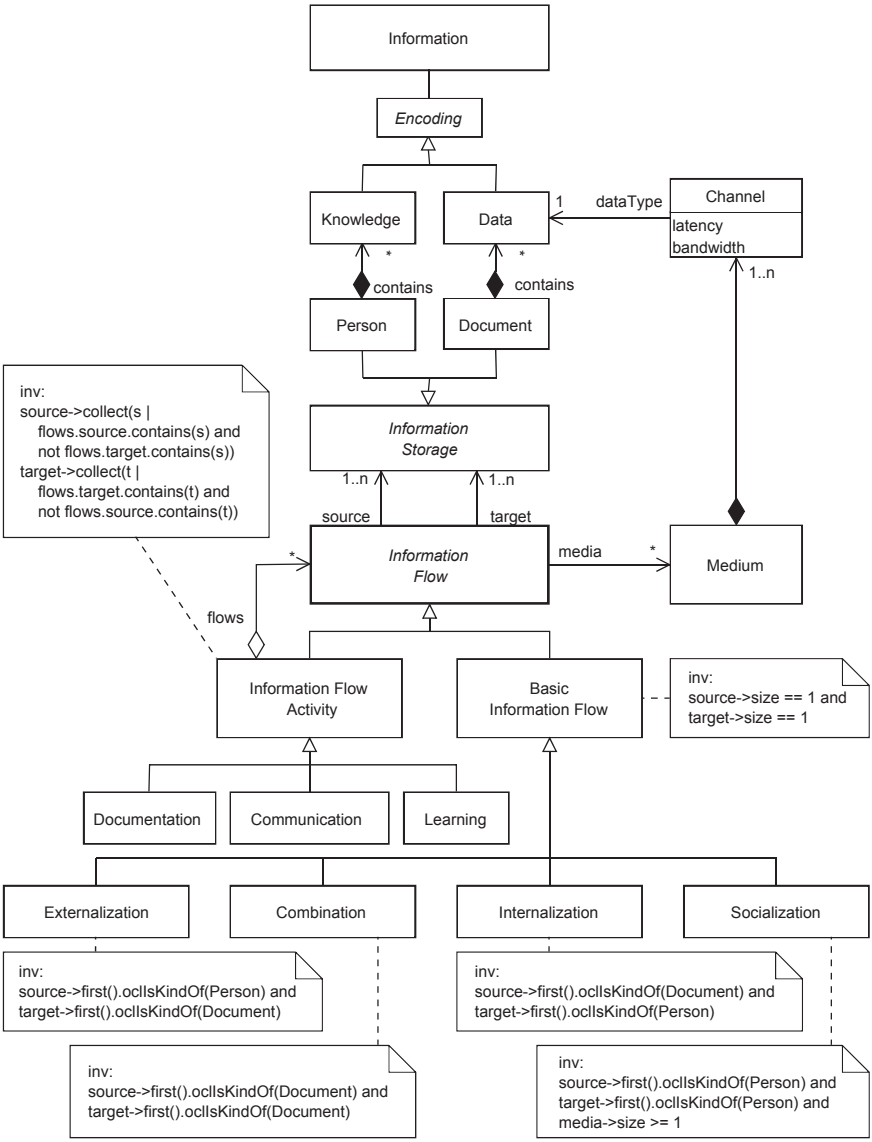


Abb. 3.13: Domänenmodell: Informationsfluss

cher sind. Ein Informationsfluss kann Medien für die Datenübertragung benötigen. Ein Medium besteht aus einem oder mehr Kanälen. Ein Kanal, der für den Transport eines bestimmten Datentyps ausgelegt ist, hat eine Kanallatenz und eine Kanalbandbreite.

Spezielle Informationsflussaktivitäten sind Dokumentation, Lernen und Kommunikation. Sozialisation ist ein elementarer Informationsfluss, der eine Person als Quell- und eine Person als Zielinformationsspeicher hat und der mindestens ein Übertragungsmedium benötigt. Externalisierung ist ein elementarer Informationsfluss, der eine Person als Quell- und ein Dokument als Zielinformationsspeicher hat. Internalisierung ist ein elementarer Informationsfluss, der ein Dokument als Quell- und eine Person als Zielinformationsspeicher hat. Kombination ist ein elementarer Informationsfluss, der ein Dokument als Quell- und ein Dokument als Zielinformationsspeicher hat.

3.6 Kommunikation

Viele Probleme der Softwareentwicklung lassen sich auf Kommunikationsprobleme zurückführen (vgl. z.B. [Curtis 1988, Kraut 1995, Boehm 2002, Silva 2010]). Daher wird Kommunikation hier als spezielle Informationsflussaktivität gesondert betrachtet. Nach Definition 3.24 ist Kommunikation ein Informationsfluss zwischen Personen. Mit Kommunikation kann Wissen zwischen Personen übertragen werden. Wissen kann nicht einfach direkt von dem Gedächtnis einer Person in das Gedächtnis einer anderen Person kopiert werden, es muss zunächst in Daten transformiert, über einen oder mehrere Kanäle zum Empfänger transportiert und schließlich vom Empfänger durch Interpretation der Daten wieder in Wissen überführt werden (vgl. Sozialisation in Kapitel 3.5.1 und Abb. 3.11).

Diese Sichtweise auf Kommunikation entspricht vom Prinzip her dem allgemeinen Kommunikationsmodell nach Shannon [Shannon 1948] (vgl. Kapitel 2.5.1): Es gibt einen Sender, der eine Nachricht über einen Kanal an einen Empfänger sendet. Im Unterschied zum allgemeinen Kommunikationsmodell wird hier aber gefordert, dass Quell- und Zielinformationsspeicher Menschen sein müssen. Zudem wird hier ein stärkeres Augenmerk auf die Schwierigkeiten der Transformation zwischen Wissen und Daten (und umgekehrt) und weniger auf die Transformation von Daten in Signale (und umgekehrt) für die Datenübertragung gelegt, da bei ersterem viel mehr Probleme in der Softwareentwicklung auftreten. Letzteres ist heute weitgehend gelöst oder durch hohe Bandbreiten der verfügbaren Datenübertragungsnetze unproblematisch.

In Abbildung 3.14 ist ein Kommunikationsmodell dargestellt, das das allgemeine Kommunikationsmodell von Shannon [Shannon 1948] erweitert. Es ist im Wesentlichen eine Abwandlung der in Abbildung 3.11 dargestellten Sozialisation. Im Vergleich zum Modell von Shannon rückt die Bedeutung des Menschen am Kommunikationsprozess in den Vordergrund. Die Bedeutung des technischen Anteils an der Kommunikation tritt zurück. Der Prozess der Kommunikation, d.h. die effektive Weitergabe von Wissen, geschieht in fünf Schritten:

1. **Kontext trennen:** Zunächst muss der Sender überlegen, *was* er *wie* als Nachricht senden möchte. Er muss den wesentlichen Inhalt identifizieren, von dem er annehmen kann, dass er ausreicht, um das zu kommunizierende Wissen beim Empfänger herzustellen. Ein Teil der Bedeu-

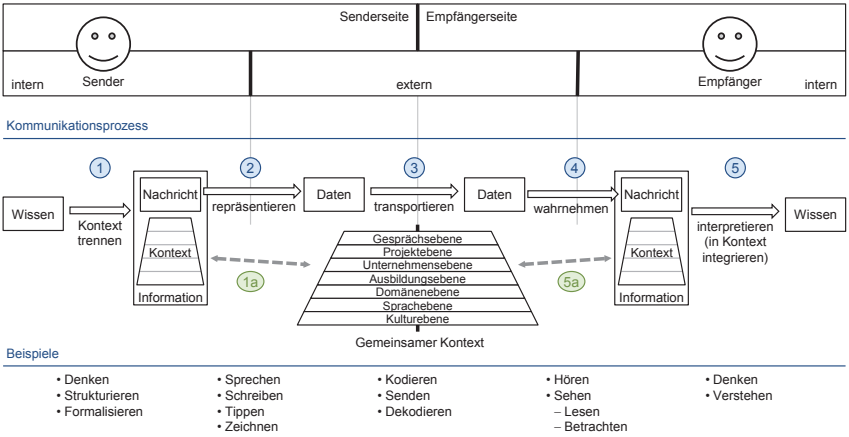


Abb. 3.14: Kommunikationsmodell

tung einer Nachricht wird nicht kommuniziert und wird als Kontextinformation beim Empfänger vorausgesetzt (1a). Sender und Empfänger müssen also eine gewisse gemeinsame Wissensbasis haben, um erfolgreich kommunizieren zu können (vgl. Common Ground [Clark 1993]). Je größer der gemeinsame Kontext von Sender und Empfänger ist, desto kleiner kann die Nachricht sein. Dem Sender fällt es dann leichter, d.h. es geht auch schneller, die zu sendende Nachricht zu identifizieren. Bei wenig gemeinsamen Kontext muss der Sender mehr Aufwand betreiben, um eine geeignete Nachricht zu finden. Die Nachricht muss mehr Informationen enthalten, die im ersten Fall als schon vorhanden vorausgesetzt werden. Eine umfangreichere Nachricht ist aufwendiger zu erstellen. Der gesamte Vorgang des Kontexttrennens ist rein kognitiv und passiert im Kopf des Senders (intern).

(2) **Repräsentieren:** Nachdem eine geeignete Nachricht identifiziert ist, muss sie vom Sender als Daten repräsentiert werden, damit sie zum Empfänger übertragen werden kann. Je nach Kommunikationskanal geschieht Repräsentation u.a. durch Sprechen, Schreiben, Tippen oder Zeichnen. Die Nachricht wird beim Repräsentieren aus dem Kopf des Senders in ein externes Medium überführt (intern → extern). Erst jetzt können sie zum Empfänger transportiert werden.

③ **Transportieren:** Die zu sendenden Daten werden vom Sender über einen Kanal zum Empfänger übertragen (Senderseite → Empfängerseite). Falls ein technischer Kommunikationskanal (d.h. mediierte Kommunikation) genutzt wird, müssen die Daten in eine für den Kanal geeignete Form transformiert werden, z.B. die Schallwellen gesprochener Sprache in elektromagnetische Signale für die Telefonleitung. Hier gelten alle Erkenntnisse aus der Kommunikationstheorie von Shannon [Shannon 1948]. Der zur Interpretation notwendige Kontext wird nicht übertragen. Er wird beim Empfänger vorausgesetzt.

④ **Wahrnehmen:** Die empfangenen Daten werden durch Aufnahme der Signale (Daten) über die Sinnesorgane des Empfängers zur empfangenen Nachricht (extern → intern). In der Softwareentwicklung spielen hauptsächlich die Sinne der auditiven und visuellen Wahrnehmung eine Rolle.

⑤ **Interpretieren:** Der letzte Schritt im Kommunikationsprozess ist die Interpretation der empfangenen Nachricht. Dazu muss der Empfänger unter Zuhilfenahme des geeigneten Kontextes die ursprüngliche Bedeutung wiederherstellen. Die Kommunikation ist nur erfolgreich, wenn Sender und Empfänger den gleichen Kontext benutzen, d.h. ein ausreichend großer gemeinsamer Kontext existiert (5a). Erst dann wird die Nachricht zu neuem Wissen. Interpretieren ist also immer die Verknüpfung von neuen Informationen (Nachricht) mit bekanntem Wissen (Kontext).

Die fünf Schritte sollen an einem Beispiel aus der Softwareentwicklung verdeutlicht werden. In einem Softwareprojekt, in dem ein Werkzeug zur Berechnung von Kreditkosten für einen Kunden aus der Bankenbranche entwickelt werden soll, halten die drei Entwickler A, B und C eine Telefonkonferenz, um sich über den aktuellen Stand im Projekt auszutauschen. Entwickler A möchte den anderen mitteilen, dass er in der vergangenen Woche eine neue Anforderung vom Kunden erhalten hat, die B und C nun implementieren sollen. Zusätzlich zu den anderen Werten auf der Übersicht der Kreditkosten sollen auch die effektiven Jahreszinsen berechnet werden. In der Schnittstellenbeschreibung der Berechnungsklasse hat er dafür bereits eine Methode vorgesehen, die B und C nun implementieren sollen.

① **Kontext trennen:** Damit B und C die Methode implementieren können, müssen sie wissen, wie der effektive Jahreszins berechnet wird. Dies möchte A nun B und C kommunizieren. Dazu muss A zunächst überlegen, wie gut sich B und C in der Terminologie des Kunden (Bankenbran-

che) auskennen. Er wählt folgende Nachricht: "Bei einer Kreditlaufzeit von n Jahren ist der effektive Jahreszins die n-te Wurzel aus dem Verhältnis von Kreditgesamtkosten und Darlehensbetrag.". Dabei geht er davon aus, dass B diese Nachricht direkt versteht, weil B schon oft an Projekten mit diesem Kunden beteiligt war. Da C erst neu in der Firma ist und das aktuelle Projekt C's erstes Projekt in der Bankenbranche ist, beschließt A zusätzlich noch einen Link auf eine Webseite zu schicken, die beschreibt, was Kreditgesamtkosten und Darlehensbetrag sind und wie sie berechnet werden.

② **Repräsentieren:** A spricht die Nachricht ins Telefon und tippt den Link der Webseite in das Chatfenster des Sofortnachrichtendienstes ICQ.

③ **Transportieren:** Die Schallwellen der gesprochenen Nachricht werden vom Telefonsystem in elektronische Signale umgewandelt und über die Telefonleitung an B und C gesendet. Dort werden sie wieder in Schallwellen zurückgewandelt und am Telefonhörer ausgegeben. Der Sofortnachrichtendienst schickt den Link über eine Internetverbindung an B und C. Dort wird der Verweis auf die Webseite als anklickbarer Link dargestellt.

④ **Wahrnehmen:** B und C hören die Nachricht aus dem Telefon. C liest den Link im Chatfenster, öffnet die Webseite und liest die Berechnungsformel.

⑤ **Interpretieren:** B interpretiert die Nachricht im Kontext seines Wissens der Bankenbranche aus früheren Projekten und versteht A damit korrekt. Das Wissen ist erfolgreich von A an B weitergegeben worden. C versucht die Nachricht zu interpretieren, hat aber kein Vorwissen im Bereich Kreditwesen und weiß daher nicht, was genau die Kreditgesamtkosten sind. Er versteht die Nachricht zunächst nicht, weiß also auch nicht, wie die Funktion zu implementieren wäre. Nun interpretiert C die Formel auf der Webseite und versteht nun basierend auf seinem Vorwissen im Bereich der Mathematik, wie die Kreditgesamtkosten berechnet werden. Nun hat er neues Wissen über die Kreditgesamtkosten erworben, welches er wiederum zur erfolgreichen Interpretation der gesprochenen Nachricht von A nutzen kann. C hat nun die Nachricht von A verstanden und in sein Wissen aufgenommen. Insgesamt war die Kommunikation von A über die Kanäle gesprochene natürliche Sprache und künstliche Sprache (math. Formel) mit B und C erfolgreich. Beide wissen nun, wie die Funktion implementiert werden kann.

Diese Sicht auf Kommunikation ist rein inhaltsbasiert. D.h. Metakommunikation (vgl. Kapitel 2.5.2) oder Kommunikation über zwischenmenschliche Beziehungen wird nicht gesondert betrachtet. Stattdessen kann sie als Spezialfall "normaler" Kommunikation gesehen werden. Zum Beispiel kann während eines persönlichen Gesprächs von Angesicht zu Angesicht ein fragender Gesichtsausdruck (Nachricht), der ein Missverständnis symbolisieren soll (Information), wie folgt abgebildet werden: ① Der Sender möchte dem Empfänger mitteilen, dass er eine vorher empfangene Nachricht nicht verstanden hat. Die zu übertragende Information soll also sein: "Ich habe nicht verstanden." Die vorherige Nachricht wird zum Kontext, der beim Empfänger bekannt sein muss, damit dieser weiß, was nicht verstanden wurde. ② Die Nachricht wird als fragender Gesichtsausdruck repräsentiert und direkt in Form von Licht (Datentyp bewegtes natürliches Bild) zum Empfänger transportiert (③). ④ Der Empfänger nimmt den Gesichtsausdruck wahr (optisch). ⑤ Zur Interpretation des empfangenen Gesichtsausdrucks benötigt er als Kontextinformation den bisherigen Gesprächsverlauf, d.h. die letzte Nachricht, und die Bedeutung bestimmter Gesichtsausdrücke. Letzteres ist typischerweise kulturell bedingt und wurde durch viele soziale Kontakte (frühere Kommunikation) erlernt. Der Empfänger weiß nun, dass seine vorangegangene Nachricht nicht verstanden wurde und kann entsprechend handeln.

Mit Hilfe des Kommunikationsmodells aus Abb. 3.14 und dem obigen inhaltlichen Beispiel können einige in der Softwareentwicklung häufig auftretende Kommunikationsprobleme erklärt werden:

Wissen wird nicht weitergegeben: Es wird zwar ein Kommunikationsversuch gestartet, aber es entsteht kein neues Wissen beim Empfänger. Dies kann auf Grund syntaktischer oder semantischer Kommunikationsprobleme geschehen.

syntaktisch: Auf syntaktischer Ebene kommen die Daten nicht oder nur unvollständig beim Empfänger an. Dem Empfänger ist bewusst, dass Daten fehlen. Auf Grund der Unvollständigkeit kann der Empfänger den Daten nicht die ursprüngliche Bedeutung zuweisen. Da dem Empfänger das Fehlen von Daten bewusst ist, kann er einen neuen Transportversuch anfordern. Im obigen Szenario könnte das z.B. durch eine schlechte Telefonverbindung passieren.

semantisch: Auf semantischer Ebene kommen die Daten zwar vollständig und unverändert beim Empfänger an, der Empfänger kann sie

aber auf Grund fehlender Kontextinformationen nicht interpretieren. Der Empfänger versteht die Nachricht nicht. Da ihm das auch bewusst ist, kann er den Sender um weiteren Erklärungen bitten. Im obigen Beispiel war das der Fall, als C versucht hat die Nachricht ohne Lektüre der Webseite zu interpretieren.

Missverständnisse: Die Kommunikation führt zu *falschem* neuen Wissen beim Empfänger, d.h. der Empfänger erhält anderes als das vom Sender intendierte Wissen. Missverständnisse entstehen, wenn der Empfänger einen anderen als den intendierten Kontext, d.h. den falschen Kontext, zur Interpretation der empfangenen Daten nutzt. Dem Empfänger ist dann nicht bewusst, dass er etwas falsch verstanden hat. Daher wird er nicht direkt nachfragen und das Missverständnis kann dem Projekt schaden, z.B. wenn eine Anforderung falsch verstanden wurde und das erst beim Kundenabnahmetest auffällt. Im obigen Beispiel hätte es zu einem Missverständnis kommen können, wenn A nicht die Webseite direkt mitgeschickt hätte und C auf Basis seines Allgemeinwissens die genaue Zusammensetzung der Kreditgesamtkosten "erraten" hätte.

Der gemeinsame Kontext spielt also eine sehr wichtige Rolle für den Erfolg von Kommunikation.

3.6.1 Gemeinsamer Kontext und Wissensunterschiede

Ein gemeinsamer Kontext, genauer gemeinsames Kontextwissen, ist Grundlage für erfolgreiche Kommunikation (vgl. Abb. 3.14). Kontextinformation kann während einer Kommunikation nur bis zu einem gewissen Grad übermittelt werden. Ein Teil des Kontextes muss also immer bereits beim Empfänger vorhanden sein, damit dieser die empfangene Nachricht korrekt interpretieren kann. Fehlt Kontextinformation, oder nimmt der Empfänger andere Kontextinformation zur Interpretation als der Sender vorgesehen hat, so kommt es zu Missverständnissen. Die Kommunikation ist dann nicht erfolgreich. Betrachtet man eine Kommunikation mit mehreren Teilnehmern, so müssen alle Teilnehmer eine gemeinsame Wissensbasis haben, damit eine Information erfolgreich zwischen allen Beteiligten kommuniziert werden kann. Gemeinsamer Kontext wird wie folgt definiert:

Definition 3.35: **Gemeinsamer Kontext**

Gemeinsamer Kontext $cc(P)$ ist die Schnittmenge des Wissens aller Kommunikationsteilnehmer $\{K_{p_1},\ldots,K_{p_n}\}$ einer Kommunikation. Man schreibt:

$$cc(P)=\bigcap_{p\epsilon P} K_p$$

Gemeinsamer Kontext schließt Wissen auf allen Ebenen ein: Wissen aus dem aktuellen Gesprächsverlauf, z.B. gewonnen durch Interpretation der letzten empfangenen Nachricht, Wissen aus dem letzten Projekt, bis hin zu impliziten Wissen, dass durch die Kultur bestimmt ist. Das Fehlen eines gemeinsamen Kontextes kann auch als Wissensunterschied zwischen den Kommunikationspartnern bezeichnet werden. Je größer der Wissensunterschied ist, desto wahrscheinlicher ist es, dass es bei der Kommunikation zu Problemen in Form von nicht weitergegebenen Wissen oder Missverständnissen kommt (siehe oben). Wissensunterschiede können je nach Größe verschiedenen Klassen zugeordnet werden, den so genannten Ebenen von Wissensunterschieden. Im Software Engineering sind folgende aufeinander aufbauende Ebenen von Wissensunterschieden relevant.

Kulturebene: Wissensunterschiede auf kultureller Ebene liegen dann vor, wenn Kommunikationsteilnehmer aus unterschiedlichen Kulturen stammen, sodass es selbst bei Kenntnis einer gemeinsamen Sprache zu Verständnisproblemen kommen kann. Ein "Ja" kann eigentlich ein "Nein" meinen. Die Bedeutung von Sprichwörtern wird meist nicht kulturübergreifend verstanden. Metaphern verlieren ihre kommunikative Kraft. Sogar Mimik und Gestik kann in unterschiedlichen Kulturen Unterschiedliches bedeuten. Ein SE-Beispiel für Wissensunterschiede auf Kulturebene ist ein Meeting von Projektpartnern aus Asien und Europa in einem global verteilten Softwareprojekt.

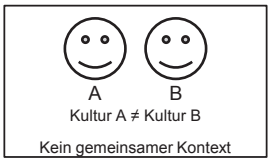


Abb. 3.15: Wissensunterschied: Kulturebene

Sprachebene: Wissensunterschiede auf sprachlicher Ebene liegen vor, wenn es keine gemeinsame natürliche Sprache gibt, auf die in der Kommunikation zurückgegriffen werden kann. Ein typisches Beispiel einer Kommunikationsaktivität im SE, bei denen Teilnehmer Wissensunterschiede auf Sprachebene haben, sind Meetings in global verteilten Projekten, bei denen englische Muttersprachler mit Entwicklern, die Englisch kaum bis gar nicht beherrschen, kommunizieren müssen. Auch wenn Muttersprachler mit Personen reden, die die verwendete Sprache nur als Fremdsprache sprechen, kann es auf Grund von großen Unterschieden im Wortschatz zu unerwarteten Missverständnissen kommen.

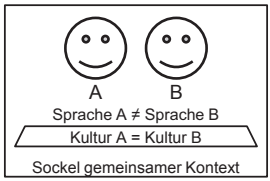


Abb. 3.16: Wissensunterschied: Sprachebene

Domänenebene: Wissensunterschiede auf Domänenebene liegen dann vor, wenn Kommunikationsteilnehmer Experten in verschiedenen Wissensgebieten sind, aber die selbe Sprache sprechen und aus dem selben Kulturkreis stammen. Beide Seiten haben tiefgründiges Wissen im eigenen Fachgebiet, aber meist nur oberflächliche Kenntnisse des jeweils anderen Gebiets. Es kann leicht zu Missverständnissen kommen, wenn Fachbegriffe auf einer Seite nicht bekannt sind, oder Abkürzungen unterschiedliche Bedeutungen in den verschiedenen Fachgebieten haben. Kommunikation ist hier schwierig, weil jede Domäne ihre eigene Fachsprache hat. Ein Kundengespräch ist ein typisches Beispiel für Kommunikation bei der die Teilnehmer aus unterschiedlichen Domänen stammen. Der Kunde ist Experte in seinem Bereich (die so genannte Anwendungsdomäne, vgl. Kapitel 3.4.2) und die Analysten sind Experten im Software Engineering. Die Wissensunterschiede auf Domänenebene sind der Hauptgrund für die großen Schwierigkeiten bei der Anforderungserhebung in der Softwareentwicklung (vgl. u.a. [Curtis 1988]).

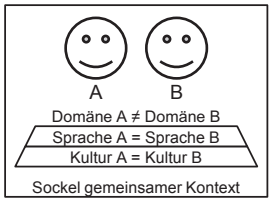


Abb. 3.17: Wissensunterschied: Domänenebene

Ausbildungsebene: Wissensunterschiede auf Ausbildungsebene liegen dann vor, wenn Kommunikationsteilnehmer zwar aus dem selben Fach sind, aber unterschiedliche Ausbildungsstände in diesem Fach haben (und dieselbe Kultur). Dies kann durch unterschiedliche Ausbildung oder unterschiedliche Berufserfahrung bedingt sein. Probleme können entstehen, wenn die jeweilige Fachsprache unterschiedlich tief verankert ist oder je nach Ausbildungshintergrund leichte Unterschiede aufweist. Beispiele aus dem SE sind Entwickler mit unterschiedlichen Abschlüssen (z.B. Fachinformatiker und Diplominformatiker) oder Entwickler mit unterschiedlicher Berufserfahrung (5 Jahre vs. Neueinsteiger). Beispiele für Begriffe aus der Domäne Software Engineering, die je nach Ausbildung unterschiedlich verstanden werden können, sind Schicht, Tier, Webservice, uvm.

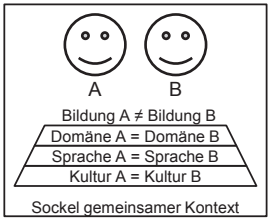


Abb. 3.18: Wissensunterschied: Ausbildungsebene

Unternehmensebene: Wissensunterschiede auf Unternehmensebene liegen dann vor, wenn Kommunikationsteilnehmer aus einem Kulturkreis, mit vergleichbarer Ausbildung und Berufserfahrung, aber aus unterschiedlichen Unternehmen stammen. Verschiedene Unternehmenskulturen können Einfluss auf das Verständnis haben, z.B. bei Verwendung von unternehmensinternen Abkürzungen. In der Softwareentwicklung können Wissensunterschiede auf Unternehmensebene immer dann zu Kommunikationsproblemen führen, wenn Software über Unternehmensgrenzen hinweg entwickelt wird.

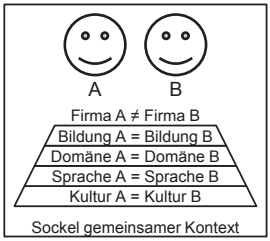


Abb. 3.19: Wissensunterschied: Unternehmensebene

Projektebene: Wissensunterschiede auf Projektebene liegen dann vor, wenn Kommunikationsteilnehmer Erfahrungen aus verschiedenen Projekten mitbringen (und ansonsten keine Unterschiede auf den vorherigen Ebenen haben). Das kann zu Problemen bei der Kommunikation führen, wenn zum Beispiel Erkenntnisse aus einem Projekt (z.B. eine bestimmte Technologie hat sich bewährt) unbewusst bei der Kommunikation in einem anderen Projekt als Kontextwissen vorausgesetzt wird.

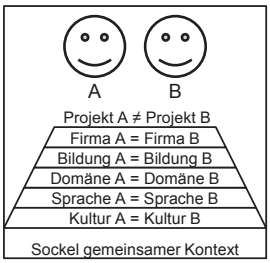


Abb. 3.20: Wissensunterschied: Projektebene

Gesprächsebene: Wissensunterschiede auf Gesprächsebene liegen dann vor, wenn Kommunikationsteilnehmer keine Unterschiede auf den vorherigen Ebenen haben, aber Teile des aktuellen Gesprächsverlaufs nicht kennen. Probleme können hier entstehen, z.B. wenn ein Entwickler zu spät zu einem projektinternen Meeting kommt und auf Grund der verpassten Informationen dem Meeting nicht mehr richtig folgen kann.

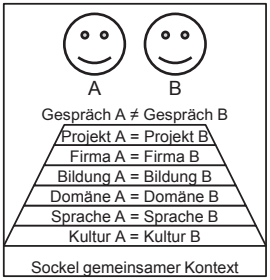


Abb. 3.21: Wissensunterschied: Gesprächsebene

Wenn zwei Kommunikationsteilnehmer aus dem selben Kulturkreis stammen, der gleichen Fachrichtung angehören, die gleiche Ausbildung und Berufserfahrung haben, im gleichen Unternehmen arbeiten, die gleichen Projekte kennen und den aktuellen Gesprächsverlauf verfolgt und verstanden haben, dann sind ihre Wissensunterschiede kleiner als auf Gesprächsebene. Dennoch kann es noch zu Missverständnissen kommen, weil ein Kommunikationsteilnehmer nie alles weiß, was die anderen Kommunikationsteilneh-

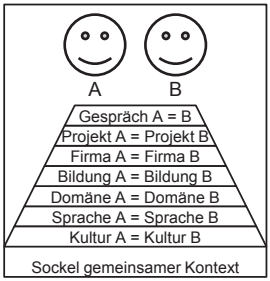


Abb. 3.22: kaum Wissensunterschiede

mer wissen. Die Grundannahme, dass es immer Wissensunterschiede zwischen zwei beliebigen Personen gibt, egal wie gut sie sich kennen, wird in folgendem Axiom festgehalten.

Axiom 3.3: **Es gibt immer Wissensunterschiede**
Das Wissen K_p zweier Personen ist immer unterschiedlich. Für beliebige Personen p_1 und p_2 mit $p_1 \neq p_2$ gilt:

$$K_{p_1} \nsubseteq K_{p_2}$$

Kommunikationsprobleme können nicht nur auf Grund von Wissensunterschieden entstehen, sondern auch, wenn die Teilnehmer auf unterschiedliche Ziele hinarbeiten.

3.6.2 Zielunterschiede

Eine weitere Quelle für Kommunikationsprobleme, die auch auftreten können, wenn die Kommunikationsteilnehmer kaum Wissensunterschiede aufweisen, sind die individuellen Zielunterschiede. Wenn die Teilnehmer unterschiedliche Ziele verfolgen, können sie den Erfolg einer Kommunikation trotz Verständnis des Gegenübers behindern. Typische Kommunikationssituationen, bei denen große Zielunterschiede den Erfolg behindern, sind Verhandlungen. In der Softwareentwicklung kann das z.B. bei Verhandlungen zwischen Entwicklern und Geldgebern (Management) vorkommen: Die Entwickler möchten zumeist ein funktionierendes qualitativ hochwertiges Produkt erstellen. Das Management hat oft das Ziel für die Produkterstellung möglichst wenig Geld zu investieren. Wenn diese beiden Parteien über Funktionskürzungen oder Budgeterhöhungen kommunizieren, kann es zu Problemen kommen.

In Anlehnung an die drei Ebenen der gegenseitigen Abhängigkeit von Straus und McGrath [Straus 1994], nämlich Kollaboration, Koordination und Konflikt, wird hier auch zwischen drei Ebenen von Zielunterschieden zwischen den Kommunikationsteilnehmern unterschieden.

Widersprüchliche Ziele (auch Konflikt): Widersprüchliche Ziele liegen vor, wenn die Erreichung des Ziels eines Kommunikationsteilnehmers dazu führt, dass nicht mehr alle Ziele der anderen Teilnehmer erreicht werden können. D.h. die Ziele der Kommunikationsteilnehmer stehen in Konflikt zueinander (vgl. [Cockburn 2007, S. 130, Fig 3-17]). Ein typisches Beispiel einer Kommunikationsaktivität im Software Engineering, bei der die Teilnehmer widersprüchliche Ziele haben, ist eine Verhandlung zwischen Entwicklern, die das Produkt funktionsvollständig und qualitativ hochwertig machen wollen, und dem Projektmanagement, das das Budget einhalten will.

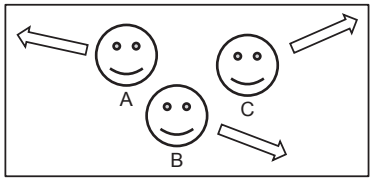


Abb. 3.23: Zielunterschied: Wiedersprüchlich

Koordinative Ziele: Koordinierte Ziele liegen vor, wenn das Erreichen des Ziels eines Kommunikationsteilnehmers dazu führt, dass die Ziele der anderen Teilnehmer zum größten Teil auch erreicht werden. D.h. die Ziele der Kommunikationsteilnehmer sind ähnlich (vgl. [Cockburn 2007, S. 130, Fig 3-18]). Ein Beispiel für Kommunikation mit koordinierten Zielen aus dem Software Engineering ist die Anforderungserhebung, bei dem Kunde und Entwickler ähnliche Ziele verfolgen. Beide möchten am Ende eine funktionierende Software, aber der Kunde möchte dafür möglichst wenig Geld ausgeben, wohingegen die Entwickler möglichst viel Geld damit verdienen wollen. Auch wenn in der Anforderungserhebung meist nicht über Budgetfragen geredet wird, spielen diese Überlegungen indirekt eine Rolle, sodass Kunde und Entwickler bezogen auf die Anforderungen zwar ähnliche, aber nicht die gleichen Ziele verfolgen.

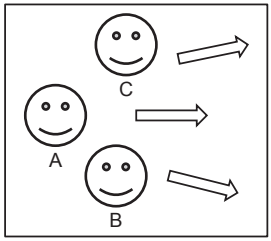


Abb. 3.24: Zielunterschied: Koordinativ

Keine Zielunterschiede (auch Kollaboration):
Keine Zielunterschiede liegen vor, wenn alle Teilnehmer das gleiche Ziel haben, sie also echt kollaborieren. Eine Zusammenarbeit hat hier echte Vorteile für alle Beteiligten, da jeder Teilschritt in Richtung des gemeinsamen Ziels alle gleichermaßen voran bringt. Ein Beispiel kollaborativer Kommunikation im SE ist die Zusammenarbeit beim Pair Programming.

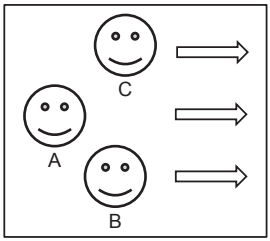


Abb. 3.25: Zielunterschied: Kollaborativ

Nicht nur die Kommunikationsteilnehmer verfolgen ihre individuellen Ziele, sondern die Kommunikation selbst verfolgt aus Sicht einer höheren Instanz ein Ziel (z.B. aus Sicht des Softwareprojekts).

3.6.3 Kommunikationsziel

Nach den Definitionen 3.17 und 3.24 spricht man nur dann von Kommunikation, wenn Wissen von einem Sender in das Wissen eines oder mehrerer Empfänger erfolgreich überführt wird. Daraus lässt sich das grundlegende Ziel von Kommunikation herleiten: die erfolgreiche Wissensweitergabe von Sender zu Empfänger. Aus Softwareentwicklungssicht sind aber noch weitere Ziele relevant, damit das oberste Ziel der Entwicklung von Software, das Lösen eines Problems eines Kunden mit Hilfe von Software, erreicht werden kann.

Üblicherweise gibt es für das Lösen eines Problems mehrere Möglichkeiten, d.h. auch unterschiedliches Wissen, was zur Lösung desselben Problems genutzt werden kann. Daher ist ein weiteres Kommunikationsziel die Wahl einer Lösungsmöglichkeit (= Wissen) aus einer gegebenen Menge von Lösungen. Dieses Ziel setzt das erste Ziel voraus. D.h. alle Kommunikationsteilnehmer müssen zunächst die zur Wahl stehenden Lösungsmöglichkeiten kennen und verstanden haben, bevor sie sich gezielt für eine entscheiden können. Das dritte Kommunikationsziel, neben der Wissensweitergabe und der Wissensauswahl, ist die Erzeugung neuen Wissens während der Kommunikation. Die Kreativität von Gruppen [Mednick 1962, Kurtzberg 2005] kann dazu führen,

dass nach der Kommunikation mehr Informationen vorhanden sind, als die Summe aller Informationen, die vor der Kommunikation vorhanden waren. Es wird also Information (in Form von Wissen) neu erzeugt.

Zusammengefasst wird in der Informationsflusstheorie zwischen den folgenden drei Softwareentwicklungs-relevanten Kommunikationszielen unterschieden, wobei das zweite und dritte Ziel die Erfüllung des ersten voraussetzen:

Informieren: Das grundlegendste Ziel von Kommunikation ist das der Informationsübermittlung, d.h. Information von einem Sender an einen oder mehrere Empfänger zu übermitteln. Das Wissen der Empfänger wird durch die neue Information erweitert. Während einer Kommunikationsaktivität (z.B. einer Konversation) können die Rollen des Senders und Empfängers mehrmals wechseln. Ein typisches Beispiel für eine Kommunikationsaktivität mit dem Ziel des Informierens in der Softwareentwicklungsdomäne ist ein Kundengespräch, bei dem der Kunde die Entwickler über seine Vision informiert.

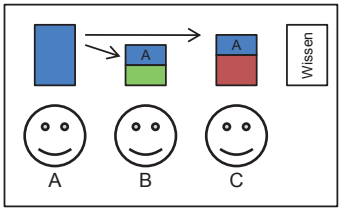


Abb. 3.26: Ziel: Informieren

Entscheiden: Ein weiteres Kommunikationsziel ist Entscheidungsfindung. Das Ergebnis einer Kommunikationsaktivität dieser Kategorie ist eine Entscheidung, d.h. eine Auswahl aus einer gegebenen Menge von Lösungsmöglichkeiten in Form von Wissen. Kommunikation mit dem Ziel der Entscheidungsfindung schließt informierende Kommunikation ein, da alle Beteiligten die gleiche Informationsbasis haben müssen, um eine informierte Entscheidung treffen zu können. Die Verhandlung der Umsetzungsprioritäten von Anforderungen mit dem Kunden ist ein typisches SE-Beispiel der Entscheidungsfindung.

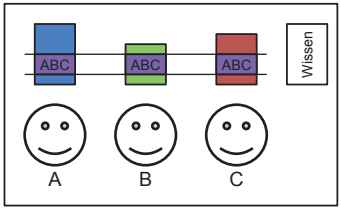


Abb. 3.27: Ziel: Entscheiden

Kreativ sein: Das dritte Kommunikationsziel ist das der Kreativität, d.h. die Schaffung neuen Wissens durch den Kommunikationsprozess. Auch kreative Kommunikation schließt informierende Kommunikation ein. Die so übermittelten Informationen sind Inspiration für den kreativen Prozess. Ein typisches Beispiel für eine kreative Kommunikationsaktivität im Software Engineering ist ein Architekturmeeting, bei dem eine Softwarearchitektur erstellt werden soll, die den Anforderungen genügt.

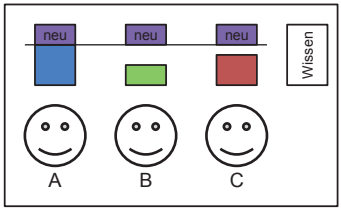


Abb. 3.28: Ziel: Kreativ sein

Mit Hilfe dieser für die Softwareentwicklung wichtigen Kommunikationsziele kann nun Kommunikationserfolg definiert werden.

3.6.4 Kommunikationserfolg

Eine Voraussetzung für erfolgreiche Kommunikation ist, dass die Empfänger das mit Hilfe der Nachricht übertragene Wissen des Senders erhalten. Dazu müssen sie die Nachricht korrekt verstehen, d.h. sie müssen sie in einem geeigneten Kontext interpretieren. Grundlage für den Kommunikationserfolg ist also, dass Wissen weitergegeben wird und dass keine Missverständnisse auftreten. Unter Berücksichtigung der oben genannten Kommunikationsziele kann Erfolg allgemein als das Erreichen des Kommunikationsziels angesehen werden. Kommunikationseeffektivität wird wie folgt definiert:

Definition 3.36: **Kommunikationseffektivität**
Für einen Informationsfluss $i \xrightarrow{t} i'$ ist die Kommunikationseffektivität $\frac{|i' \cap i|}{|i|}$ der Zielerreichungsgrad der Kommunikation.

Kommunikationseffektivität beschreibt, in wie weit das Kommunikationsziel erreicht wird. Kommunikation ist nur dann effektiv, wenn das zu kommunizierende Wissen des Senders erfolgreich (d.h. möglichst vollständig und möglichst korrekt) zu Wissen der Empfänger geworden ist. Dies ist Voraussetzung für alle Kommunikationsziele, da sie immer auch auf informierender Kommunikation aufbauen.

Da Zeitüberschreitungen ein häufiges Problem in der Softwareentwicklung sind (vgl. Kapitel 1.1.1 und u.a. [Armour 2006, Armel 2009]) ist es sinnvoll als Maß für den Erfolg einer Kommunikation nicht nur die Erreichung des Ziels, sondern auch die Dauer bis zu dessen Erreichung einzubeziehen. Kommunikationseffizienz wird daher wie folgt definiert:

Definition 3.37: **Kommunikationseffizienz**
Für einen Informationsfluss $i \xrightarrow{t} i'$ ist die Kommunikationseffizienz die Geschwindigkeit $\frac{|i|}{t}$, mit der das Kommunikationsziel erreicht wird.

Kommunikationseffizienz beschreibt, wie schnell das Kommunikationsziel erreicht wird, bzw. wieviele Informationen pro Zeiteinheit effektiv übermittelt werden. Kommunikation ist umso effizienter, je schneller eine gegebene Menge Informationen effektiv übermittelt wird.

Auf Basis von Kommunikationseffektivität und Kommunikationseffizienz wird Kommunikationserfolg wie folgt definiert.

Definition 3.38: **Kommunikationserfolg**
Kommunikationserfolg ist das effiziente und effektive Erreichen des Kommunikationsziels.

Kommunikation ist also dann erfolgreich, wenn die Empfänger in möglichst kurzer Zeit möglichst viel verstehen.

3.6.5 Zusammenfassung

In Abbildung 3.29 sind die gerade beschriebenen Eigenschaften von Kommunikation und deren Zusammenhänge noch einmal in einem Domänenmodell zusammenfassend dargestellt.

Kommunikation ist eine spezielle Informationsflussaktivität. Alle Quell- und Zielinformationsspeicher, die nach Definition 3.24 Personen sind, sind die Teilnehmer der Kommunikation. Die Schnittmenge des Wissens aller Teilnehmer bildet den gemeinsamen Kontext, auf den die Kommunikation aufbauen kann.

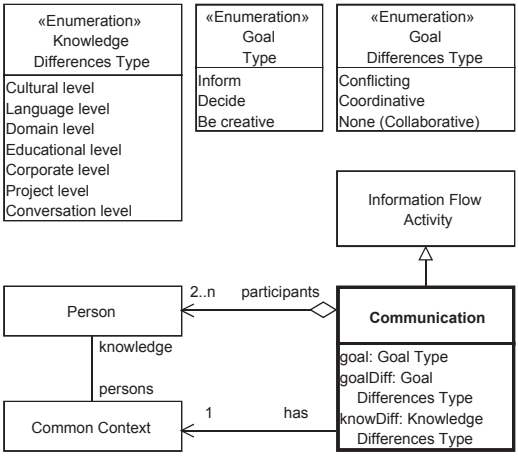


Abb. 3.29: Domänenmodell: Informationsflussaktivität Kommunikation

Je kleiner der gemeinsame Kontext ist, desto größer sind die Wissensunterschiede der Teilnehmer. Wissensunterschiede werden nach ihrer Größe in sieben Ebenen eingeteilt: (1) Kultur-, (2) Sprach-, (3) Domänen-, (4) Ausbildungs-, (5) Unternehmens-, (6) Projekt- und (7) Gesprächsebene. Weiterhin kann eine Kommunikation nach den Zielunterschieden ihrer Teilnehmer klassifiziert werden (Konflikt, koordiniert, keine). Das Ziel einer Kommunikation ist entweder Informieren, Entscheiden oder die kreative Schaffung neuen Wissens.

3.7 Informationsfluss und Softwareentwicklung

Basierend auf den im vorangegangenen Teil dieses Kapitels vorgestellten Definitionen werden nun die Begriffe Software, Softwareentwicklung und Softwareentwicklungserfolg definiert.

In Anlehnung an die Überlegungen von Brooks [Brooks 1995, S. 5], und im Gegensatz zur Definition nach IEEE [IEEE 1990] (vgl. Kapitel 1.2.2), wird hier zwischen Software und Softwareprodukt unterschieden. Dafür gibt es im Wesentlichen zwei Gründe:

1. Zur Erstellung eines Softwareprodukts ist wesentlich mehr Aufwand zu betreiben, als zur Erstellung eines einfachen Computerprogramms (vgl. [Brooks 1995, S. 5]). Daher sind für die Erstellung eines Softwareprodukts auch mehr und andere Informationsflüsse erforderlich.

2. Der Begriff Software als das Gegenteil von Hardware hat intuitiv nichts mit Dokumenten wie Betriebsanleitungen zu tun, sondern ist lediglich ein immaterielles Konstrukt, welches Computerhardware steuert. Diese intuitive Bedeutung soll durch die hier gegebene Definition weitestgehend beibehalten werden (vgl. Kapitel 2.1.1 zur Explikation).

Auf Basis der in diesem Kapitel eingeführten Begriffe werden Software und Softwareprodukt wie folgt definiert:

Definition 3.39: **Software**

Software ist ein elektronisches Dokument, das auf einem Computer ausgeführt werden kann, um Probleme eines Nutzers zu lösen.

D.h. Software ist nur das ausführbare Computerprogramm (engl.: Executable). Alle weiteren wichtigen Informationen, wie notwendige Betriebsdaten und Anleitungen, sind Bestandteil des Softwareprodukts.

Definition 3.40: **Softwareprodukt**
Ein Softwareprodukt besteht aus folgenden Dokumenten:

- Software,
- falls für die Ausführung der Software notwendig, Betriebsdaten und,
- falls für die Nutzung der Software notwendig, eine Betriebsanleitung.

Der Begriff Softwareprodukt ist damit äquivalent mit dem Begriff Software nach IEEE [IEEE 1990].

Definition 3.41: **Softwareentwicklung**
Softwareentwicklung ist die Menge der für die Erstellung eines Softwareprodukts notwendigen Informationsflüsse.

Nach dieser Definition und Definition 3.23 ist Softwareentwicklung eine Informationsflussaktivität. Abbildung 3.30 stellt Softwareentwicklung als eine solche dar. Die Hauptaufgabe der Softwareentwicklung besteht darin, aus Informationen, die die Lösung eines Problems in der Sprache der Anwendungsdomäne beschreiben, eine Lösung des Problems mit Software, Betriebsdaten und Nutzerdokumentation (Abb. 3.30 Dokumente rechts), zu erstellen. Die zu Beginn der Softwareentwicklung notwendigen Informationen aus der Anwendungsdomäne können entweder als Wissen, z.B. eines Kunden, oder als bereits dokumentierte Anforderungen vorliegen (Abb. 3.30 Informationsspeicher links).

Die Erstellung einer Lösung in der Anwendungsdomäne aus einem Problem der Anwendungsdomäne ist nicht zwingend Teil der Softwareentwicklung und damit nicht charakteristische Eigenschaft der Softwareentwicklung. Zum Beispiel kann die prinzipielle Lösung für das Problem der Multiplikation in der Anwendungsdomäne Mathematik bereits bekannt sein. Die Softwareentwicklung beschäftigt sich dann nur mit der Umsetzung der prinzipiellen Lösung in eine Lösung mit Hilfe von Software. Mit Hilfe der so entstandenen Software kann dann das Multiplikationsproblem auf einem Computer gelöst werden.

Natürlich kann die Lösungsfindung für ein Problem der Anwendungsdomäne auch Teil eines Softwareprojekts sein. Abbildung 3.31 zeigt ein im Vergleich

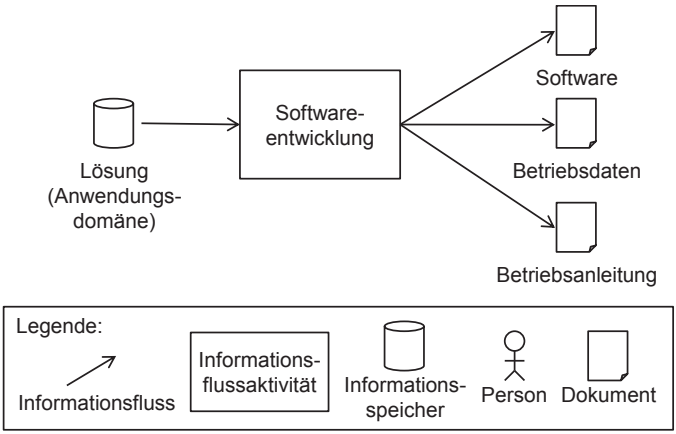


Abb. 3.30: Softwareentwicklung als Informationsflussaktivität

zu Abbildung 3.30 erweitertes Informationsflussmodell, welches Problemlösung und eine spätere Wartung des Softwareprodukts im Zusammenhang mit der eigentlichen Softwareentwicklung beschreibt. Für eine effektive und effiziente Wartung sollte im Softwareprojekt neben dem Softwareprodukt auch Entwicklerdokumentation erstellt werden. Dazu können die in der Softwareentwicklung entstandenen Zwischendokumente (siehe unten) genutzt werden.

In Abbildung 3.32 sind verschiedene Softwareentwicklungsvarianten mit unterschiedlich vielen beteiligten Informationsspeichern dargestellt. Nach Definition 3.41 ist ein minimales Softwareentwicklungsprojekt eine einzige Externalisierung (vgl. Abb. 3.32 ①). Das ist zum Beispiel der Fall, wenn sich ein Softwareentwickler für den eigenen Bedarf eine Software entwickelt (und kein Handbuch, und keine Betriebsdaten benötigt). Er kennt die eigenen Anforderungen und die Einsatzdomäne, muss daher also nicht kommunizieren. Sobald Software nicht für den eigenen Bedarf entwickelt werden soll, bzw. derjenige, der für die Lösung eines Problems eine Software benötigt, diese aber nicht selbst erstellen kann, ist Kommunikation zwischen Nutzer und Entwickler der Software notwendig (vgl. Abb. 3.32 ②). Die Kommunikation wird umso aufwendiger, je mehr Stakeholder in die Entwicklung einbezogen werden (vgl. Abb. 3.32 ③). Wenn das zu lösende Problem zu groß oder zu komplex ist, als dass es ein Entwickler alleine in akzeptabler Zeit lösen könnte, also mehrere

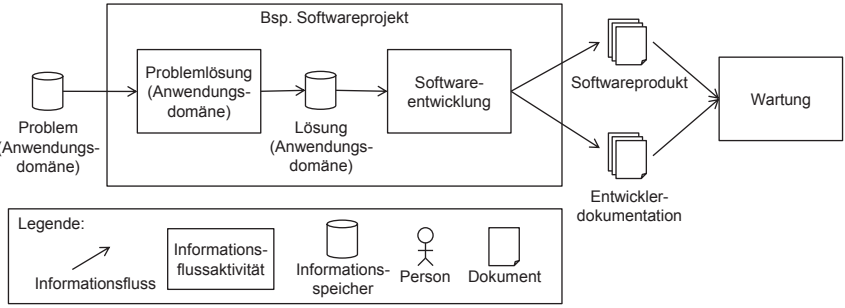


Abb. 3.31: Softwareprojekt mit Softwareentwicklung, Lösungsfindung und Dokumentation für Wartung

Entwickler benötigt werden, dann ist zur Abstimmung zwischen den Entwicklern zusätzliche Kommunikation notwendig (vgl. Abb. 3.32 ④). Weiterhin ist es bei komplexen Problemen notwendig, die Arbeit in Zwischenschritten zu erledigen, da man so die Arbeit besser zwischen den Entwicklern aufteilen und komplexe Aufgaben in für einen einzelnen Entwickler kognitiv erfassbare Einheiten zerlegen kann. Zwischenergebnisse sind dabei Dokumente (vgl. Abb. 3.32 ⑤, ⑥).

Erfolgreiche Softwareentwicklung ist unter Umständen von vielen Informationsspeichern abhängig. Wie bereits bei der Definition von Kommunikationserfolg motiviert, spielt die Zeit in der Softwareentwicklung eine wichtige Rolle (vgl. Kapitel 1.1.1). D.h. Informationsflüsse in der Softwareentwicklung sollten nicht nur effektiv, sondern auch effizient sein. Softwareentwicklungserfolg wird in Anlehnung an Kommunikationserfolg wie folgt definiert:

Definition 3.42: **Softwareentwicklungserfolg**

Softwareentwicklungserfolg ist die effiziente Erstellung eines effektiven Softwareprodukts.

Softwareentwicklungserfolg ist somit eine Kombination aus der Erstellung eines effektiven Softwareprodukts und Softwareentwicklungseffizienz. Eine effektive Software löst die Nutzerprobleme, für die sie geschaffen wurde. Anders ausgedrückt: Sie erfüllt die Anforderungen. Effiziente Softwareentwicklung

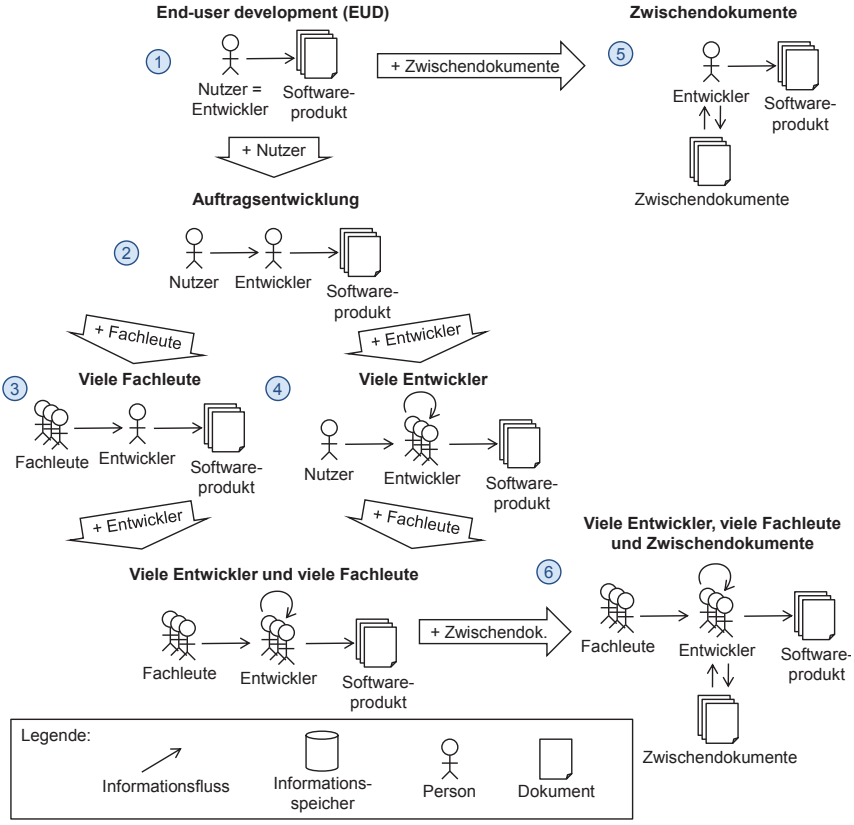


Abb. 3.32: Varianten der Softwareentwicklung mit unterschiedlicher Anzahl relevanter Informationsspeicher

ist die Erstellung eines Softwareprodukts mit möglichst wenig Aufwand, d.h. u.a. mit möglichst wenig monetären Kosten, mit möglichst wenig Personalaufwand oder in möglichst kurzer Zeit.

3.7.1 Softwareentwicklung als Problemlösen

Nach Definition 3.39 kann Software Probleme eines Nutzers lösen. Problem ist dabei bewusst sehr abstrakt gehalten, damit alle Arten von Software unter die Definition passen (von Verwaltungssoftware über Webanwendungen, Embedded-Software bis zu Computerspielen). Da das zu lösende Problem aber Einfluss auf die Softwareentwicklung, die benötigten Entwickler, die notwendige Dokumentation und Kommunikation und schließlich den Entwicklungserfolg hat, wird es hier noch etwas konkreter betrachtet.

Eine relevante Eigenschaft des Problems ist die Problemgröße. Sie beeinflusst Dauer und Kosten eines Softwareentwicklungsvorhabens. Die Problemgröße kann über die Problemkomplexität und den Problemgehalt charakterisiert werden. Diese beiden Eigenschaften haben Einfluss auf Parallelisierbarkeit und minimale Dauer der Softwareentwicklung (vgl. u.a. [Brooks 1995]). Problemgröße, Problemkomplexität und Problemgehalt werden wie folgt definiert.

Definition 3.43: **Problemgehalt**
Der Problemgehalt ist der Informationsgehalt der für die Lösung des Nutzerproblems notwendigen Informationen.

Auf Grund des rekursiven Charakters von Information kann der Problemgehalt nicht in absolute Zahlen gefasst werden. Es sind nur relative Vergleiche möglich, wenn die beiden zu vergleichenden Probleme auf einen gemeinsamen Kontext reduziert werden können (vgl. Kapitel 3.4.1).

Definition 3.44: **Problemkomplexität**
Die Problemkomplexität C_P ist die Anzahl der Abhängigkeiten $C_P(n)$, die bei der Zerlegung der Lösung des Problems in n Teile zwischen den Lösungsteilen entstehen.

Abhängigkeit zwischen zwei Lösungsteilen bedeutet, dass bei Änderung eines Lösungsteils das Problem nur noch dann gelöst werden kann, wenn auch der abhängige Lösungsteil entsprechend geändert wird.

Insgesamt ergibt sich die Definition der Problemgröße wie folgt:

> Definition 3.45: **Problemgröße**
> Die Problemgröße ist die Kombination von Problemgehalt und Problemkomplexität der für die Lösung des Nutzerproblems notwendigen Informationen.

Es ist zu beachten, dass mit Problemgröße die tatsächliche Problemgröße gemeint ist, auch wenn diese zu Beginn eines Projekts noch nicht bekannt ist. Erst nach einer ausreichenden Analyse des Problems (z.B. Anforderungserhebung) kann die tatsächliche Größe abgeschätzt und darauf basierend geplant werden. Da nur relative Vergleiche möglich sind, basieren diese Schätzungen immer auf Erfahrungswerten aus früheren ähnlichen Projekten.

3.7.2 Charakteristische Informationsflüsse der Softwareentwicklung

Softwareprojekte, bei denen die in der Einleitung beschriebenen Probleme auftreten (vgl. Kapitel 1.1.1), gehören meist der Kategorie ⑥ aus Abbildung 3.32 an: Projekte mit vielen Stakeholdern und einem komplexen zu lösenden Problem, welches viele Entwickler und viele Zwischendokumente erfordert (vgl. auch Abb. 3.33). Typische Softwareprojekte haben zusätzlich noch die Aufgabe, zunächst eine Lösung für ein gegebenes Problem in der Anwendungsdomäne zu erstellen oder Dokumentation für spätere Wartung der erstellten Software zur Verfügung zu stellen (vgl. Abb. 3.31). Zunächst werden hier die vier charakteristischen Arten von Informationsflussaktivitäten in der Softwareentwicklung selbst vorgestellt (vgl. Abb. 3.33):

① **Dokumentation des Softwareprodukts:** Dies ist die eigentliche Erstellung des Softwareprodukts, d.h. die Programmierung des Quellcodes zur Erstellung des Executables, die Erstellung der Betriebsdaten und die Erstellung der Benutzerdokumentation. Damit dies möglich ist, müssen alle relevanten Informationen (z.B. Anforderungen) bekannt sein. Mit

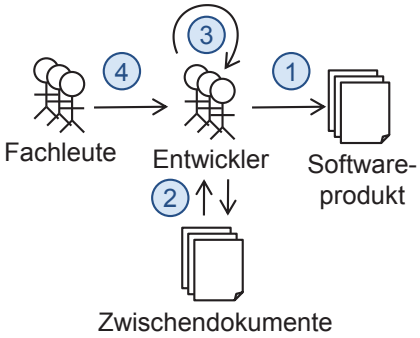


Abb. 3.33: Charakteristische Informationsflussaktivitäten der Softwareentwicklung

Dokumentation des Softwareprodukts ist hier die Informationsflussaktivität Dokumentation gemeint, also die Erstellung der Dokumente des Softwareprodukts (vgl. Def. 3.25). Es ist insbesondere *nicht* die Erstellung von Dokumenten gemeint, die die Software selbst beschreiben (mit Ausnahme der Betriebsanleitung, vgl. nächster Punkt). Das charakteristische des Informationsflusses "Dokumentation des Softwareprodukts" ist, dass das Ergebnis konkrete Informationen sind, mit deren Hilfe (Ausführen der Software) ein Nutzerproblem gelöst werden kann. D.h., bei dieser Informationsflussaktivität ist stets darauf zu achten, dass die erstellten Informationen auf Anwenderseite (Anwendungsdomäne) verstanden werden können. Die Software sollte lauffähig, fehlerfrei und korrekt (d.h. die wirklichen Anforderungen umsetzen), die Betriebsdaten vollständig und die Nutzerdokumentation vollständig und in der Sprache der Anwendungsdomäne formuliert sein. Für die Dokumentation des Softwareprodukts ist Softwaredomänenwissen notwendig, z.B. Wissen über Programmiersprachen, Werkzeuge, Frameworks, Softwareentwicklungsmethoden, Testen, Versionskontrolle etc. Bei der Erstellung der Software kombinieren die Entwickler durch einen kreativen Prozess Informationen der Anwendungsdomäne mit ihrem Wissen der Softwaredomäne (vgl. Def. 3.17), um so eine Lösung in Software zu schaffen. Beim Konkretisieren der Lösung müssen individuelle Entscheidungen getroffen werden, welche der möglichen Lösungsvarianten umgesetzt werden sollen.

② **Dokumentation von Zwischendokumenten:** Bei dieser Art von Informationsflüssen der Softwareentwicklung werden Zwischenergebnisse in Dokumenten festgehalten. Dies ermöglicht Arbeitsteilung und den Umgang mit komplexen Problemen. Das charakteristische dieser Informationsflüsse ist, dass die Informationen in den Zwischenergebnissen immer konkreter werden, d.h. der Lösungsraum wird in jedem Zwischenergebnis weiter eingeschränkt (vgl. u.a. [Pohl 2008, S. 20-22], Kapitel 2.6). Ein weiteres Merkmal ist, dass dabei die Sprache der Softwaredomäne genutzt werden kann. Für Dokumentation von Zwischendokumenten ist Softwaredomänenwissen (z.B: UML, UseCases etc.) und allgemeines Problemlösungswissen (z.B. die Fähigkeit zur Abstraktion und Modellbildung) wichtig. Dokumente, die ausschließlich für die Kommunikation zwischen Entwicklern erstellt werden, zählen nicht in diese Kategorie, sondern zur nächsten.

③ **Kommunikation zwischen Entwicklern:** Wenn die Arbeit auf mehrere Entwickler aufgeteilt wird, dann müssen diese sich abstimmen. Dazu ist Kommunikation zwischen den Entwicklern notwendig. Das charakteristische dieser Informationsflüsse ist, dass Entwickler immer die gemeinsame Sprache des Software Engineerings besitzen (unter der Annahme, dass sie SE gelernt haben), d.h. ihre Wissensunterschiede sind bei Kommunikation über Informationen der Softwaredomäne höchstens auf Ausbildungsebene. Kommunikation zwischen Entwicklern ist daher weniger fehleranfällig, als Kommunikation zwischen Fachleuten und Entwicklern. Für diesen Informationsfluss ist Kommunikations- und Projektmanagementwissen notwendig. Bei der Kommunikation zwischen Entwicklern kommt meist nicht nur informierende Kommunikation zur Abstimmung zum Einsatz, sondern auch entscheidende und kreative Kommunikation (vgl. Kapitel 3.6.3). Beim Konkretisieren der Lösung müssen projektweite Entscheidungen getroffen werden, welche der möglichen Lösungsvarianten umgesetzt werden sollen. Weiterhin können durch gemeinsame Arbeit mehrerer Entwickler an einem Problem die Vorteile kreativer Kommunikation genutzt werden (vgl. u.a. [Kurtzberg 2005]).

④ **Kommunikation mit Fachleuten:** In großen Softwareprojekten müssen nicht nur die Informationen eines Nutzers oder eines Kunden für die Erstellung der Software berücksichtigt werden, sondern Informationen vieler unterschiedlicher Fachleute eingeholt und ggf. miteinander abgestimmt werden. Das charakteristische dieser Informationsflüsse ist,

dass häufig Wissensunterschiede auf Domänenebene zwischen Fachleuten und Entwicklern vorliegen. Missverständnisse bei der Kommunikation werden dadurch wahrscheinlicher. Auch hier ist Kommunikations- und Projektmanagementwissen notwendig. Bei der Kommunikation mit Fachleuten ist wichtig, dass die Entwickler wirklich verstehen, was das Problem der Fachleute ist. Dazu ist meist ein nicht unerheblicher Teil Anwendungsdomäenwissen notwendig. Beides muss durch informierende Kommunikation durch die Entwickler erlernt werden.

Aus diesen vier charakteristischen Arten von Informationsflussaktivitäten in der Softwareentwicklung lassen sich typische Phasen der Softwareentwicklung ableiten:

1. **Implementierung** zur Erstellung der eigentlichen Software (Dokumentation der Software und Kommunikation zwischen Entwicklern, Abb. 3.33 ① & ③)

2. **Entwurf**, Architektur, Design zur Strukturierung der Lösung, um Komplexitätsbewältigung und Arbeitsteilung zu ermöglichen (Dokumentation der Zwischendokumente und Kommunikation zwischen Entwicklern, Abb. 3.33 ② & ③)

3. **Anforderungserhebung** zur Verständigung und Festlegung über das zu lösende Problem (Kommunikation mit Fachleuten und Dokumentation in Zwischendokumente, Abb. 3.33 ④ & ②)

4. **Validierung** zur Prüfung, ob die entwickelte Software auch das Problem der Anwendungsdomäne löst (Kommunikation mit Fachleuten und Dokumentation in Zwischendokumente, Abb. 3.33 ④ & ②)

Diese vier Phasen lassen sich in den meisten Softwareentwicklungsprozessen direkt oder indirekt finden, z.B.:

Wasserfall: Kodierung, Analyse und Design, Softwareanforderungen, Testen [Royce 1970]

V-Modell XT: SW-Einheit realisieren, SW-Architektur erstellen, Anforderungen festlegen, Systemelement prüfen [VModell 2009]

Rational Unified Process: Implementation, Analyse & Design, Anforderungen, Testen [RUP 2003]

Extreme Programming: Kodierung, Design, Zuhören (dem Kunden), Testen [Beck 2000, Chp. 9]

3.7.3 Abstraktheitsübergänge von Informationen in der Softwareentwicklung

Aus Informationsflusssicht ist eine wesentliche Eigenschaft der Softwareentwicklung – unabhängig davon, wieviele Entwickler an der Software arbeiten – dass Informationen auf ihrem Weg in das Produkt Software immer konkreter werden (vgl. Punkte ① und ② oben und in Abb. 3.33). Ziel der Softwareentwicklung ist es, eine konkrete Lösung in Form von Software zu schaffen (vgl. Def. 3.39 und 3.41). Daher sollte jede Softwareentwicklungsaktivität (d.h. jeder Informationsfluss) das Ziel haben, Informationen zu konkretisieren. Konkretisieren heißt Information hinzuzufügen (vgl. Kapitel 2.6). Dies kann entweder durch Kombination mehrerer Quellen geschehen, oder durch Denken eines Menschen. Der Mensch hat die Fähigkeit Wissen zu neuem Wissen zu kombinieren. Denken ist kein Informationsfluss, weil kein Speicherwechsel stattfindet. Grundlage für die Denkprozesse ist aber das Ausgangswissen, welches für den Denkprozess vorhanden sein muss. Das Ausgangswissen erhält der Mensch wiederum durch Lernen oder durch Kommunikation.

Abbildung 3.34 zeigt die wesentlichen vier Informationsverarbeitungsrichtungen in der Softwareentwicklung: ① → ② allgemeine Problemlösung, ② → ③ Erarbeitung einer Software-gestützten Lösung, ③ → ④ Lösungsumsetzung und ④ → ① Lösungsprüfung. Wie oben erläutert, muss die allgemeine Lösung eines Problems in der Anwendungsdomäne (① → ②) nicht unbedingt Teil des Softwareentwicklungsprojekts sein. Dennoch muss spätestens, wenn geprüft wird, ob die Software das geforderte Problem löst, das zu lösende Problem bekannt sein (④ → ①). Nach Abbildung 3.34 gestalten sich die Domänen-Abstraktheitsübergänge in der Softwareentwicklung wie folgt:

① → ②**:** Ausgehend von einem konkreten Problem in der Anwendungsdomäne (①) wird eine Lösung in der Anwendungsdomäne erarbeitet, die zumindest theoretisch ohne Software auskommt (②). Diese allgemeine Problemlösung ist typischerweise geeignet mehr als das konkrete Ausgangsproblem zu lösen und ist daher abstrakter (d.h. sie beschreibt einen größeren Lösungsraum).

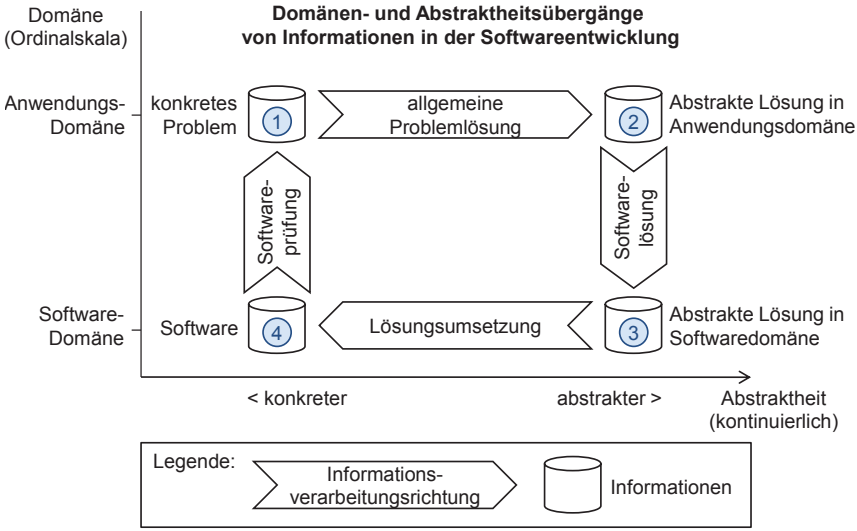


Abb. 3.34: Domäne und Abstraktheit von Information, die während der Softwareentwicklung eine wichtige Rolle spielen.

② → ③**:** Der nächste Schritt ist die Umwandlung der allgemeinen Lösung in eine Lösung mit Hilfe von Software (③), d.h. der Übergang von einer Lösungsbeschreibung in der Sprache der Anwendungsdomäne in eine Lösungsbeschreibung in der Sprache der Softwaredomäne. Dies geschieht üblicherweise während der Anforderungserhebung (vgl. u.a. [Bjoerner 2006, S. 10]). Alternativ kann, ausgehend von einem konkreten Problem in der Anwendungsdomäne, direkt eine Software-basierte Lösung erarbeitet werden (① → ③).

③ → ④**:** Ein weiterer Teil der Softwareentwicklung ist die Umsetzung des Lösungsvorschlags in Software. Dabei wird aus einer Menge von Umsetzungsmöglichkeiten eine ganz konkrete Lösung gewählt und umgesetzt (④). Häufig sind Zwischenschritte (z.B. Entwurf) auf dem Weg zur konkreten Lösung sinnvoll (nicht abgebildet), um die Lösungsumsetzung auf mehrere Entwickler verteilen zu können oder bei späteren Änderungen nicht wieder den ganzen Weg gehen zu müssen.

④ → ①**:** Abschließend sollte geprüft werden, ob die konkrete Lösung (d.h. die Software) das ursprüngliche Problem auch tatsächlich löst (④ → ①) und ob sie innerhalb des durch die abstrakte Lösung beschriebenen Lösungsraums liegt (④ → ②).

Zum besseren Verständnis der in Abbildung 3.34 dargestellten Domänen-Abstraktheits-Übergänge von Informationen in der Softwareentwicklung werden im Folgenden drei konkrete Beispiele angeführt.

Beispiel 1: Domänen-Abstraktheits-Übergänge am Beispiel von Informationen über funktionale Softwareeigenschaften: Das erste fiktive Beispiel stammt aus der Anwendungsdomäne Bankenbranche. Die SparBank möchte mehr Sparpläne verkaufen. Bei einem Sparplan wird einmalig Startkapital K_0 und dann monatlich eine Sparrate K_s eingezahlt. Das Kapital wird jährlich mit einem Zinssatz i verzinst. Nach Ende der Laufzeit n wird das Endkapital K_n, welches sich aus eingezahltem Kapital, Zins und Zinseszins ergibt, ausgezahlt. Durch einen neu zu entwickelnden Sparplanrechner für den Onlineauftritt der Bank sollen neue Kunden gewonnen werden. Die Do-mänen-Abstraktheits-Übergänge könnten in diesem Szenario wie folgt aussehen:

Allgemeine Problemlösung: ① Konkretes Anwenderproblem, beschrieben in der Sprache der Anwendungsdomäne, ohne Anwendung von Software: Zum Beispiel: Die Berechnung der monatlichen Sparrate K_s wird bisher manuell mit Papier und Taschenrechner durch Ausprobieren verschiedener Werte für K_s ermittelt → ② Abstrakte Lösung eines Anwenderproblems beschrieben in der Sprache der Anwendungsdomäne ohne Anwendung eines Computersystems. Zum Beispiel: Die monatliche Sparrate K_s kann wie folgt berechnet werden:

$$K_s = \frac{K_n - K_0 \cdot (1+i)^n}{12 \cdot (1+i)} \cdot \frac{i}{(1+i)^n - 1}$$

Softwarelösung: ② → ③ Abstrakte Lösung, beschrieben in der Sprache der Softwaredomäne; Beispiel in Pseudocode:

```
function K_s(K_n,K_0: Currency, n: Integer, i: Real):
    Currency
var
  savings, monthly: Currency
  progression: Real
begin
  progression = i/((1+i)^n - 1)
  savings = K_n - K_0(1+i)^n
  monthly = savings/(12*(1+i))
  K_s = monthly*progression
end
```

Lösungsumsetzung: ③ → ④ Konkrete Lösung, beschrieben in der Sprache der Softwaredomäne; Beispiel in Java:

```
Currency calcSavingsRate(Currency K_n, Currency K_0,
        int n, double i) {
  Currency K_s, savings,  tmpCur;
  double prog = i / (Math.pow(1+i, n) - 1);
  tmpCur = new Currency(K_0);
  tmpCur.mul(Math.pow(1+i, n));
  savings = new Currency(K_n);
  savings.sub(tmpCur);
  savings.div(12 * (1+i));
  K_s = new Currency(savings);
  K_s.mul(prog);
  return K_s;
}
```

Lösungsprüfung: ④ → ① Konkrete Lösung des Anwenderproblems durch Einsatz des Computersystems. Beispiel: Prüfen, ob die mit dem obigen Java-Algorithmus berechneten Werte mit den alten manuell erstellten Werten übereinstimmen.

Beispiel 2: Domänen-Abstraktheits-Übergänge am Beispiel von Prozessen, die durch Software unterstützt werden sollen: ① Der tatsächlich, manuell ausgeführte Prozess der Abschlussarbeitenverwaltung im Fach Informatik an der Leibniz Universität Hannover → ② natürlichsprachige Beschreibung

(in Sprache Anwendungsdomäne = universitäre Prüfungsverwaltung) eines zusammenhängenden Abschlussarbeitenprozesses → ③ Servicekomposition beschrieben mit Geschäftsprozessnotation (z.B. BPMN) → ④ Implementierung der Servicekomposition mit Programmiersprache (z.B. BPEL) → ① Prüfung, ob der implementierte Prozess mit dem tatsächlich durchgeführten Prozess übereinstimmt, z.B. indem die Abschlussarbeitenverwaltung tatsächlich mit der neuen Software durchgeführt wird.

Beispiel 3: Domänen-Abstraktheits-Übergänge am Beispiel qualitativer Softwareeigenschaften: Anwendungsdomäne Webshop: ① Die Kunden eines Webshops sollen so schnell durch die Produktbeschreibungsseiten navigieren können, dass sie nicht zur Konkurrenz wechseln, weil sie zu lange warten müssen. → ② Die Produktbeschreibungsseiten sollen in höchstens 1000 ms ausgeliefert werden. → ③ Der HTML Quellcode für die Produktbeschreibungsseiten soll auf dem Webserver in einem Cache vorgehalten werden. → ④ Nutzung eines Webservers mit Cache-Funktionalität, z.B. Apache Webserver → ① Prüfung, ob die Nutzung des Apache Webservers zu einer Auslieferungszeit der Produktbeschreibungsseiten von weniger als 1000 ms führt und ob dadurch weniger Kunden zur Konkurrenz abwandern.

3.7.4 Informationsspeicher der Softwareentwicklung

Aus den charakteristischen Informationsflüssen in der Softwareentwicklung (S. 138) ergeben sich Anforderungen an die dabei verwendeten Informationsspeicher, d.h. sowohl an die ausführenden Personen als auch an die verwendeten Dokumente. Im Folgenden wird auf diese Anforderungen in Form von Informationscharakteristika für die beteiligten Personen und Dokumente über typische Rollen und Dokumenttypen in der Softwareentwicklung eingegangen.

Rollen Aus den charakteristischen Informationsflussaktivitäten der Softwareentwicklung ergeben sich direkt zwei Rollenkategorien: Softwareentwickler und Fachleute (vgl. Abb. 3.33). Fachleute sind diejenigen, die ein Problem in ihrer Anwendungsdomäne haben, für das sie gerne eine Lösung mit Hilfe von Software hätten. Die Softwareentwickler können Problemlösungen in Form

von Software schaffen. Aus den in Abbildung 3.34 dargestellten Informationsverarbeitungsrichtungen ergeben sich drei spezielle Softwareentwicklungsrollen:

- *Anforderungsanalyst*, für den Übergang von Anwendungsdomäne zur Softwaredomäne,
- *Programmierer*, für die konkrete Umsetzung der Lösung als Software, und
- *Tester*, für die Prüfung der Lösung in Software gegen das ursprüngliche Problem.

Eine weitere Softwareentwicklungsrolle wird wichtig, wenn das zu lösende Problem auf mehrere Softwareentwickler aufgeteilt werden soll oder die Software später (längerfristig) gewartet werden muss: der *Architekt*. Ein Architekt muss die zu entwickelnde Software so strukturieren, dass einzelne Teile mit so wenig Abstimmungsaufwand wie möglich von verschiedenen Entwicklern bearbeitet werden können (vgl. [Parnas 1972]). Diese Struktur, sofern bekannt, hilft später auch bei der Wartung, schneller relevante Stellen im Quellcode finden zu können und durch Änderungen möglichst wenig neue Fehler einzubauen.

Die Fachleute lassen sich weiter in direkte Nutzer und indirekte Betroffene unterscheiden. Beide können wichtige Informationen aus der Anwendungsdomäne liefern. Bei der professionellen Softwareentwicklung nehmen der Kunde (d.h. der Geldgeber) und der Projektleiter noch eine wichtige Rolle ein. Der Kunde finanziert das Softwareentwicklungsvorhaben und gibt meist auch zeitliche Grenzen vor. Ein Projektleiter steuert im Wesentlichen den Informationsfluss innerhalb eines Softwareprojekts zwischen Entwicklern und Fachleuten.

Tabelle 3.7 gibt einen Überblick über typische in der Softwareentwicklung anzutreffende Rollen mit ihren Informationsflusscharakteristika.

Dokumente Auch die typischen Dokumente in der Softwareentwicklung ergeben sich größtenteils direkt aus den charakteristischen Informationsflussaktivitäten. Neben dem Softwareprodukt (Software, Betriebsdaten und -anleitung) werden meist auch Zwischendokumente erzeugt. Diese dienen zur Arbeitsaufteilung, Arbeitsabstimmung, Entscheidungsfindung, Kreativitätshilfe,

Tabelle 3.7: Typische Rollen in der Softwareentwicklung

Rolle	Informationsfluss-Aufgaben	Informations-Charakteristika
Software-entwickler	Ziel ist Dokumentation der Software (inkl. Arbeitsteilung und Zwischenschritte), Informationen müssen immer konkreter werden	Müssen Wissen aus SE-Domäne besitzen, müssen bei Arbeits-teilung zur Abstimmung untereinander Kommunikations-fähigkeiten aufweisen
Program-mierer	Finale Erstellung der Software	Programmierkenntnisse
Architekt	Strukturierung der Software, um Arbeitsteilung, Komplexitätsbewältigung und später Wartung zu ermöglichen bzw. zu vereinfachen	Abstraktionsfähigkeit, Modellbildung
Anforde-rungsana-lyst	Erfassung einer Problemlösung in Anwendungsdomäne und Überführung der Lösung in Softwaredomäne	Kenntnis der Anwendungsdomäne, Kommunikationsfähigkeiten (für Kommunikation mit Fachleuten)
Tester / Qualitäts-sicherung	Lösungsprüfung	Kenntnis der Anwendungsdomäne, Kommunikationsfähigkeiten (für Kommunikation mit Fachleuten)
Fachleute		
Nutzer	Liefert Problem und meist eine (theoretische) Lösung in Anwendungsdomäne	Hat Anwendungsdomänenwissen
Betroffene	Kann Informationen über (mögliche) Einflüsse, die durch Einsatz der neuen Software entstehen können, liefern	Hat Anwendungsdomänenwissen
Management / Sonstige		
Kunde	–	Liefert finanzielle und zeitliche Rahmenbedingungen, Geldgeber
Projektlei-tung	Steuert Informationsfluss	Muss Wissen aus Kommunikations- und Projektleitungsdomäne haben

kognitiven Entlastung oder Dokumentation von Zwischenergebnissen. Zudem können externalisierte Zwischenergebnisse leichter (durch den Autor und durch andere) geprüft werden. Dies sind nur einige mögliche Funktionen von Zwischendokumenten, die sich teilweise überschneiden. Ein Zwischendokument kann auch mehrere Funktionen in einem übernehmen. Entsprechend der sich aus Abbildung 3.34 ergebenden Aufgaben für die verschiedenen Softwareentwicklungsrollen ergeben sich folgende Zwischendokumententypen, die die Ergebnisse der Aufgaben festhalten können:

Anforderungsdokumente: Enthalten Informationen über das zu lösende Problem, wie das Problem in der Anwendungsdomäne (bisher) gelöst wird und wie es mit Hilfe von Software zukünftig gelöst werden können soll.

Entwurfsdokumente: Enthalten Informationen über die Struktur der Software. Die Struktur beschreibt möglichst unabhängige Teile (z.B. Komponenten) der Software und deren Abhängigkeiten (z.B. über Schnittstellenbeschreibungen).

Prüfdokumente: Enthalten Informationen über den Vergleich von Softwarelösung und Anwendungsproblem oder Anwendungslösung. Das schließt ein, was wie geprüft werden soll, welche Ergebnisse bei einer Prüfung erwartet werden und welche Ergebnisse erreicht wurden.

Auch die Projektleitung benötigt für ihre Arbeit Zwischendokumente, wie Projektpläne, Statusberichte, usw. Weitere wichtige Dokumente sind Dokumente, die als Ergebnis des Projekts über das Projektende hinaus wichtig sind. Das können z.B. Dokumente für die Wartung oder solche, die aus rechtlichen Gründen gefordert werden, sein (vgl. Tabelle 3.8).

3.7.5 Zusammenfassung

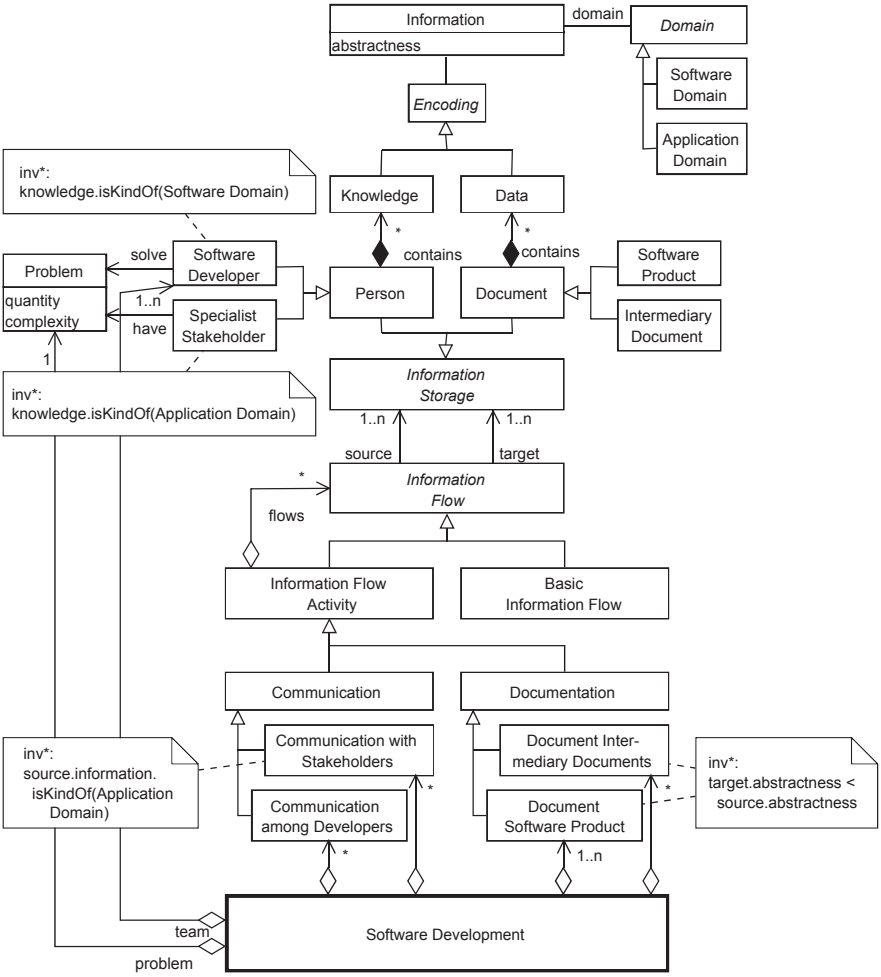
In Abbildung 3.35 (links und unten) sind die gerade beschriebenen Eigenschaften von Softwareentwicklung aus Informationsflusssicht und deren Zusammenhänge noch einmal in einem Domänenmodell zusammenfassend dargestellt.

Softwareentwicklung löst ein Problem eines Nutzerkreises mit Hilfe von Software. Ein Team von Softwareentwicklern führt dazu vier verschiedene Arten von Informationsflüssen aus:

Tabelle 3.8: Typische Dokumente in der Softwareentwicklung

Dokument-typ	Informations-Charakteristika	Anforderungen durch Informationsflussziel
Softwareprodukt		
Software	konkrete auf Computer ausführbare Problemlösung	Nutzer in Anwendungsdomäne muss Software verstehen und nutzen, d.h. sein Problem damit lösen, können
Betriebs-daten	Daten, die für Nutzung der Software benötigt werden	Muss von Software einlesbar sein. Kann für Nutzer in Anwendungs-domäne notwendig sein.
Betriebs-anleitung	Beschreibt konkret, wie die Software angewendet werden kann, um ein Problem zu lösen	Nutzer in Anwendungsdomäne muss damit in die Lage versetzt werden, die Software korrekt zu nutzen
Zwischendokumente		
Anfor-derungs-dokumente	Beschreibt abstrakte Problem-lösung in Softwaredomäne, kann auch abstrakte Lösungs-beschreibung aus Anwendungs-domäne enthalten	Softwareentwickler müssen verstehen, was die zu entwickelnde Software leisten können muss
Entwurfs-dokumente	Strukturierte und konkretisierte Lösungsbeschreibung in Softwaredomäne	Arbeit auf mehrere Software-entwickler aufteilbar machen, Art der Architektur bestimmt Kommunikationsbedarf für Abstimmung
Prüf-dokumente	Beschreibt Vergleich der kon-kreten Lösung mit konkretem Problem bzw. abstrakten Lösungsbeschreibung in Anwendungsdomäne	Softwareentwickler müssen Abweichungen vom Soll und Ursachen dafür erkennen können, um Gegenmaßnahmen ergreifen zu können.
Externe Dokumente		
siehe Zwischen-dokumente	entsprechend Zwischendokumenten	Andere Softwareentwickler, die Produkte warten oder weiterentwickeln müssen. Diese kennen meist den ursprünglichen Projektkontext nicht und brauchen daher ausführlichere Informationen

(1) Kommunikation mit Fachleuten, um das Problem zu verstehen, (2) abstimmende Kommunikation untereinander, (3) Dokumentation von Zwischenergebnissen und (4) Dokumentation des Softwareprodukts. Charakteristisch ist, dass Informationen bei der Dokumentation so lange immer konkreter werden, bis sie schließlich im Softwareprodukt das Nutzerproblem lösen können.



*Der Lesbarkeit halber kein standardkonformes OCL. Z.B. bedeutet "target.abstractness < source.abstractness", dass die in den Zielinformationsspeichern enthaltenen Informationen im Schnitt konkreter sind, als die der Quellinformationsspeicher.

Abb. 3.35: Domänenmodell: Softwareentwicklung

4 Theoreme der Informationsflusstheorie

4.1 Überblick

Nachdem die Grundlagen in Kapitel 2 vorgestellt und die Grundbegriffe der Theorie in Kapitel 3 erarbeitet wurden, werden hier Sätze und Hypothesen der Informationsflusstheorie vorgestellt, mit denen sich typische Phänomene der Softwareentwicklung erklären und vorhersagen lassen. Zusammen mit den Grundbegriffen bilden diese Theoreme die Informationsflusstheorie. In Kapitel 5 wird die Informationsflusstheorie dann einer Validitätsprüfung unterzogen, bevor sie abschließend in Kapitel 6 als Grundlage für eine praktische Softwareprozessverbesserungsmethode genutzt wird.

Nach Definition 3.41 ist Softwareentwicklung die Menge aller zur Erstellung eines Softwareprodukts notwendigen Informationsflüsse. Dabei besteht das Softwareprodukt aus drei Dokumenten: der Software selbst, den Betriebsdaten und der Nutzerdokumentation (vgl. Def. 3.39). Wie in Kapitel 3.7 beschrieben, sind die folgenden vier Informationsflussarten typisch für größere Softwareprojekte:

1. Softwareerstellung, d.h. Dokumentation der Software, ggf. der Betriebsdaten und der Betriebsanleitung
2. Komplexitätsbewältigung und Ermöglichung von Arbeitsteilung durch Dokumentation von Zwischenergebnissen
3. Abstimmung bei Arbeitsteilung durch Kommunikation zwischen Entwicklern
4. Lernen, wie das Problem gelöst werden kann und soll, durch Kommunikation mit Fachleuten

Im Zusammenspiel mit dem zu lösenden Problem und dem Softwareentwicklungsteam zeichnen diese Informationsflüsse die Softwareentwicklung aus (vgl. Abb. 3.35). Abbildung 4.1 gibt einen Überblick über die in diesem Kapitel betrachteten Theoreme. Die Theoreme beschreiben jeweils entweder einen aus den Grundbegriffen hergeleiteten oder einen hypothetischen Zusammenhang zwischen den verschiedenen Eigenschaften, die nach Informationsflusstheorie charakteristisch für die Softwareentwicklung sind (vgl. Kapitel 3.7). Die Abbildung zeigt nur hier diskutierte Zusammenhänge. Sie macht keine Aussagen über die Abwesenheit von Zusammenhängen.

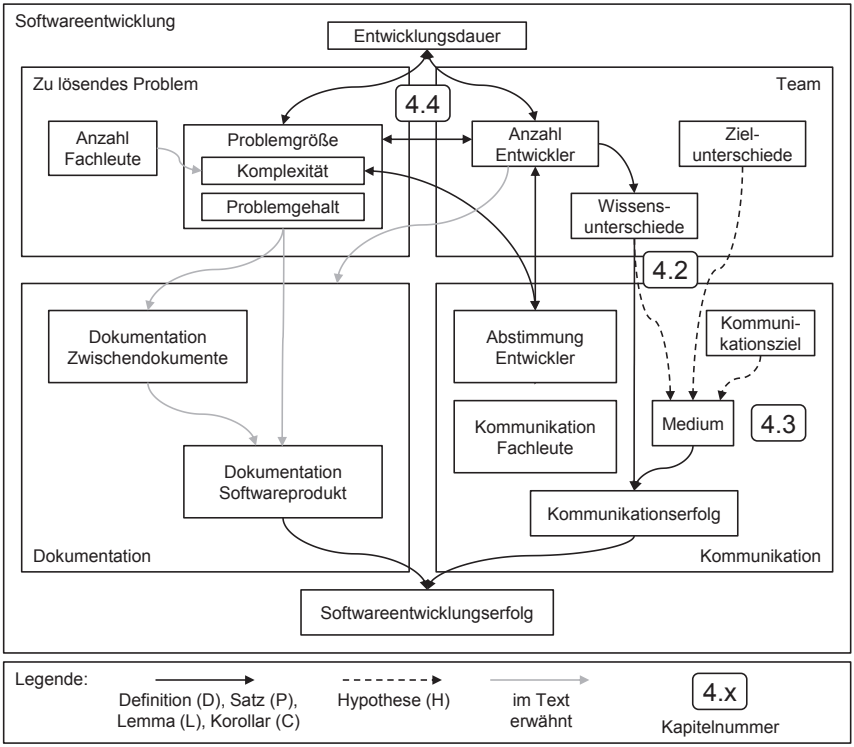


Abb. 4.1: Übersicht der Theoreme der Informationsflusstheorie anhand der Teil-Ganzes-Beziehungen (Verschachtelung in Abbildung) von Bestandteilen der Softwareentwicklung nach Abb. 3.35 und zugehörige Kapitelnummern

Es werden Theoreme über die Zusammenhänge zwischen dem zu lösenden Problem, dem Softwareentwicklungsteam, sowie den Informationsflüssen Dokumentation und Kommunikation betrachtet. Bei der Kommunikation wird besonders auf den Bereich der Medienwahl (vgl. Kapitel 4.3) eingegangen, da das Problem der Wahl eines geeigneten Kommunikationsmediums großen Einfluss auf den Erfolg der Kommunikation haben kann. Dokumentation wird hier nicht so ausführlich betrachtet, da die Software-Engineering-Forschung dort schon weiter ist, als im Bereich der Kommunikation. Das zeigt sich vor allem dadurch, dass Kommunikation als eine der Hauptursachen für die auch heute noch häufig anzutreffenden Probleme in der Softwareentwicklung gilt (vgl. Kapitel 1.1). Zusammenhänge im Bereich der Dokumentation werden daher nur nebenbei im Fließtext erwähnt, sofern dies sinnvoll für das Verständnis ist, aber nicht als Theoreme ausformuliert (vgl. graue Pfeile in Abb. 4.1).

4.2 Theoreme der Kommunikation

Wie in Kapitel 3.6 beschrieben, hängt der Erfolg von Kommunikation im Wesentlichen von drei Eigenschaften ab: dem Kommunikationsziel, dem gemeinsamen Kontext der Kommunikationsteilnehmer und den Zielunterschieden der Kommunikationsteilnehmer. Kommunikationserfolg ist als die effektive und effiziente Wissensweitergabe (im Falle der informierenden Kommunikation) an alle Kommunikationsteilnehmer definiert (vgl. Def. 3.38). In diesem Unterkapitel werden nun Theoreme über den Zusammenhang dieser Eigenschaften vorgestellt.

Zunächst werden einige Hilfssätze hergeleitet, die für die Herleitung der Sätze der Kommunikation und später der Medienwahl (vgl. Kapitel 4.3), benötigt werden. Mit den Hilfssätzen wird ein Zusammenhang zwischen dem elementaren Informationsfluss Sozialisation und der Informationsflussaktivität Kommunikation hergestellt. Die wesentlichen Unterschiede sind die potentiell größere Anzahl von Teilnehmern und die Möglichkeit, dass während einer Kommunikationsaktivität die Rollen von Sender und Empfänger mehrmals wechseln können.

Der erste Hilfssatz beschreibt den Zusammenhang zwischen Sozialisationsbandbreite und Kommunikationseffizienz.

Lemma 4.1: **Sozialisationsbandbreite und Kommunikationseffizienz**
Die Sozialisationsbandbreite korreliert positiv mit der Kommunikationseffizienz.

Herleitung. Annahme: Kombinationsbandbreiten, die bei der Kombination (vgl. Kapitel 3.5.1) evtl. vorhandener Zwischendokumente der Kommunikation (vgl. Def. 3.24) auftreten können, werden hier zur Vereinfachung als bereits in der Kanalbandbreite berücksichtigt angesehen. Mit dieser Annahme kann Kommunikation als eine Menge von Sozialisationen betrachtet werden.

Kommunikationseffizienz (Def. 3.37) beschreibt die Geschwindigkeit, mit der eine gegebene Menge Informationen effektiv kommuniziert werden kann. Bandbreite (Def. 3.30) beschreibt die Menge an Daten, die über einen Kanal pro Zeiteinheit übertragen werden kann. Nach Axiom 3.2 korreliert der Informationsgehalt positiv mit der Datenmenge, d.h. wenn Daten schneller übertragen

werden, dann werden auch mehr Informationen übertragen. Eine hohe Sozialisationsbandbreite führt daher zu einer größeren Kommunikationseffizienz und eine niedrige Sozialisationsbrandbreite führt zu einer niedrigeren Kommunikationseffizienz. □

Korollar 3.1 besagt, dass die Sozialisationsbandbreite durch das Minimum der Menge an Daten pro Zeiteinheit limitiert ist, die der Mensch externalisieren (Sender), internalisieren (Empfänger) und der Kanal übertragen kann. Aus Lemma 4.1 folgt daher auch, dass sowohl der Mensch als auch die Technik limitierender Faktor bei der Kommunikation sein kann. So kann z.B. eine Videokonferenz durch eine schlechte Verbindung (z.B. niedrige Kanalbandbreite) oder durch langsames Sprechen eines Teilnehmers (z.B. geringe Externalisierungsbandbreite, weil nicht die Muttersprache verwendet wird) verlangsamt werden.

Der zweite Hilfssatz beschreibt den Zusammenhang zwischen Sozialisationslatenz und Kommunikationseffizienz.

Lemma 4.2: **Sozialisationslatenz und Kommunikationseffizienz**
Die Sozialisationslatenz korreliert negativ mit der Kommunikationseffizienz.

Herleitung. Annahme: Zeiten, die durch die Kombination (vgl. Kapitel 3.5.1) evtl. vorhandener Zwischendokumente der Kommunikation (vgl. Def. 3.24) auftreten können, werden hier zur Vereinfachung als bereits in der Kanallatenz berücksichtigt angesehen. Mit dieser Annahme kann Kommunikation als eine Menge von Sozialisationen betrachtet werden.

Kommunikationseffizienz (Def. 3.37) beschreibt die Geschwindigkeit, mit der eine gegebene Menge Informationen effektiv kommuniziert werden kann. Latenz (Def. 3.31) ist die kürzeste Zeit, die ein Antwortdatum auf ein gesendetes Datum wieder beim Absender sein kann. Da Informationen nur in Form von Daten übertragen werden können (Def. 3.12 und Kapitel 3.5.3) und der Informationsgehalt positiv mit der Datenmenge korreliert (Axiom 3.2), hängen Sozialisationslatenz und Kommunikationseffizienz über ihre Zeitkomponente voneinander ab. Eine hohe Sozialisationslatenz führt zu einer niedrigeren Kommunikationseffizienz und eine niedrige Sozialisationslatenz führt zu einer höheren Kommunikationseffizienz. Sozialisationslatenz und Kommunikationseffizienz sind negativ miteinander korreliert. □

Korollar 3.2 besagt, dass die Sozialisationslatenz durch die Zeiten bestimmt ist, die der Mensch zur Externalisierung und Internalisierung sowie der Kanal für die Übertragung der Daten benötigt. Aus Lemma 4.2 folgt daher auch, dass sowohl der Mensch als auch die Technik Antwortgeschwindigkeiten bei der Kommunikation negativ beeinflussen können. So kann z.B. eine Telefonkonferenz durch eine schlechte Telefonverbindung (z.B. hohe Kanallatenz) oder durch verzögertes Antworten eines Teilnehmers (z.B. durch langes Überlegen) verlangsamt werden.

Mit Hilfe dieser Hilfssätze können nun die Theoreme der Kommunikation hergeleitet werden.

4.2.1 Gemeinsamer Kontext

Im Folgenden werden Theoreme über den Zusammenhang zwischen Anzahl und gemeinsamen Kontext (vgl. Kapitel 3.6.1) der Kommunikationsteilnehmer hergeleitet.

An einer Kommunikation sind nach Definition 3.24 mindestens zwei Personen beteiligt. Je mehr Personen an einer Kommunikationsaktivität beteiligt sind, desto höher ist die Wahrscheinlichkeit, dass Wissensunterschiede oder Zielunterschiede existieren (vgl. Kapitel 3.6.1 und 3.6.2). Allgemein gilt, dass der gemeinsame Kontext, auf den die Kommunikation aufbauen kann, immer kleiner als das Minimum des individuellen Wissens aller beteiligten Personen ist. Der folgende Satz beschreibt diesen Zusammenhang.

Satz 4.1: **Obere Grenze des gemeinsamen Kontexts**
Der gemeinsame Kontext cc aller Kommunikationsteilnehmer P ist kleiner als jedes individuelle Wissen K_p aller Teilnehmer.

$$\forall p \in P : cc(P) \subset K_p$$

Herleitung. Nach Definition 3.35 gilt $cc(P) = \bigcap_{p \in P} K_p$:

$$
\begin{array}{lrcll}
P_1 = \{p_1\}: & cc(P_1) & = & K_{p_1} & \\
\Rightarrow & cc(P_1) & \subseteq & K_{p_1} & \\
P_2 = \{p_1, p_2\}: & cc(P_2) & = & cc(P_1) \cap K_{p_2} & \subseteq cc(P_1) \\
 & & & cc(P_1) \cap K_{p_2} & \subset K_{p_2} \quad \text{Axiom 3.3} \\
 & & \vdots & & \\
P_n = \{p_1, \ldots, p_n\}: & cc(P_n) & = & cc(P_{n-1}) \cap K_{p_n} & \subseteq cc(P_{n-1}) \\
 & & & cc(P_{n-1}) \cap K_{p_n} & \subset K_{p_n} \quad \text{Axiom 3.3} \\
\Rightarrow \forall p \in P: & cc(P) & \subset & K_p &
\end{array}
$$

□

D.h., das kleinste Wissen limitiert den gemeinsamen Kontext. In der Realität ist der tatsächliche gemeinsame Kontext allerdings sehr viel kleiner, als das kleinste Wissen der Teilnehmer. Unter der Annahme, dass alle Kommunikationsteilnehmer erfolgreich an der Kommunikation teilnehmen sollen, folgt, dass je mehr Teilnehmer eine Kommunikationsaktivität hat, desto kleiner ist der gemeinsame Kontext bzw. desto größer sind die Wissensunterschiede.

Satz 4.2: **Gemeinsamer Kontext und Anzahl Kommunikationsteilnehmer**

Je mehr Teilnehmer P eine Kommunikation hat, desto kleiner ist der gemeinsame Kontext cc.

$$\text{für } P_n \subset P_m \text{ ist } cc(P_m) \subset cc(P_n)$$

Herleitung. Nach Definition 3.35 und $P_n \subset P_m$ gilt:

$$\begin{aligned} cc(P_m) &= \bigcap_{p \in P_m} K_p \\ &= \bigcap_{p \in P_m \cup P_n} K_p \\ &= K_{p_1} \cap \ldots \cap K_{p_n} \cap \ldots \cap K_{p_m} \\ \Rightarrow K_{p_1} \cap \ldots \cap K_{p_n} \cap \ldots \cap K_{p_m} &\subset K_{p_1} \cap \ldots \cap K_{p_n} \qquad \text{Axiom 3.3} \\ \bigcap_{p_m \in P_m} K_{p_m} &\subset \bigcap_{p_n \in P_n} K_{p_n} \\ cc(P_m) &\subset cc(P_n) \end{aligned}$$

□

Mit Hilfe dieser Sätze lassen sich z.B. die vielen Probleme großer Softwareentwicklungsprojekte begründen (vgl. Kapitel 1.1.3). In großen Projekten mit großen Teams müssen sich mehr Entwickler untereinander abstimmen als in den unproblematischeren kleinen Projekten. Nach Satz 4.2 haben große Teams nur einen kleinen gemeinsamen Kontext auf den die Kommunikation aufbauen kann. Kommunikation dauert daher länger und ist fehleranfälliger. Da Kommunikation einen wesentlichen Teil der Softwareentwicklung ausmacht (vgl. Kapitel 3.7.2 und 5.2) führt das auch zur Verlangsamung und Verteuerung von Softwareprojekten. Im folgnden Abschnitt wird besprochen, wie genau der Kommunikationserfolg vom gemeinsamen Kontext abhängt.

4.2.2 Kommunikationserfolg

In diesem Abschnitt werden die Theoreme über den Zusammenhang zwischen gemeinsamen Kontext der Kommunikationsteilnehmer und Kommunikationserfolg hergeleitet.

Satz 4.3: **Gemeinsamer Kontext und Kommunikationseffektivität**
Die Größe des gemeinsamen Kontextes korreliert positiv mit der Kommunikationseffektivität.

D.h. mit großem gemeinsamen Kontext kann effektiver, damit auch erfolgreicher, kommuniziert werden, als mit kleinem gemeinsamen Kontext.

Herleitung. Beim Kontexttrennen (vgl. Abb. 3.11, ①) muss der Sender beurteilen, welches Wissen beim Empfänger vorausgesetzt werden kann. Sei p_f die Wahrscheinlichkeit, dass der Sender dabei einen Fehler macht, d.h. dass der Kontext der gesendeten Information nicht Wissen des Empfängers ist. $p_e = 1 - p_f$ ist die Wahrscheinlichkeit, dass der Sender die Nachricht so wählt, dass der Kontext im Wissen des Empfängers vorhanden ist. Da bei einem großen gemeinsamen Kontext die Wahrscheinlichkeit größer ist, dass vorausgesetzter Kontext und vorhandener Kontext übereinstimmen, ist p_e um so größer, je größer der gemeinsame Kontext ist und um so kleiner, je kleiner der gemeinsame Kontext ist. Weiterhin gilt, je größer p_e desto größer ist die Kommunikationseffektivität und umgekehrt, weil Kontext Voraussetzung für Verständnis (vgl. Abb. 3.14 rechts), und Verständnis Voraussetzung für Kommunikationseffektivität (vgl. Kapitel 3.6.4) ist. □

Aus Satz 4.3 und der Definition von Wissensunterschieden in Kapitel 3.6.1 folgt folgendes Korollar:

Korollar 4.1: **Wissensunterschiede und Kommunikationseffektivität**
Die Kommunikation ist um so effektiver, je weniger Wissensunterschiede zwischen den Kommunikationsteilnehmern existieren.

Aus diesem Korollar folgt:

Korollar 4.2: **Bedarfs-Effektivitäts-Widerspruch der Kommunikation**
Angenommen, bei großen Wissensunterschieden besteht ein hoher Informationsbedarf und umgekehrt, dann gilt: In Situationen mit hohem Informationsbedarf kann nur ineffektiv kommuniziert werden. In Situationen mit niedrigem Informationsbedarf kann hingegen sehr effektiv kommuniziert werden.

D.h. in Situationen, bei denen ein großer Wissensunterschied besteht und daher ein hoher Informationsbedarf herrscht, ist die Wahrscheinlichkeit größer, dass es zu Unterbrechungen im Informationsfluss oder Missverständnissen (vgl. Kapitel 3.6) kommt. Ein typisches Beispiel dafür ist ein Kundengespräch zu Anfang eines Projekts. Zwischen Kunden und Entwicklern existieren üblicherweise Wissensunterschiede auf Domänenebene und es besteht ein hoher

Informationsbedarf. Die Entwickler müssen die Anwendungsdomäne lernen, um die Anforderungen des Kunden richtig verstehen zu können. Zudem müssen die Entwickler das Problem des Kunden bzw. die von ihm vorgeschlagene Lösung soweit verstehen, dass sie dafür eine Software entwickeln können. Wie von Korollar 4.2 vorhergesagt, kommt es bei Kundengesprächen aber häufig zu Missverständnissen (vgl. u.a. [Lutz 1992, Lubars 1993, Keil 1998]), d.h. die Kommunikation ist ineffektiv.

Ein ähnlicher Zusammenhang besteht auch zwischen gemeinsamen Kontext der Kommunikationsteilnehmer und Kommunikationseffizienz.

Satz 4.4: **Gemeinsamer Kontext und Kommunikationseffizienz**
Die Größe des gemeinsamen Kontextes korreliert positiv mit der Kommunikationseffizienz.

D.h. mit großem gemeinsamen Kontext kann effizienter kommuniziert werden als mit kleinem gemeinsamen Kontext.

Herleitung. Nach Def. 3.35 ist der gemeinsame Kontext das gemeinsame Wissen, auf das alle Kommunikationsteilnehmer ihre Nachrichten aufbauen können. In Kapitel 3.4 wurde beschrieben, dass je nach vorausgesetztem Kontext die zu sendende Nachricht unterschiedlich groß ausfallen kann (vgl. Abb. 3.6). Die Nachricht kann umso kleiner sein, je größer der gemeinsame Kontext ist. Unter der Annahme, dass die Zeit, die ein Mensch benötigt, um Nachricht und Kontext zu trennen (beim Senden) bzw. die Nachricht im Kontext zu interpretieren (beim Empfang), unabhängig von der Nachrichtengröße ist (vgl. Def. 3.32 und 3.33), kann ein Mensch eine kleine Nachricht schneller externalisieren und internalisieren als eine große Nachricht, weil eine kleine Nachricht schneller repräsentiert bzw. wahrgenommen (z.B. gelesen) werden kann als eine große Nachricht. Dies ergibt sich aus der Definition der Bandbreite (vgl. Def. 3.30). Zudem kann bei konstanter Kanalbandbreite eine kurze Nachricht auch schneller über einen Kanal transportiert werden, als eine lange Nachricht. Die Größe einer Nachricht steht also im direkten Zusammenhang mit der Sozialisationsbandbreite. Nach Korollar 3.1 bestimmen Kanal-, Externalisierungs- und Internalisierungsbandbreite die Sozialisationsbandbreite. Nach Lemma 4.1 korreliert die Sozialisationsbandbreite positiv mit der Kommunikationseffizienz. D.h. kleine Nachrichten, die schnell externalisiert, transportiert und wieder internalisiert werden können, führen zu effizienter

Kommunikation und große Nachrichten, die langsamer externalisiert werden können, führen zu weniger effizienter Kommunikation. □

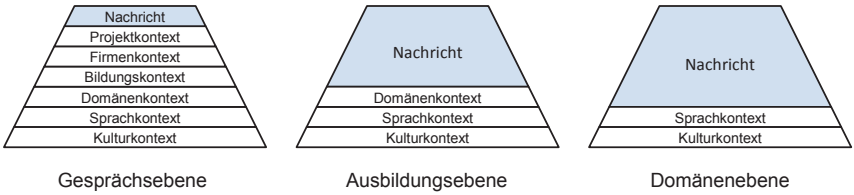


Abb. 4.2: Vergleich verschiedener Nachrichtengrößen nach Wissensunterschieden

So können z.B. Entwickler in einem gut eingearbeiteten Pair-Programming-Team effizienter kommunizieren, als Entwickler eines neu zusammengesetzten Teams, die vorher noch nie zusammengearbeitet haben, weil Pair-Programmierer durch die bisherige intensive Zusammenarbeit einen größeren gemeinsamen Kontext haben als das neue Team.

Ein weiterer Grund für den Einfluss der Größe des gemeinsamen Kontextes auf die Kommunikationseffizienz ist der erhöhte Kommunikationsbedarf, der entsteht, wenn der gemeinsame Kontext während einer Kommunikation bewusst vergrößert werden soll. Zum Beispiel helfen die Grundlagenvorstellungen in Präsentationen zu Abschlussarbeiten auf einen ausreichend großen gemeinsamen Kontext zu kommen, um den Rest des Vortrags erfolgreich kommunizieren zu können. Die Vorstellung der Grundlagen verlängert den Vortrag und wäre bei einem Publikum überflüssig, welches diese bereits beherrscht, sodass sie dort weggelassen werden könnten, was den Vortrag verkürzen würde. Abbildung 4.2 verdeutlicht den Einfluss des Wissensunterschieds auf die Nachrichtengröße für den Kontextangleich. Je nach Wissensunterschieden müssen für eine erfolgreiche Kommunikation unterschiedlich viele Informationen fließen. Z.B. müssen bei Wissensunterschieden auf Domänenebene (z.B. zwischen Kunden und Softwareentwicklern, vgl. Abb. 4.2 rechts) erst Informationen über die Domäne selbst ausgetauscht werden, bevor Projekt-relevante Informationen verstanden werden können. Für die Softwareentwicklung bedeutet das, dass die Entwickler zunächst die Anwendungsdomäne verstehen lernen müssen, bevor sie die Anforderungen richtig interpretieren können. Andererseits kann bei kleinen Wissensunterschieden die Nachricht sehr klein

ausfallen, was sehr effiziente Kommunikation ermöglicht (vgl. Abb. 4.2 links). Z.B. kann bei einem gut eingespielten Pair-Programming Paar ein Fingerzeig genügen, damit der andere weiß, was als Nächstes zu tun ist.

Aus Satz 4.4 und der Definition von Wissensunterschieden in Kapitel 3.6.1 folgt folgendes Korollar:

Korollar 4.3: **Wissensunterschiede und Kommunikationseffizienz**

Die Kommunikation ist um so effizienter, je weniger Wissensunterschiede zwischen den Kommunikationsteilnehmern existieren.

Aus diesem Korollar folgt:

Korollar 4.4: **Bedarfs-Effizienz-Widerspruch der Kommunikation**

Angenommen, bei großen Wissensunterschieden besteht ein hoher Informationsbedarf und umgekehrt, dann gilt: In Situationen mit hohem Informationsbedarf kann nur ineffizient kommuniziert werden. In Situationen mit niedrigem Informationsbedarf kann hingegen sehr effizient kommuniziert werden.

D.h. in Situationen, bei denen ein großer Wissensunterschied besteht, z.B. am Anfang eines Softwareprojekts zwischen Kunden und Entwicklern (Domänenebene), müssen viele Informationen zwischen den Kommunikationsteilnehmern ausgetauscht werden, damit die Softwareentwickler das zu lösende Problem verstehen und korrekt lösen können. In dieser Situation mit hohem Informationsbedarf kann aber nach Korollar 4.4 nur ineffizient kommuniziert werden, was dazu führen kann, dass die Anforderungserhebung ein langwieriger Prozess ist. Diese typische Situation der Softwareentwicklung wird in folgendem Korollar festgehalten. Es folgt direkt aus den Korollaren 4.1 und 4.3.

Korollar 4.5: **Domäneninterne versus Domänen-übergreifende Kommunikation**

Kommunikation mit Wissensunterschieden auf Domänenebene ist ineffektiver und ineffizienter als Kommunikation mit Wissensunterschieden, die kleiner als die Domänenebene sind.

D.h., Kommunikation zwischen Domänen ist schwieriger als Kommunikation innerhalb einer Domäne. In der Softwareentwicklung ist daher auch die Kommunikation mit Fachleuten (u.a. Kunden, Nutzer) schwieriger als die Kommunikation zwischen Entwicklern (vgl. Kapitel 3.7.2).

4.2.3 Zielinformationsspeicher

In diesem Abschnitt werden Theoreme über den Unterschied zwischen Kommunikation, bei der die Zielinformationsspeicher bekannt sind, und Kommunikation, bei der diese nicht bekannt sind, beschrieben. Bekannt bezieht sich hierbei auf das Wissen des Senders über das Vorwissen der Zielinformationsspeicher. Wenn der Sender die Zielpersonen und ihr Vorwissen kennt, dann kann er auch ungefähr den gemeinsamen Kontext abschätzen und diesen gezielt bei der Kommunikation berücksichtigen, d.h. er kann die Nachrichtengröße gezielt wählen. Beispiele für Kommunikationsaktivitäten, bei denen die Zielpersonen bekannt sind, sind Team-Meetings. Ein Beispiel für eine Kommunikationsaktivität, bei der die Zielpersonen üblicherweise nicht bekannt sind, ist Dokumentation der Software für die Wartung. Der Bekanntheitsgrad ist nicht binär. Ein Zielpersonenkreis kann mehr oder weniger bekannt sein. Z.B. kann ein Professor in einer Vorlesung das Vorwissen der Zuhörer auf Grund des Studienplans ungefähr abschätzen, sodass er in der Vorlesung auf einen größeren Kontext aufbauen kann, als er es z.B. in einem Lehrbuch über das selbe Thema machen könnte. Bei der Kommunikation mit seinen langjährigen Mitarbeitern kann er hingegen das Vorwissen noch präziser abschätzen als in der Vorlesung. Die folgenden Sätze stellen einen Zusammenhang zwischen dem Bekanntheitsgrad der Zielinformationsspeicher und Kommunikationserfolg her.

Satz 4.5: **Bekanntheitsgrad Zielspeicher und Kommunikationseffektivität**

Der Bekanntheitsgrad der Zielinformationsspeicher korreliert positiv mit der Kommunikationseffektivität.

D.h., je bekannter der Empfängerkreis bei der Kommunikation ist, desto gezielter kann der Sender auf deren Vorwissen aufbauen, sodass weniger Missverständnisse auftreten und seltener zu wenig Informationen übermittelt werden.

Herleitung. Der Bekanntheitsgrad der Zielinformationsspeicher beschreibt das Wissen des Senders über das Vorwissen der Empfänger. Wenn der Sender das Vorwissen der Empfänger kennt, kann er den gemeinsamen Kontext für die Kommunikation abschätzen. Wenn der Sender den gemeinsamen Kontext kennt, dann kann er darauf aufbauend gezielt Nachrichten formulieren (vgl. Kapitel 3.4). D.h., es ist unwahrscheinlicher, dass der Sender zu wenig Informationen schickt. Eine ausreichende Informationmenge erhöht die Wahrscheinlichkeit effektiver Kommunikation. Weiterhin vermindern gezielt formulierte Nachrichten die Wahrscheinlichkeit, dass Missverständnisse auftreten. Weniger Missverständnisse bedeuten nach Definition 3.36 eine höhere Kommunikationseffektivität. □

Satz 4.6: **Bekanntheitsgrad Zielspeicher und Kommunikationseffizienz**
Der Bekanntheitsgrad der Zielinformationsspeicher korreliert positiv mit der Kommunikationseffizienz.

D.h., je bekannter der Empfängerkreis bei der Kommunikation ist, desto gezielter kann der Sender auf deren Vorwissen aufbauen, sodass weniger überflüssige Nachrichten die Kommunikation unnötig verlangsamen.

Herleitung. Der Bekanntheitsgrad der Zielinformationsspeicher beschreibt das Wissen des Senders über das Vorwissen der Empfänger. Wenn der Sender das Vorwissen der Empfänger kennt, kann er den gemeinsamen Kontext für die Kommunikation abschätzen. Wenn der Sender den gemeinsamen Kontext kennt, dann kann er darauf aufbauend gezielt Nachrichten formulieren (vgl. Kapitel 3.4). D.h., es ist wahrscheinlicher, dass der Sender nicht zu viele Informationen schickt. Würde er zu viele Informationen schicken, würde das bei konstanter Bandbreite nach Definition 3.30 dazu führen, dass die Kommunikation länger dauert. Bei gleicher zu übermittelnder Informationsmenge ist nach Definition 3.37 schnelle Kommunikation effizienter als langsame Kommunikation. Insgesamt gilt dann, dass ein hoher Bekanntheitsgrad zu effizienter Kommunikation und ein niedriger Bekanntheitsgrad zu ineffizienter Kommunikation führt. □

Aus Satz 4.6 lässt sich auch folgern, dass es effizienter ist bei der Suche nach einer bestimmten Information jemanden zu fragen, der die Information hat und den eigenen Kontext kennt, als selbst danach zu suchen. Der Befragte

kann gezielt auf die Frage antworten. Das ist meist schneller, als Informationen selbst in einer großen Datenmenge zu suchen, weil der direkte Abruf aus dem menschlichen Gedächtnis sehr effizient ist.

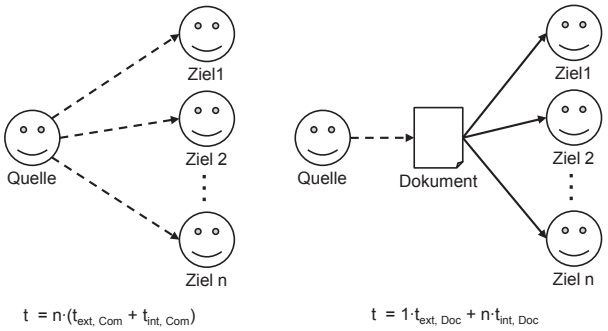


Abb. 4.3: Vergleich von direkter Kommunikation und Kommunikation über Dokumentation bei vielen Zielinformationsspeichern (in FLOW-Notation, vgl. Kapitel 6.1.3)

Insgesamt besteht bei unbekanntem Empfängerkreis die Gefahr, dass entweder zu wenig Informationen geschickt werden, was zu Missverständnissen führen kann, oder zu viele Informationen geschickt werden, was zu einer unnötigen Verlängerung der Kommunikationsaktivität führt.

Diese negativen Effekte treten insbesondere bei Kommunikation über Dokumentation auf (vgl. Abb. 4.3 rechts). Benutzt man Dokumente als Kommunikationsmittel, dann sind während der Erstellung des Dokuments meist nicht alle Leser, d.h. Zielinformationsspeicher, bekannt. Um verständlich für möglichst viele Leser zu sein, werden viele Kontextinformationen in das Dokument integriert. Das können z.B. Motivation, Grundlagen oder Glossare sein. Die vielen Zusatzinformationen führen dazu, dass die Dokumentation sehr zeitaufwendig ist. Dieser Aufwand lohnt sich erst ab einer gewissen Anzahl von Empfängern. Abbildung 4.3 zeigt eine Gegenüberstellung von Kommunikation mit Dokumentation und direkter Kommunikation ohne Zwischendokumente für einen großen Empfängerkreis. Ein weiterer Nachteil umfangreicher Dokumentation ist, dass die Empfänger relevante Informationen erst aufwendig suchen müssen (vgl. Satz 4.6). Kommunikation über Dokumente ist daher meist ineffektiver und ineffizienter als direkte Kommunikation ohne Zwischendokumente.

Insgesamt gilt: Aus Sicht des Senders lohnt sich Dokumentation erst ab einer gewissen Anzahl von Informationszielen. Wenn die Anzahl der Zielpersonen n größer als das Verhältnis aus Externalisierungszeit der Dokumentation $t_{\text{ext,Doc}}$ und Externalisierungszeit der direkten Kommunikation $t_{\text{ext,Com}}$, d.h. $n > \frac{t_{\text{ext,Doc}}}{t_{\text{ext,Com}}}$. Dabei ist allerdings zu beachten, dass nach Satz 4.2 die Externalisierungszeit der Dokumentation mit steigender Anzahl der Zielpersonen tendenziell länger wird, weil mehr Kontextinformationen dokumentiert weden müssen. Aus Sicht der Informationsziele ist nach den Sätzen 4.5 und 4.6 gezielte direkte Kommunikation immer besser. Letzteres setzt allerdings Personen voraus, die die gesuchte Information kennen und für eine Befragung verfügbar sind. Das ist nicht immer der Fall. Die Extreme-Programming-Praktik des On-Site-Customers [Beck 2000] nutzt z.B. diesen Vorteil von gezielter direkter Kommunikation.

4.3 Theoreme der Medienwahl

In der Softwareentwicklung sollte möglichst effektiv und effizient kommuniziert werden. Effektiv, weil fehlende Informationen und Missverständnisse zu einem Produkt führen, welches das Problem des Kunden nicht oder nicht vollständig löst. Effizient, weil Entwicklungszeit meist begrenzt (da teuer) ist. Die Wahl eines geeigneten Kommunikationsmediums hat großen Einfluss auf Kommunikationseffektivität und -effizienz und ist damit auch eine für die Softwareentwicklung relevante Problemstellung. Der folgende Satz bildet die Grundlage für die Theorie der Medienwahl.

Satz 4.7: **Kommunikationserfolg und Medienwahl**

Kommunikationserfolg hängt zu einem großen Teil von der Wahl eines geeigneten Kommunikationsmediums ab.

Herleitung. Nach Definition 3.38 ist Kommunikationserfolg effektive und effiziente Kommunikation. Daher wird zunächst der Einfluss der Medienwahl auf die Kommunikationseffizienz und anschließend der Einfluss auf die Kommunikationseffektivität gezeigt.

Zusammenhang von Medienwahl und Kommunikationseffizienz:

- Ein Medium besteht aus $1..n$ Kanälen (Definition 3.28).
- Ein Kanal hat eine Latenz t_l (Definition 3.31) und eine Bandbreite (Definition 3.30).
- Der verwendete Kommunikationskanal bestimmt den Datentyp, in dem die Daten repräsentiert werden müssen (Definition 3.27).
- Somit hat der Datentyp auch Einfluss auf Externalisierungs- und Internalisierungsbandbreiten (vgl. Tabelle 3.6), die der Mensch für Repräsentation und Wahrnehmung benötigt.
- D.h. das gewählte Medium hat Einfluss auf die Sozialisationsbandbreite (Korollar 3.1) und auf die Sozialisationslatenz (Korollar 3.2).
- Nach den Lemmata 4.1 und 4.2 haben Sozialisationsbandbreite und -latenz Einfluss auf die Kommunikationseffizienz.

Zusammenhang von Medienwahl und Kommunikationseffektivität:

- Die Sozialisationslatenz (Korollar 3.2) hat auch Einfluss auf die Geschwindigkeit, mit der Missverständnisse den Kommunikationspartnern mitgeteilt werden können. Falls es Wissensunterschiede gibt, die dazu führen, dass eine Nachricht nicht verstanden werden kann, und dieser Unterschied erkannt wird, bestimmt die Sozialisationslatenz, wie schnell das Problem dem Sender mitgeteilt werden und er entsprechend darauf reagieren kann.
- Missverständnisse durch Wissensunterschiede, d.h. fehlenden gemeinsamen Kontext, haben direkten Einfluss auf die Kommunikationseffektivität (Satz 4.3 und Definition 3.36).

□

Die Grundidee der hier vorgestellten Theoreme zur Medienwahl ist, Medien getrennt für Inhaltsübermittlung und Steuerung zu wählen. Steuerung ist der Teil der Kommunikation, der für die sozialen Interaktionen – insbesondere das Erkennen und Auflösen von Missverständnissen – verantwortlich ist. Falls die beiden unabhängig voneinander gewählten Medien nicht zufällig die gleichen sind, werden während der Kommunikation beide Medien gleichzeitig benutzt. In vielen Situationen in der Softwareentwicklung wird das – ohne es explizit so zu nennen – schon so gemacht. Zum Beispiel ist Pair Programming eine Kommunikationsaktivität, bei der das steuernde Medium von Angesicht zu Angesicht ist und das inhaltliche Medium der gemeinsam genutzte Computer, genauer der darauf dargestellte Quellcode. Die folgende Hypothese fasst das noch einmal zusammen.

Hypothese 4.1: **Kommunikationssubaktivitäten**

Die Wahl je eines geeigneten Kommunikationsmediums entsprechend der Anforderungen der Kommunikationssubaktivitäten

- *Inhaltsübertragung* und
- *Steuerung*

führt zu einem größeren Kommunikationserfolg, als die Wahl eines einzelnen Kommunikationsmediums basierend auf den Kommunikationsaktivität-übergreifenden Gesamtanforderungen.

Daher ist es sinnvoll Kommunikationsmedien für Kommunikationsaktivitäten getrennt für die folgenden Kommunikations*sub*aktivitäten zu wählen:

1. **Inhaltsübertragung:** Die Hauptaufgabe von Kommunikation ist Informationsübermittlung zwischen den Kommunikationsteilnehmern. Der Inhalt einer Nachricht muss vom Sender zum Empfänger übertragen werden. Auch in der Softwareentwicklung schließt das neben Projektinformation soziale Information (z.B. zum Kennenlernen, Vertrauen aufbauen) und Information zum Ausbau des gemeinsamen Kontexts (auch Common Ground [Clark 1993]) ein.

2. **Steuerung:** Steuerung ist Kommunikation über Verlauf und Zwischenerfolge einer Kommunikationsaktivität. Steuernde Kommunikation schließt Informationen darüber ein, ob eine Nachricht verstanden oder nicht verstanden wurde. Steuerung ist Meta-Kommunikation in Bezug auf Inhaltsübertragung. Durch aktive gezielte Steuerung kann die Informationsübermittlung effektiver und effizienter gestaltet werden. Z.B. kann durch gezielte Moderation Beteiligung und Sprecherwechsel der Teilnehmer verbessert werden. Die Moderation kann durch einen Kommunikationsteilnehmer oder einen dedizierten Moderator vorgenommen werden. Aber auch ohne Moderator gibt es viele Möglichkeiten zur Steuerung. Missverständnisse können durch Mimik, Gestik oder verbale Rückfragen verdeutlicht werden.

Beide Subaktivitäten folgen jeweils dem Kommunikationsmodell aus Abbildung 3.14, wobei die Rollen von Sender und Empfänger während einer Kommunikationsaktivität mehrmals wechseln können. Es wird zwischen diesen beiden Subaktivitäten unterschieden, weil sie unterschiedliche Anforderungen an die verwendeten Kommunikationskanäle haben. In vielen Situationen ist es sinnvoll, mehr als einen Kommunikationskanal zu nutzen, um die Subaktivitäten zur Inhaltsübermittlung und zur Steuerung optimal zu unterstützen. Die folgenden Fragen müssen geklärt werden, um geeignete Kommunikationsmedien wählen zu können.

1. Welche Anforderungen an die Kanäle bestehen für die Kommunikationssubaktivitäten?

2. Welche Faktoren haben Einfluss auf die Kanalanforderungen?

3. Wie ist der Zusammenhang zwischen Kanaleigenschaften und Kommunikationserfolg, jeweils für die Kommunikationssubaktivitäten Inhaltsübermittlung und Steuerung?

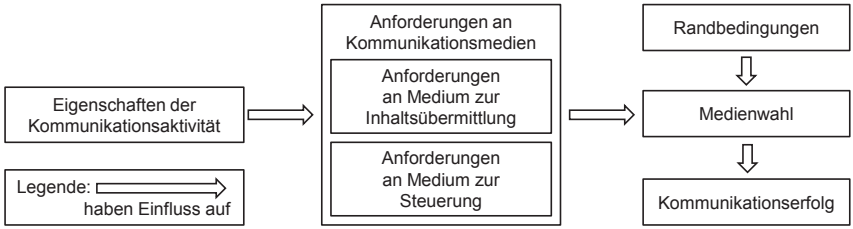


Abb. 4.4: Kette von Einflüssen bei der Medienwahl, Übersicht

Abbildung 4.4 zeigt einen Überblick der unterschiedlichen Einflüsse, die nach der hier vorgestellten Theorie bei der Medienwahl eine Rolle spielen. Die Kommunikationsaktivität, also der eigentliche Kommunikationsvorgang, hat bestimmte Charakteristika, die Einfluss auf die Anforderungen an die zu nutzenden Kommunikationsmedien haben. Die Anforderungen bestimmen, welche Medien am geeignetsten für die gegebene Kommunikationsaktivität sind. Randbedingungen, die Kommunikationsaktivität-übergreifend gelten, haben zusätzlich Einfluss auf die Medienwahl. So können z.B. Budgetgrenzen oder technische Einschränkungen dafür sorgen, dass bestimmte Medien nicht genutzt werden können. Welches Medium, bzw. welche Medien letztendlich zur Kommunikation genutzt werden und wie stark diese von den Anforderungen abweichen, hat Einfluss auf den Ablauf und den Erfolg der Kommunikation.

Die Wahl geeigneter Kommunikationsmedien für die Inhaltsübermittlung und für die Steuerung der Kommunikation hängt im Wesentlichen vom Ziel der Kommunikationsaktivität, den Unterschieden in den individuellen Zielen der Teilnehmer und den Wissensunterschieden der Teilnehmer ab, bzw. der Größe des gemeinsamen Kontexts auf den die Kommunikation aufbauen kann (vgl. Abb. 4.5 links).

In den folgenden Abschnitten wird gezielt auf die Eigenschaften der Kommunikationsaktivität (Abb. 4.5 links), ihr Einfluss auf die Medienwahl (Abb. 4.5 Mitte) und schließlich ihr Einfluss auf den Kommunikationserfolg (Abb. 4.5 rechts unten) eingegangen.

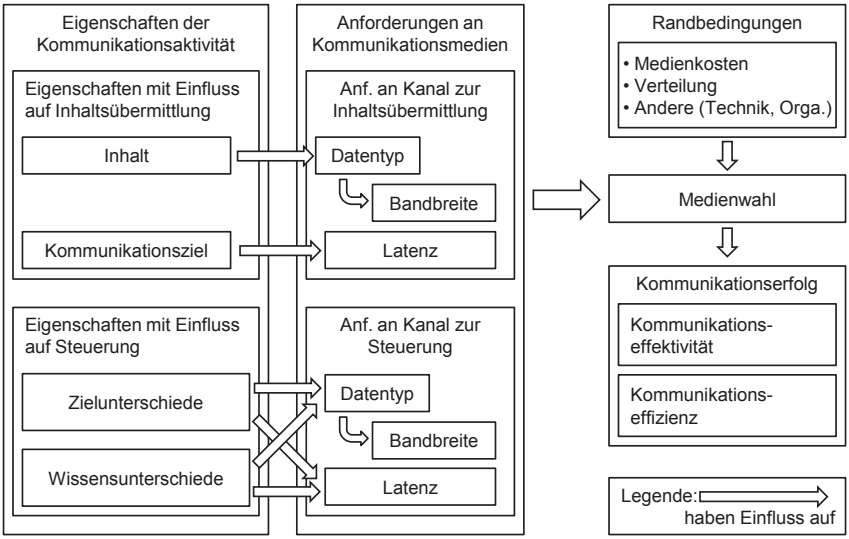


Abb. 4.5: Kette von Einflüssen bei der Medienwahl, Detailsicht

4.3.1 Einfluss des Kommunikationsinhalts

Nach Definition 3.8 ist ein Datentyp eine Klassifikation von Daten nach Dimension, Sinn und Ursprung. Zudem ist nach Definition 3.27 ein Kanal für einen bestimmten Datentyp ausgelegt. Weiterhin hat ein Kanal eine bestimmte Bandbreite. Der Datentyp hat Einfluss auf die Mindestbandbreite des Kommunikationskanals, falls diese die Kommunikationseffizienz nicht negativ beeinflussen soll. Der folgende Satz beschreibt diesen Einfluss.

Satz 4.8: **Datentyp und Mindestbandbreite**

Hat ein Kanal weniger Bandbreite als das Minimum aus Internalisierungs- und Externalisierungsbandbreiten aller Kommunikationsteilnehmer, also $b_{\text{chn}}(d) < \min(b_{\text{ext}}(d), b_{\text{int}}(d))$, so mindert das die Kommunikationseffizienz.

Herleitung. Nach Korollar 3.1 ist die Sozialisationsbandbreite:

$$b_{\text{soc}}(d) < \min(b_{\text{ext}}(d), b_{\text{chn}}(d), b_{\text{int}}(d))$$

Mit

$$b_{\text{chn}}(d) < \min(b_{\text{ext}}(d), b_{\text{int}}(d))$$

folgt

$$b_{\text{soc}}(d) < b_{\text{chn}}(d)$$

Lemma 4.1 besagt, dass die Sozialisationsbandbreite positiv mit der Kommunikationseffizienz korreliert, d.h. je größer die Bandbreite, desto größer ist die Kommunikationseffizienz. Umgekehrt gilt, dass eine zu geringe Bandbreite auch die Kommunikationseffizienz mindert. □

Aus Satz 4.8 folgt, dass die Kanalbandbreite, abhängig vom Datentyp, immer größer als die Internalisierungs- und Externalisierungsbandbreite gewählt werden sollte, um eine Kommunikation nicht unnötig zu verlangsamen.

$$b_{\text{chn}}(d) > \max(b_{\text{ext}}(d), b_{\text{int}}(d))$$

In Tabelle 3.6 sind typische Externalisierungs- und Internalisierungsbandbreiten für die verschiedenen Datentypen aufgelistet. Um die Kommunikation nicht zusätzlich zu verlangsamen, sollte also in Abhängigkeit vom kommunizierten Datentyp die Kanalbandbreite größer als die dort angegebenen Werte gewählt werden. Dies gilt sowohl für den Kanal der Inhaltsübermittlung als auch für den Kanal zur Steuerung.

Der Inhalt, der kommuniziert werden soll, hat Einfluss auf den Datentyp für den Kommunikationskanal der Inhaltsübermittlung. Der folgende Satz beschreibt diesen Zusammenhang.

Satz 4.9: **Kommunikationsinhalt und Datentyp**

Passt der Datentyp des Kommunikationskanals nicht zum kommunizierten Inhalt, so verlangsamt das Externalisierung und Internalisierung und führt tendenziell zu mehr Missverständnissen.

Herleitung. Stimmt der Datentyp des Inhalts nicht mit dem Datentyp des Kanals überein, so muss der Sender zunächst eine andere für den Datentyp des Kanals geeignete Nachricht erzeugen und repräsentieren. Erzeugen und Repräsentieren sind Zusatzaufwände, die die Externalisierung verlangsamen (vgl.

Tabelle 4.1: Relativer Aufwand der Umwandlung von Datentypen

von	1D-A-N	1D-V-N	1D-V-A	2D-V-N	2D-V-A
nach					
1D-A-N	–	leicht	mittel	schwer	sehr schwer
1D-V-N	leicht	–	leicht	schwer	sehr schwer
1D-V-A	leicht	leicht	–	schwer	schwer
2D-V-N	schwer	schwer	schwer	–	schwer
2D-V-A	leicht	leicht	leicht	mittel	–

Legende: 1D-A-N Gesprochene natürliche Sprache, 1D-V-N Geschriebene natürliche Sprache, 1D-V-A Geschriebene künstliche Sprache, 2D-V-N Natürliches Bild, 2D-V-A Künstliches Bild

Def. 3.32 und Korollar 3.2). Auf Empfängerseite müssen die im falschen Datentyp vorliegenden Daten internalisiert werden. Vor allem das Interpretieren dauert dabei länger, als beim zum Inhalt passenden Datenformat (vgl. Def. 3.33 und Korollar 3.2).

Weiterhin erhöht die Notwendigkeit ein zusätzliches Datenformat verstehen zu müssen, die Wahrscheinlichkeit, dass dem Empfänger das dafür notwendige Wissen fehlt. Dies erhöht wiederum die Wahrscheinlichkeit, dass es zu Missverständnissen kommt. □

Aus Satz 4.9 folgt, dass der Datentyp des Kommunikationskanals für die Inhaltsübermittlung passend zum Kommunikationsinhalt gewählt werden sollte (vgl. u.a. [Dennis 2008]).

Die verschiedenen Datentypen können mit unterschiedlich hohem Aufwand ineinander überführt werden. Zum Beispiel ist es aufwändiger ein natürliches Bild, z.B. von einer Person, in natürlicher Sprache zu beschreiben als gesprochene natürliche Sprache in geschriebene natürliche Sprache umzuwandeln. Allgemein gilt, dass 1-dimensionale Daten ohne großen Aufwand in höherdimensionalen Datentypen repräsentiert werden können, sofern sich die grundlegenden Symbole nicht ändern müssen. Tabelle 4.1 zeigt geschätzte relative Aufwände bei der Umwandlung der wichtigsten Datentypen.

Für den Kanal zur Steuerung der Kommunikation gilt eine Einschränkung bezüglich der sinnvoll verwendbaren Datentypen.

Hypothese 4.2: **Datentyp und Steuerungseffizienz**

Zur Steuerung von Kommunikation eignen sich ausschließlich natürliche Datentypen. Dies sind in absteigender Reihenfolge ihrer Steuerungseffizienz:

- Bewegtes Natürliches Bild (3D visuell natürlich). Zum Beispiel werden Mimik und Gestik als bewegtes natürliches Bild kodiert.
- Gesprochene natürliche Sprache (1D auditiv natürlich).
- Geschriebene natürliche Sprache (1D visuell natürlich).

Durch die natürlich gegebene Fähigkeit des Menschen Daten mit dem Datentyp bewegtes natürliches Bild effizient verarbeiten zu können, eignet er sich besonders gut zur Steuerung von Kommunikation. Auch wenn der Mensch keine so große Bandbreite zur Verarbeitung auditiver Daten hat wie zur Verarbeitung visueller Daten, eignet sich gesprochene natürliche Sprache auch sehr gut zur Steuerung von Kommunikation, da der Mensch von Kind auf an den Umgang mit diesem Datentyp gewöhnt ist. Künstliche Sprachen oder gar künstliche Bilder eignen sich hingegen kaum zur Steuerung von Kommunikation, weil der Mensch dies erst mühsam erlernen müsste und es danach dennoch anstrengend, weil unnatürlich [Kock 2005], wäre, Kommunikation so abzustimmen.

Aus Satz 4.9 folgt das Korollar 4.6, welches zusammenfasst, wann es sinnvoller ist künstliche statt natürliche Datentypen zu nutzen. In der Softwareentwicklung sind diese unumgänglich. So sind z.B. Quellcode und UML-Diagramme häufig auftretende künstliche Datentypen. Bei der Medienwahl sollte daher immer auch der Datentyp des Inhalts berücksichtigt werden.

Korollar 4.6: **Künstlich effizienter als natürlich**

Unter den folgenden Voraussetzungen ist die Nutzung künstlicher Datentypen für die Kommunikation effizienter als die Nutzung der natürlichen Sprache.

- Wenn der zu kommunizierende Inhalt in einem künstlichen Datentyp vorliegt.
- Wenn der gemeinsame Kontext keine natürliche Sprache enthält, aber eine künstliche Sprache (z.B. Mathematik, Programmiersprachen) oder höherdimensionale künstliche Datenformate (z.B. UML)

Aus der Definition der Zielunterschiede in Kapitel 3.6.2 und Hypothese 4.2 folgt Hypothese 4.3.

Hypothese 4.3: **Zielunterschiede und Datentyp für Steuerung**

Je nach Zielunterschieden der Kommunikationsteilnehmer sollte mindestens[a] einer der folgenden Datentypen für den steuernden Kommunikationskanal gewählt werden:

Widersprüchliche Ziele: Bewegtes natürliches Bild (3D visuell natürlich)

Koordinierte Ziele: Gesprochene natürliche Sprache (1D auditiv natürlich)

Kollaborative Ziele: Geschriebene natürliche Sprache (1D visuell natürlich)

um die Kommunikation effektiv steuern zu können.

[a] Reihenfolge entsprechend Hypothese 4.2

Widersprüchliche Ziele stellen die höchsten Anforderungen an den Kommunikationskanal für die Steuerung. Nach Hypothese 4.2 ist der 3D-visuell-natürliche-Kanal am besten dafür geeignet. Koordinierte Ziele haben nur geringe

Anforderungen an die steuernde Kommunikation. Daher reicht hier der gesprochene natürliche Sprache. Zudem fällt dem Menschen Steuerung über gesprochene Sprache leicht. Bei kollaborativen Zielen reicht geschriebene natürliche Sprache zur Steuerung aus, falls diese überhaupt notwendig ist.

Aus der Definition der Wissensunterschiede in Kapitel 3.6.1 und Hypothese 4.2 folgt Hypothese 4.4.

Hypothese 4.4: **Wissensunterschiede und Datentyp für Steuerung**

Je nach Wissensunterschieden der Kommunikationsteilnehmer sollte mindestens[a] einer der folgenden Datentypen für den steuernden Kommunikationskanal gewählt werden:

Kulturebene: Bewegtes natürliches Bild (3D visuell natürlich)

Sprachebene: Bewegtes natürliches Bild (3D visuell natürlich)

Domänenebene: Bewegtes natürliches Bild (3D visuell natürlich)

Ausbildungsebene: Gesprochene natürliche Sprache (1D auditiv natürlich)

Unternehmensebene: Gesprochene natürliche Sprache (1D auditiv natürlich)

Projektebene: Geschriebene natürliche Sprache (1D visuell natürlich)

Gesprächsebene: Geschriebene natürliche Sprache (1D visuell natürlich)

um die Kommunikation effektiv steuern zu können.

[a] Reihenfolge entsprechend Hypothese 4.2

Wissensunterschiede auf Kultur-, Sprach- und Domänenebene stellen die höchsten Anforderungen an den Kommunikationskanal für die Steuerung. Nach Hypothese 4.2 ist ein bewegtes natürliches Bild am besten dafür geeignet. Wissensunterschiede auf Ausbildungs- oder Unternehmensebene hingegen haben geringere Anforderungen an die steuernde Kommunikation. Daher reicht hier gesprochene natürliche Sprache. Bei Wissensunterschieden auf Projekt- oder Gesprächsebene reicht geschriebene natürliche Sprache zur Steuerung aus.

4.3.2 Einfluss von Kommunikationsziel, Ziel- und Wissensunterschieden

Die Latenz hat dann Einfluss auf den Kommunikationserfolg, wenn während einer Kommunikation Information in verschiedene Richtungen fließen muss, d.h. wenn Sender und Empfänger ihre Rollen mindestens einmal wechseln müssen, um das Kommunikationsziel zu erreichen. Dies kann entweder direkt durch das Kommunikationsziel gegeben sein oder indirekt, wenn Missverständnisse erkannt und ausgeräumt werden müssen (d.h., wenn Steuerung notwendig ist). Der direkte Zusammenhang zwischen Kommunikationsziel und Anforderungen an die Antwortgeschwindigkeit des Kommunikationskanals für die Inhaltsübermittlung ist in folgender Hypothese beschrieben.

Hypothese 4.5: **Kommunikationsziel und Latenzanforderungen für Inhaltsübermittlung**

Je nach Kommunikationsziel sollten mindestens folgende Latenzen für den Kommunikationskanal der Inhaltsübermittlung gewählt werden:

Kreativ sein: sehr geringe Latenz ($< 100\,\text{ms}$)

Entscheiden: geringe Latenz ($< 3\,\text{s}$)

Informieren: Keine Latenzanforderungen

um erfolgreich zu kommunizieren.

Nach Definition des Kommunikationsziels in Kapitel 3.6.3 ist schnelles Feedback wichtig für die Kreativität. Daher hat dieses Kommunikationsziel die höchsten Latenzanforderungen. Weiterhin ist Feedback bei der Entscheidungsfindung wichtig. Dies resultiert in mittleren Latenzanforderungen. Ist das Ziel nur das Informieren, so fließt Information nur in eine Richtung. Latenzanforderungen spielen keine Rolle.

Latenzanforderungen für den steuernden Kanal resultieren aus den Ziel- und Wissensunterschieden der Kommunikationsteilnehmer.

Hypothese 4.6: **Zielunterschiede und Latenzanforderungen für Steuerung**

Je nach Zielunterschieden der Kommunikationsteilnehmer sollte mindestens eine der folgenden Latenzen für den steuernden Kommunikationskanal gewählt werden:

Widersprüchlich: sehr geringe Latenz (< 100 ms)

Koordiniert: mittlere Latenz (< 10 min.)

Kollaborativ: Keine Latenzanforderungen

um die Kommunikation effektiv steuern zu können.

Aus der Definition von Zielunterschieden in Kapitel 3.6.2 können die sich aus den Zielunterschieden ergebenden Latenzanforderungen für die Steuerung wie folgt begründet werden.

Widersprüchliche Ziele: Da es in Kommunikationsaktivitäten mit Zielkonflikten leicht passieren kann, dass die Kommunikation auf eine emotionale Ebene abrutscht, gibt es hier die höchsten Anforderungen an die Kommunikations*sub*aktivität der Steuerung. Teilnehmer müssen schnell Reaktionen der anderen Teilnehmer erkennen und richtig deuten können, um darauf angemessen – in ihrem Interesse – reagieren zu können. Geringe Latenzen und ein visueller Kanal sind daher sehr wichtig.

Koordinierte Ziele: Anforderungen an das Medium der steuernden Kommunikationssubaktivität sind hier geringer als im Konfliktfall. Eine gewisse Feedbackgeschwindigkeit ist aber wünschenswert, um alle Ansichten besser koordinieren zu können.

Keine Zielunterschiede (auch Kollaboration): Antwortgeschwindigkeit des Kanals für den steuernden Kommunikationsteil spielt hier nur insofern eine Rolle, dass Fortschritte rechtzeitig abgestimmt werden können, um Doppelarbeit zu vermeiden.

Hypothese 4.7: **Wissensunterschiede und Latenzanforderungen für Steuerung**

Je nach Wissensunterschieden der Kommunikationsteilnehmer sollte mindestens eine der folgenden Latenzen für den steuernden Kommunikationskanal gewählt werden:

Kulturebene: sehr geringe Latenz (< 100 ms)

Sprachebene: sehr geringe Latenz (< 100 ms)

Domänenebene: sehr geringe Latenz (< 100 ms)

Ausbildungsebene: geringe Latenz (< 3 s)

Unternehmensebene: mittlere Latenz (< 10 min.)

Projektebene: hohe Latenz (< 1 d)

Gesprächsebene: hohe Latenz (< 1 d)

um die Kommunikation effektiv steuern zu können.

Aus der Definition von Wissensunterschieden in Kapitel 3.6.1 resultieren die folgenden Anforderungen an den Kanal zur Steuerung der Kommunikation.

Kulturebene: Wissensunterschiede auf Kulturebene stellen die höchsten Anforderungen an ein Kommunikationsmedium. Häufig können sie *nicht* einfach durch die Wahl eines geeigneten Mediums überwunden werden. Selbst bei Nutzung der reichhaltigsten und schnellsten Kanäle, wie z.B. bei direkter Kommunikation von Angesicht zu Angesicht am Whiteboard, müssen alle Beteiligten sehr vorsichtig mit ihrem kommunikativen Verhalten umgehen, sowohl verbal als auch non-verbal.

Sprachebene: Medienanforderungen an die Kommunikationssubaktivität zur Steuerung sind hoch. Ein visueller Kanal, der die zeitnahe Übertragung eines bewegten natürlichen Bildes (z.B. Video) ermöglicht, kann durch schnelle Übertragung von Mimik und Gestik helfen, Missverständnisse, die bei einem rein auditiven Kanal unentdeckt bleiben würden, schnell zu entdecken.

Domänenebene: Medienanforderungen an die Kommunikationssubaktivität

zur Steuerung sind hoch. Zeitnahes Feedback ist wichtig, damit Missverständnisse erkannt und beseitigt werden können.

Ausbildungsebene: Probleme können entstehen, wenn die jeweilige Fachsprache unterschiedlich tief verankert ist oder je nach Ausbildungshintergrund leichte Unterschiede aufweist. Auch hier ist ein gewisses Maß an Feedback wichtig, um Missverständnisse erkennen zu können.

Unternehmensebene: Daraus resultierende Missverständnisse sind relativ leicht zu erkennen, was in moderaten Medienanforderungen bezüglich der Sub-Aktivität zur Steuerung der Kommunikation resultiert.

Projektebene: Missverständnisse oder fehlende Informationen sind üblicherweise leicht zu identifizieren. Der Steuerungskanal stellt daher keine hohen Anforderungen, so lange überhaupt ein Rückkanal vorhanden ist.

Gesprächsebene: Auch bei Wissensunterschieden auf Gesprächsebene sind Missverständnisse sehr leicht zu identifizieren. Die Möglichkeit zur Steuerung der Kommunikation ist zwar sinnvoll, muss aber nicht besonders schnell sein.

4.3.3 Einfluss von Randbedingungen

Randbedingungen sind Einflüsse, die meist Kommunikationsaktivität-übergreifend gelten, z.B. projektweit, und die die Wahl eines Mediums direkt betreffen. In der Softwareentwicklung beeinflussen die folgenden Randbedingungen häufig die Medienwahl:

Medienkosten: Mit dem Einsatz von Medien sind Kosten verbunden. Dies sind sowohl technische als auch organisatorische Kosten. Für ein persönliches Meeting müssen alle Teilnehmer an einen Ort gebracht werden. Dies kann Reisekosten verursachen. Elektronische Kommunikationsmedien können Anschaffungs- und Betriebskosten verursachen. Begrenzte Budgets führen dazu, dass nicht immer das zu den Anforderungen passende Medium gewählt werden kann. Dies führt wiederum zu einem geringeren Kommunikationserfolg.

Verteilung: Wenn sich die Teilnehmer einer Kommunikationsaktivität nicht am selben Ort befinden, ist ein Gespräch von Angesicht zu Angesicht

nicht ohne Weiteres möglich. Es müssen dann entweder alle Teilnehmer an einen Ort gebracht, oder auf mediierte Kommunikation ausgewichen werden. Im verteilten Fall sind die Kosten für das Medium Angesicht zu Angesicht sehr hoch. Verteilung beeinflusst also auch die Medienkosten. Zudem kann Verteilung einen sehr starken Einfluss auf die minimal mögliche Antwortgeschwindigkeit haben, falls sie Zeitzonenübergreifend ist. Meist sind dann nur noch asynchrone Medien wie E-Mail oder Dokumente nutzbar.

Andere: Technische oder organisatorische Einschränkungen können auch dazu führen, dass ein nach den Anforderungen geeignetes Medium nicht eingesetzt werden kann. Zum Beispiel sind Videokonferenzsysteme oft ausgebucht oder limitierte Netzwerke verhindern den Einsatz hochauflösender Videoübertragungssysteme gänzlich. Ein weiterer Kommunikationsaktivität-übergreifender Einfluss ist, wie die Nutzung eines Mediums erlernt wurde bzw. wie vertraut die Kommunikationsteilnehmer mit einem Medium sind (vgl. u.a. [Dennis 2008]). Z.B erlernt jeder die Kommunikation von Angesicht zu Angesicht von Kindheit an. Es wurde erlernt, wie man sich dabei verhält. Die viele Übung führt zu einem sehr natürlichen Umgang mit diesem Medium (vgl. [Kock 2005]). Telefon ist ein Medium, dessen Umgang in unserer Gesellschaft auch recht vertraut ist. Die sogenannten "Digital Natives" können mit neuen Medien natürlicher umgehen, als die vorherigen Generationen. Umgekehrt kann der Umgang mit bestimmten Medien auch wieder verlernt werden, z.B. das Schreiben von Briefen. Weitere Randbedingungen sind vorstellbar.

4.3.4 Zusammenfassung der Einflussfaktoren

In Tabelle 4.2 sind noch einmal alle Einflussfaktoren mit ihren charakteristischen Werten und Beispielen aus der Softwareentwicklung zusammengefasst. Die Faktoren können sich unterschiedlich stark auf die Anforderungen für den steuerunden und den inhaltlichen Kanal auswirken (vgl. Abb. 4.5). So sind zum Beispiel die Anforderungen für das steuernde Medium in einem Anforderungsverhandlungsgespräch, bei dem die Teilnehmer Zielkonflikte und Wissensunterschiede auf der Domänenebene haben, viel höher, als die Anforderungen für das Medium zur Übertragung der eigentlichen Inhalte.

Tabelle 4.2: Wichtige Einflussfaktoren auf die Medienwahl im Software Engineering

Faktor	**Charakteristische Werte:** SE Beispiel
Inhalt	**Gesprochene natürliche Sprache:** Telefonat
	Geschriebene natürliche Sprache: E-Mail
	Geschriebene künstliche Sprache: Quellcode
	Künstliche Abbildungen: UML-Klassendiagramm
	Bewegtes natürliches Bild: Angesicht zu Angesicht
Ziel der Kommunikationsaktivität	**informieren:** Anforderungselicitation
	kreativ sein: Softwarearchitektur erstellen
	entscheiden: Designalternative wählen
Zielunterschiede	**Widerspruch:** Perfektionistische Entwickler vs. limitiertes Budget
	Koordiniert: Kunde und Entwickler wollen funktionierende Software
	Kollaboration: Entwickler beim Pair Programming
Wissensunterschiede	Unterschiede auf
	Kulturebene: Entwickler aus Asien und Europa
	Sprachebene: Entwickler aus Frankreich und Deutschland
	Domänenebene: Kunde (z.B. Bankenbranche) und Entwickler
	Ausbildungsebene: Entwickler mit SE-Abschluss von unterschiedlichen Universitäten
	Unternehmensebene: Entwickler unterschiedlicher Unternehmen
	Projektebene: Entwickler in unterschiedlichen Projekten
	Gesprächsebene: Entwickler, der das letzte Meeting verpasst hat
Verteilung	**lokal:** Entwickler im selben Raum
	verteilt: Entwickler in verschiedenen Räumen

4.4 Theoreme der Problemgröße

Softwareentwicklung ist das Lösen von Problemen mit Hilfe von Software (vgl. Kapitel 3.7). Für die Planung und Durchführung von Softwareentwicklungsprojekten ist es wichtig zu wissen, wie schwierig das zu lösende Problem ist und wie sich diese Schwierigkeit auf die Parameter des Softwareprojekts auswirken kann. Wichtige, da kostenintensive, Parameter der Softwareentwicklung sind die Projektlaufzeit und die Anzahl benötigter Softwareentwickler. In diesem Kapitel werden die Zusammenhänge zwischen Problemgröße, Anzahl Softwareentwickler, Entwicklungsdauer und Abstimmungsaufwand erläutert (vgl. Abb. 4.1) wie sie sich aus den Grundbegriffen der Informationsflusstheorie ergeben. Die hier vorgestellten Sätze können z.B. bei der Planung helfen, die Projektlaufzeit gegen die Teamgröße abzuwägen. Schließlich eignen sich die Theoreme auch, um das von Brooks provokant formulierte Gesetz *"Adding manpower to a late software project makes it later."* [Brooks 1995, S. 232] und die zugehörige "Badewannenkurve" (vgl. Abb. 5.1 rechts) mit Hilfe der Informationsflusstheorie zu erklären und sogar einzuschränken, für welche Art Softwareprojekte Brook's Law gilt (vgl. Kapitel 5.1.1, Abb. 5.2).

Im Folgenden werden einige Fragestellungen aufgelistet, die im Zusammenhang mit der Komplexität des mit Software zu lösenden Problems stehen (vgl. Def. 3.44). Die Fragen sind in der Softwareentwicklung z.B. für die Planung und Aufwandsschätzung von Softwareprojekten relevant.

- Wie komplex darf ein zu lösendes Problem höchstens sein, damit es von einer gegebenen Anzahl an Entwicklern in einer gegebenen Zeit gelöst werden kann?
- Wie lange dauert ein Projekt, bei gegebener Anzahl an Entwicklern und gegebener Problemkomplexität? Oder anders ausgedrückt: Was ist die minimale Zeit zur Lösung eines gegebenen Problems?
- Wie viele Entwickler benötigt man, um ein gegebenes Problem in einer gegebene Zeit zu lösen?

Die Eigenschaften Problemkomplexität, Entwicklungsdauer, Anzahl Entwickler und Abstimmungsaufwand spielen bei der Beantwortung dieser Fragen eine entscheidende Rolle (vgl. Abb. 4.1). Um die Zusammenhänge zwischen diesen Eigenschaften besser ausdrücken zu können, werden zunächst einige Variablen definiert:

I: Sei I der Informationsgehalt aller Informationen, die bei der Softwareentwicklung insgesamt fließen müssen, sobald die Entwickler das zu lösende Problem kennen und verstanden haben. I enthält also die Dokumentation des Softwareprodukts, die Dokumentation von Zwischendokumenten und die Kommunikation zur Abstimmung zwischen den Entwicklern. I enthält *nicht* die Kommunikation mit Fachleuten.

I_C: Sei I_C der Teil von I, der während der Softwareentwicklung zur Abstimmung der Entwickler fließen muss. I_C kann auch als Abstimmungsaufwand oder Kommunikationsoverhead (vgl. [Brooks 1995]) bezeichnet werden.

I_{Dev}: Sei I_{Dev} der Teil von I, der ohne Abstimmung direkt in die Entwicklung des Softwareprodukts und der Zwischendokumente fließt. I_{Dev} kann auch als Entwicklungsaufwand bezeichnet werden. Es gilt $I_{\text{Dev}} = I - I_C$.

w_C: Sei w_C der relative Anteil des Abstimmungsaufwands I_C vom Gesamtaufwand I. Es gilt $w_C = I_C / I$.

w_{Dev}: Sei w_{Dev} der relative Anteil des Entwicklungsaufwands I_{Dev} vom Gesamtaufwand I. $w_{\text{Dev}} = I_{\text{Dev}} / I$.

t: Sei t die Entwicklungsdauer des Softwareentwicklungsprojekts. Die Entwicklungsdauer t wird je nach Projektgröße üblicherweise in Tagen, Monaten oder Jahren angegeben.

n: Sei n die Anzahl der Softwareentwickler.

I_1: Sei I_1 der Informationsgehalt der Informationen, die ein durchschnittlicher Softwareentwickler in einer Zeiteinheit von t (Tag, Monat, Jahr) verarbeiten kann. Zur Vereinfachung wird angenommen, dass I_1 konstant und für alle Entwickler gleich ist.

t_1: Sei t_1 die Entwicklungsdauer, in der ein Entwickler I_{Dev} alleine bewältigen kann, d.h. ohne Arbeitsteilung und ohne Abstimmungsaufwand. Es gilt $I_{\text{Dev}} = t_1 \cdot I_1$. t_1 kann als ein Maß für die Problemgröße des mit Software zu lösenden Problems (vgl. Def. 3.45) betrachtet werden. Mit t_1 kann der Entwicklungsaufwand bei der Planung von Softwareprojekten geschätzt werden, z.B. als die Anzahl geschätzter Entwicklermonate, die ohne Abstimmungsoverhead in die reine Entwicklung des Softwareprodukts investiert werden müssen.

C_P: Sei C_P die Problemkomplexität nach Definition 3.44.

C: Sei C die Abstimmungskomplexität, die sich aus dem Abstimmungsbedarf bei der Softwareentwicklung ergibt. Nach Definition 3.14 ist $C(n)$ die Anzahl der Abstimmungsabhängigkeiten, die sich durch die Verteilung der Arbeit auf n Entwickler ergibt.

t_C: Sei t_C die Zeit, in der eine Abstimmungsabhängigkeit mit Hilfe von Kommunikation aufgelöst werden kann.

Einige Zusammenhänge ergeben sich bereits direkt aus den Definitionen der Variablen.

Für $n = 1$ gilt $w_{\text{Dev}} = 1$:

$$I = I_{\text{Dev}} = t_1 \cdot I_1 \tag{4.1}$$

Für $n > 1$ gilt $w_{\text{Dev}} + w_C = 1$:

$$\begin{aligned} I &= I_{\text{Dev}} + I_C && (4.2)\\ n \cdot t \cdot I_1 &= w_{\text{Dev}} \cdot n \cdot t \cdot I_1 + w_C \cdot n \cdot t \cdot I_1 && (4.3)\\ &= t_1 \cdot I_1 + C(n) \cdot t_C \cdot I_1 && (4.4) \end{aligned}$$

Die Abstimmungskomplexität C ist vergleichbar mit der Laufzeit- oder Speicherplatzkomplexität von Algorithmen. Letztere beschreiben den Ressourcenverbrauch (Rechenzeit, Speicherverbrauch) von Algorithmen in Abhängigkeit einer Eingabegröße. Die Abstimmungskomplexität der Softwareentwicklung beschreibt den "Kommunikationsverbrauch", d.h. den Abstimmungsaufwand, der abhängig vom zu lösenden Problem und der Anzahl der Softwareentwickler ist (siehe unten).

Problemkomplexität und Abstimmungskomplexität können ebenso wie Algorithmen in verschiedene Komplexitätsklassen eingeordnet werden, welche durch asymptotische Schranken definiert sind. Dazu sei die untere Schranke für die Komplexitäten der Softwareentwicklung in Anlehnung an die O-Notation der Informatik wie folgt definiert:

$$f \in \Omega(g), \text{ wenn } \exists n_0 > 0 \text{ und } \exists M > 0, \text{ sodass } \forall n \geq n_0 : |f(n)| \geq M \cdot |g(n)| \tag{4.5}$$

Aus verschiedenen Gründen wird Software nicht nur von einem einzelnen, sondern von mehreren Softwareentwicklern entwickelt. Einerseits kann das

Problem so umfangreich sein, dass ein Entwickler für dessen Lösung zu lange brauchen würde. Andererseits kann das Problem so komplex sein, dass ein Entwickler alleine es nicht kognitiv erfassen kann (vgl. u.a. [Brooks 1995]). In beiden Fällen muss die Entwicklung so aufgeteilt werden, dass alle Entwickler parallel an der Lösung des Problems arbeiten können, aber möglichst wenig Abstimmungsaufwand zwischen den Teilaufgaben entsteht. Im Folgenden wird daher davon ausgegangen, dass die Software durch den Entwurf immer so zerlegt wird, dass alle Entwickler parallel arbeiten können. Diese Annahme lässt sich u.a. damit begründen, dass sich in vielen Softwareprojekten die Struktur der Organisation, d.h. auch die Struktur des Entwicklungsteams, in der Struktur der Software wiederfindet (vgl. Conway's Law in Kapitel 5.1.6 und [Conway 1968, Herbsleb 2003]). Für eine Aufteilung des Problems in n Teile ergibt sich folgender Zusammenhang zwischen Problem- und Abstimmungskomplexität.

Satz 4.10: **Abstimmungskomplexität**

Die Abstimmungskomplexität C ist mindestens so komplex, wie die Problemkomplexität C_P.

$$C \in \Omega(C_P)$$

Herleitung. Die Problemkomplexität ist nach Definition 3.44 die Anzahl der Abhängigkeiten, die bei der Zerlegung des Problems in n Teillösungen zwischen den Teillösungen bestehen. Wenn in einem Softwareprojekt mit n Entwicklern jeder Entwickler genau eine Teillösung bearbeitet, dann müssen mindestens $C_P(n)$ Abhängigkeiten abgestimmt werden. D.h. es gibt ein $M \geq 1$, sodass $C(n) > M \cdot C_P(n)$. M kann als die Anzahl der Abstimmungen gedeutet werden, die zur Auflösung einer Abhängigkeit zwischen Teillösungen notwendig sind. Mit $C(n) > 0$ und $C_P(n) > 0$ gilt dann auch:

$$\exists n_0 = 1 > 0 \text{ und } \exists M > 0, \text{ sodass } \forall n \geq 1 : |C(n)| \geq M \cdot |C_P(n)|$$

□

D.h., ab einer gewissen Problemkomplexität und einer gewissen Entwickleranzahl lässt sich Abstimmungsaufwand bei der Softwareentwicklung nicht mehr vermeiden (vgl. essenzielle Komplexität [Brooks 1987]). Nach oben ist die Abstimmungskomplexität nicht beschränkt. So kann z.B. ein schlechter Entwurf, oder gar kein Entwurf, dazu führen, dass die Abstimmungskomplexität in

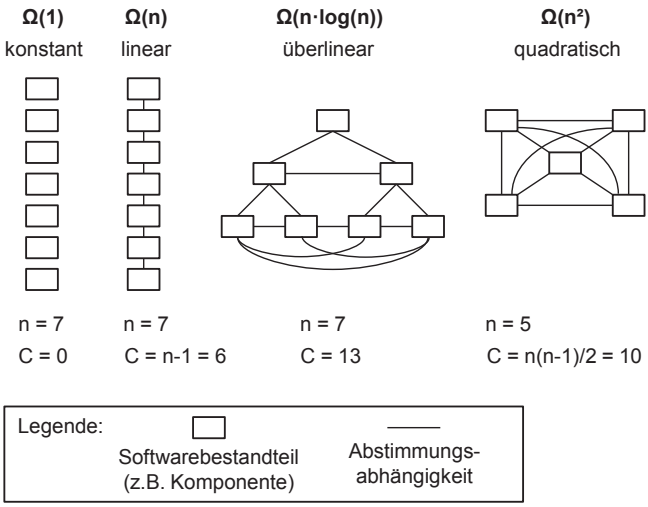


Abb. 4.6: Vergleich verschiedener Abstimmungskomplexitäten

$\Omega(n^2)$ ist, obwohl das Problem nur eine Komplexität von $\Omega(n)$ hat (vgl. akzidentielle Komplexität [Brooks 1987]). Komplexe Probleme können also nur mit kleinen Teams mit relativ wenig Abstimmungsaufwand gelöst werden, sofern ihre individuelle kognitive Leistungsfähigkeit und der Zeitplan das zulassen.

Abbildung 4.6 zeigt einen Vergleich von Beispielen von Softwarearchitekturen mit verschiedenen Abstimmungskomplexitäten. Eine Abstimmungskomplexität in $\Omega(1)$ wird nur erreicht, wenn alle Softwarebestandteile völlig unabhängig voneinander sind. Dieser Fall ist sehr unrealistisch für die Softwareentwicklung. Eine Komplexität in $\Omega(n)$ lässt sich z.B. mit einer Schichtenarchitektur erreichen, bei der jede Schicht von einem Entwickler entwickelt wird und diese nur von der darunter liegenden Schicht abhängig ist. Realistischer ist der überlineare Fall in $\Omega(n \log(n))$. Auch hier ist eine baumartige Schichtenarchitektur vorstellbar, bei der die unteren Schichten von mehreren Entwicklern bearbeitet werden. Quadratische Abstimmungskomplexität ($\Omega(n^2)$) und mehr ergibt sich, wenn jeder Softwarebestandteil abhängig von jedem anderen Softwarebestandteil ist.

Abstimmungsaufwand lässt sich in vielen Projekten also nicht vermeiden. Das

gilt insbesondere für große Projekte oder bei komplexen Problemstellungen. So wächst zum Beispiel der Abstimmungsaufwand in einem $\Omega(n^2)$-Projekt quadratisch mit der Anzahl der Softwareentwickler. Es stellt sich die Frage, ab welcher Anzahl von Entwicklern, unter Berücksichtigung anderer relevanter Faktoren, der Abstimmungsaufwand so groß wird, dass keine Zeit für die eigentliche Entwicklung mehr übrig ist. Der folgende Satz beschreibt, wie der Abstimmungsaufwandsanteil w_C von Entwickleranzahl, Entwicklungsdauer, Abstimmungskomplexität und Abstimmungszeit abhängt.

Satz 4.11: **Abstimmungsaufwandsanteil**

Der Abstimmungsaufwandsanteil w_C ergibt sich wie folgt aus Abstimmungskomplexität C, Abstimmungszeit t_C, Entwickleranzahl n und Entwicklungsdauer t:

$$w_C = \frac{C(n) \cdot t_C}{n \cdot t}$$

Herleitung. Mit Formel 4.2 gilt:

$$\begin{aligned} I &= I_{\text{Dev}} + I_C \\ n \cdot t \cdot I_1 &= n \cdot t \cdot w_{\text{Dev}} \cdot I_1 + C(n) \cdot t_C \cdot I_1 \\ \Leftrightarrow w_{\text{Dev}} &= \frac{n \cdot t - C(n) \cdot t_C}{n \cdot t} \\ w_{\text{Dev}} &= 1 - \frac{C(n) \cdot t_C}{n \cdot t} \\ \Leftrightarrow w_C &= \frac{C(n) \cdot t_C}{n \cdot t} \end{aligned}$$

□

Abbildung 4.7 zeigt einen Vergleich der Entwicklung des Abstimmungsaufwandsanteils w_C in Abhängigkeit von der Anzahl der eingesetzten Entwickler für unterschiedlich komplexe Probleme. Es ist zu erkennen, dass insbesondere Probleme, die in $\Omega(n^2)$ liegen, der Abstimmungsaufwandsanteil schnell, d.h. überlinear, gegen 100 % geht. Die anderen Parameter wie Abstimmungszeitanteil und Entwicklungsdauer beeinflussen nur die Lage der Kurven, nicht aber ihr asymptotisches Verhalten.

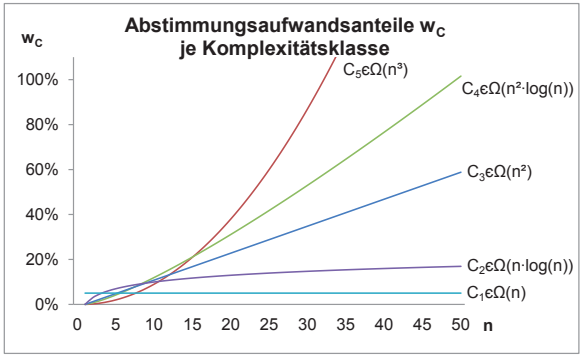


Abb. 4.7: Relativer Kommunikationsoverhead in Abhängigkeit von der Anzahl der für die Softwareentwicklung eingesetzten Entwickler für unterschiedlich komplexe Probleme

Der folgende Satz beschreibt die Entwicklungsdauer in Abhängigkeit von Problemgröße, Entwickleranzahl, Abstimmungskomplexität und Abstimmungszeit.

Satz 4.12: **Entwicklungsdauer**

Die Entwicklungsdauer t ergibt sich wie folgt aus Problemgröße t_1, Entwickleranzahl n, Abstimmungskomplexität C und Abstimmungszeit t_C:

$$t = \frac{t_1 + C(n) \cdot t_C}{n}$$

Herleitung.

$$\begin{aligned} I &= I_{\text{Dev}} + I_C \\ n \cdot t \cdot I_1 &= t_1 \cdot I_1 + C(n) \cdot t_C \cdot I_1 \\ \Leftrightarrow t &= \frac{t_1 + C(n) \cdot t_C}{n} \end{aligned}$$

□

In Kapitel 5.1.1 findet sich ein Anwendungsbeispiel der Formel aus Satz 4.12. Abbildung 5.2 auf Seite 203 zeigt, wie man mit Hilfe dieser Formel die berühmte "Badewannenkurve" (vgl. Abb. 5.1 rechts) und Brook's Law *"Adding manpower to a late software project makes it later"* [Brooks 1995, S. 232] erklären kann.

Der folgende Satz beschreibt, wie die Entwickleranzahl in Abhängigkeit von Problemgröße, Entwicklungsdauer, Abstimmungszeit und Abstimmungskomplexität bestimmt werden kann.

Satz 4.13: **Entwickleranzahl**

Die Entwickleranzahl n steht in folgendem Verhältnis zu Problemgröße t_1, Entwicklungsdauer t, Abstimmungskomplexität C und Abstimmungszeit t_C:

$$1 = \frac{t_1 + C(n) \cdot t_C}{n \cdot t}$$

Herleitung.

$$\begin{aligned} I &= I_{\text{Dev}} + I_C \\ n \cdot t \cdot I_1 &= t_1 \cdot I_1 + C(n) \cdot t_C \cdot I_1 \\ \Leftrightarrow 1 &= \frac{t_1 + C(n) \cdot t_C}{n \cdot t} \end{aligned}$$

□

Mit Hilfe der Formel aus Satz 4.13 kann z.B. bestimmt werden, ab welcher Entwickleranzahl die Entwicklungsdauer eines Projekts in Abhängigkeit der Abstimmungskomplexität wieder steigt, indem die Nullstellen der ersten Ableitung dieser Formel bestimmt werden (vgl. Kapitel 5.1.1 S. 202).

Mit diesen Formeln können alle eingangs genannten Fragen beantwortet werden, sofern alle relevanten Parameter bekannt sind. Dabei ist zu beachten, dass die Formeln nur dann sinnvoll eingesetzt werden können, wenn man Problemkomplexität, Abstimmungsaufwandsanteil und Abstimmungszeit quantifizieren kann. Das ist, wie in Kapitel 3.4.1 beschrieben, nicht absolut, sondern nur relativ zu ähnlichen Projekten möglich. Die Formeln können aber mit Hilfe von Erfahrungswerten aus vergleichbaren Projekten, bei denen die Werte

bekannt sind, sinnvoll eingesetzt werden. Das folgende Beispiel soll das verdeutlichen (vgl. [Glickstein 2008] für ein ähnliches Beispiel):

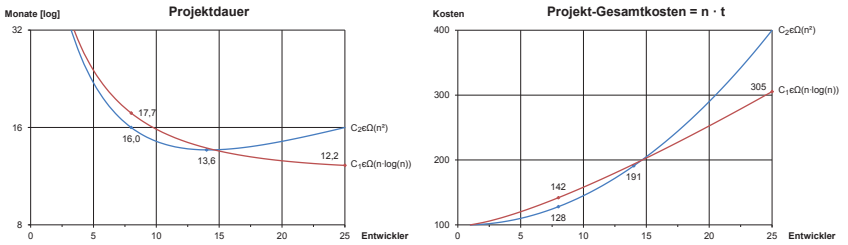


Abb. 4.8: Beispielverlauf der Projektdauer in Monaten (links) und der Gesamtkosten in Entwicklermonaten (rechts) in Abhängigkeit der Entwickleranzahl für Projekte mit überlinearer und quadratischer Abstimmungskomplexität

Beispiel: Auf Grund der Erfahrungen aus dem Vorgängerprojekt wird der reine Entwicklungsaufwand für das neue Projekt auf $t_1 = 100$ Monate geschätzt, d.h. $I_{\text{Dev}} = 100 I_1$. Im Entwurf wird eine Architektur entwickelt, die zu einer Abstimmungskomplexität $C_1 \in \Omega(n \log(n))$ führt. Weiterhin besteht die Möglichkeit die Architektur des Vorgängerprojekts zu übernehmen, welche zu einer Abstimmungskomplexität $C_2 \in \Omega(n^2)$ führen würde. Die Abstimmungszeit t_C wird auf den Erfahrungswert von einem Monat geschätzt. D.h., es wird davon ausgegangen, dass über das Projekt verteilt ungefähr ein Monat zur Abstimmung einer Architekturabhängigkeit benötigt wird. Auf Basis dieser fiktiven Werte sollen Überlegungen verdeutlicht werden, die mit Hilfe der hier vorgestellten Sätze möglich sind.

Abbildung 4.8 zeigt mögliche Verläufe der Projektdauer (links) und der Gesamtkosten (rechts) für die beiden Architekturvarianten. Die Grafik zeigt, dass das Projekt mit quadratischer Abstimmungskomplexität mit einem kleinen Team schneller zum Ziel kommt als das überlineare Projekt. Die kürzest mögliche Projektlaufzeit des quadratischen Projekts ist 13,6 Monate. Dazu bräuchte man 14 Entwickler. Das Projekt würde dann insgesamt 191 Entwicklermonate kosten. Mit der anderen Architekturvariante wäre man erst mit 15 Entwicklern und mehr schneller. Z.B. wäre mit 25 Entwicklern eine Projektlaufzeit von 12,2 Monaten möglich. Allerdings würde diese Variante

Tabelle 4.3: Vergleich verschiedener Kosten-Zeit-Varianten für die beiden Architekturvarianten des Beispielprojekts

Variante	**Abstimmungs-komplexität** C	**Entwickler** n	**Projektdauer** t	**Gesamtkosten** $n \cdot t$
minimale Kosten	$C_1(1) = C_2(1) = 0$	1	100	100
minimale Zeit	$C_1 \in \Omega(n \log(n))$	30	12	361
Kompromiss	$C_2 \in \Omega(n^2)$	8	16	128

305 Entwicklermonate kosten. Ein guter Kompromiss aus möglichst geringer Projektlaufzeit und akzeptablen Kosten wäre z.B. ein Projekte mit 8 Entwicklern. Das $\Omega(n^2)$-Projekt wäre dann in 16,0 Monaten mit Kosten in Höhe von 128 Entwicklermonaten zu bewältigen. Das $\Omega(n \log(n))$-Projekt wäre bei 8 Entwicklern etwas langsamer (17,7 Monate) und leicht teurer (142 Entwicklermonate). Tabelle 4.3 stellt verschiedene Varianten des Beispiels noch einmal gegenüber.

Insgesamt lässt sich feststellen, dass hohe Abstimmungskomplexität zu einem hohen Kommunikationsoverhead führt. D.h., es geht viel Zeit für abstimmende Kommunikationsaktivitäten verloren und es bleibt wenig Zeit für die eigentliche Entwicklungsarbeit. Das Beispiel verdeutlicht auch, dass eine Möglichkeit die Abstimmungskomplexität zu reduzieren darin besteht, die Arbeit auf möglichst wenige Entwickler zu verteilen (vgl. Tabelle 4.3 Variante minimale Kosten). Ist dies nicht möglich, z.B. wenn ein fixer Liefertermin feststeht, dann sollte das Ziel des Entwurfs sein, eine Architektur zu schaffen, die die Abstimmungskomplexität möglichst nah an die untere Komplexitätsschranke nach Satz 4.10 bringt.

Eine Einschränkung, die bei Anwendung der hier vorgestellten Formeln für die Planung von Softwareprojekten beachtet werden muss, ist, dass diese nur die reine Softwareentwicklung betrachten, ausgehend von bekannten Anforderungen. Der Aufwand, der durch Anforderungserhebung und Lernen der Anwendungsdomäne sowie durch Lernen neuen SE-Wissens entsteht (vgl. [Brooks 1995]) sind in diesen Formeln nicht berücksichtigt und müssen gesondert betrachtet werden.

5 Prüfung der Informationsflusstheorie

Nachdem die Grundbegriffe der Theorie in Kapitel 3 und die Theoreme der Theorie in Kapitel 4 vorgestellt wurden, wird die Theorie in diesem Kapitel einer Prüfung unterzogen. Nach der Validitätsprüfung wird im folgenden Kapitel 6 auf Basis der Informationsflusstheorie die FLOW-Methode als ein Beispiel zur praktischen Nutzbarmachung der Theorie erarbeitet.

Nach [Popper 2002, S. 7-8] lassen sich vier Richtungen unterscheiden, nach denen eine Prüfung einer Theorie durchgeführt wird:

1. Prüfung auf innere Widerspruchsfreiheit
2. Prüfung der wissenschaftlichen Form (z.B.: Sind die Sätze minimal? D.h., wiederholen sie sich nicht in anderer Form?)
3. Prüfung des wissenschaftlichen Fortschritts (z.B. durch Vergleich mit anderen Theorien des Forschungsbereichs)
4. Empirische Anwendung der aus der Theorie abgeleiteten Folgerungen (z.B.: Lassen sich aus der Theorie hergeleitete Prognosen in der Realität wiederfinden?)

Neben diesen allgemeinen Kriterien der Güte einer Theorie gibt es noch spezielle Anforderungen an Theorien für die Softwareentwicklung. So sagen z.B. Ludewig und Lichter [Ludewig 2010], Easterbrook et al. [Easterbrook 2008] und Broy [Broy 2011], dass die präzise Definition zentraler Begriffe im Software Engineering besonders wichtig ist:

> *Der präzise Umgang mit den Begriffen ist im Software Engineering besonders wichtig.* [Ludewig 2010, S. 40]

> *A good theory precisely defines the theoretical terms* [Easterbrook 2008]

> *Software engineering also deals with numerous abstract concepts that are often hard to understand because of the imprecise terms used to describe them* [...]. *This is why theory matters – it helps determine and evaluate the concepts that provide the basis for identifying terminology and developing engineering methods.* [Broy 2011]

Präzise Begriffe sind also auch Voraussetzung für die Entwicklung neuer Verbesserungsmethoden. Zudem ermöglicht eine präzise definierte Theorie Entwicklung, Durchführung und Auswertung von empirischen Studien. Folgende Zitate zeigen, dass das gerade im Software Engineering noch nicht ausreichend geschieht.

> *By defining the key terms, the results of empirical studies can be compared.*
>
> [...]
>
> *Software Engineering researchers have traditionally been very poor at making theories explicit [Joergensen 2004]. Many of the empirical studies conducted over the past few decades fail to relate the collected data to an underlying theory. The net result is that results are hard to interpret, and studies cannot be compared.* [Easterbrook 2008]
>
> [...] *we believe that an increased emphasis on theory can have a major impact on empirical investigations by providing a mechanism whereby results become more cumulative.* [Herbsleb 2003]
>
> *A mechanistic model* [or theoretical model], *supposedly backed by the nature of the system under study and verified by means of experimentation, is a much stronger position than a model obtained empirically and not backed by the theory of the phenomenon.* [Juristo 2001, S. 43]

Nach Boehm sollte eine Software-Engineering-Theorie die Bereiche Informatik, Management von Softwareprojekten und persönliche, kulturelle und ökonomische Werte, die bei der Entwicklung von Softwaresystemen eine Rolle spielen, abdecken [Boehm 2006]. Zudem wird nach Boehm eine solche Theorie wegen des immer vorhandenen menschlichen Faktors weniger formal sein, als Theorien aus anderen Wissenschaften (z.B. der Mathematik) [Boehm 2006].

Nach Jacobson und Meyer sollte eine Software-Engineering-Theorie methodenunabhängig sein und Theoreme über fundamentale Zusammenhänge liefern, um eine grundlegende Theorie der Softwareentwicklung zu sein [Jacobson 2009].

> *The model should be method-independent (describing the problem, not the solution); and it should not only include definitions and axioms but – the part too often missing in formal models – theorems stating fundamental properties of all systems and all viable methods.* [Jacobson 2009]

Auf innere Widerspruchsfreiheit und wissenschaftliche Form wurde bei der Formulierung der Theorie in den Kapiteln 3 und 4 geachtet. Zudem bilden die Definitionen des 3. Kapitels eine präzise Terminologie. Wie diese bei der Planung, Durchführung, Auswertung und Interpretation von empirischen Studien genutzt werden können, wird beispielhaft anhand einer für diese Arbeit durchgeführten empirischen Studie in Kapitel 5.2 gezeigt. Durch eine Interpretation der Ergebnisse ausgewählter aktueller Studien aus der Software-Engineering-Literatur aus Sicht der Informationsflusstheorie wird in Kapitel 5.3 die Erreichung des eingangs gesetzten Ziels eine grundlegende Theorie der Softwareentwicklung zu schaffen (vgl. Kapitel 1.2.3) überprüft. Die Prüfung des wissenschaftlichen Fortschritts und der externen Widerspruchsfreiheit wird anhand eines Vergleichs der Informationsflusstheorie mit verwandten Software-Engineering-Theorien in Kapitel 5.1 durchgeführt. D.h., der Vergleich mit verwandten Theorien zeigt nicht nur Neuerungen und Unterschiede, sondern auch, dass die Informationsflusstheorie nicht im Widerspruch zu etablierten SE-Erkenntnissen steht.

5.1 Vergleich mit verwandten Theorien der Softwareentwicklung

In diesem Unterkapitel wird die Informationsflusstheorie mit verwandten Theorien des Software Engineering verglichen. Einerseits werden Gemeinsamkeiten und eventuell vorhandene Widersprüche diskutiert, um zu prüfen, ob die Theorie im Widerspruch zu etablierten Software-Engineering-Erkenntnissen steht. Andererseits werden Unterschiede herausgearbeitet, um zu prüfen, ob die Informationsflusstheorie einen wissenschaftlichen Fortschritt darstellt. Dazu wird jede verwandte Theorie kurz allgemein mit ihren Kernideen beschrieben. Danach werden Gemeinsamkeiten und Unterschiede im Vergleich zur Informationsflusstheorie diskutiert.

Als verwandt werden Theorien mit folgenden Eigenschaften betrachtet:

- Sie sind durch ihren Titel als Software-Engineering-Theorie gekennzeichnet. In diese Kategorie fallen folgende Theorien:
  - *Theory-W Software Project Management* [Boehm 1989]
  - *Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering* [Herbsleb 2003]
  - *An Initial Theory of Value-Based Software Engineering* [Boehm 2006]
  - *A Theory of Shared Understanding for Software Organizations* [Aranda 2010]
- Sie sind durch Theoreme (d.h. Hypothesen, Sätze, Gesetze) gekennzeichnet, die zur Verbesserung von Softwareentwicklungsvorhaben eingesetzt werden können. In diese Kategorie fallen folgende Theorien:
  - *The Mythical Man-Month. Essays on Software Engineering* [Brooks 1974, Brooks 1995]
  - *Media, Tasks, and Communication Processes: A Theory of Media Synchronicity* [Dennis 1999, Dennis 2008]
  - *The Laws of Software Process: A New Model for the Production and Management of Software* [Armour 2003]
  - *The Triptych Process Model – Process Assessment and Improvement* [Bjoerner 2006]

- *Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity* [Cataldo 2008]
- *Agile Software Development. The Cooperative Game* [Cockburn 2007]

Die im Folgenden vorgestellten Theorien sind in der Reihenfolge ihrer Erstveröffentlichung sortiert.

5.1.1 The Mythical Man-Month

Das Buch "The Mythical Man-Month" [Brooks 1995] beschreibt nicht direkt eine Theorie des Software Engineering. Es ist ein Bericht von Brooks Softwareentwicklungserfahrungen, die er zwischen 1964 und 1965 als Manager des IBM OS/360 Projekts gesammelt hat. In dem Buch versucht er die Erkenntnisse aus seinen Erfahrungen anderen Softwareentwicklern und Managern mitzuteilen. Dabei formuliert er auch Gesetze und Hypothesen. Die wohl bekannteste Aussage aus dem Buch ist *Brook's Law*, welches er bewusst provokativ einfach wie folgt formuliert:

> *Adding manpower to a late software project makes it later.* [Brooks 1995, S. 25]

Die eigentliche Aussage, die sich hinter diesem Gesetz verbirgt, ist, dass es keinen linearen Zusammenhang zwischen Anzahl von Entwicklern und Projektlaufzeit gibt (vgl. Abb. 5.1 links), da das Problem der Softwareentwicklung nicht in unabhängige Teile zerlegt werden kann, die ohne zusätzlichen Abstimmungsaufwand parallel erledigt werden können. Es ist also immer Kommunikation notwendig, um die Arbeiten verschiedener Entwickler miteinander abzustimmen (vgl. Abb. 5.1 Mitte). Je mehr Entwickler an einem Projekt beteiligt sind, desto mehr Kommunikationsoverhead ist notwendig. Bei komplexen Aufgaben kann es sogar passieren, dass das Projekt mit steigender Entwickleranzahl sogar wieder länger dauert (vgl. Abb. 5.1 rechts). Die Schlussfolgerung von Brooks ist, dass der Entwicklermonat[1] als Metrik für den Projektaufwand nur ein Mythos ist. Dieses zentrale Argument hat dann auch zur Namensgebung des Buches geführt.

In einem zusätzlichen Kapitel der 20-jährigen Jubiläumsausgabe des Buchs hat Brooks nochmal seine wichtigsten Hypothesen zusammengefasst [Brooks

[1] d.h. die Arbeit, die ein Entwickler in einem Monat leisten kann, engl. man-month

1995, Kap. 18]. Aus diesen 202 Theoremen sollen hier nur kurz einige für den Vergleich mit der Informationsflusstheorie relevante genannt werden:

Programm-System-Produkt: Die Erstellung eines Programm-System-Produkts, also einer getesteten, leicht erweiterbaren, wartbaren und gut dokumentierten Software, die aus mehreren Komponenten besteht, ist ungefähr neunmal aufwendiger als die Erstellung eines undokumentierten, ungetesteten Einzelcomputerprogramms für den Privatgebrauch.

Der Mythos Mannmonat: "Der Entwicklermonat ist ein trügerischer und gefährlicher Mythos, weil er impliziert, dass Entwickler und Monate austauschbar sind" [Brooks 1995, S. 231].

Kommunikationsoverhead: Die Aufteilung von Aufgaben auf mehrere Entwickler verursacht zusätzlichen Kommunikationsaufwand: Schulungs- und Abstimmungskommunikation.

Brook's Law: "Adding manpower to a late software project makes it later" [Brooks 1995, S. 232] (vgl. Abb. 5.1 rechts).

Neue Entwickler: Das Hinzufügen neuer Entwickler zu einem laufenden Projekt erhöht den Gesamtaufwand aus drei Gründen: (1) Zusatzaufwand und Unterbrechungen durch die notwendige Neustrukturierung des Projekts, (2) Schulung der neuen Entwickler und (3) zusätzlicher Abstimmungsaufwand.

Kleine Teams: Ein kleines präzises Team ist am besten – so wenig verschiedene Köpfe wie möglich

Architektur: Eine einfache, klare, konsistente und gut dokumentierte Architektur des Systems mindert den Abstimmungsaufwand.

Kommunikation: Teams sollten vielfältig miteinander kommunizieren: informell, in regelmäßigen Projektmeetings, in technischen Briefings und anderen gemeinsamen formalen Dokumenten.

Organisation: Die Aufgabe von Organisationen ist die Reduktion von Abstimmungsaufwand. Baum-artige Organisationsstrukturen funktionieren in der Softwareentwicklung nicht so gut, weil dort meist Graph-artige Kommunikationsabhängigkeiten existieren.

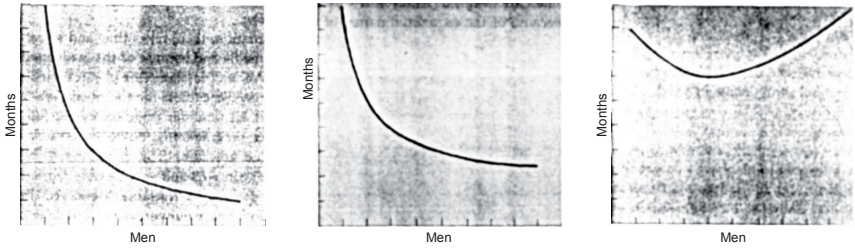


Abb. 5.1: Entwicklungsdauer versus Anzahl Softwareentwickler für perfekt aufteilbare Aufgabe (links), Aufgabenaufteilung, die Kommunikation erfordert (Mitte) und Aufgabe mit komplexen Wechselbeziehungen (rechts), aus [Brooks 1995, S. 16-19]

Dokumentation: Dokumentation ist wichtig für jedes Softwareprojekt. Der Vorgang des Schreibens fordert hunderte Mini-Entscheidungen, die zu klaren diskutierbaren Ergebnissen führen. Zudem können Dokumente zur Fortschrittskontrolle und als Warnmechanismus genutzt werden.

Software löst Nutzerprobleme: Der Programmierer befriedigt eher ein Nutzerbedürfnis als ein greifbares Produkt zu liefern (nach [Cosgrove 1971]).

Flexible Organisationen: Um auf Änderungen, die während eines Softwareprojekts immer auftreten können, flexibel reagieren zu können, sollte auch die Organisation des Projekts flexibel sein.

Software ist komplex: Softwaresysteme gehören wohl zu den komplexesten Dingen – im Sinne der Anzahl verschiedener Arten von Teilen – die die Menschheit baut.

Bezug zur Informationsflusstheorie Insgesamt gibt es sehr viele Überschneidungen und Ähnlichkeiten zwischen den Theoremen aus The Mythical Man-Month und der Informationsflusstheorie:

- Die Komplexität der Aufgabe hat Einfluss auf den Abstimmungsaufwand (vgl. Satz 4.10).

- Der Abstimmungsaufwand hat Einfluss auf die Projektlaufzeit (vgl. Satz 4.12).
- Eine einfache Architektur kann den Abstimmungsaufwand mindern (vgl. Kapitel 4.4).
- Informelle, direkte und formale Kommunikation ist wichtig für den Erfolg von Softwareentwicklung (vgl. Kapitel 3.7.2).
- Softwareentwicklung ist Problemlösen (vgl. Def.3.39).
- Dokumentation in der Softwareentwicklung ist Konkretisierung von Informationen (hunderte Mini-Entscheidungen schränken den Lösungsraum ein, siehe oben, vgl. Kapitel 3.7.3 und Abb. 3.34).

Mit Hilfe der Theoreme aus Kapitel 4.4 und dem Konzept der Abstimmungskomplexität lassen sich die von Brooks skizzierten Projektlaufzeiten (vgl. Abb. 5.1) aus der Informationsflusstheorie herleiten. Abbildung 5.2 zeigt einen Vergleich der Gesamtprojektlaufzeit in Abhängigkeit von der Anzahl der eingesetzten Entwickler für unterschiedlich komplexe Probleme. Es ist zu erkennen, dass z.B. Projekte mit einer Abstimmungskomplexität in $\Omega(n^2)$ ab einer bestimmten Anzahl von Entwicklern insgesamt länger dauern, obwohl die Arbeit auf mehr Entwickler verteilt werden kann.

Weiterhin kann mit Hilfe der Informationsflusstheorie eingeschränkt werden, auf welche Art von Softwareprojekten Brook's Law (vgl. Abb. 5.1 rechts) zutrifft. Alle Projekte, die ein Problem mit überlinearer Komplexität lösen müssen, haben die Eigenschaft, dass sich ab einer bestimmten Anzahl von Entwicklern beim Hinzufügen weiterer Entwickler die Gesamtprojektlaufzeit verlängert, statt sich durch die Arbeitsteilung weiter zu verkürzen. Der Punkt, an dem sich das Hinzufügen von mehr Entwicklern nicht mehr lohnt, lässt sich mit der Formel aus Satz 4.12 bestimmen: $t = \frac{t_1 + C(n) \cdot t_C}{n}$. Dazu muss zunächst die Funktion für die Abstimmungskomplexität $C(n)$ in die Formel eingesetzt werden. Anschließend bestimmt man die Nullstellen über die erste Ableitung dieser Funktion. Allgemein gilt:

$$0 = \left(\frac{t_1 + C(n) \cdot t_C}{n} \right)'$$

Am Beispiel einer quadratischen Abstimmungskomplexität mit $C(n) = n(n-1)/2 \in \Omega(n^2)$ (vgl. [Brooks 1995, S. 18]) sieht das wie folgt aus:

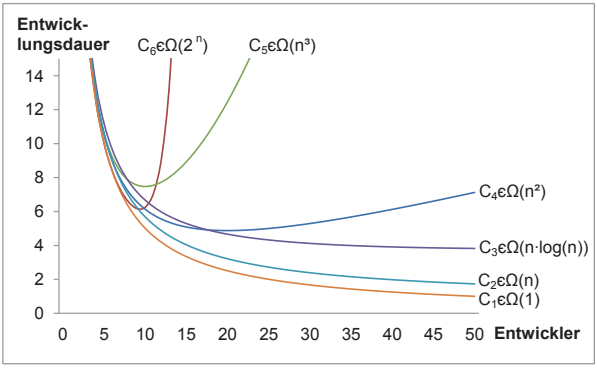


Abb. 5.2: Entwicklungsdauer in Abhängigkeit von der Anzahl der für die Softwareentwicklung eingesetzten Entwickler für unterschiedlich komplexe Probleme (in Anlehnung an Darstellung von [Brooks 1995, S. 16-19])

$$
\begin{aligned}
t &= \frac{t_1 + C(n)\cdot t_C}{n} \\
&= \frac{t_1 + n(n-1)/2\cdot t_C}{n} \\
&= n^{-1}t_1 + \frac{t_C}{2}n - \frac{t_C}{2} \\
t' &= -n^{-2}t_1 + \frac{t_C}{2} \\
\Rightarrow 0 &= -n^{-2}t_1 + \frac{t_C}{2} \\
\Leftrightarrow n &= \sqrt{\frac{2t_1}{t_C}}
\end{aligned}
$$

Setzt man z.B. für $t_1 = 50$ und $t_C = \frac{1}{4}$ ein, dann erhält man mit $n = 20$ die Entwickleranzahl, ab der das Projekt mit der Abstimmungskomplexität $C_4 \in \Omega(n^2)$ aus Abbildung 5.2 wieder länger dauert.

Abgrenzung zur Informationsflusstheorie Zu den wesentlichen Unterschieden zur Informationsflusstheorie gehört, dass der Fokus in The Mythical Man-Month auf dem Management von Softwareprojekten liegt, wohingegen er bei der Informationsflusstheorie stärker auf dem Wesen von Softwareentwicklung liegt. Zudem ist die Informationsflusstheorie systematischer, formaler und baut stärker auf Erkenntnissen aus der Literatur auf. Brooks listet zwar 202 Theoreme auf, liefert aber keine grundlegenden Definitionen. Er benutzt vielmehr die natürliche Sprache und einige wenige SE-Termini, um seine Hypothesen zu formulieren. Die Hypothesen stammen hauptsächlich aus einer einzigen Projekterfahrung und wenigen Software-Engineering-Publikationen. Die Informationsflusstheorie kann hingegen auf 40 Jahre mehr Software-Engineering-Literatur zurückgreifen. Das zeigt sich z.B. auch darin, dass einige der damals adressierten Dinge heute gelöst sind. So ist z.B. der Einsatz von Hochsprachen und Versionsverwaltungssystemen heute Standard. Brooks sagt zwar, dass Kommunikation ein wesentlicher Teil der Softwareentwicklung ist, er beschreibt aber nicht so detailliert wie die Informationsflusstheorie unter welchen Umständen wie kommuniziert werden sollte (vgl. z.B. Medienwahl in Kapitel 4.3).

5.1.2 Theory W

Die Theory W ist eine Softwareprojektmanagementtheorie, die Boehm als konsequente Weiterentwicklung der Managementtheorien Theory X, Y und Z sieht [Boehm 1989]. Nach Theory X ist der effizienteste Weg, eine Aufgabe zu erledigen, die Aufgabe in eine wohl-orchestrierte Sequenz von Aktivitäten aufzuteilen, innerhalb der der Mensch so effizient und berechenbar wie eine Maschine arbeitet. Management hält dieses System hauptsächlich durch Zwang am Laufen. Theory X ist eine schlechte Langzeitstrategie, da sie die Kreativität der Mitarbeiter hemmt. Daher wurde in Theory Y postuliert, dass das Management Kreativität und Einzelunternehmertum stimulieren sollte, um die Unternehmen langfristig adaptiv zu halten. Durch die vielen individuellen Initiativen, die um begrenzte Ressourcen konkurrierten, kam es in durch Theory Y gemanagten Unternehmen zu mehr Konflikten. In Theory Z sollten diese Konfliktpotentiale durch die frühe Entwicklung gemeinsamer Werte und Entscheidungsfindung durch Konsens vermieden werden. Theory Z konzentriert sich dabei auf die Organisationsebene.

In der Softwareentwicklung gibt es aber durch die vielen unterschiedlichen Sta-

keholder wie Entwickler, Kunden, Nutzer, Wartungspersonal etc. auch Konfliktpotentiale auf anderen Ebenen. Theory W geht dieses Problem an, indem es Anleitungen liefert, die beschreiben, wie Softwareprojektmanager es schaffen, aus allen Beteiligten "Gewinner" ("Theory W: Make Everyone a Winner" [Boehm 1989]) zu machen. D.h. *"statt das Management als Autokrat (Theory X), als Coach (Theory Y) oder als Moderator (Theory Z) zu charakterisieren, charakterisiert Theory W die primäre Aufgabe des Managers als Vermittler zwischen seinen verschiedenen Kundenkreisen und als Präsentator von Projektlösungen mit Gewinn-Konditionen für alle Beteiligten."* [Boehm 1989]. Das Hauptziel des Managers ist es also, Konfliktpotentiale aufzuspüren und möglichst oft in Win-Win-Situationen umzuwandeln.

Boehm räumt ein, dass es in der Softwareentwicklung durchaus schwierig sein kann, immer Win-Win-Situationen zu schaffen. Sein Ansatz, Win-Win-Situationen zu schaffen, basiert auf Arbeiten aus dem Bereich der Verhandlungsführung. Folgende Schritte sollen Win-Win-Vorbedingungen schaffen und sowohl den Softwareentwicklungsprozess als auch die Software selbst strukturieren (übersetzt aus [Boehm 1989]):

1. Win-Win-Vorbedingungen etablieren
    a) Verstehe, wie Menschen gewinnen wollen;
    b) Etabliere angemessene Erwartungen;
    c) Passe die Aufgabe den Gewinnkonditionen der Leute an;
    d) Schaffe eine unterstützende Umgebung;
2. Einen Win-Win-Entwicklungsprozess strukturieren
    a) Schaffe einen realistischen Prozessplan;
    b) Nutze den Plan, um das Projekt zu steuern;
    c) Identifiziere und manage Win-Lose- und Lose-Lose-Risiken;
    d) Beteilige Leute durchgängig;
3. Ein Win-Win-Softwareprodukt strukturieren
    a) Passe das Produkt den Gewinnkonditionen von Nutzern und Wartungspersonal an;

Dabei sind nach Boehm die ersten drei Unterschritte (2.a, 2.b, 2.c) für die Softwareentwicklung besonders wichtig. Sie werden zusätzlich zum Hauptprinzip "Make everyone a winner" in den folgenden zwei Nebenprinzipien hervorgehoben (übersetzt aus [Boehm 1989]):

1. Plane den Flug und fliege den Plan (2.a und 2.b).
2. Identifiziere und manage deine Risiken (2.c).

Pläne wie Projektpläne, Testpläne oder Konfigurationsmanagementpläne dokumentieren die gegenseitige Verpflichtung auf Schaffung und Einhaltung einer Menge von Win-Win-Bedingungen aller Projektbeteiligten und sie sind die Basis zum Aufspüren von Abweichungen von Win-Win-Bedingungen und zur Einleitung von Gegenmaßnahmen durch das Projektmanagement. Gegenmaßnahmen sollten unter Anwendung der Schritte aus der obigen Aufzählung versuchen, für alle Beteiligten Win-Win-Situationen zu schaffen, oder, falls das nicht möglich ist, die Verluste zu minimieren und alle Beteiligten davon zu überzeugen, dass diese Verluste im Vergleich zu anderen Gegenmaßnahmen minimal sind. Risikomanagement ist in der Softwareentwicklung wichtig, um sicherzustellen, dass die Pläne realistisch sind. Risikomanagement lenkt die Aufmerksamkeit des Projektmanagers auf die Stellen des Projekts, an denen Verletzungen der Win-Win-Bedingungen am wahrscheinlichsten sind. Die Schritte und Prinzipien der Theory W bieten spezifische Hilfestellung sowohl bei taktischen als auch bei strategischen Projektmanagementaufgaben [Boehm 1989].

Zusammenfassend kann laut Boehm das Hauptprinzip der Theory W ("make everyone a winner") und die beiden Nebenprinzipien ("plan the flight and fly the plan" und "identify and manage your risks") genutzt werden, um Erklärungen für Probleme eines gegebenen Softwareentwicklungsprojekts zu liefern und um Möglichkeiten zur Vermeidung dieser Probleme herzuleiten. Theory W ist eine Projektmanagementtheorie mit Fokus auf die Besonderheiten von Softwareentwicklungsprojekten, d.h. insbesondere die große Anzahl von Stakeholdern mit unterschiedlichen Zielen. Neben den grundlegenden Schritten und Prinzipien verweist Boehm auf bekannte Techniken des Software Engineering (z.B. Reviews, Versionsverwaltung) und Projektmanagements (z.B. Risikomanagement), um Win-Win-Situationen zu erreichen. Damit liefert die Theory W ein Framework, mit dem *"man die meisten scheinbar unverbundenen Software-Management-Aktivitäten herleiten kann […]"* (übersetzt aus [Boehm 1989]).

Bezug zur Informationsflusstheorie Aus Sicht der Informationsflusstheorie könnte man das Ziel der Theory W wie folgt beschreiben: Softwareentwicklung ist charakterisiert durch viele verschiedene Stakeholder (vgl. Kapitel 3.7.4 und Tabelle 3.7) die alle verschiedene Ziele verfolgen (vgl. Kapitel 3.6.2). Theory W beschreibt eine Managementstrategie, die das Ziel verfolgt, die Kommunikation in der Softwareentwicklung zu verbessern, indem in Konflikt stehende Ziele der Stakeholder in kollaborative Ziele umgewandelt werden sollen oder ganz ausgeräumt werden sollen (Kooperation). Boehm geht davon aus, dass Softwareprojekte mit weniger Konflikten zwischen den Teilnehmern erfolgreicher sind. Dies deckt sich mit Vorhersagen der Informationsflusstheorie. Nach den Hypothesen 4.3, 4.6 und Satz 4.7 haben hohe Zielunterschiede der Kommunikationsteilnehmer einen negativen Einfluss auf den Kommunikationserfolg. Nach Kapitel 3.7 beruht Softwareentwicklung zu großen Teilen auf Kommunikation. Damit haben hohe Zielunterschiede auch einen negativen Einfluss auf den Erfolg von Softwareentwicklung. Umgekehrt führt eine Verminderung der Zielunterschiede zu einer Verbesserung der Softwareentwicklung.

Abgrenzung zur Informationsflusstheorie Die Informationsflusstheorie liefert also eine Begründung für die hinter der Theory W stehenden Grundannahmen. Damit ist die Informationsflusstheorie grundlegender und komplementär zur Theory W. Das Hauptprinzip und die beiden Nebenprinzipien der Theory W können unter der Grundannahme als drei Theoreme aufgefasst werden. Boehm liefert in der Theorie nur eine Definition, die von Gefährdungspotnezial (engl. risk exposure). Alle anderen Zusammenhänge werden in natürlicher Sprache und ein wenig SE-Terminologie erläutert. Theory W geht auf die Besonderheiten der Softwareentwicklung ein, indem die typischen vielen verschiedenen Stakeholder berücksichtigt werden. Wie sich diese unterscheiden und was diese Unterschiede für Auswirkungen auf die Entwicklung von Software haben, wird nicht betrachtet. Die Informationsflusstheorie stellt hingegen z.B. den typischen Wissensunterschied zwischen Entwicklern (Softwaredomäne) und Nutzern (Anwendungsdomäne) heraus, den es während der Entwicklung durch Kommunikation zu überwinden gilt (vgl. Kapitel 3.7).

5.1.3 Media Synchronicity Theory

Die Media Synchronicity Theory ist keine Theorie des Software Engineering. Sie findet dort aber Anwendung, z.B. in der Forschung über die Kommunikation in global verteilten Softwareentwicklungsprojekten (u.a. [Niinimaeki 2010]). Da sich ein Großteil der Informationsflusstheorie mit dem Thema Kommunikation und Medienwahl beschäftigt (vgl. Kapitel 4.2 und 4.3) und dieses Thema insbesondere in der global verteilten Softwareentwicklung eine große Rolle spielt, wird die Media Synchronicity Theory dennoch als verwandte Theorie einbezogen.

Auf die Media Synchronicity Theory (MST) [Dennis 2008] wurde bereits in den Grundlagen eingegangen (vgl. Kapitel 2.5.4). Die Grundidee der Media Synchronicity Theory ist, dass sich alle Gruppenkommunikationsprozesse aus Kombination der Grundkommunikationsprozesse *Übermittlung* und *Fokussierung* zusammenstellen lassen. Ein Zusammenhang zwischen der zu lösenden Aufgabe und Kommunikationserfolg wird über die Mediensynchronität hergestellt. Mediensynchronität ist als der Grad zu dem Individuen zur selben Zeit an der selben Aufgabe arbeiten, d.h. einen gemeinsamen Fokus haben, definiert [Dennis 1999]. Medien haben fünf Eigenschaften, die Einfluss auf die Mediensynchronität haben. Dies sind Feedback, Parallelität, Symbolvielfalt, Verifizierbarkeit und Wiederverarbeitbarkeit (vgl. Kapitel 2.5.4). In der Neuauflage der Theorie von 2008 [Dennis 2008] werden sieben Theoreme über die Auswirkung der Medienwahl unter Berücksichtigung der fünf Eigenschaften und der Grundkommunikationsprozesse formuliert.

P1: Das erste Theorem ist gleichzeig die Hauptaussage der MST. Der Kommunikationserfolg hängt davon ab, wie gut Mediensynchronität und Grundkommunikationsprozess zusammen passen.

   a) Für den Grundkommunikationsprozess der Übermittlung führt ein Medium mit *niedriger* Synchronität zu effizienterer Kommunikation.

   b) Für den Grundkommunikationsprozess der Fokussierung führt ein Medium mit *hoher* Synchronität zu effizienterer Kommunikation.

P2: Die Geschwindigkeit, mit der über ein Medium *Feedback* gegeben werden kann, erhöht den gemeinsamen Fokus und hat daher positiven Einfluss auf die Mediensynchronität.

P3: *Parallelität*, d.h. die Anzahl voneinander unabhängiger Kommunikationen, die gleichzeitig über ein Medium geführt werden können, vermindert den gemeinsamen Fokus und hat daher negativen Einfluss auf die Mediensynchronität.

P4: Natürliche und passende Symbolmengen haben einen positiven Einfluss auf die Mediensynchronität:

a) Medien mit *natürlichen Symbolmengen* (physisch, visuell, verbal) haben eine höhere Synchronität als Medien mit weniger natürlichen Symbolmengen (geschrieben).

b) Medien mit *Symbolmengen*, die gut zum Inhalt der Kommunikation passen, verbessern Informationsübertragung und -verarbeitung, und erhöhen daher die Mediensynchronität.

P5 Verifizierbarkeit senkt den gemeinsamen Fokus und hat daher negativen Einfluss auf die Mediensynchronität.

P6 Wiederverarbeitbarkeit senkt den gemeinsamen Fokus und hat daher negativen Einfluss auf die Mediensynchronität.

P7 Neben den Grundkommunikationsprozessen (vgl. P1) haben auch die *Vertrautheit* der Kommunikationsteilnehmer miteinander, mit der Aufgabe und mit dem genutzten Medium Einfluss auf den Bedarf an Mediensynchronität.

a) Zusammenarbeit auf Basis etablierter Normen, an bekannten Aufgaben und mit vertrauten Medien hat einen geringen Bedarf an hoher Mediensynchronität.

b) Zusammenarbeit ohne etablierte Normen, an neuen unbekannten Aufgaben und mit ungewohnten Medien hat den größten Bedarf an hoher Mediensynchronität.

Da die meisten Kommunikationsaktivitäten sowohl aus Informationsübermittlungsprozessen als auch aus Informationsfokussierungsprozessen bestehen und die Medien, die sich gut für Übermittlung eigenen, meist nicht so gut für Fokussierung geeignet sind, und umgekehrt, schlussfolgert die Media Synchronicity Theory, dass die Nutzung eines einzigen Mediums oft nicht zum bestmöglichen Kommunikationsergebnis führt und Medienwechsel bei der Durchführung der meisten Aufgaben angebracht sind [Dennis 1999]. In manchen Situation kann auch eine Kombination von Medien sinnvoll sein [Karim 2005, Wat-

son 2007, Dennis 2008]. Welche Situationen das sind und wie Medien kombiniert werden sollten, wird allerdings nicht aufgeführt.

Eine Studie von DeLuca und Valacich über neu zusammengesetzte Teams, die eine Prozessverbesserungsaufgabe unter Nutzung von Medien niedriger Synchronität lösen sollten, konnte die meisten Theoreme der Media Synchronicity Theory belegen [DeLuca 2006].

Bezug zur Informationsflusstheorie Media Synchronicity Theory und Informationsflusstheorie weisen Ähnlichkeiten in den grundlegenden Medieneigenschaften auf. So ist z.B. die Möglichkeit schnelles Feedback zu geben eng verwandt mit der Latenz nach Informationsflusstheorie (vgl. Def. 3.31). Es wird auch zwischen natürlichen und künstlichen Symbolmengen bzw. Datentypen (vgl. Def. 3.8) unterschieden. Die Grenze zwischen natürlich und künstlich verläuft allerdings unterschiedlich: bei der MST zwischen gesprochen und geschrieben, bei der Informationsflusstheorie zwischen natürlichen und künstlichen (geschriebenen) Sprachen. Beide Theorien empfehlen das verwendete Medium an den Kommunikationsinhalt anzupassen (vgl. Satz 4.9). Zudem empfehlen beide Theorien unter Umständen auch mehrere Medien gleichzeitig zu nutzen.

Abgrenzung zur Informationsflusstheorie MST nennt zwar die Möglichkeit in bestimmten Situationen auch mehrere Medien parallel zu benutzen, führt aber nicht genauer auf, wie das geschehen soll. Im Gegensatz dazu besagt die Informationsflusstheorie, dass immer genau zwei Medien genutzt werden sollten, je eines für Inhaltsübermittlung und Steuerung der Kommunikation (vgl. Hypothese 4.1). Diese Trennung ist in der Softwareentwicklung besonders wichtig, weil der Inhalt meist nicht mit natürlichen Datentypen übertragen werden kann, z.B. Quellcode oder UML-Diagramme. Steuerung der Kommunikation über diese Inhalte muss aber wieder mit natürlichen Datentypen erfolgen, weil der Mensch das mit Hilfe von künstlichen Datentypen üblicherweise nicht beherrscht (vgl. Hypothese 4.2). Ein weiterer Unterschied ist die Beschreibung der Eigenschaften der Kommunikationsaktivität und ihre Auswirkungen auf die Medienwahl. MST charakterisiert Aufgaben anhand zweier Grundkommunikationsprozesse. Diese haben Einfluss auf die Anforderungen an die Mediensynchronität (vgl. P1 oben). Die Informationflusstheorie charakterisiert die Aufgabe anhand von Ziel, Inhalt und Ziel- und Wissensunterschieden der Teilnehmer (vgl. Kapitel 4.3). Jede dieser Eigenschaften hat

direkten Einfluss auf bestimmte Kanaleigenschaften (vgl. Abb. 4.5). Bei der MST werden die Aufgabeneigenschaften hingegen zunächst auf eine Metrik, die Mediensynchronität, reduziert. Auf Basis der Synchronität wird dann ein Medium gewählt. Die Informationsflusstheorie ermöglicht daher eine differenziertere Betrachtung bei der Medienwahl. Da die MST nur einen Teil der für die Softwareentwicklung relevanten Bereiche abdeckt, hat sie auch weniger Theroeme (7) und Definitionen (12).

5.1.4 The Cooperative Game

Cockburn definiert Softwareentwicklung wie folgt:

> *People inventing, deciding, communicating,*
>
> solving a problem *they don't yet understand (which keeps changing)*
>
> creating a solution *they don't yet understand (which keeps changing)*
>
> expressing ideas *in languages they don't understand (which keep changing)*
>
> *to an interpreter unforgiving of error*
>
> making decisions *with limited resources*
>
> *and every choice has economic consequences.* [Cockburn 2009]

Demnach ist Softwareentwicklung das Lösen von unbekannten und sich ständig ändernden Problemen in einem ökonomischen Kontext indem Menschen erfinden, Entscheidungen treffen und kommunizieren. Die folgenden vier Punkte zeichnen nach Cockburn [Cockburn 2009] moderne Softwareentwicklung aus:

- Softwareentwicklung ist ein kooperatives Spiel
- Softwareentwicklung ist ein Handwerk
- Schlanke Prozesse sind wichtig
- Entwicklung ist Wissenserwerb

Im Folgenden werden diese vier Punkte näher betrachtet.

Softwareentwicklung als kooperatives Spiel Nach Cockburn ist Softwareentwicklung vergleichbar mit einem kooperativen Spiel [Cockburn 2001, Cockburn 2007]. Kooperative Spiele sind endlich und zielgerichtet. Das Ziel ist nur durch Zusammenarbeit der Spieler erreichbar. In der Softwareentwicklung gibt es zwei in Konflikt stehende Unterziele: die Software zu liefern und sich auf das nächste Spiel vorzubereiten (z.B. ausreichend Dokumentation für Wartung). Die drei möglichen Züge der Spieler in diesem Spiel sind:

1. erfinden
2. entscheiden
3. kommunizieren

Die Situationen, mit denen die Spieler konfrontiert werden, wiederholen sich kaum oder nie, sodass es schwierig ist, Strategien zu entwickeln. Je nach Projektgröße und -kritikalität sind aber unterschiedliche Vorgehensweisen bei der Entwicklung sinnvoll. Insbesondere bei einer großen Anzahl von Entwicklern sind andere Mechanismen zur Abstimmung notwendig, als bei kleinen Teams. Prinzipiell sind reichhaltige Medien mit Feedback-Möglichkeit zur Kommunikation vorzuziehen. Entfernung zwischen den Entwicklern macht die Kommunikation ineffizienter und damit das Projekt teurer. Soziale Aspekte bestimmen maßgeblich die Geschwindigkeit eines Projekts [Cockburn 2009]:

- Können die Entwickler erkennen, wenn etwas ihre Aufmerksamkeit erfordert?
- Interessiert es sie genug, dass sie etwas dagegen unternehmen?
- Können sie die Informationen effektiv weitergeben?

Softwareentwicklung als Handwerk Die Sicht auf Softwareentwicklung als Handwerk stellt die Fähigkeiten der Entwickler und das Medium (Software) in den Vordergrund. Wichtige Handwerke der Softwareentwicklung sind [Cockburn 2009]:

- Entscheiden, was entwickelt werden soll
- Management (Menschen und Projekte)
- Modellierung
- Entwicklung der externen Sicht

- Grobentwurf (Architektur)
- Feinentwurf (Programmierung)
- Die Arbeit validieren

Die Fähigkeiten dieser Handwerke erlernt ein Entwickler in drei – dem asiatischen Kampfsport entlehnten – Phasen [Cockburn 2001, S. 17]:

Shu: Eine einzelne spezielle Technik lernen und diese Schritt für Schritt ausführen können.

Ha. Viele Techniken sammeln und je nach Situation die passende Technik anwenden können.

Ri: Übergeordnete Prinzipien hinter den Techniken erkennen und mit diesen Prinzipien eigene Techniken für neue Situationen entwickeln können.

Softwareentwicklung und schlanke Prozesse Softwareentwicklung ist wie Produktion (im Gegensatz zur Projektarbeit), wenn man als Produktionseinheit die nicht-validierte Entscheidung betrachtet. Abbildung 5.3 zeigt einen Ausschnitt des Flusses von Entscheidungen durch ein Softwareprojekt. Die Nutzer entscheiden, welche Funktionen sie wünschen. Die Programmierer entscheiden, wie die Funktionen umgesetzt werden. Die Tester entscheiden, ob die umgesetzten Funktionen den geforderten entsprechen usw. Feedback-Schleifen stellen sicher, dass Probleme und Einschränkungen in zukünftigen Entscheidungen berücksichtigt werden können. Die Prinzipien des kontinuierlichen Flusses und der Kontrolle der Entscheidungs-Warteschlangen aus der schlanken Produktion sollen helfen, Softwareprojekte am Laufen zu halten.

Softwareentwicklung und Wissenserwerb Ein wichtiger Teil der Softwareentwicklung ist Wissenserwerb. Das zu lösende Problem, die eigene Lösung und die Sprachen und Werkzeuge zur Lösungsumsetzung werden zu Anfang eines Projekts nicht richtig verstanden und müssen erst erlernt werden. Zudem ändern sich Problem, Lösung und Werkzeuge während eines Projekts meist. Daher schlägt Cockburn vor, möglichst früh im Projekt in Wissenserwerb zu investieren und so Risiken zu minimieren. Dies kann z.B. durch frühe, kontinuierliche Integration geschehen (vgl. Abb. 5.4 links). Wenn genug Wissen erworben wurde und die Risiken minimiert wurden, kann der eigentliche Geschäftswert geschaffen werden. Weiterhin erlaubt diese Vorgehensweise, dass

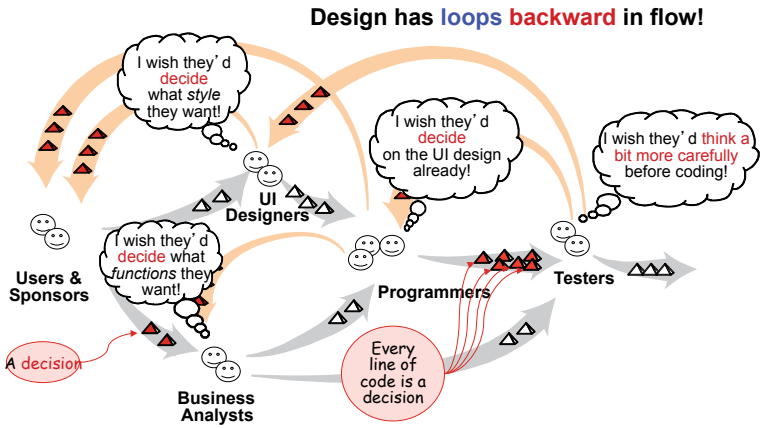


Abb. 5.3: Softwareentwicklung als Fluss von Entscheidungen (aus Cockburns "Short Theory of Designing in Teams" [Cockburn 2010a])

man gegen Ende eines Projekts die Freiheit hat, zu entscheiden, ob man pünktlich oder noch etwas mehr Zeit investieren und dafür mehr Geschäftswert liefern möchte.

Bezug zur Informationsflusstheorie Auch Cockburn [Cockburn 2007] hat Wissensunterschiede auf Ausbildungs- und Domänenebene (vgl. Kapitel 3.6.1) als Ursache für Projektprobleme identifiziert: *"The Cultural Gap Theme: 'Our user interface designers all have Ph.D.s in psychology and sit together several floors above the programmers. There is an educational, a cultural, and a physical gap between them and the programmers.'"* [Cockburn 2007, S. 232]. Insbesondere die Sicht auf Softwareentwicklung als Produktionsprozess mit Flüssen von Entscheidungen (vgl. Abb. 5.3), kommt der Informationsflusstheorie sehr nahe, wenn man Entscheidungen als eine Art von Informationen betrachtet (vgl. Kapitel 3.7.2 und FLOW-Notation in Kapitel 6.1.3). Cockburns Meinung, dass man in der Softwareentwicklung zunächst so viel wie möglich über das Problem lernen sollte, bevor man damit beginnt, Werte in Form von Softwarelösungen zu schaffen, findet sich in der Informationsflusstheorie in Korollar 4.3 wieder. Aus Korollar 4.3 folgt, dass man möglichst zu Anfang des Projekts die Wissensunterschiede zwischen den Entwicklern und insbesondere auch zwischen Entwicklern und Stakeholdern aus der Anwendungsdomäne minimie-

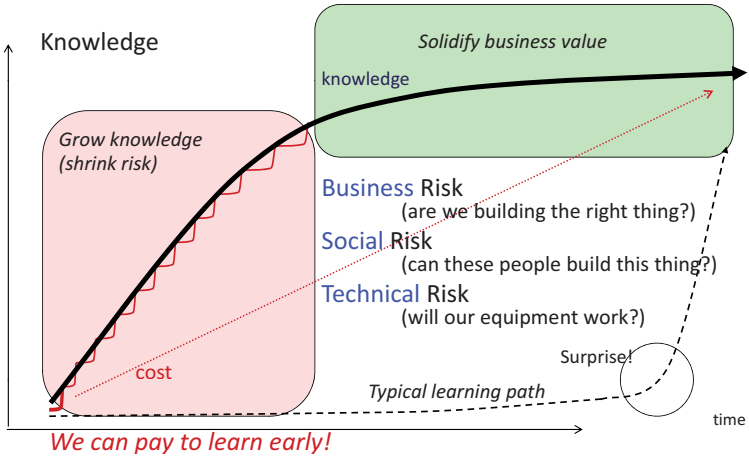


Abb. 5.4: Früh lernen, spät Werte schaffen vs. früh Werte schaffen, spät lernen (nach Cockburns "Knowledge Acquisition Theory", aus [Cockburn 2009])

ren sollte. D.h., es zahlt sich im weiteren Verlauf des Projekts durch bessere Kommunikationseffizienz aus, wenn die Entwickler am Anfang des Projekts versuchen, die Anwendungsdomäne – zumindest teilweise – zu erlernen. Insgesamt sieht Cockburn ähnliche Aspekte wie die Informationsflusstheorie als zentral für die Softwareentwicklung an, z.B. Menschen, Kommunikation, Wissenserwerb und (Entscheidungs-) Flüsse.

Abgrenzung zur Informationsflusstheorie Die Theorie nach Cockburn vereint vier Sichtweisen auf Softwareentwicklung, die unabhängig voneinander zu sein scheinen. Zentrale Begrifflichkeiten wie Entscheidungen und Wissen werden nicht klar definiert. Die zentralen Konzepte werden auch nicht in einen Gesamtzusammenhang gebracht. Zum Beispiel bleibt unklar, wie genau Cockburn den Zusammenhang zwischen Entscheidungsflüssen, Wissenserwerb und Kommunikation sieht. Cockburns Theorie ist genereller als die Informationsflusstheorie, weil er z.B. auch Motivationsaspekte integriert. Die Informationsflusstheorie hingegen ist umfangreicher, weil sie mehr Zusammenhänge auch auf niedrigerem Abstraktionsniveau erklärt. Zählt man die Vorträ-

ge von Cockburn als Teil seiner Theorie der Softwareentwicklung, dann kann man zwei Definitionen von Softwareentwicklung identifizieren (eine Langfassung, vgl. oben, und eine Kurzfassung, vgl. [Cockburn 2009]). Die Theorie enthält keine expliziten Theoreme. Die implizite Annahme scheint zu sein, dass Softwareentwicklung erfolgreicher ist, wenn sie mit den Prinzipien, die sich aus den vier unterschiedlichen Sichweisen ergeben, durchgeführt wird.

5.1.5 The Laws of Software Process

> *"Software is knowledge."* [Armour 2003, S. xv]

In seinem Buch "The Laws of Software Process" [Armour 2003] beschreibt Armour seine Sicht auf die Softwareentwicklung. Seiner Meinung nach ist Softwareentwicklung so problematisch, weil viele Unternehmen Software als Produkt sehen, anstatt als Medium für das eigentliche Produkt: Wissen [Armour 2003]. Software ermöglicht Wissen ausführbar zu machen und somit Probleme zu lösen. Das Produkt der Softwareentwicklung ist nicht direkt die Software, sondern die Erlangung des dafür notwendigen Wissens. D.h. Softwareentwicklung ist Wissenserwerb, oder umgekehrt, die Verminderung von Unwissenheit [Armour 2003, S. 235]. Armour unterscheidet das in der Softwareentwicklung zu beschaffende Wissen nach dem Grad der Unbekanntheit in fünf Kategorien. Die fünf Grade der Unbekanntheit nennt er Grade der Ignoranz (engl. Five Orders of Ignorance) [Armour 2003, S. 7].

0. Grad (0OI): Das Fehlen von Ignoranz: 0OI ist erreicht, wenn Wissen nachweisbar vorhanden ist, z.B. das Wissen über eine Programmiersprache nachgewiesen durch erfolgreiche Anwendung dieser Sprache.

1. Grad (1OI): Das Fehlen von Wissen: 1OI ist das bewusste Fehlen von Wissen. 1OI ist die normale Ignoranz. Z.B. kann einem Programmierer bewusst sein, dass er eine bestimmte Programmiersprache nicht beherrscht.

2. Grad (2OI): Das Fehlen von Bewusstsein: 2OI ist das fehlende Bewusstsein über das Fehlen von Wissen. Ein Beispiel für den Einfluss von 2OI auf die Softwareentwicklung ist die Schwierigkeit der Aufwandschätzung. Man kann zwar den Aufwand zur Erlangung fehlenden Wissens einplanen (1OI), aber nicht für das fehlende Wissen, von dem man noch nicht weiß, das es fehlt (2OI). Gute Schätzungen enthalten daher immer noch einen Puffer.

3. Grad (3OI): Das Fehlen von Prozess: 3OI ist das Fehlen eines Prozesses zur Erlangung des Wissens darüber, dass Wissen fehlt. Viele Prozesse der Softawreentwicklung sind 3OI Prozesse. Sie sollen Softwareentwicklern die Möglichkeit geben, herauszufinden, was sie nicht wissen, das sie nicht wissen. D.h. Softwareentwicklungsprozesse liefern kein direkt nutzbares Wissen, sondern helfen nur die richtigen Fragen zu stellen. [Armour 2000]

4. Grad (4OI): Meta-Ignoranz: 4OI liegt vor, wenn man die fünf Grade der Ignoranz nicht kennt.

Basierend auf der Grundidee der fünf Grade der Ignoranz formuliert Armour einige Gesetzmäßigkeiten für Softwareentwicklungsprozesse, die helfen sollen, das Wesen der Softwareentwicklung besser zu verstehen [Armour 2003, Chp. 1]. Diese Gesetze sind von Armour bewußt humorvoll formuliert, was aber nichts an ihrem Wahrheitsgehalt ändert [Armour 2003].

Das erste Softwareprozessgesetz: Prozesse ermöglichen uns nur das zu tun, von dem wir schon wissen, wie es geht.

Korollar des ersten Softwareprozessgesetzes: Du kannst keinen Prozess für etwas haben, was du noch nie getan hast oder von dem du nicht weißt, wie es geht.

Die reflexive Erstellung von Systemen und Prozessen:

1. Die einzige Möglichkeit effektive Systeme zu bauen ist die Anwendung effektiver Prozesse.

2. Die einzige Möglichkeit effektive Prozesse zu erstellen ist die Konstruktion effektiver Systeme.

Lemma der ewigen Verspätung: Die einzigen Prozesse, die wir für das aktuelle Projekt einsetzen können, wurden in früheren Projekten definiert, die anders als das jetzige waren.

Das zweite Softwareprozessgesetz: (vgl. Regel der Prozessverzweigung) Wir können Softwareprozesse nur in zwei Stufen definieren: zu vage oder zu einengend.

Die Regel der Prozessverzweigung: Softwareprozessregeln werden üblicherweise in zwei Stufen beschrieben: eine allgemeine Darstellung der Regel

und ein spezifisches detailliertes Beispiel (z.B. das zweite Softwareprozessgesetz).

Die Doppelhypothese des Wissenserwerbs:

1. Hypothese Eins: Wir können Wissen nur in einer Umgebung erwerben, die dieses Wissen enthält.

2. Hypothese Zwei: Die einzige Möglichkeit die Validität von Wissen festzustellen, ist es mit einer anderen Wissensquelle zu vergleichen.

Armours Beobachtung über Softwareprozesse: Was alle Entwickler wirklich wollen ist eine rigorose, eiserne, engstirnige, konkrete, universelle, absolute, ganzheitliche, endgültige und vollständige Menge von Prozessregeln, die sie missachten können.

Das dritte Softwareprozessgesetz: (auch bekannt als das Schuhfabrikantenangehörigkeitsgesetz) Die letzte Wissensart, die für die Implementierung in ein ausführbares Softwaresystem in Betracht gezogen wird, ist das Wissen, wie man Wissen in ein ausführbares Softwaresystem implementiert.

Die Zwillingsziele der optimalen Auflösung:

1. Das einzige natürliche Ziel eines Softwareprozessverbesserungsteams sollte sein, sich selbst so schnell wie möglich überflüssig zu machen.

2. Das Endergebnis einer andauernden Entwicklung und Anwendung effektiver Prozesse ist, dass sie niemand wirklich benutzen muss.

Armour schlussfolgert aus diesen Grundeigenschaften von Prozessen in der Softwareentwicklung, dass die Softwareentwicklungsindustrie folgendes in Bezug auf Softwareentwicklungsprozesse tun sollte [Armour 2003]:

- Es sollte zwischen bekannten (0OI, 1OI) und unbekannten (2OI, 3OI) Softwareentwicklungsaktivitäten unterschieden werden, und für jede unterschiedliche Arten und Granularitäten von Prozessen definiert werden.

- Es sollte nicht mehr versucht werden monolithische Prozesse zu definieren.

- Prozesse sollten nur aus echten Entwicklungserfahrungen heraus entwickelt werden, um bewiesenermaßen echte Probleme zu lösen.
- Die Einschränkungen früherer Prozesse sollten erkannt werden und es sollte daraus gelernt werden.
- Prozesse, die 2OI angehen, sollten kreativen Freiraum vorsehen.
- Zusätzlich zum kreativen Freiraum sollten "Prozesslabore" eingeführt werden, um verschiedene Prozessmöglichkeiten zu erkunden.
- Prozessverbesserungsgruppen sollten zeitlich befristet und klar definierte Abschlusskriterien haben.
- Es sollten Metriken entwickelt werden, mit denen der tatsächliche Nutzen eines Prozesses nachgewiesen werden kann.
- Erstelle Wissensmanagementsysteme (beides, in ausführbarer und lesbarer Form) und stelle diese zur Verfügung.
- So viel wie möglich automatisieren. Nicht nur das Zielsystem sollte autmoatisiert werden, sondern auch das Prozesswissen, das auf dem Weg dorthin erlangt wurde.

Weiterhin beschreibt Armour, wie sich der Aufmerksamkeitsfokus im Laufe eines Projekts entwickelt. Abbildung 5.5 zeigt einen Zusammenhang des Aufmerksamkeitsfokus (Locus of attention) zwischen Aufgaben auf Projekt- und Aktivitätsebene. Die Aussage der Abbildung ist, dass der Aufmerksamkeitsfokus während der Entwicklung zwar stark schwanken kann, z.B. zwischen High-Level Anforderungen und detailliertem Design, sich aber im Mittel (Trendlinie) gegen Projektende immer mehr der konkreten Lösung widmet.

Bezug zur Informationsflusstheorie Abbildung 5.5 zeigt, dass auch Armour die Konkretisierung und den Domänenübergang von Informationen (abstrakte Anforderungen → konkrete Lösung, Problem- und Lösungsdomäne) als charakteristische Eigenschaft der Softwareentwicklung sieht (vgl. Kapitel 3.7.3). Weiterhin stellt Armour heraus, dass SE-Methoden meist keine Antworten, sondern Fragen liefern. D.h. sie helfen nicht Arbeit zu minimieren, sondern sie führen zu mehr Arbeit [Armour 2000]. Diese Sicht kann durch die Informationsflusstheorie noch ergänzt werden. Demnach ist ein wesentlicher Teil der Softwareentwicklung das Lernen der Anwendungsdomäne und

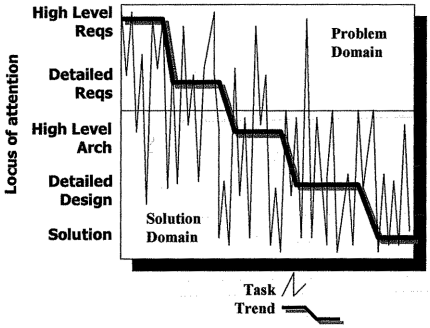


Abb. 5.5: Zusammenhang des Aufmerksamkeitsfokus (Locus of attention) zwischen Aufgaben auf Projekt- und Aktivitätsebene aus [Armour 2003, S. 71], nach [Curtis 1988]

der Anforderungen. Zu Beginn eines Projekts in einer neuen Anwendungsdomäne kann den Entwicklern noch nicht klar sein, was sie alles über die Domäne wissen müssen, um das Nutzerproblem tatsächlich lösen zu können (2OI). 3OI Methoden helfen dann, in der richtigen Projektphase die richtigen Personen zu fragen. Z.B. helfen die kurzen Iterationen und das Feedback des On-Site Customers im Extreme Programming [Beck 2000] Anforderungen zu identifizieren, die man so nicht erwartet hätte, sofern er wirklich aus der Anwendungsdomäne kommt. Anders ausgedrückt, SE-Methoden können nur SE- oder Prozess-Wissen liefern, aber nicht Anwendungsdomänenwissen (vgl. Abb. 3.10). Folgendes Zitat von Armour offenbart eine weitere Ähnlichkeit zur Informationsflusstheorie.

> *The problem arises when we think the code, rather than the knowledge in the code, is the product.* [Armour 2000]

Demnach ist es problematisch für die Softwareentwicklung, wenn die Entwickler den Quellcode und nicht das Wissen, dass in diesem Quellcode steckt, als das Produkt ihrer Arbeit sehen. Die darin enthaltene Unterscheidung zwischen Wissen und Daten (Quellcode) spiegelt sich in der Informationsflusstheorie im Informationsbegriff wieder. Daten alleine sind nicht hilfreich, erst mit dem richtigen Kontext werden sie zur wervollen Information (vgl. Def. 3.11). Auch die Idee, dass Software ein Nutzerproblem löst, ist die gleiche (vgl. Def. 3.39).

Abgrenzung zur Informationsflusstheorie Die Informationsflusstheorie ist formaler, umfangreicher und grundlegender als das Modell von Armour. Fasst man die Beschreibungen von Software als Wissensmedium, von Softwareentwicklung als Wissenserwerb und jede Stufe der Ignoranz als Definitionen auf, so liefert Armour sieben Definitionen und zehn Theoreme. Armour fokussiert sich starkt auf den Zusammenhang von Prozessen und Softwareentwicklung. Er sagt nicht viel darüber aus, wie Softwareentwicklung konkret besser gemacht werden kann (außer z.B. kreativen Freiraum einräumen und Wissensmanagement betreiben). Im Gegensatz zur Informationsflusstheorie sagt er nicht viel darüber aus, wie das Wissen in der Softwareentwicklung erlangt werden kann, z.B. durch Kommunikation. Er sagt zwar, dass es einen Unterschied zwischen Daten und Wissen gibt, führt aber nicht aus, worin der genau besteht. Armour definiert den zentralen Begriff Wissen nicht explizit. Für ihn ist Wissen etwas, das in etwas anderem (z.B. Daten, Software) enthalten ist, und für etwas sinnvolles genutzt werden kann [Armour 2003, S. 207].

5.1.6 Empirical Theory of Coordination

Die Grundannahme der Empirical Theory of Coordination (ETC) nach Herbsleb und Mockus ist, dass Koordination der Ingenieursarbeit der Schlüssel für erfolgreiches Software Engineering ist [Herbsleb 2003]. Dabei spielt gute Kommunikation eine wichtige Rolle. Die ETC geht davon aus, dass es eine einzige (sehr große, aber endliche) Menge von Entscheidungen gibt, die ein Softwareprojekt ausmachen. Softwareentwicklung kann dann als eine Kombination von Belegungen dieser Entscheidungen, die den Anforderungen des Projekts genügen, gesehen werden. Dabei werden Zwischenwerte, d.h. die teilweise Erfüllung der Anforderungen, nicht berücksichtigt. Eine Kombination von Belegungen für eine Menge Entscheidungen erfüllt die Anforderungen oder erfüllt sie nicht. Durch Festlegung von Belegungen für Entscheidungen schreitet Softwareentwicklung voran. Nach jeder Belegung bleiben weniger Entscheidungen übrig, bis alle Entscheidungen getroffen (d.h. belegt) wurden und das resultierende Produkt die Anforderungen erfüllt oder nicht erfüllt. Während der Entwicklung können Belegungen rückgängig gemacht und durch andere ersetzt werden. Entscheidungen können sich gegenseitig beeinflussen und einschränken. Die Belegung einer Entscheidung limitiert meist die Wahl bei anderen Entscheidungen. Verletzt die Wahl einer Entscheidung eine Einschränkung, so müssen Belegungen rückgängig gemacht werden, oder die Anforderungen können nicht mehr erfüllt werden. Das Problem der Koordination in der Softwa-

reentwicklung ist dann, zwischen den vielen Entwicklern sicherzustellen, dass die gegenseitigen Einschränkungen der getroffenen Entscheidungen erkannt werden und darauf korrekt reagiert wird.

Herbsleb und Mockus leiten mit Hilfe einer mathematischen Formulierung dieser Gedanken zwei bekannte "Gesetzmäßigkeiten" des Software Engineering her [Herbsleb 2003]:

1. **Das Prinzip der Modularität:** Die Partitionierung von Entscheidungen in nicht-überlappende Module. Ein Modul bietet Information Hiding (vgl. [Parnas 1972]), wenn Entscheidungen außerhalb des Moduls keinen Einfluss auf Entscheidungen innerhalb des Moduls haben. Partitionierungen von Entscheidungen in Module können z.B. durch Architekturen erzeugt werden.

2. **Conway's Law:** Eine weitere Partitionierung von Entscheidungen entsteht in großen Projekten durch Organisationseinheiten wie Sub-Teams oder Individuen, die Entscheidungen mit relativ wenigen Auswirkungen auf andere Organisationseinheiten treffen. Conway's Law besagt nun, dass sich die Kommunikationsstruktur der Organisation in der Struktur des entwickelten Systems wiederspiegelt (vgl. [Conway 1968]). D.h. Partitionierungen von Entscheidungen, die aus Organisationseinheiten entstanden sind, finden sich im System wieder (z.B. durch Module).

Herbsleb und Mockus nennen diese zwei Prinzipien als Beispiele, wie man die Theorie nutzen kann, um bekannte Zusammenhänge im SE auszudrücken. So stellen sie z.B. einen Zusammenhang zwischen Kommunikation und Modularität nach Parnas [Parnas 1972] wie folgt her:

> *Modules, or "work items" as Parnas defined them to be, address how work may be split among teams in a way that does not impose unreasonable requirements for coordination and communication among teams.* [Herbsleb 2003]

Basierend auf der Idee der ETC kann also der Zweck von modularer Softwarearchitektur mit Information Hiding zwischen den Modulen als Minimierung der Abstimmungsaufwände zwischen den Entwicklern, die an den Modulen arbeiten, beschrieben werden.

In einer in dem gleichen Papier vorgestellten Studie werden aus diesen Prinzipien zwei Hypothesen hergeleitet und geprüft [Herbsleb 2003]:

1. Entwickler, die von vielen Personen Arbeit zugewiesen bekommen, haben eine niedrigere Produktivität.
2. Die Dauer der Bearbeitung von Änderungsaufträgen, die Änderungen an verschiedenen Modulen erfordern, ist größer als bei Änderungsaufträgen, die Änderungen an nur einem Modul erfordern.

Die Analyse der Studie liefert Unterstützung für beide Hypothesen [Herbsleb 2003].

In einem weiteren Papier von Herbsleb et al. [Herbsleb 2006] wird die Idee der ETC erweitert und weitere Hypothesen aufgestellt und geprüft. Die nicht reduzierbaren gegenseitigen Abhängigkeiten von Entscheidungen im Software Engineering werden als verteiltes Bedingungserfüllungssystem (engl.: Distributed Contraint Satisfaction Problem (DCSP)) beschrieben. Demnach kann Koordination als die Ausführung eines "Algorithmus" gesehen werden, der ein DCSP löst. In DCSPs sind Entscheidungen in ein Netzwerk von Abhängigkeiten eingebunden und werden von potenziell vielen Agenten getroffen. Eine Lösung des DCSP erfordert dann eine Zusammenarbeit dieser Agenten, d.h. Entwickler. Herbsleb et al. nennen folgende SE-Phänomene, die mit der Theory of Coordination erklärt werden können [Herbsleb 2006]:

- (Produkt-) Architektur hat einen enormen Einfluss auf Kommunikationsmuster. [Sosa 2004]
- Änderungen an der Architektur können die Effektivität der Koordination von Entwicklungsarbeit ernsthaft beeinträchtigen. [Henderson 1990]
- Die Struktur eines Systems ist ein Abbild der Kommunikationsstruktur der Organisation, die das System entwickelt hat. [Conway 1968]

Die folgenden Hypothesen werden auf Basis des DCSP-Ansatzes aus [Herbsleb 2006] aufgestellt:

H1: Hohe Dichte von Abhängigkeiten erhöht die Dauer von Änderungen am Quellcode.

H2: Arbeit im Rahmen von hoher Abhängigkeitsdichte produziert wahrscheinlich mehr Fehler als Arbeit im Rahmen niedriger Abhängigkeitsdichte.

H3: Arbeit im Rahmen von hoher Abhängigkeitsdichte erfordert mehr Aufwand als Arbeit im Rahmen niedriger Abhängigkeitsdichte.

H4: Die Verteilung von Entscheidungen mit starker gegenseiter Abhängigkeit auf viele Personen erhöht die Dauer von Änderungen am Quellcode.

H5: Die Verteilung von Entscheidungen mit starker gegenseiter Abhängigkeit auf viele Personen erhöht die Anzahl von Fehlern.

H6: Die Verteilung von Entscheidungen mit starker gegenseiter Abhängigkeit auf viele Personen verursacht mehr Arbeitsaufwand als bei der Verteilung von Entscheidungen mit starker gegenseiter Abhängigkeit auf wenige Personen.

In der in [Herbsleb 2006] durchgeführten Studie konnte Unterstützung für H1, H4, H5 und H6 gefunden werden. Zur Unterstützung von H2 und H3 konnten mit den genutzten Metriken keine eindeutigen Ergebnisse gefunden werden. Für H1 und H4 konnte gezeigt werden, dass sowohl hohe Abhängigkeitsdichte als auch die Verteilung von stark voneinander abhängigen Entscheidungen auf viele Entwickler zu signifikant längerer Entwicklungszeit führen. Umso mehr Entwickler an einer Datei arbeiten, desto mehr Fehler enthält diese später im Betrieb (H5). H6 konnte belegt werden, indem gezeigt wurde, dass Arbeit an Quellcode, zu dem viele Entwickler beigetragen haben, signifikant weniger produktiv war als Arbeit an Quellcode, zu dem wenige Entwickler beigetragen haben.

Bezug zur Informationsflusstheorie Der Ausgangspunkt der ETC findet sich auch in der Informationsflusstheorie wieder: Abhängigkeiten zwischen Arbeitspakten führen zu Abstimmungbedarf (vgl. Kapitel 4.4). Auch nach ETC spielt Architektur eine wichtige Rolle bei der klaren Aufteilung und Strukturierung des zu lösenden Problems, sodass es von vielen Entwicklern in Zusammenarbeit gelöst werden kann. Eine klare von allen verstandene Architektur hilft dabei Entscheidungen so zu treffen, dass sie möglichst wenig in Konflikt mit Entscheidungen anderer stehen. Ähnlich ist, dass getroffene Entscheidungen den noch zur Verfügung stehenden Lösungsraum einschränken. D.h. auch nach ETC ist das charakteristische Merkmal der Softwareentwicklung, dass Informationen im Projekt immer konkreter werden (vgl. Kapitel 3.7.3).

Die Sätze der Informationsflusstheorie aus Kapitel 4.4 können genutzt werden, um zu zeigen, dass eine Architektur, die sich entsprechend Conway's Law an hierarchischen Organisationsstrukturen ausrichtet (vgl. Abb. 5.6), eine gute Idee ist, weil der Abstimmungsaufwand dann nur linear mit der Anzahl der

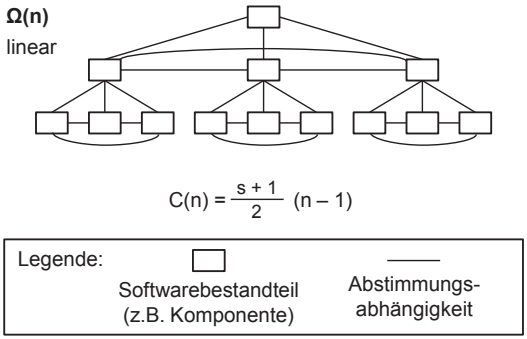


Abb. 5.6: Beispiel einer hierarchischen Softwarearchitektur, die zu linearer Abstimmungskomplexität führt

Softwareentwickler wächst, die gemeinsam an der Lösung eines Problems arbeiten. Ein Softwareprojekt, das die Entwicklung entsprechend Conway's Law [Conway 1968] in n Komponenten mit Abstimmungsabhängigkeiten entsprechend einer hierarchisch aufgebauten Organisation mit Hierachietiefe d und Verzweigungsgrad s auf n Entwickler aufteilt, hat die folgende Abstimmungskomplexität:

$$C(n) = \frac{s+1}{2}(n-1) \in \Omega(n)$$

Herleitung. Der Hierarchietiefe d gibt an, wieviele Ebenen eine Hierarchie hat. Der Verzweigungsgrad s gibt an, wieviele Kindknoten jeder Knoten maximal haben kann. Es wird angenommen, dass Abhängigkeiten zwischen Eltern- und Kindknoten bestehen und dass auf jeder Ebene alle Knoten von allen anderen Knoten mit dem selben Elternknoten abhängig sind. Die Abbildung 5.6 zeigt ein Beispiel einer solchen Hierarchie mit drei Ebenen $d = 3$ und Verzweigungsgrad $s = 3$.

Die Anzahl der Entwickler n ergibt sich wie folgt aus d und s:

$$
\begin{aligned}
n &= \sum_{x=0}^{d} s^x && \text{geometrische Reihe} \\
&= \tfrac{s^{d+1}-1}{s-1}, s \neq 1 \\
\Leftrightarrow s^{d+1} &= n(s-1)+1 \\
\Leftrightarrow (d+1)\log(s) &= \log(n(s-1)+1) \\
\Leftrightarrow d &= \tfrac{\log(n(s-1)+1)}{\log(s)} - 1 \\
&= \log_s(n(s-1)+1) - 1
\end{aligned}
$$

Für die Abstimmungskomplexität $C(n)$ gilt dann:

$$
\begin{aligned}
C(n) &= \sum_{x=0}^{d-1} s^x \cdot \sum_{x=1}^{s}(x-1) + (n-1) && \text{Sub-Graphen + Hierarchie} \\
&= \tfrac{s^d-1}{s-1} \cdot \tfrac{s(s-1)}{2} + (n-1) && \text{geom. und endliche Reihe} \\
&= \tfrac{s^{d+2}-s^{d+1}-s^2+s}{2(s-1)} + (n-1) \\
&= \tfrac{s^{d+1}(s-1)-s(s-1)}{2(s-1)} + (n-1) \\
&= \tfrac{s^{d+1}-s}{2} + (n-1) && \text{mit } d = \log_s(n(s-1)+1) - 1 \\
&= \tfrac{s^{\log_s(n(s-1)+1)}-s}{2} + (n-1) \\
&= \tfrac{(n(s-1)+1)-s}{2} + (n-1) \\
&= \tfrac{ns-n-s+1}{2} + (n-1) \\
&= (n-1)\tfrac{s-1}{2} + (n-1) \\
&= \tfrac{s+1}{2} \cdot (n-1) \in \Omega(n)
\end{aligned}
$$

□

Damit ist gezeigt, dass hierarchische Organisationsstrukturen zu linearer Abstimmungskomplexität führen können, wenn das Problem auch eine lineare Komplexität hat (vgl. Satz 4.10) und die Lösung nach Conway's Law entsprechend der Organisationsstruktur aufgeteilt wird.

Abgrenzung zur Informationsflusstheorie Die ETC liefert eine Erklärung, warum Information Hiding [Parnas 1972] und Software, die entsprechend Conway's Law strukturiert ist, sich positiv auf den Abstimmungsaufwand bei der Softwareentwicklung auswirken. Wie oben beschrieben, kann diese Erklärung mit Hilfe der Informationsflusstheorie noch erweitert werden. Die Abstimmungskomplexität in hierarchischen Strukturen wächst nur linear mit der Anzahl der eingesetzten Entwickler, sofern nur Abhängigkeiten entlang der Hierarchieebenen und zwischen Arbeitsteilen innerhalb eines Teil-Baums bestehen. In dem Sinne ist die Informationsflusstheorie grundlegender als die ETC.

Die Informationsflusstheorie ist umfangreicher und weniger mathematisch als die ETC. Der Einfluss von Kommunikation zwischen Entscheidern auf die Wahrscheinlichkeit Entscheidungen zu treffen, die Einschränkungen aus bereits gefällten Entscheidungen *nicht* verletzen, wird auf die Kommunikationshäufigkeit (Frequenz) reduziert. Dauer, genutzte Medien oder Wissensunterschiede zwischen den Kommunikationsteilnehmern spielen keine Rolle. Wenn Entscheidungen als einziges Element zur Analyse von Softwareprojekten benutzt werden, können nur Effekte auf Grund von Abstimmungsproblemen gefunden werden (vgl. Kapitel 3.6.3 zum Unterschied zw. entscheiden und informieren). D.h., der wichtige Wissensgewinn zu Anfang eines Projekts (z.B. Anwendungsdomäne kennenlernen) kann damit nur schlecht oder gar nicht abgebildet werden. Dies sind aber nach Informationsflusstheorie (vgl. Kapitel 3.7) und nach Meinung anderer Softwareingenieure (vgl. u.a. [Curtis 1988, Armour 2003, Cockburn 2007]) relevante, da charakteristische Phasen der Softwareentwicklung, die einen großen Anteil an den noch heute anzutreffenden Problemen des Software Engineering (vgl. Kapitel 1.1) haben. Mit der von der Informationsflusstheorie vorgeschlagenen Betrachtung von Informationsflüssen, insbesondere von Kommunikation, als Kern der Softwareentwicklung, lassen sich sowohl die Phänomene erklären, die durch Koordination von Entscheidungen auftreten, als auch die Phänomene, die durch das Erlernen neuer Informationen entstehen. Herbsleb et al. nennen diese Einschränkung sogar explizit:

> *The focus of the theory is exclusively on coordinating the technical work.*
> [Herbsleb 2006]

In [Herbsleb 2006] wird zwar erwähnt, dass die Anforderungserhebung besonders anfällig für Koordinations- und Kommunikationsprobleme ist, es wird aber nicht näher erläutert, wie die Theorie dort helfen kann. Selbst wenn

man die tatsächliche Anforderung des Kunden als eine Belegung einer Entscheidung sieht und die falsche Interpretation dieser Anforderung durch den Entwickler als eine Belegung einer Entscheidung, die mit der ersten im Konflikt steht, hilft die Theorie nicht zu erklären bzw. vorzuschlagen, wie man diesen Konflikt erkennen und vermeiden kann. Demnach ist die Informationsflusstheorie grundlegender als die Theory of Coordination. Die Studien in [Herbsleb 2003, Herbsleb 2006] basieren auf Change Request Daten, d.h. es werden Wartungsprojekte analysiert und nicht Softwareentwicklungsprojekte. Die Theorie wird also nicht für die Softwareentwicklung, sondern für die Softwarewartung evaluiert.

Insgesamt können ETC und Informationsflusstheorie als komplementäre Theorien gesehen werden: Der übergeordnete Kommunikationsprozess vom Kunden zur finalen Software wird von der Informationsflusstheorie beschrieben; Phänomene der eigentlichen Entwicklungsarbeit (vgl. Dokumentation der Software) können durch die ETC beschrieben werden.

5.1.7 Value-Based Software Engineering

Nach [Boehm 2006] ist die Theory of Value-Based Software Engineering eine Kombination aus der Theory W (vgl. Kapitel 5.1.2), der Nutzentheorie, der Entscheidungstheorie, der Abhängigkeitstheorie und der Kontrolltheorie. Im Zentrum steht die Theory W. Die Kernaussage der Theory W ist nach [Boehm 2006] das Geschäftserfolgs-Theorem:

> *Enterprise Success Theorem:*
>
> *Your enterprise will succeed if and only if*
>
> *It makes winners of your success-critical stakeholders.* [Boehm 2006]

Demnach ist Softwareentwicklung nur dann erfolgreich, wenn alle erfolgskritischen Stakeholder zufrieden sind. Die vier anderen Theorien sollen helfen, Win-Win-Situationen zu schaffen und aufrechtzuerhalten. Welche Aufgabe jede der Theorien hat, lässt sich aus dem Win-Win-Erzielungs-Theorem ableiten:

> *Win-Win Achievement Theorem: Making winners of your success-critical stakeholders requires:*
>
> 1. *Identifying all of the success-critical stakeholders (SCSs).*

2. *Understanding how the SCSs want to win.*
3. *Having the SCSs negotiate a win-win set of product and process plans.*
4. *Controlling progress toward SCS win-win realization, including adaptation to change.*

[Boehm 2006]

Damit die erfolgskritischen Stakeholder zu Gewinnern werden können, müssen sie zunächst identifiziert werden. Dann muss verstanden werden, wie diese gewinnen wollen. Danach müssen die Stakeholder Pläne zur Erreichung von Win-Win-Situationen verhandeln. Abschließend muss der Fortschritt der Win-Win-Realisierung, inklusive Berücksichtigung von Änderungen, kontrolliert werden. Für jeden dieser vier Schritte wird der Einsatz einer speziellen Theorie vorgeschlagen. Zur Identifikation der erfolgskritischen Stakeholder wird die Abhängigkeitstheorie benutzt. Die Nutzentheorie hilft zu verstehen, wie Stakeholder gewinnen wollen. Den Einfluss von Werten der Stakeholder auf ihre Entscheidungen kann man mit Hilfe der Entscheidungstheorie bestimmen. Schließlich wird die Kontrolltheorie genutzt, um den Fortschritt zu prüfen und auf Änderungen reagieren zu können.

Zusammenfassend lässt sich feststellen, dass VBSE mit Hilfe der Theory W bekannte Theorien der Wirtschaftswissenschaften so zusammen bringt, dass sie im Kontext der Besonderheiten der Softwareentwicklung (insbesondere viele unterschiedliche Stakeholder) zur erfolgreichen Durchführung von Softwareprojekten genutzt werden können.

Bezug und Abgrenzung zur Informationsflusstheorie Da die Theory W im Mittelpunkt der VBSE steht, gelten auch die gleichen Gemeinsamkeiten und Unterschiede wie bei der Theory W (vgl. Kapitel 5.1.2). Durch die Integration von Theorien aus den Wirtschaftswissenschaften wird der Fokus auf wirtschaftliche und Projektmanagement-Zusammenhänge verstärkt. Menschen stehen durch die Theory W zwar im Mittelpunkt, diese beschreibt aber eher *was* für erfolgreiche Softwareentwicklung erreicht werden muss (Win-Win-Situationen zwischen allen Stakeholdern), aber nicht *wie* das geschieht. Eine zu Grunde liegende Annahme scheint zu sein, dass Stakeholder mit aufeinander abgestimmten und nicht in Konflikt stehenden Zielen (d.h. Win-Win-Situation) zu erfolgreicheren Softwareprojekten führen. D.h. unter Anderem auch, dass motivierte Entwickler erfolgreicher Software entwickeln. Was sie

zur eigentlichen Entwicklung tun müssen, ist nicht Teil der Theorie. Damit ist der Fokus der Theorie genereller und Projektmanagement-spezifischer als der der Informationsflusstheorie.

Beide Theorien können sich aber ergänzen: Die VBSE stellt sicher, dass die verschiedenen Werte aller relevanten Stakeholder bekannt und beachtet werden, und die Informationsflusstheorie hilft, die Kommunikation zwischen den Stakeholdern zu verbessern, sodass durch die Kombination beider Theorien die Wahrscheinlichkeit erfolgreicher Softwareentwicklung steigt.

5.1.8 The Triptych Process Model

Bjørner beschreibt in drei Bänden [Bjoerner 2006-1, Bjoerner 2006-2, Bjoerner 2006-3] einen sehr umfangreichen, sehr formalen und dokumentenzentrierten Softwareentwicklungsprozess. Bjørner nennt den Prozess Triptych Process Model [Bjoerner 2006, Bjoerner 2006-3, Chp. 31]. Er beschreibt das Wasserfall-artige, Plan-getriebene und möglicherweise in Iterationen ausgeführte Vorgehen wie folgt:

> *The triptych approach to software engineering proceeds on the basis of carefully monitored and controlled possibly iterated progression through domain engineering and requirements engineering to software design.* [Bjoerner 2006]

Triptych steht also für die drei Phasen Domänen-Engineering, Requirements Engineering und Softwaredesign (vgl. Abb. 5.7). Softwareentwicklung ist das Durchlaufen dieser drei Phasen in der vorgegebenen Reihenfolge [Bjoerner 2006-1]. Ergeben sich Änderungen in einer Phase können Prozesschritte in vorangegengenen Phasen wiederholt werden (vgl. REDOs in Abb. 5.7).

Ergebnis aller Phasen und Zwischenschritte sind Dokumente. Nach Bjørner sind Dokumente das zentrale Element der Softwareentwicklung, in denen sich Wissen und Fortschritt manifestiert. Während der Softwareentwicklung müssen sehr viele Dokumente erstellt werden. Die folgenden beiden Zitate unterstreichen den Stellenwert von Dokumenten in der Theorie von Bjørner:

> *Common to all three phases of software development are that they primarily manifest themselves in documents.* [...] [the figures] *illustrate the breadth, depth and quite substantial number of such resulting documents.* [Bjoerner 2006]

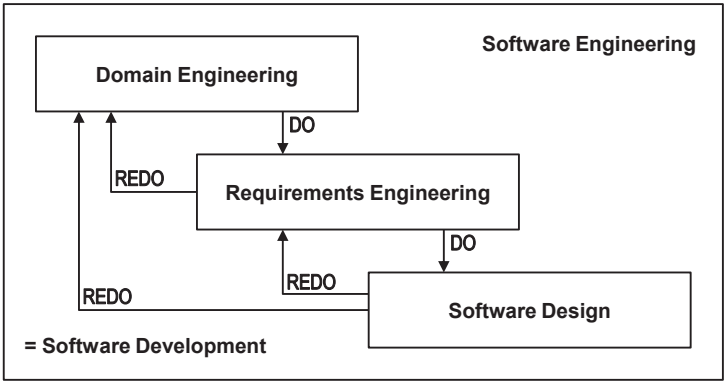


Abb. 5.7: Übersicht des Triptych Process Model aus [Bjoerner 2006]

> *There is nothing else* emanating from steps, stages and phases* [in software development] *than documents, on paper or electronically.*
>
> **Strictly speaking: Understanding also emerges, and so do closer relations between client (acquirer, customer) and developer (deliverer, provider), etcetera. But, contractwise, unless, for example, education and training is also part of a project, documents are the only tangible goods delivered!* [Bjoerner 2006-1, S. 14]

Bjørner gesteht zwar ein, dass im Softwareentwicklungsprozess auch Wissen weitergegeben wird ("understanding emerges"), dies ist aber aus Vertragssicht meist nur ein Nebenprodukt. Daher schließt er, dass Dokumente das einzige Ergebnis der verschiedenen Softwareentwicklungsaktivitäten sind. D.h. Softwareentwicklung besteht im Wesentlichen aus Dokumentationen. Nach Bjørner kann Verständnis nur über Dokumentieren erlangt werden, was foglendes Zitat zeigt:

> *The description, prescription or design work to be done in the phase* [...] *rely on assumptions and dependencies. These must be fully understood, hence documented before any proper development takes place.* [Bjoerner 2006]

Softwareentwicklung kann informell, formell oder mit einem beliebigen Zwischenwert der beiden Stufen verfolgt werden [Bjoerner 2006]:

0. **Informelle Entwicklung:** Informelle Entwicklung bedeutet, dass keine Formalisierung von Domänenbeschreibung, Anforderungsvorschrift und Softwarespezifikation vorgesehen sind. D.h. Verifikation kann nur durch

Testen und nicht durch formale Beweise oder Model Checking geschehen.

Auf der semi-formal zu formal Skala gibt es die folgenden drei Punkte:

1. **Systematische Entwicklung:** Systematische Entwicklung formalisiert Domänenbeschreibung, Anforderungsvorschrift und Softwarespezifikation. Mehr Formalisierung ist nicht vorgesehen.

2. **Rigorose Entwicklung:** Rigorose Entwicklung erweitert systematische Entwicklung durch Festlegung kritischer Eigenschaften und möglicherweise durch Skizzierung oder Durchführung von Beweisen oder Model Checking einiger Eigenschaften.

3. **Formale Entwicklung:** Formale Entwickung erfordert, dass alle notwendigen Eigenschaften (inklusive Korrektheit) formal beschrieben und entweder bewiesen oder formal modellgeprüft sind.

Wobei das Triptych-Paradigma nur die letzten drei Formen der Softwareentwicklung (1-3) erlaubt [Bjoerner 2006].

Bezug zur Informationsflusstheorie Die drei Phasen des Triptych Process Model finden sich auch in der Informationsflusstheorie wieder (vgl. Kapitel 3.7.2). Insbesondere die Hervorhebung des Domänen-Engineerings als eigenständige Prozessphase lässt sich gut mit den Vorhersagen der Informationsflusstheorie in Einklang bringen. Nach Informationsflusstheorie ist Kommunikation zwischen verschiedenen Domänen sehr schwierig und fehleranfällig (vgl. Korollar 4.5). Da bei der Softwareentwicklung die Anwendungsdomäne meist nicht die Softwaredomäne ist, ist auch die Anforderungserhebung schwierig und fehleranfällig. Daher ist es sinnvoll, dass die Entwickler zunächst die Anwendungsdomäne lernen, um die Wissensunterschiede zu den Fachleuten zu verkleinern, was wiederum zu erfolgreicherer Kommunikation und schließlich zu erfolgreicherer Softwareentwicklung führt.

Eine weitere Ähnlichkeit ist die Idee der Abstraktheits- und Domänenübergänge von Informationen während der Softwareentwicklung (vgl. Kapitel 3.7.3). Nach beiden Theorien geschieht der Übergang zwischen der Sprache der Anwendungsdomäne und der Sprache der Softwaredomäne üblicherweise in der Anforderungsphase (vgl. Abb. 3.34 ② → ③). Bjørner unterscheidet drei Arten von Anforderungen, die diesen Übergang wiederspiegeln [Bjoerner 2006-1]:

> *Domain requirements* [...] *are requirements that pertain solely to domain phenomena, i.e., they are requirements whose professional terms are domain terms.*
>
> *Interface requirements* [...] *are requirements that pertain both to the domain and to the machine to be built, to the interface between the machine and the domain, human users of the domain as well as (other) natural phenomena and man-made equipment of the domain.*
>
> [...]
>
> *Machine requirements* [...] *are requirements that primarily pertain to the machine to be built, that is, the software + hardware of the desired computing system.* [Bjoerner 2006-1, S. 10]

Es gibt also Domänen-, Schnittstellen- und Maschinenanforderungen. Domänenanforderungen werden in der Sprache der Domäne verfasst, Maschinenanforderungen in der Sprache der Problemlösungsdomäne (hier Softwaredomäne) und Schnittstellenanforderungen stellen die Verbindung zwischen den beiden Domänen her, sodass dafür beide Sprachen benutzt werden. Alle folgenden Informationen (Architektur, Entwurf, Implementierung) sind dann nur noch in der Softwaredomäne. Die Konkretisierung von Domänenbeschreibung über Anforderungen zur Softwarespezifikation beschreibt Bjørner wie folgt:

> *A domain description is only an abstraction, or a model of some reality, but it is not that reality, whereas a requirements prescription is intended to be a precise exact model of the software to be implemented.* [Bjoerner 2006-1, S. 13]

Abgrenzung zur Informationsflusstheorie Wesentliche Unterschiede zwischen dem Triptych Process Model und der Informationsflusstheorie sind:

- Die sehr umfangreiche und formale Beschreibung der Triptych-Theorie mit 485 Definitionen in drei Bänden [Bjoerner 2006-1, Bjoerner 2006-2, Bjoerner 2006-3] mit großem Anteil mathematischer Formalismen und formaler Sprachen.
- Das Hervorheben der Wichtigkeit formaler Methoden in allen Phasen der Softwareentwicklung bei Bjørner [Bjoerner 2006-1].
- Das während der Entwicklung von Software sehr viele und möglichst formale Dokumente erzeugt werden müssen [Bjoerner 2006-1].

- Der Mensch und Kommunikation als Teil der Softwareentwicklung werden von Bjørner nur beiläufig erwähnt.

Da die Triptych-Theorie immer die Erstellung von sehr vielen Dokumenten fordert, erscheint sie wirtschaftlich gesehen nur für große Projekte geeignet. In diesem Sinne deckt die Informationsflusstheorie mehr Arten von Softwareentwicklungsprojekten ab, z.B. auch agile Softwareentwicklung.

5.1.9 Socio-Technical Congruence

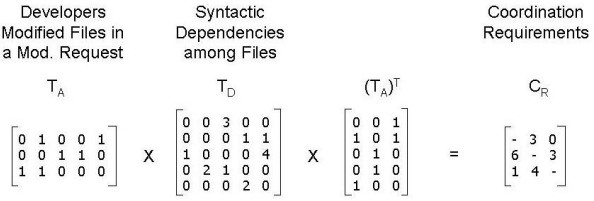


Abb. 5.8: Beispiel zur Berechnung von Koordinationsanforderungen aus [Cataldo 2008]

Die zentrale Hypothese der Socio-Technical Congruence Theory (STC) ist, dass die Produktivität der Softwareentwicklung steigt, wenn die Koordinationsanforderungen (engl. coordination requirements) mit den tatsächlich ausgeführten Koordinationsaktivitäten übereinstimmen (d.h., wenn sie kongruent sind) [Cataldo 2008]. Die STC löst eine sich aus der ETC [Herbsleb 2003] (vgl. Kapitel 5.1.6) ergebende Fragestellung. Es liefert eine Möglichkeit Koordinationsqualität in einem Projekt zu messen. Dazu definiert die Theorie eine Kongruenzmetrik. Für die Metrik werden Koordinationsanforderungen aus Entwickler-Software-Beziehungen berechnet, z.B. Entwickler haben Quellcode-Dateien bearbeitet (vgl. T_A in Abb. 5.8), Quellcode-Dateien haben Abhängigkeiten untereinander (vgl. T_D in Abb. 5.8), Quellcode-Dateien werden von Entwicklern bearbeitet (vgl. $(T_A)^T$ in Abb. 5.8): daraus lässt sich berechnen, welcher Entwickler seine Arbeit mit welchem anderen Entwickler koordinieren muss (vgl. C_R in Abb. 5.8).

Tatsächlich ausgeführte Koordinationsaktivitäten werden aus Organisationsstruktur (und die sich daraus ergebeneden Kommunikationsmöglichkeiten) oder

tatsächlichen Kommunikationsevents abgeleitet, z.B. aus Zugehörigkeit zu einem lokalen Team, aus IM-Chats oder aus Ticketmanagement-Systemen (MR, vergleichbar mit Bugzilla). Die Hypothese, dass Kongruenz zwischen Kommunikationsanforderungen und tatsächlicher Kommunikation zu Produktivitätssteigerung führt, konnte von Cataldo et al. [Cataldo 2008] durch eine schwache aber signifikante Korrelation zwischen socio-technischer Kongruenz und Dauer der Behebung eines Defekts belegt werden. Kongruente Teams waren durchschnittlich 32 % schneller als inkongruente Teams.

Kwan et al. [Kwan 2011] haben eine weitere STC Studie durchgeführt. Diese hat nicht zu so eindeutigen Ergebnissen geführt, wie die Studie von Cataldo et al. [Cataldo 2008]. Zum Beispiel haben sie einen unerwarteten signifikanten Zusammenhang zwischen Socio-technischer Inkongruenz und Build-Erfolg im IBM Rational Team Concert Projekt festgestellt. D.h. inkongruente Koordinationsanforderungen und Koordinationswirklichkeit haben zu signifikant höherem Build-Erfolg geführt als übereinstimmende Anforderungen und gemessene tatsächliche Kommunikation. Kwan et al. liefern folgende Erklärung für dieses unerwartete Ergebnis: Die nicht gemessene Kommunikation über Informationen aus der Entwicklungsumgebung, die speziell für die Bedürfnisse verteilter Softwareentwicklung entwickelt wurde, kann ausgereicht haben, die Arbeit abzustimmen und das System erfolgreich zu bauen.

Bezug und Abgrenzung zur Informationsflusstheorie Da die STC auf der gleichen Grundannahme beruht, wie die ETC (vgl. Kapitel 5.1.6), sind auch die Gemeinsamkeiten und Unterschiede zur Informationsflusstheorie ähnlich. Abhängigkeiten zwischen Arbeitspaketen (egal ob Komponenten oder Klassen) führen zu Koordinationsbedarf (vgl. Kapitel 4.4). Auch die STC ist wie die ETC eine mathematische Theorie. Die STC definiert vier Matrizen und eine Kongruenzmetrik. Die Grundannahme, dass kongruente Kommunikation die Produktivität erhöht, kann als ein Theorem angesehen werden. Wie die ETC deckt die STC im Vergleich zur Informationsflusstheorie nur einen Ausschnitt der Softwareentwicklung ab. Kommunikation mit Fachleuten wird nicht betrachtet, obwohl die Metrik dabei sicherlich auch angewendet werden könnte.

Die Informationsflusstheorie kann das unerwartete Ergebnis der Studie von Kwan et al. erklären [Kwan 2011]. Das untersuchte IBM Projekt ist ein Open-Source-Projekt, welches eine Software zur Unterstützung von Softwareentwicklung entwickelt. D.h. Anwendungs- und Lösungsdomäne sind die glei-

che. Kommunikation zwischen Fachleuten und Entwicklern ist unproblematischer als in anderen Projekten (vgl. Korollar 4.5). Die Entwickler arbeiten verteilt und kommunizieren hauptsächlich über Kommentare an Einträgen des Ticketmanagementsystems. D.h., inhaltliche Kommunikation geschieht über den Quellcode und andere in der Entwicklungsumgebung integrierte Dokumente, und steuernde Kommunikation geschieht hauptsächlich über textuelle Kommentare (vgl. Kapitel 4.3). Nach Satz 4.10 bestimmt die Problemkomplexität die Abstimmungskomplexität. D.h., ein einfaches Problem bedarf wenig Abstimmung, welche mit wenigen Kommentaren erfolgen kann, und dennoch zu einer erfolgreichen Lösung führen kann, weil die inhaltlichen Informationen im Quellcode ausreichen. Ein komplexes Problem bedarf viel Abstimmungsaufwand. Die Textnachrichten mit relativ hoher Latenz reichen dazu nicht aus (vgl. Hypothesen 4.5 und 4.7), was die Fehlschläge auch bei kongruenter Kommunikation erklären könnte. Da die Informationsflusstheorie dieses aus Sicht der STC unerwartete Ergebnis erklären kann, ist sie grundlegender als die STC.

Die STC ist stark auf eine Metrik fokussiert, sie beschäftigt sich also primär damit, wie man den Kommunikationsbedarf bestimmen kann. Die Theorie, die erklärt, warum die Metrik sinnvoll ist, ist sekundär. Die Informationsflusstheorie verfolgt gegenläufige Herangehensweise. Sie hat einen primären Fokus auf grundlegende theoretische Zusammenhänge, die sekundär zur Herleitung von Metriken genutzt werden können (vgl. Kapitel 5.2.4 unten). Durch den starken Fokus auf theoretische Zusammenhänge fällt anschließend aber die Interpretation empirischer Ergebnisse leichter.

5.1.10 A Theory of Shared Understanding for Software Organizations

In seiner Theorie beschreibt Aranda das Konzept des gemeinsamen Verständnisses als den Schlüssel zur Erklärung und Lösung von Koordinations- und Kommunikationsproblemen in der Softwareentwicklung [Aranda 2010]. Aranda definiert Koordination und Kommunikation basierend auf der Idee des gemeinsamen Verständnisses wie folgt [Aranda 2010]:

Gemeinsames Verständnis: Zwei oder mehr Teilnehmer haben ein gemeinsames Verständnis über eine Situation, wenn die Teile ihrer mentalen

Modelle, die sich aus dieser Situation ergeben haben, funktional equivalent sind. Funktionale Equivalenz bedeutet, dass die Modelle die gleichen Erklärungen und Vorhersagen über eine Situation liefern.

Koordination: Koordination zwischen Teilnehmern einer Situation besteht aus dem Teilen und Verhandeln über das Verständnis ihrer Ziele und Pläne.

Kommunikation: Kommunikation zwischen Teilnehmern einer Situation besteht aus dem Teilen und Verhandeln über das Verständnis ihres Status und Kontextes.

Eine Situation ist dabei eine Episode von Interaktionen und die Umgebung in der diese stattfinden [Cody 1985]. Die Theorie basiert auf der Grundannahme, dass effektive Koordination und Kommunikation essenziell für den Erfolg von Softwareentwicklung ist. Softwareentwickler sind ständig damit konfrontiert ein *gemeinsames Verständnis* über ihre Ziele, ihre Pläne, ihren Status und ihren Kontext zu teilen und zu verhandeln. Diese Bemühungen sind die Ursache für Koordinations- und Kommunikationsprobleme. Gemeinsames Verständnis wird von vier Eigenschaften von Teaminteraktionen beeinflusst [Aranda 2010]:

Synchronie: Teilnehmer teilen ihr Verständnis in zeitlicher Nähe zu der Situation, die einer Handlung bedarf.

Nähe: Teilnehmer teilen ihr Verständnis in physischer Nähe zueinander und zu der Situation.

Verhältnismäßigkeit: Teilnehmer teilen ihr Verständnis in Einklang mit ihrer Verantwortlichkeit für die Situation.

Reife: Teilnehmer teilen ihr Verständnis unter Ausnutzung vorheriger Verhaltensmuster.

Softwareorganisationen, die Werte, Strukturen und Praktiken haben, welche diese vier Eigenschaften unterstützen, haben weniger Koordinations- und Kommunikationsprobleme. Aus den Definitionen von Koordination und Kommunikation und den vier Eigenschaften von Teaminteraktionen leitet Aranda sechs Folgerungen auf Konstrukte ab, die häufig in der Software-Engineering-Forschung betrachtet werden [Aranda 2010]:

Prozesse: Ein Softwareentwicklungsprozess ist üblicherweise ein asynchroner, entfernter, asymmetrischer Koordinationsmechanismus.

Dokumentation: Dokumentation ist üblicherweise ein asynchroner, entfernter, asymmetrischer Kommunikationsmechanismus.

Praktiken & Werkzeuge: Praktiken und Werkzeuge sollten auf ihren positiven oder negativen Einfluss auf die vier Eigenschaften hin evaluiert werden.

Lokale Zusammenarbeit: Lokale Zusammenarbeit fördert Synchronie und Nähe direkt und Verhältnismäßigkeit und Reife indirekt.

Organisationswachstum: Das Wachstum einer Softwareorganisation führt zu einer unerwünschten Formalisierung, die wiederum zu asynchroner, entfernter, asymmetrischer Interaktion führt.

Kohäsion: Gruppenzusammenhalt ermöglicht die harmonische Nutzung von synchronen, nahen, symm.etrischen und reifen Kommunikationsstrategien.

Bezug zur Informationsflusstheorie Die vier Eigenschaften Synchronie, Nähe, Verhältnismäßigkeit und Reife sind aus Sicht der Informationsflusstheorie indirekte Eigenschaften von Kommunikationsaktivitäten und Medien. Synchronie kann über die Latenz abgebildet werden (vgl. Def. 3.31). Physische Nähe kann durch die möglichen Datentypen für Steuerung und Inhaltsübermittlung abgebildet werden (vgl. Kapitel 4.3). Die Verhältnismäßigkeit könnte durch Zielunterschiede nachgebildet werden (vgl. Kapitel 3.6.2), weil Personen, die Verantwortung für eine Aufgabe haben wahrscheinlich andere Ziele verfolgen, als Personen, die keine Verantwortung für eine Aufgabe tragen, diese aber dennoch ausführen müssen. Reife lässt sich durch die Größe des gemeinsamen Kontexts abbilden, der über die Zeit immer größer wird, wenn Entwickler viel miteinander kommunizieren (vgl. Sätze 4.5 und 4.6). Insgesamt spielt direkte lokale effiziente Kommunikation eine wichtige Rolle in der Softwareentwicklung, sowohl bei Aranda als auch (unter bestimmten Voraussetzungen) in der Informationsflusstheorie (vgl. Kapitel 4.3). Daraus lässt sich ableiten, dass der Mensch zentraler Faktor der Softwareentwicklung ist.

Abgrenzung zur Informationsflusstheorie Die Theory of Shared Understanding deckt den kreativen kommunikationsintensiven Teil der Softwareentwicklung ab, betrachtet aber den konstruktiven nicht. Sie behauptet sogar, dass Dokumentation schädlich ist. Nach Informationsflusstheorie ist Dokumentation ein essentieller Bestandteil, ohne den Softwareentwicklung nicht möglich wäre (vgl. Kapitel 3.7.2). Die Theory of Shared Understanding liefert

insgesamt sieben Definitionen (Gemeinsames Verständnis, Kommunikation, Koordination und die vier Interaktionseigenschaften) und sieben Theoreme (Grundannahme und die sechs Folgerungen). Die Informationsflusstheorie ist umfangreicher und etwas formaler (z.B. in der Kennzeichnung und Formulierung der Theoreme).

5.1.11 Zusammenfassung

Tabelle 5.1 zeigt eine Übersicht über alle betrachteten verwandten Theorien. Es wird jeweils die zentrale Aussage, die Formalität und die Anzahl von explizit genannten Definitionen und Theoremen angegeben. Es ist zu erkennen, dass alle betrachteten Theorien (bis auf The Triptych Process Model) im Vergleich zur Informationsflusstheorie relativ wenige Definitionen angeben. Daraus lässt sich ableiten, dass die Informationsflusstheorie umfangreicher ist. Eine umfangreiche Terminologie ermöglicht, dass viele unterschiedliche Softwareentwicklungsphänomene beschrieben werden können. Dies ist wiederum eine wichtige Voraussetzung für eine grundlegende Theorie. Die Triptych-Theorie nach Bjørner ist zwar formaler und umfangreicher als die Informationsflusstheorie, schließt aber auf Grund ihrer Striktheit bezüglich Anzahl und Formalität geforderter Dokumente viele wirtschaftlich relevante Softwareprojekte aus.

Neben einer grundlegenden Terminologie bieten die vielen Theoreme der Informationsflusstheorie einen Ansatzpunkt zur empirischen Überprüfung. Umso präziser diese formuliert sind, desto leichter können messbare Metriken abgeleitet werden. Dies erleichtert Planung und Auswertung von empirischen Studien. Die Informationsflusstheorie bewegt sich formal zwischen den mathematischen und umgangssprachlichen Theorien. D.h. auch, dass Studien mit "mittlerem" Aufwand geplant und durchgeführt werden können. Für die mathematisch formulierten Zusammenhänge (vgl. ETC, STC und Triptych) können zwar relativ leicht Metriken erarbeitet werden, indem z.B. Textnachrichten aus großen Repositories ausgelesen und analysiert werden. Die damit erzielten Ergebnisse können aber meist schlecht auf die Hauptaussage der Theorie generalisiert werden. Z.B. können auf Textnachrichten basierte Zusammenhänge nicht auf allgemeine Kommunikation generalisiert werden, wenn in der Abstimmung auch andere Medien (z.B. direkt von Angesicht zu Angesicht) genutzt wurden. Davon ist aber in den meisten Softwareprojekten auszugehen.

Tabelle 5.1: Vergleich von Theorien des Software Engineering

Theorie	**Zentrale Aussage**	**Formalität**[1]	**Definitionen**[2] **/ Theoreme**[3]
Mythical Man Month [Brooks 1995]	Zeit ≠ Entwicklermonate	natürliche Sprache	0 / 202
Theory W [Boehm 1989]	Mach jeden zum Gewinner	natürliche Sprache	1 / 3
Media Synchronicity Theory [Dennis 2008]	Kommunikationserfolg hängt von Mediensynchronität ab	Fachsprache (eigene)	12 / 7
The Cooperative Game [Cockburn 2007]	Softwareentwicklung ist ein kooperatives Spiel	natürliche Sprache	2 / 4
The Laws of Software Process [Armour 2003]	Softwareentwicklung ist Wissenserwerb	Fachsprache (eigene)	7 / 10
Theory of Coordination [Herbsleb 2003]	Softwareentwicklung ist Koordinationsproblem	mathematisch	4 / 2
Value-Based Software Engineering [Boehm 2006]	Erfolgskritische Stakeholder müssn gewinnen	Fachsprache (SE & PM)	0 / 2
The Triptych Process Model [Bjoerner 2006]	Softwareentwicklung ist Dokumentation	Fachsprache (eigene) und mathematisch	485 / 0
Socio-Technical Congruence [Cataldo 2008]	Softwareentwicklungserfolg hängt von kongruenter Kommunikation ab	mathematisch	5 / 1
A Theory of Shared Understanding [Aranda 2010]	Softwareentwicklungserfolg hängt vom gemeinsamen Verständnis ab	Fachsprache (eigene)	7 / 7
Informationsflusstheorie	Softwareentwicklung ist Informationsfluss	Fachsprache (eigene)	48 / 32

[1] Formalität im Sinne des Fachwissens, welches zum Verständnis der Theorie vorausgesetzt wird: natürliche Sprache, Fachsprache (z.B. SE, Projektmanagement (PM), oder eigens definierte), mathematisch
[2] Explizit gekennzeichnete Definitionen oder Axiome
[3] Explizit gekennzeichnete Behauptungen, Hypothesen, Sätze oder Gesetze

5.2 Eigene empirische Untersuchung – Medienwahl und Softwareprojekterfolg

In diesem Abschnitt wird eine eigene empirische Evaluation der Informationsflusstheorie vorgestellt. Es wird der Zusammenhang zwischen bestimmtem Kommunikationsverhalten in Softwareprojekten und Softwareprojekterfolg untersucht. Ziel der Studie ist es, die in Kapitel 4.3 vorgestellten Hypothesen 4.2, 4.3, 4.4, 4.5, 4.6, und 4.7 empirisch zu prüfen. Dazu wird die Kommunikation der Entwickler untereinander (vgl. Kapitel 3.7.2) in einem Experiment untersucht. Für die Studie wurden 130 Entwickler aus 26 kontrollierten studentischen Softwareprojekten zu ihrem Kommunikationsverhalten befragt.

5.2.1 Ziel der Studie und testbare Hypothesen

Ziel dieser Studie ist die empirische Evaluation eines Teils der in dieser Arbeit vorgestellten Theorie. Der zu untersuchende Teil lässt sich als Ziel mit Hilfe der von Wohlin et al. [Wohlin 2000, S. 42] vorgeschlagenen Schablone wie folgt beschreiben:

> Analysiere *den Einfluss von Kommunikation*
> auf den *Erfolg von Softwareprojekten*
> aus der Perspektive des *Software-Engineering-Forschers*.

Wohlin et al. folgend ist der nächste Schritt nach der Definition des Ziels der Studie die Formulierung von Hypothesen und die Wahl geeigneter Variablen [Wohlin 2000, S. 47].

Zunächst werden die Hypothesen aus Kapitel 4 in testbare Hypothesen H_i überführt. Dabei entspricht der Index i der Nummer der zu Grunde liegenden Hypothese aus Kapitel 4.3. Die Hypothesen 4.2, 4.3, 4.4, 4.5, 4.6 und 4.7 machen alle eine Aussage über den Einfluss unterschiedlicher Kommunikationsparameter und der Wahl bestimmter Medieneigenschaften auf den Kommunikationserfolg (vgl. Def. 3.38). Kommunikationserfolg ist nur schwer direkt messbar. Dazu müsste man den Wissensgewinn der Personen nachweisen können (vgl. Def. 3.2 und 3.24). Daher wird hier der Softwareprojekterfolg direkt gemessen. Auf den Projekterfolg haben zwar noch viele andere Dinge

Einfluss, aber ohne funktionierende Kommunikation kann kein Softwareprojekt (ab einer bestimmten Größe) erfolgreich sein (vgl. u.a. [Brooks 1974, Curtis 1988, Kraut 1995, Carmel 1999, Boehm 2002, Cockburn 2007, Silva 2010]). Um die folgenden Hypothesen messbar zu machen, werden sie daher über Softwareprojekterfolg statt Kommunikationserfolg definiert.

H_{42}: *Alternativhypothese:* Es gibt einen Zusammenhang zwischen dem gewählten Datentyp zur Steuerung der Kommunikation und Softwareprojekterfolg.

Nullhypothese: Es gibt keinen Zusammenhang zwischen dem gewählten Datentyp zur Steuerung der Kommunikation und Softwareprojekterfolg.

H_{43}: *Alternativhypothese:* Es gibt einen Zusammenhang zwischen dem auf Basis von Zielunterschieden gewählten Datentyp zur Steuerung der Kommunikation und Softwareprojekterfolg.

Nullhypothese: Es gibt keinen Zusammenhang zwischen dem auf Basis von Zielunterschieden gewählten Datentyp zur Steuerung der Kommunikation und Softwareprojekterfolg.

H_{44}: *Alternativhypothese:* Es gibt einen Zusammenhang zwischen dem auf Basis von Wissensunterschieden gewählten Datentyp zur Steuerung der Kommunikation und Softwareprojekterfolg.

Nullhypothese: Es gibt keinen Zusammenhang zwischen dem auf Basis von Wissensunterschieden gewählten Datentyp zur Steuerung der Kommunikation und Softwareprojekterfolg.

H_{45}: *Alternativhypothese:* Es gibt einen Zusammenhang zwischen der auf Basis des Kommunikationsziels gewählten Latenz des Kanals zur Inhaltsübermittlung und Softwareprojekterfolg.

Nullhypothese: Es gibt keinen Zusammenhang zwischen der auf Basis des Kommunikationsziels gewählten Latenz des Kanals zur Inhaltsübermittlung und Softwareprojekterfolg.

H_{46}: *Alternativhypothese:* Es gibt einen Zusammenhang zwischen der auf Basis von Zielunterschieden gewählten Latenz des Kanals zur Steuerung der Kommunikation und Softwareprojekterfolg.

Nullhypothese: Es gibt keinen Zusammenhang zwischen der auf Basis von Zielunterschieden gewählten Latenz des Kanals zur Steuerung der Kommunikation und Softwareprojekterfolg.

H_{47}**:** *Alternativhypothese:* Es gibt einen Zusammenhang zwischen der auf Basis von Wissensunterschieden gewählten Latenz des Kanals zur Steuerung der Kommunikation und Softwareprojekterfolg.

Nullhypothese: Es gibt keinen Zusammenhang zwischen der auf Basis von Wissensunterschieden gewählten Latenz des Kanals zur Steuerung der Kommunikation und Softwareprojekterfolg.

Aus diesen Hypothesen ergeben sich folgende Variablen, für die geeignete Metriken gefunden werden müssen:

- Aufgabe
  - Problemgröße (für Vergleichbarkeit der Projekte)
- Team
  - Entwickleranzahl
  - Zielunterschiede
  - Wissensunterschiede
- Kommunikationsverhalten
  - Medien zur Steuerung (Datentypen, Latenzen)
  - Medien zur Inhaltsübermittlung (Latenzen)
  - Teilnehmer der Kommunikation
  - Häufigkeit der Kommunikation
- Softwareprojekterfolg

Abbildung 5.9 gibt einen Überblick über die Variablen und ihren Zusammenhang. Die zu testenden Hypothesen sind an den entsprechenden Stellen markiert.

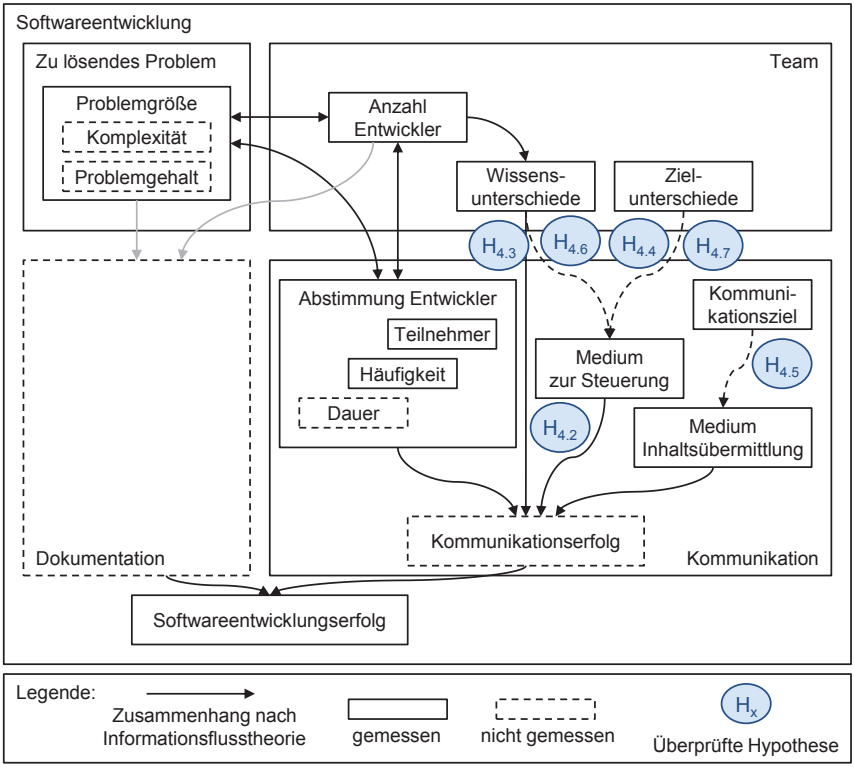


Abb. 5.9: Überblick über die Variablen der Studie und ihr hypothetischer Zusammenhang (vgl. Abb. 4.1)

5.2.2 Empirische Strategie

Als übergreifende empirische Strategie wurde das Quasi-Experiment (vgl. u.a. [Wohlin 2000, S. 9]) gewählt. In einem Quasi-Experiment werden die Werte der unabhängigen Variablen *nicht* gezielt und zufällig festgelegt, sondern teilweise durch die vorhandenen Eigenschaften der Studienteilnehmer gegeben. Insbesondere wird das Kommunikationsverhalten der Entwickler in der hier präsentierten Studie *nicht* gezielt gesteuert, sondern ihr natürliches Verhalten betrachtet. Weiterhin hat die Studie folgende für die Interpretation der Ergebnisse wichtige Eigenschaften:

Kontrollierte Variablen: Experimente können zur Prüfung von Theorien eingesetzt werden [Wohlin 2000, S. 15]. Dazu werden alle Variablen, bis auf die zu untersuchenden, kontrolliert, sodass der untersuchte Effekt isoliert betrachtet werden kann. In dieser Studie wurden nicht alle Variablen kontrolliert (sieh unten), um möglichst realistische Ergebnisse zu bekommen. Damit aber Ergebnisse generalisiert und statistische Methoden angewandt werden können, wurden dennoch wichtige Faktoren gezielt eingestellt. So sind alle untersuchten Projekte ungefähr gleich umfangreich. Es arbeiten zwischen 4 und 6 Studenten (Median 5) an einer Aufgabe. Alle Projekte dauern gleich lang. Alle Projekte haben den gleichen Entwicklungsprozess verfolgt (vgl. Kapitel 5.2.3 unten).

Realistische Umgebung: Es wurde versucht, möglichst realistische Softwareprojekte als Analysegegenstand des Experiments zu wählen, um leichter glaubhaft machen zu können, dass die festgestellten Zusammenhänge nicht nur unter Laborbedingungen auftreten. Daher wurden in dieser Studie Softwareprojekte untersucht, die ein echtes Problem lösen sollten. Die Projekte enthielten die typischen Phasen der Softwareentwicklung (Anforderungen, Entwurf, Implementierung), die die Studenten selbständig durchführen mussten. Es gab keine Musterlösungen, realistischen Zeitdruck, die Möglichkeit des Fehlschlagens sowie zeitlich eingeschränkte Kunden- und Betreuerverfügbarkeiten (vgl. Kapitel 5.2.3 unten).

Fragebögen: Da die Teilnehmer des Experiments während der Durchführung möglichst wenig von ihrer eigentlichen Aufgabe abgelenkt werden sollten, wurden einige Metriken mit Hilfe von Fragebögen im Nachhinein erhoben (vgl. Kapitel 5.2.4 unten). Dies ist Vergleichbar mit dem Vorgehen bei einer begründenden Studie[2] nach [Wohlin 2000, S. 11]. Diese eignen sich, um Aussagen über das Verhalten einer Population zu überprüfen. Betrachtet man Softwareentwickler als eine Population und die Definitionen und Theoreme der Informationsflusstheorie als Aussagen über das Verhalten dieser Population während der Softwareentwicklung, so eignen sich fragebogenbasierte Metriken auch zur Prüfung zumindest von Teilen der Informationsflusstheorie.

Ein Ziel der Informationsflusstheorie ist es grundlegende Zusammenhänge in der Softwareentwicklung abzubilden. Da studentische Softwareprojekte auch

[2] engl. explanatory survey

Softwareprojekte sind, müssen die Vorhersagen der Informationsflusstheorie auch in studentischen Projekten nachweisbar sein, sofern die Theorie valide ist. Zudem wurde bei den studentischen Projekten darauf geachtet, dass sie so realitätsnah wie möglich sind (z.B. echtes Problem). Vollständig kontrollierte Experimente eignen sich nur bedingt, um einen Zusammenhang zwischen Medienwahl und Gesamtprojekterfolg zu ermitteln, weil die sehr vielen Einflussfaktoren in der Softwareentwicklung nur sehr schwer kontrolliert werden können. Um Ergebnisse komulieren zu können müssen dennoch wichtige Faktoren eingestellt werden, was dem hier gewählten Quasi-Experiment entspricht.

5.2.3 Kontext der Studie

Für die Studie wurden 26 Projekte mit insgesamt 130 Entwicklern untersucht. Bei den befragten Entwicklern handelt es sich um Studenten der Leibniz Universität Hannover, die im Rahmen ihres Informatikstudiums (bzw. ihres Mathematikstudiums mit Studienrichtung Informatik) ein Softwareprojekt absolvieren mussten. Nach Lehrplan wird das Softwareprojekt im 5. Semester des Bachelorstudiums durchgeführt. Die Studenten haben dann bereits zwei Programmiervorlesungen (Scheme und Java), die Vorlesung Grundlagen der Softwaretechnik und die Vorlesung Softwarequalität gehört. Das Softwareprojekt dauert ein Semester. Ein Softwareprojektteam besteht typischerweise aus fünf Studenten.

Es ist ein Dokumenten-zentrierter Entwicklungsprozess vorgeschrieben, bei dem in den ersten vier Wochen zunächst eine Anforderungsspezifikation, in den folgenden drei Wochen entweder ein Entwurf oder Prototypen inkl. Prototypendokumentation und in den letzten sechs Wochen das eigentliche Softwareprodukt implementiert werden (vgl Abb. 5.10). Für Anforderungsspezifikation, Entwurfsdokument und Prototypendokumentation werden den Studenten Templates zur Verfügung gestellt, die Struktur und Inhalt der Dokumente vorgeben. Am Ende des Projekts hat der Kunde eine Woche Zeit das Produkt in Betrieb zu nehmen und ggf. die Studenten bei dabei auftretenden Problemen zu Rate zu ziehen. Nach Inbetriebnahme entscheidet der Kunde, ob er das Produkt abnimmt oder nicht.

Die drei Hauptphasen Anforderungen, Entwurf bzw. Prototyp und Implementierung werden jeweils durch formale Quality Gates (vgl. [Flohr 2008])

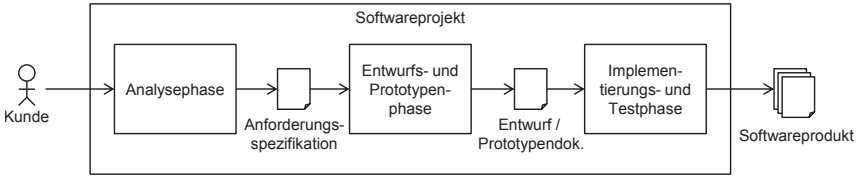


Abb. 5.10: Informationsfluss im Softwareprojekt des Informatikstudiums der Leibniz Universität Hannover (in Anlehnung an Abb. 3.31)

abgeschlossen. Dadurch soll ein Minimum an Qualität gesichert werden, sodass die folgenden Phasen mit höherer Wahrscheinlichkeit auch erfolgreich sind. Der vorgegebene Prozess sieht keine Inkremente oder Iterationen vor. Anforderungen können sich aber auch nach der Anforderungserhebungsphase noch ändern. Späte Änderungen, d.h. Änderungen nach dem ersten Quality Gate, dürfen und sollen in der Implementierung berücksichtigt werden.

Die Kunden der Projekte sind meist Mitarbeiter des Fachgebiets Software Engineering der Leibniz Universität Hannover (teilweise aber auch extern) mit einem echten Interesse an der zu entwickelnden Software. Jede Aufgabe wird an ein bis drei Teams ausgegeben. Den Teams wird ein Mitarbeiter des Fachgebiets als Berater für technische Fragen und Fragen zum Prozess zugeordnet. Die Kommunikation mit dem Kunden und mit dem Berater wird durch Zeitkarten künstlich beschränkt, um zu simulieren, dass in vielen professionellen Projekten Kunde und interne Berater nicht unbegrenzt verfügbar sind. Zudem soll der Zeitdruck bei Kundengesprächen dafür sorgen, dass sich die Studenten gut darauf vorbereiten. Jedes Team hat sechs Zeitkarten á 15 Minuten für Kundengespräche und sechs Zeitkarten á 30 Minuten für Beratergespräche, die sie über die gesamte Projektlaufzeit einsetzen dürfen. Zusätzlich gibt es eine so genannte Eskalationskarte. Sie gibt den Studenten die Möglichkeit bei Problemen, die sie nicht mehr alleine teamintern lösen können, offizielle Hilfe von außerhalb zu bekommen. Zur Verwaltung von Dokumenten und des Quellcodes wird den Studenten ein Versionsverwaltungssystem (Subversion) zur Verfügung gestellt.

Eine ausführlichere Beschreibung des Softwareprojekts im Studiengang Informatik der Leibniz Universität Hannover findet sich in [Schneider 2005b, Luebke 2005b].

5.2.4 Metriken

In diesem Abschnitt werden Metriken zur Bestimmung der Variablen aus Abbildung 5.9 näher erläutert.

5.2.4.1 Projekterfolgsmetrik

Nach Definition 3.42 ist Softwareentwicklungserfolg die effiziente Erstellung eines effektiven Softwareprodukts. D.h. Softwareentwicklung ist umso erfolgreicher, je schneller das richtige (d.h. den Anforderungen entsprechende) Produkt erstellt wird. Auch nach der Standish Group [CHAOS 1995] sind Softwareprojekte erfolgreich, wenn sie

- termingerecht sind,
- im Budget liegen und
- alle spezifizierten Anforderungen erfüllt wurden.

Ob alle Anforderungen umgesetzt wurden, wird im Softwareprojekt einmal durch einen Checklistenpunkt im dritten QualityGate geprüft (Wurden alle Abnahmetests bestanden?) und zum Anderen durch die Abnahme des Kunden. Beide Metriken alleine liefern nicht zu hundert Prozent ein verlässliches Ergebnis. Im QualityGate wird nicht jeder Abnahmetest durchgeführt, sondern nur die Unterschrift des Quality Agents geprüft, die bestätigt, dass das Team die Tests durchgeführt hat und dabei keine Fehler gefunden wurden. Der Kunde hat bei der Abnahme auch Spielraum und entscheidet meist zu Gunsten der Studenten, wenn das Produkt seine Anforderungen zumindest einigermaßen erfüllt. Durch Kombination der beiden Metriken kann dennoch ein Maß für den Erfüllungsgrad der Anforderungen mit ausreichender Zuverlässigkeit gebildet werden. Da es für studentische Softwareprojekte kein Budget gibt, welches überschritten werden kann, und die Projektlaufzeit durch das Semester beschränkt ist, können Termin- und Budgetüberschreitungen nicht gemessen werden. Es ist noch nie vorgekommen, dass Studenten bereits vor Ende der Implementierungsphase fertig waren und eine erfolgreiche Abnahme durchgeführt haben. Um die Realität nachzubilden, in der Softwareprojekte als zumindest teilweise erfolgreich angesehen werden, die zwar Budget oder Zeitplan nicht eingehalten haben, aber das Produkt dennoch eingesetzt wird, gelten hier auch diejenigen Projekte als erfolgreich, die nach Einschätzung des Kunden mit einem Monat Nacharbeit unter Softwareprojektbedingungen wie

geplant eingesetzt werden könnten. Als nicht erfolgreich gelten Projekte, die die Ziele des Kunden nicht erfüllen und der Kunde nicht glaubt, dass sich das nach einem Monat Nacharbeit noch ändern könnte. Die nicht erfolgreichen Projekte werden noch weiter unterteilt in Projekte, die das dritte QualityGate im ersten Versuch, im zweiten Versuch oder gar nicht bestanden haben. Es ergibt sich also folgende Metrik zur Bestimmung des Projekterfolgs für Softwareprojekte des Informatikstudiums der Leibniz Universität Hannover:

Erfolgreich: Die Ziele des Kunden wurden erreicht. Die Software könnte direkt wie beabsichtigt eingesetzt werden.

Eingeschränkt erfolgreich: Die Ziele des Kunden wurden nicht ganz erreicht. Aber mit einem Monat Nacharbeit unter Softwareprojektbedingungen könnte die Software einsatzbereit gemacht werden.

Nicht erfolgreich: Die Ziele des Kunden wurden nicht erreicht.

- **QG 3.1 bestanden:** Das dritte QualityGate wurde im ersten Versuch bestanden.
- **QG 3.2 bestanden:** Das dritte QualityGate wurde im zweiten Versuch bestanden.
- **QG 3 nicht bestanden:** Das dritte QualityGate wurde nicht bestanden, weder beim ersten noch beim zweiten Versuch.

Der in dieser Studie verwendete Fragebogen zur Bestimmung des Projekterfolgs befindet sich im Anhang (vgl. Anhang A.1).

5.2.4.2 Metrik zur Bestimmung der Problemgröße

Da im Softwareprojekt der Leibniz Universität Hannover Produkte entwickelt werden sollen, für die ein echter Bedarf besteht, werden viele verschiedene Produkte gefordert. Jedes Projekt wird dabei höchstens dreimal ausgegeben. Die Projekte sind teilweise unterschiedlich aufwendig, weil sie von unterschiedlichen Kunden stammen und weil sich die tatsächlichen Anforderungen erst während des Projekts herauskristallisieren[3].

[3] D.h., auch wenn versucht wird den Studenten vergleichbar aufwendige Projekte zu geben, kann es passieren, dass sich während des Projekts herausstellt, dass die Aufgabe doch mehr oder weniger aufwendig ist, als geplant.

Für die Auswertung der Ergebnisse ist es wichtig, dass die Projekte so ähnlich wie möglich sind. Um sicherzustellen, dass die Projekte ungefähr gleich umfangreich sind, ist eine Metrik zur Bestimmung der Problemgröße notwendig. Problemgehalt und -komplexität können nur schwer bestimmt werden (vgl. Kapitel 3.7.1). Wie in Kapitel 4.4 beschrieben, ist eine Expertenschätzung eine sinnvolle Möglichkeit die Problemgröße zu bestimmen. Die Rollen des Kunden und des Betreuers werden im Softwareprojekt von Mitarbeitern des Fachgebiets besetzt, die die Lehrveranstaltung schon aus vorangegangenen Semestern kennen. Zur Bestimmung der Problemgröße wurden daher die Kunden und Betreuer der jeweiligen Projekte nach Projektende befragt, wie viele Studenten ihrer Meinung nach notwendig gewesen wären, um die geforderte Software innerhalb der 6 Wochen Implementierungsphase zu programmieren. Dabei sollten sie davon ausgehen, dass den Studenten alle Anforderungen bekannt sind. Weiterhin sollte von Studenten ausgegangen werden, die im 5. Bachelor-Semester des Informatikstudiums sind und die Grundlagenvorlesungen (Scheme, Java, Softwaretechnik und Softwarequalität) gehört und bestanden haben, und die grundlegende Programmierkenntnisse besitzen. Grundlegende Programmierkenntnisse heißt dabei insbesondere, dass sie sich während der sechs Wochen Bearbeitungszeit in weiterführende Technologien und Frameworks (z.B. Android-Entwicklung oder Web-Entwicklung) erst einarbeiten müssen. Als Maß für die Problemgröße der Softwareprojekte wird der Mittelwert der Antworten von Kunde und Betreuer benutzt. Mit Hilfe der Ergebnisse dieser Metrik kann überprüft werden, ob ausreichend viele Entwickler an der Aufgabe gearbeitet haben. Zudem kann damit festgestellt werden, ob bei den fehlgeschlagenen Projekten die Problemgröße und nicht mangelnde Kommunikation Ursache für das Fehlschagen gewesen ist.

5.2.4.3 Metriken zur Bestimmung von Ziel- und Wissensunterschieden

Zielunterschiede Die Bestimmung von Zielunterschieden (vgl. Kapitel 3.6.2) zwischen den Teammitgliedern ist objektiv nur sehr schwer möglich. Direkte Befragungen sind während des Projekts nicht zielführend, da die Studenten dann noch in der Prüfungssituation sind und ihre wahren Ziele nicht preisgeben werden. Nach Projektende sind die Studenten für Befragungen nicht mehr erreichbar. Daher wird hier eine Metrik zur Bestimmung der Zielunterschiede verwendet, die aus den Symptomen, z.B. Konflikten, vorhandene Zielunterschiede ableitet.

Am Ende des Softwareprojekts müssen alle Teams mit Hilfe der LID-Technik [Schneider 2000] ihre Erfahrungen im Projekt in einer zweistündigen Sitzung diskutieren. Diese Erfahrungen werden mit Hilfe eines Templates strukturiert in so genannten LIDs dokumentiert. Mit Hilfe der LIDs und anderen organisatorischen Daten der Lehrveranstaltung werden Zielunterschiede wie folgt abgeleitet:

Konflikt: Mit in Konflikt stehenden Zielen werden diejenigen Teams bewertet, die:

- während der Implementierungsphase offiziell eskaliert haben,
- in früheren Phasen (Anforderungen, Entwurf) eskaliert haben und im LID kein Anzeichen für eine spätere Einigung zu finden ist,
- im LID explizit angegeben haben, dass es in der Implementierungsphase Konflikte gab.

Koordiniert: Koordinierte Ziele haben Teams, die:

- in frühen Phasen (Anforderungen, Entwurf) einen Konflikt hatten (siehe oben), aber im LID Anzeichen zu finden sind, dass sie sich bis zur Implementierung wieder geeinigt haben,
- im LID unterschiedliche Ziele angegeben haben. Zum Beispiel, wenn ein Student angibt, dass er die Veranstaltung nur besucht, um die Leistungspunkte zu erhalten, und ein anderer Student angibt, dass er etwas lernen möchte.

Kollaborativ (keine): Alle anderen Teams werden als kollaborative Teams bewertet.

Wissensunterschiede Bei der Anmeldung zur Lehrveranstaltung müssen die Softwareprojektteilnehmer ihre Fähigkeiten auf der Skala *keine*, *wenig* oder *viel* einschätzen (vgl. Abb. 5.11). Unter anderem müssen sie dabei ihre Erfahrung bei der Java-Entwicklung einschätzen. Auf Basis der Selbsteinschätzung der eigenen Java-Erfahrung wurde der für die Implementierungsphase relevante Wissensunterschied (vgl. Kapitel 3.6.1) wie folgt bestimmt:

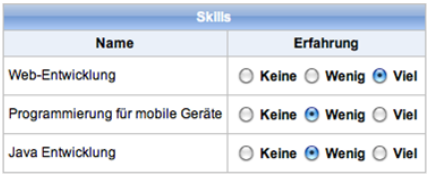


Abb. 5.11: Ausschnitt aus Softwareprojektanmeldung: Ankreuzmöglichkeiten zur Einschätzung der eigenen Fähigkeiten

Domäne: Wenn im Team mindestens ein Student ist, der angibt keine Java-Erfahrung zu haben, und mindestens ein Student ist, der angibt viel Java-Erfahrung zu haben, dann wird der Wissensunterschied auf Domänenebene festgelegt.

Ausbildung: Bei kleineren Erfahrungsunterschieden, d.h. nur Studenten, die keine bis wenig oder wenig bis viel Java-Erfahrung haben, wird der Wissensunterschied auf Ausbildungsebene festgelegt. Die Studenten befinden sich dann auf unterschiedlichen Ebenen in ihrer Programmierausbildung.

Projekt: Wenn alle Studenten den gleichen Erfahrungswert bei der Java-Programmierung haben, werden Wissensunterschiede auf Projektebene festgelegt. Es wird nicht die *Gesprächsebene* gewählt, da die Teams vom Fachgebiet zusammengestellt werden und daher nicht davon auszugehen ist, dass die Studenten über viel gemeinsame Projekterfahrung verfügen.

5.2.4.4 Metriken zur Bestimmung des Kommunikationsverhaltens

Das Kommunikationsverhalten der Softwareprojektteams wurde bestimmt, indem jeder Student am Ende der Lehrveranstaltung einen Fragebogen ausfüllen musste. Es wurde je Phase nach der Kommunikation mit dem Kunden und nach Team-interner Kommunikation gefragt. Die Studenten mussten angeben, mit wie vielen ihrer Teamkollegen sie über welches Medium, wie oft kommuniziert haben. Abbildung 5.12 zeigt einen Ausschnitt aus dem Fragebogen zur Bestimmung des Kommunikationsverhaltens im Softwareprojekt.

Die Teilnehmer des Softwareprojekts wurden zur Nutzung folgender Medien m befragt:

Art der Kommunikation		Anzahl der am Gespräch beteiligten Personen	Häufigkeit der Gespräche (unabhängig von der Dauer)		
Kommunikation mit Möglichkeit zum direkten Feedback	Teamsitzung (alle vor Ort)	mit dem gesamten Team	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
		mit mehr als 2 Personen	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
		mit 2 Personen	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
	Videokonferenz (z.B. Skype)	mit dem gesamten Team	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
		mit mehr als 2 Personen	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
		mit 2 Personen	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
	Telefonkonferenz (nur Audio, Skype ohne Video, Telefon)	mit dem gesamten Team	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
		mit mehr als 2 Personen	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
		mit 2 Personen	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
	Instant Messenger (nur Text, z.B. ICQ, MSN)	mit dem gesamten Team	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
		mit mehr als 2 Personen	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat
		mit 2 Personen	O > 1x/Tag O 1-2x/Woche O gar nicht	O 1x/Tag O 2-3x/Monat	O 3-6x/Woche O 1x/Monat

Abb. 5.12: Ausschnitt aus Fragebogen zur Bestimmung des Kommunikationsverhaltens im Softwareprojekt

- Angesicht zu Angesicht (alle vor Ort, engl. face-to-face, F2F)
- Videokonferenz
- Telefon
- Sofortnachrichten (engl. instant messaging, IM)
- E-Mail
- verschiedene Dokumente (u.a. Forum, Wiki, Spezifikationen, etc.)

Videokonferenz wurde von nur einem Team regelmäßig genutzt. Dieses Team hat aber die gleichen Werte bei F2F-Kommunikation angegeben, wie bei Videokonferenzen, weil immer ein Teammitglied zu den Treffen vor Ort per Videokonferenz zugeschaltet wurde. Die Videokonferenzdaten können daher bei der weiteren Analyse ausgeschlossen werden.

Die Anzahl der am Gespräch beteiligten Personen n_K ist in drei Kategorien unterteilt. Es besteht für jedes Medium die Möglichkeit anzugeben, ob es zur Kommunikation mit dem gesamten Team (Team), mit einem Teil des Teams, aber mehr als 2 Personen (>2), oder mit genau zwei Personen (2) genutzt wurde.

Je Medium m und je Kommunikationsteilnehmerkategorie n_K können folgende Werte für die Kommunikationshäufigkeit angekreuzt werden: gar nicht (0), 1x/Monat (1), 2-3x/Monat (2), 1-2x/Woche (3), 3-6x/Woche (4), 1x/Tag (5), $>$1x/Tag (6).

Die so gewonnenen Daten je Entwickler werden zu Projekt-spezifischen Kommunikationshäufigkeiten $x_K(m, n_K)$ aggregiert, indem je Medium und Anzahl der Kommunikationsteilnehmer der Median der angekreuzten Kommunikationshäufigkeiten bestimmt wird. So werden Ausreißer ausgefiltert. Wenn zum Beispiel ein Entwickler angibt, sich mehrfach am Tag mit allen Teammitgliedern persönlich zu treffen, aber alle anderen angeben, sich nur ein bis zweimal pro Woche persönlich zu treffen, dann erhält man über den Median den der Realität wahrscheinlich am nächsten kommenden Wert "1-2x/Woche".

Hilfsmetriken Um später leichter die Metriken für die Hypothesentests formulieren zu können, werden folgende Hilfsmetriken definiert:

Team: Ausschließlich Kommunikationshäufigkeiten von Kommunikation mit dem gesamten Team.

$$x_{\text{Team}}(m) = x_K(m, \text{Team}) \tag{5.1}$$

max: Maximum der Kommunikationshäufigkeiten aus Kommunikation mit dem gesamten Team, Kommunikation mit mehr als zwei Personen und Kommunikation mit genau zwei Personen.

$$x_{max}(m) = max(x_K(m, \text{Team}), x_K(m, >2), x_K(m, 2)) \tag{5.2}$$

gesprochen: Nur Medien mit Kanälen, die gesprochene natürliche Sprache übertragen können.

$$x_{\text{speech}}(n_K) = max(x_K(\text{F2F}, n_K), x_K(\text{Telefon}, n_K)) \tag{5.3}$$

geschrieben: Nur Medien mit Kanälen, die geschriebene natürliche Sprache übertragen können.

$$x_{\text{written}}(n_K) = max(x_K(\text{IM}, n_K), x_K(\text{E-Mail}, n_K)) \tag{5.4}$$

schnelles Feedback: Nur Medien mit Kanälen, die schnelles Feedback ermöglichen

$$x_{\text{instant}}(n_K) = max(x_K(\text{F2F}, n_K), x_K(\text{Telefon}, n_K), x_K(\text{IM}, n_K)) \tag{5.5}$$

steuernd: Maximum aus allen Medien mit Kanälen, die zur steuernden Kommunikation eingesetzt werden können. Die in dieser Studie betrachteten Medien, die zur Steuerung der Kommunikation genutzt werden können, sind Face-to-face, Telefon, Sofortnachrichten und E-Mail. Die anderen Dokumenten-basierten Medien wie Spezifikationen oder Wikis werden auf Grund ihrer relativ hohen Latenzen und der kurzen Projektlaufzeit (sechs Wochen Implementierungsphase) als nicht für die Steuerung geeignet angesehen.

$$x_{\text{control}}(n_K) = max(x_K(\text{F2F}, n_K), x_K(\text{Telefon}, n_K), x_K(\text{IM}, n_K), x_K(\text{E-Mail}, n_K)) \tag{5.6}$$

Aggregationen von Medien und Aggregationen von Teilnehmern können auch zusammengefasst werden. Z.B. beschreibt folgende Metrik die Kommunikationshäufigkeiten des gesamten Teams über Medien mit gesprochener natürlicher Sprache.

$$x_{\text{Team,speech}} = max(x_K(\text{F2F}, \text{Team}), x_K(\text{Telefon}, \text{Team}))$$

Für den Hypothesentest werden folgende zusammengesetzte Metriken definiert:

Zielunterschiede und Medium zur Steuerung Die Metrik zur Bestimmung der Kommunikationshäufigkeit von Medien, die nach Hypothese 4.3 gewählt werden, wird wie folgt festgelegt:

$$x_{4.3}(gd) = \begin{cases} x_{\text{Team}}(\text{F2F}) & \text{, für } gd = \text{konflikt} \\ x_{\text{Team,speech}} & \text{, für } gd = \text{koordiniert} \\ x_{\text{Team,control}} & \text{, für } gd = \text{keine} \end{cases} \tag{5.7}$$

Die Metrik zur Bestimmung der Kommunikationshäufigkeit von Medien, die nach Hypothese 4.6 gewählt werden, wird wie folgt festgelegt:

$$x_{4.6}(gd) = \begin{cases} x_{\text{Team}}(\text{F2F}) & \text{, für } gd = \text{konflikt} \\ x_{\text{Team,instant}} & \text{, für } gd = \text{koordiniert} \\ x_{\text{Team,control}} & \text{, für } gd = \text{keine} \end{cases} \tag{5.8}$$

Wissensunterschiede und Medium zur Steuerung Die Metrik zur Bestimmung der Kommunikationshäufigkeit von Medien, die nach Hypothese 4.4 gewählt werden, wird wie folgt festgelegt:

$$x_{4.4}(kd) = \begin{cases} x_{\text{Team}}(\text{F2F}) & \text{, für } kd = \text{Domäne} \\ x_{\text{Team,speech}} & \text{, für } kd = \text{Ausbildung} \\ x_{\text{Team,control}} & \text{, für } kd = \text{Projekt} \end{cases} \tag{5.9}$$

Die Metrik zur Bestimmung der Kommunikationshäufigkeit von Medien, die nach Hypothese 4.7 gewählt werden, wird wie folgt festgelegt:

$$x_{4.7}(kd) = \begin{cases} x_{\text{Team}}(\text{F2F}) & \text{, für } kd = \text{Domäne} \\ x_{\text{Team,instant}} & \text{, für } kd = \text{Ausbildung} \\ x_{\text{Team,control}} & \text{, für } kd = \text{Projekt} \end{cases} \tag{5.10}$$

5.2.4.5 Zusammenfassung

Tabelle 5.2 fasst die in der hier vorgestellten empirischen Studie verwendeten Metriken zusammen. Die Metriken werden mit den folgenden Attributen (nach [Jedlitschka 2005]) beschrieben:

Name: Name der Metrik für spätere Referenzierung.

Skalenniveau: Das Skalenniveau (nominal, ordinal, metrisch) bestimmt die statistischen Verfahren, die zur Analyse der Ergebnisse der Metrik genutzt werden können.

Wertebereich: Der Wertebereich beschreibt die möglichen Werte der Metrik.

Messregeln: Die Messregeln beschreiben (kurz), wie die Metrik erhoben wird, d.h. wie die Werte aus Beobachtungen oder anderen Daten abgeleitet werden.

5.2.5 Ergebnisse

Für die hier vorgestellte Studie wurden Softwareprojekte aus dem Wintersemester 2009/2010 (WS 09/10) und Softwareprojekte aus dem Wintersemester 2010/2011 (WS 10/11) ausgewertet. Es wurden Daten von 130 Studenten, 26 Teams und 15 verschiedenen Aufgaben ausgewertet.

Tabelle 5.3 gibt einen Überblick über die Projekte, Teamgrößen, Problemgrößen und Projekterfolge. Tabelle 5.3 lässt sich entnehmen, dass die Projekte im Median eine Teamgröße von 5 Studenten hatten, dass das durchschnittliche Projekt (Median) eingeschränkt erfolgreich ist (siehe oben), aber dass die Projektfehlschlagsrate im industriellen Durchschnitt von ca. 20 % liegt (vgl. u.a. [Emam 2008]).

Tabelle 5.4 zeigt Korrelationen verschiedener Kommunikationsmetriken der Implementierungsphase mit dem Projekterfolg.

Tabelle 5.2: Metriken der empirischen Studie

Metrik	**Skalentyp**	**Wertebereich**	**Messregeln**
Erfolg	ordinal	0: nein 1: nein+ 2: nein++ 3: ja- 4: ja	Fragebogen (vgl. Text S. 248 und Anhang A.1)
Erfolg (binär)	ordinal	0: nein 1: ja	Erfolg: {ja-, ja} → ja, {nein, nein+, nein++} → nein
Zielunterschiede gd	ordinal	0: keine 1: koordiniert 2: widersprüchlich	Eskalationen und LIDs (vgl. Text S. 250)
Wissensunterschiede kd	ordinal	1: Projekt 2: Ausbildung 3: Domäne	Unterschiede aus Selbsteinschätzung (vgl. Text S. 251)
Problemgröße	absolut / metrisch	$\mathbb{N}^+$	Expertenbefragung (vgl. Text S. 249)
Kommunikationsteilnehmer n_K	ordinal	Team: ganzes Team >2: mehr als zwei 2: genau zwei	Fragebogen (vgl. Abb. 5.12)
Kommunikationshäufigkeit $x_K(m, n_K)$	ordinal	0: gar nicht 1: 1x/Monat 2: 2-3x/Monat 3: 1-2x/Woche 4: 3-6x/Woche 5: 1x/Tag 6: $>$1x/Tag	Angekreuzter Wert x_K auf Fragebogen (vgl. Abb. 5.12) für Medium m* und Kommunikationsteilnehmer n_K (vgl. Text S. 254)
Zusammengesetzte Metriken			
$x_{4.3}(gd)$, $x_{4.6}(gd)$, $x_{4.4}(kd)$, $x_{4.7}(kd)$	ordinal	wie $x_K(m, n_K)$	vgl. Formeln 5.7, 5.8, 5.9 und 5.10

* $m \in$ {Angesicht zu Angesicht (F2F), Telefon, Sofortnachrichten (IM), E-Mail}

Tabelle 5.3: Überblick der in der empirischen Studie analysierten Projekte

Projekt	**Entwickler / Problemgröße**	**Erfolg**	**binär**	**Zielunterschiede**	**Wissensunterschiede**
Wintersemester 2009/2010					
Ameisen Arena 1	5 / 5	nein	nein	konflikt	Domäne
Ameisen Arena 2	5 / 5	nein+	nein	koord.	Domäne
AMP 1	5 / 3	ja-	ja	keine	Projekt
AMP 2	5 / 3	ja	ja	konflikt	Projekt
AMP 3	5 / 3	ja-	ja	keine	Domäne
Confrak	5 / 4	ja-	ja	koord.	Ausbildung
ConTexter 1	5 / 6	ja	ja	keine	Ausbildung
ConTexter 2	5 / 6	ja-	ja	koord.	Ausbildung
MDash 1	6 / 4	ja	ja	keine	Domäne
MDash 2	5 / 4	ja-	ja	koord.	Ausbildung
SE-Lib 1	5 / 4	ja-	ja	keine	Domäne
SE-Lib 2	5 / 4	ja-	ja	keine	Projekt
Seminar-Bewerter 1	5 / 3	ja	ja	konflikt	Domäne
Seminar-Bewerter 2	5 / 3	nein+	nein	keine	Projekt
SysFeedback	5 / 3	ja	ja	keine	Ausbildung
Wintersemester 2010/2011					
Anforderungsmetriken	5 / 4	ja	ja	keine	Ausbildung
FLOW-Editor	3 / 6	ja-	ja	koord.	Ausbildung
ProjectLog 1	6 / 4	ja	ja	keine	Ausbildung
ProjectLog 2	5 / 4	nein	nein	keine	Ausbildung
ProjectLog 3	4 / 4	ja-	ja	konflikt	Ausbildung
Shared Whiteboard 1	5 / 3	ja-	ja	keine	Ausbildung
Shared Whiteboard 2	5 / 3	ja-	ja	keine	Ausbildung
SmartKit	6 / 5	nein	nein	keine	Ausbildung
UIM 1	5 / 6	ja	ja	keine	Domäne
UIM 2	5 / 6	ja	ja	keine	Projekt
Zeus	5 / 5	ja	ja	keine	Ausbildung
Median ($\tilde{x}$) bzw. Durchschnitt ($\bar{x}$)	$\tilde{n} = 5 / 4$	$\tilde{e} =$ ja-	$\bar{e}_b =$ 81 %	$\tilde{gd} =$ keine	$\tilde{kd} =$ Ausbildung

Tabelle 5.4: Korrelationen von Kommunikationshäufigkeiten in der Implementierungsphase mit Softwareprojekterfolg

Korrelation ρ[1] ($n = 26$)	Erfolg	Erfolg (binär)
x_{Team}(F2F)	†0,36	*0,48
x_{max}(F2F)	†0,34	†0,39
x_{Team}(Telefon)	−0,04	−0,08
x_{max}(Telefon)	−0,03	−0,18
x_{Team}(IM)	−0,32	$^{\dagger}-0,35$
x_{max}(IM)	$^{\dagger}-0,36$	*−0,46
x_{Team}(E-Mail)	−0,04	−0,08
x_{max}(E-Mail)	−0,05	−0,03
$x_{\text{Team,speech}}$	†0,37	**0,51
$x_{\text{max,speech}}$	0,27	†0,37
$x_{\text{Team,written}}$	−0,08	−0,14
$x_{\text{max,written}}$	−0,20	−0,29
$x_{\text{Team,instant}}$	−0,11	0,00
$x_{\text{max,instant}}$	*−0,41	*−0,40
$x_{4.3}(gd)$	0,11	0,07
$x_{4.6}(gd)$	0,11	0,07
$x_{4.4}(kd)$	†0,38	**0,54
$x_{4.7}(kd)$	0,32	*0,47

$^{\dagger}p < 0{,}1$ $^{*}p < 0{,}05$ $^{**}p < 0{,}01$

[1] Spearmans Rangkorrelationskoeffizient ρ

5.2.6 Interpretation

5.2.6.1 Belege für Hypothese 4.2

Hypothese 4.2 macht eine Aussage über die Steuerungseffizienz verschiedener Datentypen. Demnach ist ein bewegtes natürliches Bild effizienter zur Steuerung der Kommunikation als gesprochene natürliche Sprache und gesprochene natürliche Sprache ist effizienter als geschriebene natürliche Sprache. Tabelle 5.4 zeigt eine hoch signifikante Korrelation zwischen Kommunikation, die gesprochene natürliche Sprache zur Steuerung nutzt, und binärem Projekterfolg ($x_{\text{Team,speech}}$). Zwischen Kommunikation, die geschriebene natürliche Sprache zur Steuerung nutzt, und Projekterfolg besteht hingegen kein signifikanter Zusammenhang. Die Nullhypothese von $H_{4\,2}$ kann mit einer Fehlerwahrscheinlichkeit von $\alpha < 1\%$ verworfen werden. Dies kann wiederum als Indiz zur Unterstützung von Hypothese 4.2 (gesprochen ist besser als geschrieben) gedeutet werden.

5.2.6.2 Belege für Hypothesen 4.3 und 4.6

Hypothese 4.3 macht eine Aussage über den Zusammenhang zwischen Zielunterschieden der Kommunikationsteilnehmer und den notwendigen Datentyp für den Kanal der Steuerung. Demnach sollte bei widersprüchlichen Zielen das bewegte natürliche Bild und bei koordinierten Zielen gesprochene natürliche Sprache zur Steuerung verwendet werden. Sind keine Zielunterschiede vorhanden, z.B. bei kollaborativen Teams, reichen Medien mit geschriebener natürlicher Sprache zur Steuerung aus. In dieser Studie konnte mit den verwendeten Metriken kein signifikanter Zusammenhang zwischen Zielunterschied-basierter Medienwahl und Projekterfolg festgestellt werden (vgl. Tabelle 5.4, $x_{4\,3}(gd)$). Die Nullhypothese von $H_{4\,3}$ kann nicht verworfen werden.

Hypothese 4.6 macht eine Aussage über den Zusammenhang zwischen Zielunterschieden der Kommunikationsteilnehmer und den Latenzanforderungen an den Kanal zur Steuerung. Koordinierte Ziele bedingen geringe Latenz und widersprüchliche Ziele eine sehr geringe Latenz für den Kanal zur Steuerung der Kommunikation. Wie in Tabelle 5.4 ($x_{4\,6}(gd)$) dargestellt, konnte keine signifikante Korrelation zwischen Zielunterschied-basierter Wahl der Latenz von Steuerungsmedien und Projekterfolg festgestellt werden. Die Nullhypothese von $H_{4\,6}$ kann nicht verworfen werden.

Betrachtet man die Mediennutzung mit geringer Latenz nicht nur für Meetings mit dem gesamten Team, sondern auch für Meetings mit einem Teil des Teams (>2) und bidirektionalen Konversationen (2), dann besteht sogar eine signifikante negative Korrelation mit dem Projekterfolg (vgl. Tabelle 5.4, $x_{\text{max,instant}}$). Dieser Effekt stammt von Teams, deren Entwickler häufig via Sofortnachrichtendienst mit nur einer anderen Person gechattet haben. D.h., dass Projekte tendenziell weniger erfolgreich waren, bei denen die Entwickler während der Implementierung hauptsächlich bidirektional gechattet haben. Hier scheint also die teilweise Ausgrenzung des Teams negativen Einfluss auf den Projekterfolg zu haben.

Insgesamt konnte für die beiden Hypothesen 4.3 und 4.6, die Aussagen über die Medienwahl auf Basis von Zielunterschieden machen, keine Unterstützung gefunden werden. Dies kann neben der Tatsache, dass die Hypothesen evtl. nicht zutreffen, noch andere Ursachen haben. Zum Einen ist die Metrik zur Bestimmung der Zielunterschiede nicht sehr genau. Tatsächlich vorhandene Zielunterschiede können durch die Analyse der LIDs im Nachhinein übersehen worden sein. Zum Anderen können andere Einflüsse die Effekte aus Zielunterschieden überlagert haben (siehe folgenden Abschnitt).

5.2.6.3 Belege für Hypothesen 4.4 und 4.7

Nach Hypothese 4.4 sollte je nach Wissensunterschieden der Kommunikationsteilnehmer ein anderer Datentyp zur Steuerung der Kommunikation genutzt werden. Für Wissensunterschiede auf Domänenebene ist demnach ein Kanal für bewegtes natürliches Bild sinnvoll, für Unterschiede auf Ausbildungsebene ein Kanal für gesprochene natürliche Sprache zweckmäßig und für Unterschiede auf Projektebene ein Kanal für geschriebene natürliche Sprache ausreichend. Tabelle 5.4 zeigt einen hoch signifikanten Zusammenhang zwischen Kommunikation über Medien, die nach Hypothese 4.4 für die Steuerung der Kommunikation gewählt wurden und Projekterfolg (binär, vgl. Tabelle 5.4, $x_{44}(kd)$). Die Nullhypothese von H_{44} kann mit einer Fehlerwahrscheinlichkeit von $\alpha < 1\%$ verworfen werden. Dies kann wiederum als Indiz zur Unterstützung von Hypothese 4.4 gedeutet werden.

Hypothese 4.7 beschreibt den Zusammenhang zwischen Wissensunterschieden und Latenzanforderungen für den Kanal zur Steuerung. Bei Wissensunterschieden auf Domänenebene sind sehr geringe Latenzen wichtig. Bei weniger Wissensunterschieden spielt die Latenz keine so große Rolle mehr. Es konnte

ein signifikanter Zusammenhang zwischen auf Grund von Wissensunterschieden benötigten Latenzen für den Kanal zur Steuerung der Kommunikation und binärem Projekterfolg festgestellt werden (vgl. Tabelle 5.4, $x_{47}(kd)$). Die Nullhypothese von H_{47} kann mit einer Fehlerwahrscheinlichkeit von $\alpha < 5\%$ verworfen werden. Dies kann wiederum als Indiz zur Unterstützung von Hypothese 4.7 gedeutet werden.

5.2.6.4 Belege für Hypothese 4.5

Nach Hypothese 4.5 gibt es einen Zusammenhang zwischen Kommunikationsziel und den Latenzanforderungen an den Kanal für die Inhaltsübermittlung. Ist das Kommunikationsziel die kreative Schaffung von Information, so ist eine niedrige Latenz für den Inhaltskanal wichtig, damit die Kommunikationsteilnehmer zeitnah Änderungen des Inhalts mitbekommen und diese in ihren kreativen Prozess einbeziehen können (vgl. Kapitel 3.6.3). Implementierung kann als kreative Kommunikation angesehen werden, wenn die Entwickler zusammen an der Erstellung der Software arbeiten und diese Zusammenarbeit nicht nur zur Abstimmung der Teilergebnisse, sondern auch zur kreativen Wissenserzeugung genutzt wird.

Betrachtet man die Implementierungsphase der hier untersuchten Softwareprojekte als kreative Kommunikation, so müsste nach Hypothese 4.5 die Latenz des Kanals zur Übermittlung des Quellcodes (Inhaltskanal) gering sein, um Kreativität zu ermöglichen. Unter der Annahme, dass erfolgreiche Kommunikation Einfluss auf den Projekterfolg hat, müsste dann auch die Latenz für den Quellcodekanal Einfluss auf den Projekterfolg haben. Das im Softwareprojekt zur Verteilung des Quellcodes eingesetzte Subversion bietet allerdings nur eine recht hohe Latenz. Es ist davon auszugehen, dass die Studenten nur in persönlichen Meetings von Angesicht zu Angesicht auch den Quellcode direkt, d.h. mit geringer Latenz, gesehen haben (z.B. gemeinsam am Notebook). Daher müsste auch ein Zusammenhang zwischen Kommunikationshäufigkeit von Angesicht zu Angesicht in der Implementierungsphase und Projekterfolg erkennbar sein. Tatsächlich konnte in der Studie eine signifikante positive Korrelation zwischen persönlicher Kommunikation (F2F) und Projekterfolg festgestellt werden (vgl. Tabelle 5.4, x_{Team}(F2F) und Erfolg(binär)). Die Nullhypothese von H_{45} kann mit einer Fehlerwahrscheinlichkeit von $\alpha < 5\%$ verworfen werden. Dies kann wiederum unter den oben genannten

Bedingungen als Indiz zur Unterstützung von Hypothese 4.5 gedeutet werden.

5.2.7 Validitätsdiskussion

Zur besseren Einordnung der Ergebnisse dieser Studie werden hier Bedrohungen der Validität nach Wohlin et al. [Wohlin 2000] diskutiert. Da in dieser Studie ein Teil einer Theorie evaluiert wird, sind interne Validität und Konstruktvalidität von besonderer Bedeutung [Wohlin 2000, S. 74].

Interne Validität Einige in dieser Studie nicht explizit betrachteten Faktoren können Einfluss auf den Projekterfolg haben. Im Folgenden werden diese erläutert und es wird diskutiert, warum die Ergebnisse dennoch als valide angesehen werden können:

Dokumentation: Wie in Abbildung 5.9 dargestellt wurde in dieser Studie Dokumentation nicht näher betrachtet. Die Qualität von Zwischendokumenten kann aber einen Einfluss auf den Projekterfolg haben. In kleineren Projekten, wie den hier betrachteten Softwareprojekten, können Dokumentationsprobleme durch Kommunikation abgeschwächt werden. So ist es zum Beispiel bei den hier betrachteten Projekten im Falle schlecht dokumentierter Anforderungen noch in der Implementierungsphase möglich gewesen mit dem Kunden zu kommunizieren und so dennoch korrekte Software zu liefern.

Helden: Helden sind Studenten, die durch große Fähigkeiten oder übermäßigen Einsatz das Projekt alleine oder zu zweit bewältigen können (engl. heroic effort)

Mangelnde Programmierfähigkeiten: Mangelnde Programmier- und SE-Kenntnisse können dazu führen, dass Projekte trotz ausreichender Kommunikation scheitern. Dieser Effekt wurde durch Zulassungsvoraussetzungen im SWP 10/11 gemindert. Seit dem Wintersemester 2010/2011 müssen Studenten die Grundlagenvorlesungen Programmieren I (Scheme), Programmieren II (Java), Grundlagen der Softwaretechnik und Softwarequalität bestanden haben, um am Softwareprojekt teilnehmen zu dürfen.

Mangelnde Sprachfähigkeiten: Mangelnde Sprachkenntnisse verhindern die Integration einzelner Studenten in das Team. So können zusätzlich Konflikte entstehen.

Mangelnde soziale Kompetenz: Mangelnde soziale Kompetenz kann die effektive Zusammenarbeit behindern. Daraus resultierende Konflikte können zu Projektfehlschlägen führen. Diese Konflikte können meist nicht durch Team-interne Kommunikation gelöst werden. Für solche Fälle gibt es im Softwareprojekt den Mechanismus der Eskalation, der, wenn er rechtzeitig von den Studenten angewandt wird, dazu führen kann, dass durch Intervention von außen interne Konflikte gelöst werden können und das Projekt normal weiter laufen kann.

Mangelnde Motivation: Auf Grund mangelnder Motivation arbeiten einige Studenten nicht produktiv am Softwareprojekt mit. Daraus resultierende Konflikte können wie oben mit Eskalationen gelöst werden. Zudem ist die Motivation durch Studiengebühren und viele Leistungspunkte für die Lehrveranstaltung Softwareprojekt meist auf einem hohen Niveau.

Andere Veranstaltungen: Die Studenten besuchen parallel zum Softwareprojekt noch andere Veranstaltungen. Das kann dazu führen, dass sie nicht genug Zeit in das Softwareprojekt investieren. Sofern sich die Studenten an den empfohlenen Lehrplan für das Bachelorstudium halten, sollten sie allerdings ausreichend Zeit für das Softwareprojekt haben.

Leichte Aufgabe: Die Annahme, dass der Projekterfolg stark mit dem Kommunikationserfolg zusammen hängt, kann falsch sein. Insbesondere wenn die gegebene Aufgabenstellung so leicht ist, dass sie ein Student in der gegebenen Zeit alleine lösen kann, dann ist wenig bis keine Kommunikation zur Abstimmung notwendig. Die Ergebnisse der Metrik zur Abschätzung des Projektaufwands deuten aber an, dass alle Aufgaben mindestens so umfangreich waren, dass Kommunikation erfolgskritisch war (vgl. Tabelle 5.3).

Die meisten Bedrohungen der internen Validität konnten direkt durch das Studiendesign gemindert werden. Der verfälschende Einfluss von Bedrohungen wie Helden, mangelnde Motivation oder mangelnde soziale Kompetenz wurden durch die große Anzahl betrachteter Projekte (26) und Studenten (130) minimiert.

Konstruktvalidität In dieser Studie wurde je Variable nur eine Metrik verwendet, mit Ausnahme der Bestimmung der Anforderungserfüllung (vgl. S. 248). Dies kann dazu führen, dass die Werte der Variablen durch Eigenheiten der Metrik verzerrt werden.

Weiterhin kann die Verwendung von Fragebögen zur Datenerhebung dazu geführt haben, dass Studenten mehr Kommunikation angegeben haben als sie tatsächlich durchgeführt haben. Dem wurde durch Nutzung des Medians über alle Antworten aus einem Team versucht entgegenzuwirken. Eine detailliertere Erhebung des Kommunikationsverhaltens der Studenten, zum Beispiel durch direkte Beobachtung oder Auswertung von E-Mail- oder Chat-Logs, war aus zeitlichen und rechtlichen Gründen nicht möglich.

Schlussfolgerungsvalidität Die Fragebogen-basierten Metriken sind subjektiv beeinflusst. D.h. das gemessene Kommunikationsverhalten muss nicht unbedingt mit dem tatsächlichen Kommunikationsverhalten übereinstimmen. Das kann dazu führen, dass tatsächlich vorhandenen Effekte nicht gefunden werden, oder es werden Effekte gefunden, die in Wirklichkeit gar nicht existieren. Mit dieser Bedrohung ist in allen Fragebogen-basierten Studien zu rechnen und muss durch möglichst sorgfältige Vorbereitung der Fragebögen und Durchführung der Befragung entgegengewirkt werden. Dies wurde auch in dieser Studie berücksichtigt. Zudem wurden die Studenten nicht direkt bei der Kommunikation während des Projekts beobachtet (siehe oben), sodass Details wie Missverständnisse auf Basis von Wissensunterschieden nicht direkt gemessen werden konnten.

Externe Validität In dieser Studie wurden studentische Projekte untersucht. Dies schränkt die Generalisierbarkeit der Ergebnisse auf industrielle Projekte ein. Der Vorteil ist aber, dass die Umgebung leichter kontrolliert werden kann, um nicht steuerbare Einflüsse zu minimieren, und leichter eine große Anzahl vergleichbarer Projekte betrachtet werden können. So konnten in der hier vorgestellten Studie Daten aus 26 Projekten mit ähnlicher Größe und ähnlichen Bedingungen ausgewertet werden. So war es möglich auch statistisch signifikante Ergebnisse zu erzielen. Eine so große Anzahl vergleichbarer industrieller Projekte zu untersuchen ist sehr schwierig und sehr teuer. Insgesamt lässt sich feststellen, dass ein Quasi-Experiment mit studentischen Projekten einen guten Mittelweg zwischen realistischen Bedingungen und ausreichender Kontrolle darstellt. Das sehen u.a. auch Juristo und Moreno so, indem sie sagen,

dass "klinische [d.h. industrielle] Studien durch Praktiker genauso wichtig wie Laborversuche durch Forscher sind." [Juristo 2001, S. 8].

5.2.8 Zusammenfassung

Ziel dieser Studie war es, den Einfluss von Kommunikation auf den Erfolg von Softwareprojekten zu analysieren. Dabei sollten einige Hypothesen aus Kapitel 4 geprüft werden. Es konnten signifikante Hinweise zur Unterstützung von vier der sechs untersuchten Hypothesen gefunden werden. Insbesondere scheint die Wahl geeigneter Kommunikationsmedien zur Steuerung von Kommunikation stark von den Wissensunterschieden der Teilnehmer abhängig zu sein. Zudem konnte der aus der Literatur bekannte starke Einfluss von direkter und gesprochener Kommunikation auf den Projekterfolg belegt werden. Das Ergebnis der Studie bekräftigt also eine der Grundannahmen der Arbeit, nämlich dass Kommunikation eine zentrale Rolle in der Softwareentwicklung spielt.

5.3 Interpretation anderer Studien aus Informationsflusssicht

In diesem Kapitel werden Ergebnisse aus anderen empirischen Studien aus Sicht der Informationsflusstheorie neu interpretiert. Damit wird gezeigt, dass die Definitionen und Theoreme der Informationsflusstheorie grundlegend genug sind, um charakteristische Softwareentwicklungsphänomene erklären zu können. Die Auswahl der Studien beschränkt sich auf das vergangene Jahrzehnt, um sicherzustellen, dass die Informationsflusstheorie auch aktuelle Phänomene beschreiben kann. Die Auswahl und Präsentation der empirischen Studien wird anhand der Struktur von Kapitel 4 vorgenommen, d.h. es werden Studien zur Kommunikation (vgl. Kapitel 5.3.1), zur Medienwahl (vgl. Kapitel 5.3.2) und zur Problemgröße (vgl. Kapitel 5.3.3) analysiert. Je Unterkapitel werden dabei nur ein oder zwei Übersichtsarbeiten präsentiert, um eine repräsentative Stichprobe der in den vergangenen Jahren aktuellen Themen zu bekommen.

5.3.1 Empirische Untersuchungen zur Kommunikation

Kommunikationsprobleme treten besonders häufig in verteilten Projekten zu Tage. Zudem wird im Moment viel im Bereich der global verteilten Softwareentwicklung geforscht. Daher wird die hier präsentierte Metastudie aus dem Bereich des Global Software Engineering (GSE) gewählt.

In einer systematischen Literaturauswertung haben da Silva et al. aktuelle Herausforderungen und Lösungen im Management von verteilten Softwareprojekten identifiziert. Die 54 untersuchten Primärstudien wurden zwischen 1998 und 2009 publiziert. 40 Studien (74 %) wurden nach 2006 veröffentlicht [Silva 2010]. Von den 54 Primärstudien sind 22 (41 %) empirisch (d.h. Ergebnisse stammen aus direkten Beobachtungen oder Experimenten), 20 (37 %) sind theoretisch oder konzeptionell (d.h. Ergebnisse stammen aus eigenen Erfahrungen oder der Literatur), 11 (20 %) sind Berichte aus der industriellen Praxis und eine Studie ist ein weiteres systematisches Literaturreview [Silva 2010].

Im Folgenden werden die am häufigsten identifizierten Herausforderungen der verteilten Softwareentwicklung aufgelistet und mit Hilfe der Informationsflusstheorie eine Erklärung geliefert, warum diese Herausforderung im GSE

zu finden ist. Es wird sich auf die Herausforderungen beschränkt, die in mindestens 20 % (11) der betrachteten Studien genannt wurden [Silva 2010].

Effektive Kommunikation (34 Studien): Nach Satz 4.3 gibt es einen Zusammenhang zwischen dem gemeinsamen Kontext der an einer Kommunikation beteiligten Personen und Kommunikationseffektivität. In verteilten Projekten müssen Personen miteinander kommunizieren, die sich häufig noch nicht so gut kennen oder durch die Verteilung nicht mehr so gut kennen. Das gilt insbesondere für global verteilte Projekte. D.h., es ist von Wissensunterschieden auf Projektebene und mehr auszugehen. Je größer die Wissensunterschiede sind, desto geringer ist die Kommunikationseffektivität (vgl. Korollare 4.1 und 4.2). Zudem ist nach Satz 4.5 die Kommunikationseffektivität negativ beeinflusst, wenn die Zielinformationsspeicher unbekannt sind. Eine sich daraus ergebende neue Hypothese ist daher, dass die Kommunikation in global verteilten Projekten, bei denen die Teams in früheren Projekten schonmal zusammengearbeitet haben, sehr viel effektiver ist, als bei neu zusammengestellten verteilten Teams. Ein weiterer negativer Einfluss auf die Kommunikationseffektivität können die Zielunterschiede sein, die bei der Zusammenarbeit mehrerer Firmen auftreten können (vgl. Kapitel 3.6.2).

Kulturunterschiede (31 Studien): Der gerade beschriebene Zusammenhang ist nach Korollar 4.1 bei Wissensunterschieden auf Kulturebene am größten. Selbst wenn eine gemeinsame Sprache vorhanden ist, kann es auf Grund von kulturellen Unterschieden dennoch zu Missverständnissen kommen (vgl. Kapitel 3.6.1).

Koordination (23 Studien): Bei verteilter Arbeit ist es nicht mehr so leicht den Status der Arbeit am anderen Standort mizubekommen. D.h., es stellen sich ohne geeignete Gegenmaßnahmen unweigerlich Wissensunterschiede auf Projektebene ein. Diese haben negativen Einfluss auf die Kommunikationseffektivität und damit auch negativen Einfluss auf die Koordination.

Zeitzonenunterschiede (19 Studien): Zeitzonenunterschiede erschweren synchrone Kommunikation oder verhindern sie ganz. D.h. es können nur Kommunikationsmittel mit großer Latenz (vgl. Kapitel 3.5.3) genutzt werden. Nach Hypothese 4.7 sind aber bei großen Wissensunterschieden wie sie im GSE auftreten gerade Medien mit geringer Latenz wichtig, um effektiv kommunizieren zu können.

Vertrauen (13 Studien): Fehlendes Vertrauen in die Kommunikationsteilnehmer führt dazu, dass wichtige Informationen gar nicht oder sogar bewusst falsch kommuniziert werden. Nach Informationsflusstheorie kann Vertrauen als indirekter Einfluss auf die Zielunterschiede der Kommunikationsteilnehmer interpretiert werden (vgl. Kapitel 3.6.2). Zielunterschiede haben nach den Hypothesen 4.3, 4.6 und Satz 4.7 Einfluss auf den Kommunikationserfolg.

Asymmetrische Prozesse, Richtlinien und Standards (13 Studien): Prozesse, Richtlinien und Standards, die zwischen den Standorten nicht zusammenpassen, führen häufig auch dazu, dass es keine Möglichkeiten zur Kommunikation und Koordination gibt. Das führt dann wieder zu den oben beschriebenen Problemen. Zudem können viele Asymmetrien zu einer großen Abstimmungskomplexität führen (vgl. Kapitel 4.4), die wiederum den Abstimmungsaufwand (vgl. Satz 4.11) und die Entwicklungsdauer (vgl. Satz 4.12) vergrößern kann.

Räumliche Entfernung (13 Studien): Räumliche Entfernung ist nur eine weitere Ursache für die bereits oben diskutierten Probleme wie Koordinationsprobleme und ineffektive Kommunikation.

IT-Infrastruktur (13 Studien): Problematische IT-Infrastruktur deutet auf das Fehlen eines standortübergreifenden Inhaltskanals hin (vgl. Hypothese 4.1). Wenn die Inhalte, an denen gearbeitet wird, nicht an allen Standorten zugreifbar sind, dann kann darüber nicht richtig kommuniziert werden (vgl. Satz 4.9). Weiterhin kann nicht wirklich gemeinsam an einem Produkt gearbeitet werden, d.h. der essenzielle Informationsfluss der Dokumentation des Softwareprodukts ist gestört (vgl. Kapitel 3.7.2).

Unterschiedliche Wissensebenen oder Wissenstransfer (11 Studien): Unterschiedliche Wissensebenen, d.h. Wissensunterschiede in Informationsflussterminologie, haben direkten Einfluss auf den Kommunikationserfolg (vgl. Kapitel 3.6.1 und Korollare 4.1, 4.3). Probleme beim Wissenstransfer sind Kommunikationsprobleme (vgl. Kapitel 4.2).

Im Folgenden werden die am häufigsten identifizierten Lösungen für die Herausforderungen der verteilten Softwareentwicklung aufgelistet. Mit Hilfe der Informationsflusstheorie wird eine Erklärung geliefert, warum diese Vorschläge das Problem lösen können. Es wird sich auf die Lösungen beschränkt, die in mindestens 10 % (6) der betrachteten Studien genannt wurden [Silva 2010].

Bereitstellung von und Schulung in Kollaborationswerkzeugen (10 Studien): Der Einsatz von Kollaborationswerkzeugen soll helfen die Kommunikationsprobleme zu minimieren. Aus Informationsflusssicht scheint dieses Vorgehen allerdings nur ein Symptom zu lindern und nicht die Ursache anzugehen. Eine Maßnahme, die gezielt Wissensunterschiede verkleinert, wäre eine Lösung, die direkt die Ursache des Problems beseitigt, und damit auch direkt das Kommunikationsproblem lösen könnte.

Einsatz mehrerer Kommunikationsmedien inkl. F2F (8 Studien): Der Einsatz von verschiedenen, auf die Situation angepassten, Kommunikationsmedien ist nach Kapitel 4.3 ein sinnvolles Vorgehen. Da, wie oben bereits erwähnt, in verteilten Projekten häufig große Wissensunterschiede existieren, ist nach Hypothese 4.2 direkte Kommunikation von Angesicht zu Angesicht wichtig für die Steuerung der Kommunikation.

Arbeitsteilung über wohl definierte Module (7 Studien): Wie in Kapitel 4.4 und 5.1.6 erläutert, ist es sinnvoll, die Arbeit so aufzuteilen, dass zwischen den einzelnen Teilen möglichst wenig Abstimmungsabhängigkeiten existieren. Um so weniger Abstimmungsabhängigkeiten existieren, desto leichter kann die Arbeit auf verschiedene Standorte verteilt werden.

Schulung in Kulturunterschieden (7 Studien): Die Schulung in Kulturunterschieden hilft, den Wissensunterschied von der Kulturebene auf die Domänenebene oder weniger zu reduzieren. Nach den Korollaren 4.1 und 4.3 kann dann erfolgreicher kommuniziert werden.

Kommunikationsprotokolle (6 Studien): In verteilten Umgebungen müssen häufig Medien eingesetzt werden, deren Nutzung für viele Kommunikationsteilnehmer ungewohnt ist (z.B. Skype oder Wikis). Protokolle, die die Nutzung dieser Medien festlegen, können helfen, dass die Teilnehmer die ungewohnten Medien schneller effektiv nutzen können (vgl. Kapitel 4.3.3).

Mitarbeiterführung (6 Studien): Eine Möglichkeit Problemen bei der Abstimmung entgegenzuwirken ist, die Mitarbeiter gezielt zu motivieren und zu fördern. Aus Informationsflusssicht kann das helfen, die Zielunterschiede der Kommunikationsteilnehmer zwischen den verschiedenen Standorten zu minimieren. D.h., wenn das Management sich standortübergreifend einig ist, kann es durch Maßnahmen der Mitarbeiterfüh-

rung dafür sorgen, dass die Kommunikation auf den tieferen Ebenen besser funktioniert.

Informelle Interaktionen fördern (6 Studien): Informelle Interaktionen, im speziellen informelle Kommunikation, hilft bei der Verkleinerung von Wissensunterschieden. Selbst wenn über nicht-projektrelevante Dinge kommuniziert wird sorgt die Interaktion für eine Vergrößerung des gemeinsamen Kontextes, der wiederum zu erfolgreicherer Kommunikation führt (vgl. Sätze 4.3 und 4.4).

Ein weiteres Ergebnis der Auswertung von da Silva et al. ist, dass sehr viele Studien (19, 35 %) zur Lösung der Kommunikationsprobleme eher traditionelle Werkzeuge wie E-Mail, Telefon und Sofortnachrichtendienste einsetzen [Silva 2010]. Dabei werden aber in den meisten Fällen nicht die Auswirkungen des Einsatzes dieser Werkzeuge berichtet. Bis auf zwei der oben genannten Vorschläge (Kulturschulung und Module) scheint keine der Lösungen die Ursachen der Probleme der verteilten Softwareentwicklung anzugehen.

Insgesamt konnte gezeigt werden, dass sich alle betrachteten Probleme der verteilten Softwareentwicklung und alle betrachteten Lösungen mit Hilfe der Definitionen und Theoreme der Informationsflusstheorie erklären lassen. Die Informationsflusstheorie ist also eine geeignete Grundlage zur Beschreibung aktueller Probleme und Lösungen im Global Software Engineering.

5.3.2 Empirische Untersuchungen zur Medienwahl

Im Bereich der Medienwahl konnte keine aktuelle Metastudie gefunden werden. Daher werden zunächst zwei aktuelle Studien über die Medienwahl in global verteilten Softwareprojekten [Niinimaeki 2009, Niinimaeki 2010], eine Studie über die Medienwahl in der Anforderungsnegotiation [Erra 2010] und anschließend eine Literaturauswertung aus dem Jahr 1997 vorgestellt [Bordia 1997] und mit den Vorhersagen der Informationsflusstheorie verglichen.

5.3.2.1 Text versus Audio

In einer Studie über acht global verteilte Softwareentwicklungsprojekte haben Niinimaeki et al. [Niinimaeki 2009] Faktoren untersucht, die die Wahl zwischen Text-basierter und Audio-basierter Kommunikation beeinflussen. Es

wurden 57 Interviews mit Projektbeteiligten durchgeführt. Als wichtigste Faktoren wurden Rolle im Projekt, Sprachfähigkeiten, Kommunikationskontext und Aufgabeneigenschaften identifiziert. Im Folgenden werden diese Faktoren erläutert und es wird mit Hilfe der Informationsflusstheorie erklärt, warum diese in global verteilten Projekten eine Rolle spielen.

Rolle: Ein Ergebnis der Studie ist, dass technisches Personal Text- über Audio-basierte Kommunikation bevorzugt. Als Begründung wird u.a. die Notwendigkeit genannt, auch Informationen über technische Arbeitsartefakte kommunizieren zu müssen. Diese technischen Artefakte, wie Quellcode, Konfigurationsdateien oder Testfälle, sind meist komplex und nur schwer verbal ausdrückbar. Nach Informationsflusstheorie ist aber nicht die Rolle (technisch vs. leitend) der ausschlaggebende Faktor, sondern nach Hypothese 4.1 und Satz 4.9 der zu kommunizierende Inhalt. In Tabelle 4.1 sind unterschiedliche Aufwände für die Konvertierung zwischen den verschiedenen Datentypen aufgeführt. Der Tabelle ist zu entnehmen, dass die Umwandlung von 1-dimensional visuell künstlichen Daten (z.B. Quellcode) in 1D auditiv natürliche Daten aufwändiger ist als in 1D visuell natürliche Daten. Das könnte der eigentliche Grund für die in der Studie festgestellte Bevorzugung von Text-basierten Medien durch das technische Personal sein.

Sprachfähigkeiten: Bezüglich der Sprache wurden zwei Beobachtungen gemacht. (1) Da Englisch als standortübergreifende Sprache genutzt wurde und für beide Seiten Englisch nicht die Muttersprache ist haben beide Seiten einen anderen Akzent. Die unterschiedlichen Akzente haben dazu geführt, dass die Gegenseite in Audio-basierten Gesprächen nur sehr schwer verstanden wurde. (2) Zudem hat die Selbsteinschätzung der eigenen Fremdsprachenfähigkeiten dazu geführt, dass Sprachkanäle gemieden und Textkanäle bevorzugt wurden. Zunächst sollte festgehalten werden, dass die Probanden in der Studie die Medien frei wählen konnten. Es wird keine Aussage darüber gemacht, wie gut diese Wahl unter Effektivitäts- und Effizienzgesichtspunkten ist. Zu beiden Punkten macht die Informationsflusstheorie keine Aussage. Sie stehen aber auch nicht im Widerspruch zu ihr. Insbesondere beim zweiten Punkt ist zu bezweifeln, ob das die effektivere Strategie ist. Durch das Ausweichen auf einen Text-basierten Kanal können zwar Sprachprobleme besser versteckt werden, aber auch Missverständnisse sind schwerer identifizierbar (vgl. Hypothese 4.2). Letzteres ist aber gerade bei Sprachproblemen sehr wichtig.

Kommunikationskontext und Aufgabeneigenschaften: Als Kommunikationskontext werden die Eigenschaften der zu lösenden Aufgabe und die Gesamtsituation des Projekts gesehen. Bei informierenden Aufgaben ohne großen Interpretationsspielraum wurden Text-basierte Medien bevorzugt. Bei kreativen komplexen Aufgaben hingegen wurde das Telefon bevorzugt. Dieses Ergebnis deckt sich mit der Media Richness Theory [Daft 1987]. Bei wichtigen projektrelevanten Informationen, wie Entscheidungen oder Anforderungsänderungen, wurde die ursprüngliche Information zwar auditiv kommuniziert, danach wurde sie zu Dokumentationszwecken aber noch einmal schriftlich festgehalten. Bei dringenden Anfragen wurde das Telefon Text-basierten Medien vorgezogen, um direkt eine Antwort zu erhalten. Zusammenfassend wurden also drei Eigenschaften der zu bewältigenden Aufgabe identifiziert, die Einfluss auf die Medienwahl haben: (1) Mehrdeutigkeit, (2) Projektrelevanz und (3) Dringlichkeit. Das Äquivalent zur Mehrdeutigkeit der Media Richness Theory in der Informationsflusstheorie sind das Kommunikationsziel (vgl. Kapitel 3.6.3) und die Wissensunterschiede der Kommunikationsteilnehmer (vgl. Kapitel 3.6.1). Ist das Ziel informieren und haben die Teilnehmer nur wenig Wissensunterschiede reicht nach Hypothese 4.4 ein Text-basierter Kanal aus. Ist hingegen Kreativität das Ziel so benötigt man nach Hypothese 4.5 einen Kanal mit niedriger Latenz, was in der Studie wahrscheinlich bei den Audio-Kanälen gegeben war. Dringende Angelegenheiten, bei denen schnelles Feedback erforderlich ist, werden durch Kombination aus Kommunikationseffizienz (Def.: 3.37) und niedriger Latenz (Def.: 3.31) abgedeckt.

In einer weiteren Studie haben Niinimaeki et al. [Niinimaeki 2010] 12 global verteilte Softwareentwicklungsprojekte mit folgenden Fragestellungen untersucht:

1. Wie werden unterschiedliche Medien in GSD Projekten genutzt?
2. Für welchen Zweck werden diese Medien genutzt?
3. Kann Media Synchronicity Theory helfen Medien in GSD auszuwählen?

Auch in dieser Studie wurde festgestellt, dass Instant Messaging (IM) von technischem Personal für z.B. Source Code bevorzugt wurde. Nach Informationsflusstheorie ist die Rolle aber nur indirekt für diese Wahl verantwortlich. Der

textuelle Kanal ist einfach besser für den technischen Inhalt ausgelegt als gesprochene Sprache (vgl. Def. 3.8 und Tabelle 4.1). Technisches Personal muss öfter technische Inhalte kommunizieren als z.B. das Management und bevorzugt daher Text-basierte Medien.

5.3.2.2 3D natürlich vs. 3D künstlich vs. 1D natürlich

In [Erra 2010] vergleichen Erra und Scanniello die Medien Angesicht-zu-Angesicht, eine 3-dimensionale virtuelle Umgebung (basierend auf SecondLive) und strukturierten Text-Chat in Bezug auf Effektivität und Effizienz der Anforderungsnegotiation miteinander. Dabei hat sich gezeigt, dass eine Anforderungsnegotiation von Angesicht zu Angesicht im Durchschnitt weniger Zeit benötigt als bei Nutzung der anderen beiden Medien. Ein überraschendes Ergebnis ist, dass beim Einsatz der virtuellen Umgebung zur Anforderungsnegotiation mehr Probleme identifiziert und gelöst werden konnten, als bei Angesicht-zu-Angesicht und Text-Chat. Dabei wurde in der virtuellen Umgebung fast ausschließlich Text-Chat zur Kommunikation benutzt, Audio nur sehr selten am Anfang in der Kennenlernphase. Bezüglich der Qualität der in der Negotiation entstandenen Anforderungen konnte kein deutlicher Unterscheid zwischen den Kommunikationsmedien festgestellt werden.

Um diese Ergebnisse mit Hilfe der Informationsflusstheorie interpretieren zu können sind noch einige Angaben zu den Probanden notwendig. An der Studie haben 44 Bachelorstudenten der Informatik teilgenommen. 32 haben Vorkenntnisse aus einer SE-Vorlesung und haben als Anforderungsanalysten fungiert, die restlichen 12 hatten Vorkenntnisse aus einer Betriebssystemvorlesung und haben die Rolle des Kunden übernommen. Die Kunden hatten keine explizite SE-Vorbildung.

Nach der Definition in Kapitel 3.6.1 wären die Wissensunterschiede zwischen Analyst und Kunde auf der Ausbildungsebene einzuordnen. Da die Studenten im Studium ähnlich weit fortgeschritten waren, die Kunden zwar keine SE-Vorkenntnisse hatten, aber davon auszugehen ist, dass die Analysten Betriebssystem-Vorbildung hatten, kann man evtl. von Wissensunterschieden nur auf Projektebene ausgehen. Das Kommunikationsziel (vgl. Kapitel 3.6.3) einer Negotiation ist es Entscheidungen bezüglich der Anforderungen zu treffen und die individuellen Zielunterschiede (vgl. Kapitel 3.6.2) sind bei dieser studentischen Gruppe als koordiniert oder gar kollaborativ einzustufen. Der Inhalt

der Negotiation sind Anforderungen, wobei in der Studie als Endergebnis tabellarische Use Cases gefordert waren (schriftlich).

Die Ergebnisse der Studie können nun wie folgt neu interpretiert werden: Nach den Hypothesen 4.3 und 4.4 ist ein Kanal für geschriebene natürliche Sprache ausreichend für die Steuerung der Kommunikation. Dieser war bei allen drei untersuchten Medien gegeben. Das erklärt, dass es keine Unterschiede bezüglich der Effektivität zwischen den Medien gab. Alle Gruppen haben ihre Aufgabe erfolgreich gelöst. Der Zeitunterschied zu Gunsten der Angesicht zu Angesicht Gruppen ist mit den erhöhten Latenzanforderungen des Kommunikationsziels (vgl. Hypothese 4.5) und der sehr effizienten Steuerung des 3D visuell natürlichen Kanals (vgl. Hypothese 4.2) zu erklären.

Das überraschende Ergebnis, dass die virtuellen Teams während der Negotiation mehr Probleme identifizieren und lösen konnten, lässt sich mit der Informationsflusstheorie so erklären: In der virtuellen Umgebung war neben Audio- und Text-Chat auch eine Möglichkeit zur Dokumentation der Use Cases vorhanden. D.h., es war ein Kanal vorhanden, der sehr gut zum finalen Inhalt passt. Da die Use Cases so wahrscheinlich relativ früh dokumentiert wurden und dann zum Review verfügbar waren, konnten Probleme schneller identifiziert werden. Im Angesicht-zu-Angesicht-Fall fehlte dieser inhaltlich passende Kanal und im strukturierten Text Chat wurden die Use Cases erst in einer zweiten Phase formal dokumentiert.

Zusammenfassend kann festgestellet werden, dass die relevantesten Ergebnisse der Erra-Studie schlüssig mit der Informationsflusstheorie beschrieben werden können.

5.3.2.3 Metastudie: Kommunikation von Angesicht zu Angesicht versus Computer-mediierte Kommunikation

In einem Literaturreview über 18 Studien zum Vergleich von direkter Kommunikation von Angesicht zu Angesicht mit Computer-mediierter Kommunikation (CMC) hat Bordia [Bordia 1997] zehn Theoreme hergeleitet, die zentrale Ergebnisse der untersuchten Studien zusammenfassen. Einige dieser Theoreme lassen sich auch mit Hilfe der Informationsflusstheorie erklären. In den betrachteten Studien ist mit CMC meist Text-basierte Kommunikation über E-Mail oder Instant Messaging gemeint. D.h. es wird ein 1D visuell natürlicher Datentyp zur Kommunikation benutzt.

Theoreme 1, 2 und 5: *CMC Teams brauchen länger, um eine Aufgabe zu erledigen.*

Als Hauptgrund für den häufig festgestellten Zeitunterschied bei der Bewältigung von Kommunikationsaufgaben wird die im Vergleich zum Sprechen langsamere Tippgeschwindigkeit genannt. Dieses Ergebnis lässt sich auch mit Hilfe der Informationsflusstheorie erklären. Nach Tabelle 3.6 beträgt die Externalisierungsbandbreite für gesprochene natürliche Sprache mehr als das Dreifache der Externalisierungsbandbreite für getippte geschriebene natürliche Sprache. Nach Lemma 4.1 hat das negative Auswirkungen auf die Kommunikationseffizienz.

Theorem 3 und 5: *CMC Teams sind besser bei der Erzeugung von Ideen.*

Als Grund für die bessere Produktivität von CMC Teams bei der Erzeugung von nicht-redundanten Ideen werden weniger Unterbrechungen bei der Erzeugung und leichtere Auswertbarkeit der Nachrichten genannt. Dies lässt sich zwar nicht direkt aus der Informationsflusstheorie herleiten, stellt aber auch keinen Widerspruch dar. Z.B. könnte man das Phänomen folgendermaßen erklären: Die sehr hohe Bandbreite des Angesicht-zu-Angesicht-Kanals führt bei vielen Kommunikationsteilnehmern dazu, dass Individuen viel Zeit damit verbringen den anderen zuzuhören, zuzusehen, und die vielen Informationen zu verarbeiten, und dadurch weniger Zeit haben über eigene Ideen nachzudenken oder keine Chance bekommen diese den anderen mitteilen zu können.

Theorem 4 und 9: *CMC Teams weisen eine gleichmäßigere Beteiligung und hemmungsloseres Verhalten auf.*

Als Grund für die gleichförmigere Beteiligung der Kommunikationsteilnehmer und das hemmungslosere Verhalten in CMC Teams wird die verminderte Bedeutung von Statusunterschieden genannt. Aus Informationsflusssicht können Statusunterschiede als indirekter Einfluss auf Zielunterschiede interpretiert werden. Die Kommunikationsteilnehmer müssen abwägen, ob sie bei der Kommunikation die Ziele des anwesenden Vorgesetzten oder die der Kommunikationsaktivität verfolgen. In Computer-mediierter Kommunikation ist der Status der Teilnehmer meist nicht direkt erkenntlich. D.h. alle können sich auf das eigentliche Kommunikationsziel konzentrieren, was zu einem größeren Kommunikationserfolg führt. Zudem macht die Informationsflusstheorie keine Aussage darüber, wie sich Unterschiede in der Beteiligung auf den Kommunikationserfolg auswirken. Es entsteht also kein Widerspruch.

Theorem 6, 10 und 7: *CMC Teams weisen weniger Kompromissbereitschaft, weniger Meinungsänderungen und mehr Verständnisschwierigkeiten auf.*
Als Grund für die geringere Kompromissbereitschaft wird u.a. das Fehlen eines klaren Verständnisses über die Präferenzen der anderen Kommunikationsteilnehmer in CMC Teams genannt. Wenn der Standpunkt des Gegenüber nicht bekannt ist, ist es auch unwahrscheinlicher, dass man sich darauf einlässt. Gleiches gilt für Meinungsänderungen. Diese Verständnisschwierigkeiten bei Text-basierter Kommunikation können mit der Informationsflusstheorie erklärt werden. Durch die geringe Bandbreite rein Text-basierter Kommunikation dauert es länger Ideen zu übermitteln und es ist schwieriger Missverständnisse zu erkennen und zu beseitigen. Nach den Hypothesen 4.2 und 4.3 ist der 2-dimenionale visuelle natürliche Kanal am Besten für die Steuerung einer Kommunikation bei widersprüchlichen Zielunterschieden geeignet.

Theorem 8 wurde nicht in den Vergleich einbezogen, weil es dabei um die Bewertung der Kommunikationspartner und des Kommunikationsmediums ging, und nicht um den Kommunikationserfolg selbst.

Insgesamt lässt sich feststellen, dass alle Theoreme entweder mit der Informationsflusstheorie erklärt werden können oder sie nicht im Widerspruch zu ihr stehen.

5.3.3 Empirische Untersuchungen zur Problemgröße

Studien, deren Ergebnisse mit Hilfe der Theoreme aus Kapitel 4.4 interpretiert werden können, finden sich im Bereich der Aufwandsschätzung von Softwareprojekten. Im vergangenen Jahrzehnt gab es zu dem Thema eine große Metastudie [Jorgensen 2007].

In einem systematischen Literaturreview haben Jørgensen und Shepperd 304 Artikel aus 76 Journalen über Studien zum Thema Kostenschätzung in der Softwareentwicklung identifiziert [Jorgensen 2007]. Ein wesentliches Ergebnis dieser Metastudie ist, dass Methoden, die auf Expertenschätzungen aufbauen, in der Industrie sehr viel verbreiteter sind, als formale Schätzmethoden, obwohl letztere auch schon sehr lange existieren. Dennoch werden Methoden, die auf Expertenschätzungen basieren, wissenschaftlich nicht entsprechend betrachtet.

> *In spite of the fact that formal estimation models have existed for many years, the dominant estimation method is based on expert judgment. Further, available evidence does not suggest that the estimation accuracy improves with use of formal estimation models. Despite these factors, current research on, e.g., expert estimation is relatively sparse and we believe that it deserves more research effort.* [Jorgensen 2007]

Dieses Ergebnis deckt sich mit den Vorhersagen der Informationsflusstheorie. Demnach sind Schätzungen nur relativ zu vergangenen Projekten sinnvoll (vgl. Kapitel 4.4). Die Formeln aus Kapitel 4.4 können nur dann sinnvoll eingesetzt werden, wenn Problemkomplexität, Abstimmungsaufwandsanteil und Abstimmungszeit quantifiziert werden können. Das ist, wie in Kapitel 3.4.1 beschrieben, nicht absolut, sondern nur relativ zu ähnlichen Projekten möglich. Die Formeln können aber mit Hilfe von Erfahrungswerten aus vergleichbaren Projekten, bei denen die Werte bekannt sind, sinnvoll eingesetzt werden. Z.B. könnte die Problemgröße, wie in Kapitel 5.2 gezeigt, aus einer Expertenbefragung ermittelt werden. Die Problemkomplexität könnte von Experten aus der Architektur und der Verteilung der darin beschriebenen Module auf die Entwickler abgeleitet werden. Mit Hilfe von Problemkomplexität und Abstimmungszeit könnten dann die anderen Parameter (Zeit, Entwickler, Kosten) gegeneinander abgewogen werden (vgl. Beispiel in Kapitel 4.4, Abb. 4.8 und Tab. 4.3). D.h., nur Experten, die viel Projekterfahrung haben, können überhaupt sinnvolle Schätzungen liefern. Das erklärt auch, warum formale Modelle (ohne Experten-Eingaben) keine genaueren Schätzungen liefern können.

Die Metastudie liefert keine Beschreibung der wissenschaftlich und praktisch dominanten Schätzmethoden. Daher werden an dieser Stelle keine Vergleiche mit einzelnen Schätzmethoden präsentiert.

5.4 Prüfungsergebnis

In diesem Abschnitt wird zusammengefasst, was die in diesem Kapitel vorgestellte Prüfung der Informationsflusstheorie ergeben hat. Im Folgenden wird das Prüfungsergebnis anhand der allgemeinen Qualitätskriterien einer Theorie nach Popper [Popper 2002, S. 7-8] und den SE-spezifischen Kriterien zusammengefasst (vgl. [Broy 2011, Ludewig 2010, Easterbrook 2008, Boehm 2006, Herbsleb 2003, Juristo 2001] und S. 195).

Allgemeine Kriterien:

Prüfung des wissenschaftlichen Fortschritts: In Kapitel 5.1 wird die Informationsflusstheorie mit zehn verwandten Theorien verglichen. Der Vergleich zeigt, dass die Informationsflusstheorie grundlegender und umfangreicher als neun der zehn untersuchten Theorien ist. Zudem zeigt der Vergleich, dass die Informationsflusstheorie nicht im Widerspruch mit etabliertem Software-Engineering-Wissen steht. Teilweise konnten mit Hilfe der Informationsflusstheorie auch neue Software-Engineering-Erkenntnisse hergeleitet werden, z.B. die Erkenntnis, dass es gut für die Softwareentwicklung ist, wenn hierarchische Organisationsstrukturen entsprechend Conway's Law in Softwarearchitekturen übernommen werden, da sie zu linearer Abstimmungskomplexität führen können (vgl. Kapitel 5.1.6). Die Informationsflusstheorie ist daher auch ein wissenschaftlicher Fortschritt.

Empirische Anwendung: In Kapitel 5.2 wird eine eigene Studie zur empirischen Prüfung der Informationsflusstheorie vorgestellt. In der Studie werden sechs Hypothesen der Informationsflusstheorie evaluiert. Es werden statistisch signifikante Belege für vier Hypothesen präsentiert. Durch das verwendete Sudiendesign werden gezielt Vorteile von fallstudienbasierten und experimentellen Untersuchungen genutzt (vgl. Kapitel 5.2.2) und so Bedrohungen der Validität minimiert (vgl. Kapitel 5.2.7). Weiterhin werden in Kapitel 5.3 Ergebnisse anderer empirischer Studien mit Vorhersagen der Informationsflusstheorie verglichen bzw. mit Hilfe der Informationsflusstheorie neu interpretiert. Es hat sich gezeigt, dass die Informationsflusstheorie in den meisten Fällen das Ergebnis der Studien erklären kann.

SE-spezifische Kriterien:

Präzise Terminologie: Nach [Easterbrook 2008, Broy 2011] ist eine präzise Terminologie wichtig für die Entwicklung und Auswertung von empirischen Studien. Die in Kapitel 5.2 vorgestellte Studie wird auf Basis der Definitionen aus Kapitel 3 und der Hypothesen aus Kapitel 4 entwickelt. Es werden testbare Hypothesen und Metriken abgeleitet. Die Ergebnisse werden in Bezug auf die testbaren Hypothesen und die dahinter liegende Informationsflusstheorie interpretiert. Zudem ermöglichen die Definitionen und Theoreme den Entwurf weiterer Studien zur Überprüfung der Informationsflusstheorie.

Bereiche Management und Mensch: Nach Boehm sollte eine Software-Engineering-Theorie die Bereiche Informatik, Management sowie persönliche, kulturelle und ökonomische Werte, die bei der Entwicklung von Software eine Rolle spielen, abdecken [Boehm 2006]. Der Bereich Informatik ist zwar nicht prominent in der Theorie vertreten, aber z.B. durch die Definition von Software (vgl. Def. 3.39) oder die große Bedeutung des Softwaredomänenwissens für die Entwicklung (vgl. Kapitel 3.4.2) eingebunden. Die Sicht auf Softwareentwicklung als Informationsfluss (vgl. Def. 3.41) oder die Klassifikation der möglichen Wissensunterschiede zwischen den verschiedenen Rollen eines Softwareprojekts (vgl. Kapitel 3.6.1 und 3.7.4) gibt dem Management von Softwareprojekten neue Ansatzpunkte zur Verbesserung (vgl. u.a. Kapitel 6.3). Der Mensch spielt in der Informationsflusstheorie eine zentrale Rolle, ohne den keine Softwareentwicklung möglich ist. Dies zeigt sich z.B. an der Definition von Wissen (vgl. Def. 3.2) und der Definition und dem großen Stellenwert von Kommunikation (vgl. Def. 3.24 sowie die Kapitel 3.6 und 4.2). Ökonomische Betrachtungen werden z.B. durch den Effizienzanteil an Kommunikations- und Softwareentwicklungserfolg (vgl. Defs. 3.38 und 3.42) und den Theoremen zur Problemgröße (vgl. Kapitel 4.4) ermöglicht.

Methodenunabhängigkeit: Nach Jacobson und Meyer sollte eine Software-Engineering-Theorie Probleme und nicht deren Lösung beschreiben [Jacobson 2009]. Die in Kapitel 5.3 präsentierte Neu-Interpretation von Studienergebnissen aus der Software-Engineering-Literatur zeigt, dass die Informationsflüsssicht – insbesondere die Ebenen der Wissensunterschiede (vgl. Kapitel 3.6.1) und ihr Einfluss auf den Kommunikationserfolg (vgl. Korollare 4.1 und 4.3) – es ermöglicht, die Ursache von

Problemen zu erklären. Damit ist die Informationsflusstheorie methodenunabhängig.

Fundamentale Sätze der Softwareentwicklung: Wie in Kapitel 4 beschrieben, liefert die Theorie auch Sätze, die grundlegende Prinzipien der Softwareentwicklung beschreiben, z.B.:

- den Zusammenhang von Anzahl an Kommunikationsteilnehmern und Wissensunterschieden (vgl. Satz 4.2),
- die Schwierigkeit der Kommunikation zwischen Fachleuten und Softwareentwicklern (vgl. Korollar 4.5),
- die Auswirkungen der Medienwahl auf den Kommunikationserfolg (vgl. Satz 4.7) oder
- den Zusammenhang zwischen Problemkomplexität und Abstimmungskomplexität (vgl. Satz 4.10).

Diese Theoreme können als fundamentale Sätze der Softwareentwicklung angesehen werden, die in allen Softwareprojekten gültig sind.

Zusammenfassend kann festgestellt werden, dass die Informationsflusstheorie sowohl der Prüfung allgemeiner Qualitätskriterien als auch der Prüfung SE-spezifischer Kriterien standhält und somit eine valide Theorie der Softwareentwicklung darstellt.

6 Der FLOW-Ansatz

In den Kapiteln 3 bis 5 wurde die Informationsflusstheorie vorgestellt und geprüft. Nun kann sie von Forschern des Software Engineering zur Analyse und Prognose von Softwareentwicklungsphänomenen genutzt werden. In diesem Kapitel wird mit Hilfe der Informationsflusstheorie eine Methode entwickelt, die von Praktikern in realen Softwareprojekten genutzt werden kann, um diese zu analysieren und zu verbessern.

Im Zentrum der FLOW-Methode steht ein 3-phasiger Verbesserungsprozess, der Strategien und Verfahren zur (1) Erhebung, (2) Analyse und (3) Verbesserung von Informationsflüssen beschreibt. Im Forschungsprojekt InfoFLOW[1] wurden verschiedene Techniken und Werkzeuge entwickelt (vgl. Kapitel 6.3 für Beispiele), die konkrete Aktivitäten für die praktische Anwendung der Methode in den einzelnen Phasen des Verbesserungsprozesses beschreiben.

Die FLOW-Methode baut auf den Grundideen von FLOW auf. FLOW wurde bereits vor der Informationsflusstheorie von Schneider [Schneider 2004, Schneider 2005, Schneider 2005a] entwickelt. Die Informationsflusstheorie ist aus FLOW hervorgegangen. Sie beschreibt den theoretischen Hintergrund von FLOW. Im folgenden Unterkapitel 6.1 werden zunächst die FLOW-Grundlagen erläutert und in einen Zusammenhang mit der Informationsflusstheorie gebracht. Anschließend wird die FLOW-Methode in Kapitel 6.2 vorgestellt. In Kapitel 6.3 werden ausgewählte FLOW-Techniken beschrieben und ihre Nützlichkeit in Bezug auf die Verbesserung von Softwareprojekten evaluiert.

6.1 FLOW-Grundlagen

FLOW wurde ursprünglich von Schneider [Schneider 2004, Schneider 2005, Schneider 2005a] als Mittel zur Softwareprozessverbesserung entwickelt. Auch

[1] http://www.se.uni-hannover.de/infoflow

bei FLOW stehen Informationsflüsse in Softwareentwicklungsprojekten als Analysegegenstand im Mittelpunkt. FLOW basiert auf einigen Grundannahmen, die im folgenden beschrieben werden:

- Informationsflüsse sind das Bindeglied zwischen dokumentenlastigen (z.B. V-Modell XT) und kommunikationsintensiven (z.B. agil) Ansätzen zur Softwareentwicklung (vgl. [Schneider 2005a]), denn egal wie entwickelt wird, die Informationen, die der Kunde in Form von Anforderungen im Kopf hat, müssen möglichst vollständig und möglichst schnell in das Endprodukt Software gelangen.
- Die grafische Modellierung von Informationsflüssen erleichtert deren Analyse (vgl. [Schneider 2005]).
- Modelle dienen in erster Linie als Diskussionsgrundlage und sind daher so einfach wie möglich zu halten.
- Inhalte werden nur grob modelliert, z.B. fließen Informationen über Anforderungen oder Entwurfsentscheidungen. (vgl. [Schneider 2004])
- Erfahrungen sind eine besonders wertvolle Art von Information und werden daher als eigener Informationstyp modelliert. (vgl. [Schneider 2004, Schneider 2005])
- Informationen haben einen Aggregatzustand. In Anlehnung an feste und flüssige Materie sind Informationen fest, wenn sie permanent gespeichert sind (z.B. in Dokumenten), oder flüssig, wenn sie flüchtig sind (z.B. im Gedächtnis einer Person). (vgl. [Schneider 2004, Schneider 2005])

Der letzte Punkt, die Metapher des Aggregatzustands von Information, ist der zentrale Gedanke von FLOW und wird daher im folgenden Abschnitt genauer erläutert.

6.1.1 Metapher der Aggregatzustände von Information

Eine wesentliche Eigenschaft von FLOW ist die Unterscheidung von festen und flüssigen Informationen und abgeleiteten Flüssen. Die Metapher ist darauf ausgerichtet, schnell zwischen wesentlichen Eigenschaften von Informationsspeichern und -flüssen in verschiedenen Medien unterscheiden zu können.

Feste Informationen sind für diejenigen, die sie benötigen, unverändert wiederholt zugreifbar und verständlich, und das auch noch nach längerer Zeit. Flüssige Informationen hingegen sind nach einer gewissen Zeit nicht mehr abrufbar, z.B. wenn sie vergessen wurden. Flüssige Informationen haben aber den Vorteil, dass sie leichter, d.h. schneller, weitergegeben werden können, weil bei ihnen nicht darauf geachtet werden muss, dass sie jeder verstehen und abrufen kann (vgl. Sätze 4.5 und 4.6). Die Aggregatzustände von Information werden wie folgt definiert.

Definition 6.1: **Feste Information**

Feste Informationen sind Informationen, die in einem Betrachtungsbereich von allen jederzeit *abgerufen* und *verstanden* werden können.

Definition 6.2: **Flüssige Information**

Flüssige Informationen sind Informationen, die nicht fest sind.

Die Unterscheidung zwischen fest und flüssig hängt vom Betrachtungsbereich ab. Ein Betrachtungsbereich ist wie folgt definiert.

Definition 6.3: **Betrachtungsbereich**

Ein Betrachtungsbereich ist

- eine Menge von Personen und
- eine Zeitspanne.

D.h. Informationen sind in einem Betrachtungsbereich dann fest, wenn sie

- von allen Personen in diesem Betrachtungsbereich und
- zu jedem Zeitpunkt innerhalb der Zeitspanne des Betrachtungsbereichs

abgerufen und verstanden werden können.

Mit Hilfe der Begriffe der Informationsflusstheorie (vgl. Kapitel 3) können die Aggregatzustände von Information wie folgt erklärt werden:

Aus der Menge an Personen des Betrachtungsbereichs lässt sich der gemeinsame Kontext (vgl. Def. 3.35) bestimmen. Dieser gibt wiederum vor, auf welcher Basis innerhalb dieses Personenkreises kommuniziert werden kann (vgl. Kapitel 3.6). Je größer der gemeinsame Kontext ist, desto effizienter ist die Kommunikation (vgl. Satz 4.4). Je besser der Zielpersonenkreis bekannt ist, desto geringer kann der Aufwand für die Erstellung von Dokumenten für diesen Personenkreis (vgl. Sätze 4.5 und 4.6) ausfallen. Da aber nach Satz 4.2 der gemeinsame Kontext mit steigender Anzahl von Personen tendenziell kleiner wird, ist es auch tendenziell schwieriger feste Informationen für einen größeren Personenkreis zu schaffen.

Die Zeitspanne bestimmt, wie lange die Informationen zugreifbar und verstehbar sein müssen, um noch als fest klassifiziert werden zu können. Sind Informationen in Form von Daten gespeichert, dann sind sie prinzipiell wiederholt zugreifbar, weil Daten nach Definition 3.7 in der physischen Welt existieren und dort wahrgenommen werden können. Eine Ausnahme besteht, wenn die Daten nicht direkt vom Menschen wahrgenommen werden können, sondern Hilfsmittel zum Auslesen der Daten erforderlich sind. Fehlen diese Hilfsmittel, z.B. ein Diskettenlaufwerk zum Auslesen von Disketten, oder sind die physischen Zustände der Daten nach einer bestimmten Zeit nicht mehr unterscheidbar, z.B. wenn die Magnetisierung der Daten auf Disketten über die Zeit verloren gegangen ist, dann kann auch auf die Informationen, die durch die Daten repräsentiert werden, nicht mehr zugegriffen werden. Sind Informationen im betrachteten Personenkreis bei allen im Gedächtnis gespeichert, dann kann es sein, dass einige Personen diese Informationen innerhalb des betrachteten Zeitraums wieder vergessen. In beiden Fällen (nicht mehr zugreifbare Daten oder vergessenes Wissen) wären die Informationen für diesen Zeitraum als flüssig zu klassifizieren. D.h. auch, dass Informationen, die ausschließlich im Gedächtnis von Personen gespeichert sind, nur dann als fest einzustufen sind, wenn sie von allen Personen über die gesamte Zeitspanne des Betrachtungsbereichs nicht vergessen werden.

Die Unterscheidung zwischen festen und flüssigen Informationsspeichern ist also nicht direkt aus der Unterscheidung zwischen physiologischen (Wissen) und physischen (Daten) Informationsspeichern ableitbar (vgl. Abb. 3.3), sondern ist von den Eigenschaften (d.h. dem Aggregatzustand) der gespeicherten Informationen in Bezug auf einen Betrachtungsbereich abhängig. Feste und flüssige Informationsspeicher werden wie folgt definiert.

Tabelle 6.1: Eigenschaften und Beispiele fester und flüssiger Informationsspeicher

Aggregatzustand	Eigenschaften	Beispiele[1]
Fest	Vorteile: - Wiederholt abrufbar - Langfristig zugänglich - Dritten zugänglich - Dritte können etwas damit anfangen Nachteile: - Speicherung kostet Zeit und Aufwand - Abruf kostet Zeit und Aufwand	- Anforderungsspezifikation (analog oder elektronisch) - Audio- / Video-Aufnahmen von Projekt-Meetings - Screencasts der im Projekt erstellten Software mit Erklärungen, was gerade gezeigt wird - Quellcode der Software
Flüssig	Vorteile: - Können leicht (effektiv und effizient) gespeichert und weitergegeben werden - Können leicht (effektiv und effizient) abgerufen und verstanden werden, wenn der notwendige Kontext vorhanden ist Nachteile: - Gehen leicht verloren (Information selbst oder Kontext wird vergessen) - Können leicht missverstanden werden, wenn der notwendige Kontext fehlt	- Allgemein, Informationen im Gedächtnis von Personen - informelle E-Mails - Chat-Protokolle - Notizzettel - Whiteboard-Notizen

[1] Typische Beispiele, unter der Annahme, dass der Betrachtungsbereich ein Projekt ist. D.h., die Menge der Personen im Betrachtungsbereich sind alle Entwickler des Projekts und die Zeitspanne ist die Projektlaufzeit.

Definition 6.4: **Fester Informationsspeicher**

Ein fester Informationsspeicher ist ein Informationsspeicher, der ausschließlich feste Informationen enthält.

Definition 6.5: **Flüssiger Informationsspeicher**

Ein flüssiger Informationsspeicher ist ein Informationsspeicher, der nicht fest ist.

Der Aggregatzustand des Flusses von Informationen ergibt sich aus den Quellspeichern des Flusses.

Definition 6.6: **Fester Informationsfluss**

Ein fester Informationsfluss ist ein Informationsfluss, bei dem Quellinformationsspeicher fest sind.

Definition 6.7: **Flüssiger Informationsfluss**

Ein flüssiger Informationsfluss ist ein Informationsfluss, bei dem Quellinformationsspeicher flüssig sind.

Aus der Unterscheidung zwischen fester und flüssiger Information und den Sätzen 4.5 und 4.6 ergeben sich bestimmte Eigenschaften für Informationen in den beiden Aggregatzuständen. Tabelle 6.1 listet diese Eigenschaften mit einigen typischen Beispielen auf.

6.1.2 Erfahrung

Nach Schneider [Schneider 2004, Schneider 2005] nehmen Erfahrungen eine Sonderrolle in FLOW ein. Erfahrungen sind besondere Informationen, die helfen können, bestimmte Softwareentwicklungsaufgaben zu beschleunigen, oder Fehler nicht noch einmal zu begehen. Daher sind Erfahrungen eine wertvolle Informationsart für die Softwareentwicklung und werden gesondert betrachtet.

Nach Schneider [Schneider 2009, S. 14] ist die Besonderheit einer Erfahrung das persönliche Involviertsein derjenigen Person, die die Erfahrung macht. Das persönliche Erleben führt dazu, dass das durch die Erfahrung gewonnene Wissen tiefer abgespeichert wird und somit leichter und längerfristig abrufbar ist (vgl. Kapitel 2.2.2.6). Eine Erfahrung besteht aus drei wesentlichen Teilen [Schneider 2009, S. 14]:

1. **einer Beobachtung:** Durch persönliches Erleben machen Menschen Beobachtungen, z.B. bei der täglichen Programmierarbeit.

2. **einer Emotion:** Manche Beobachtungen lösen eine Emotion aus, z.B., wenn etwas nicht wie erwartet funktioniert.

3. **einer Folgerung:** Das Erlebnis und die Emotion führen zu einer Reflexion über das Erlebte, was wiederum zu einer Schlussfolgerung führt, z.B. der Folgerung, wie das erlebte Problem in Zukunft vermieden werden kann.

Nach dieser Beschreibung ist Erfahrung zunächst Wissen und damit an eine Person gebunden (vgl. Def. 3.2). Dieses Wissen kann aber auch für andere Personen wertvoll sein. Damit eine Erfahrung auch für andere möglichst hilfreich bleibt, sollte neben der Folgerung auch der Beobachtungs- und der Emotionsteil mitgeteilt werden. Natürlich hat die Mitteilung einer Emotion nicht denselben Effekt wie das persönliche Erleben einer Emotion, dennoch wird hier davon ausgegangen, dass durch die Mitteilung der ursprünglichen Emotion der Empfänger durch das Mitfühlen zumindest eine kleine eigene Emotion erlebt, die wiederum zum besseren Verständnis der Information führt. Insgesamt kann man die folgenden drei Stufen unterscheiden, wie tief sich erlerntes Wissen einprägt:

1. **Etwas selber erfahren haben (echtes Erfahrungswissen):** Durch die echte eigene Erfahrung wird das Wissen mit mehr Zugriffspfaden (vgl. Kapitel 2.2.2.6) gespeichert, sodass es sehr viel leichter, schneller und langfristiger abgerufen werden kann. Eigene Erfahrungen sind die wertvollsten Erfahrungen. Dies gilt auch in der Softwareentwicklung.

2. **Information bestehend aus Beobachtung, Emotion und Folgerung:** Durch Mitteilung von Beobachtung und Emotion kann man sich in den Anderen hineinversetzen. Dadurch hat man evtl. eine kleine eigene

Emotion, die wiederum dazu führt, dass Wissen (in dem Fall die Folgerung) umfangreicher abgespeichert wird. Um den größtmöglichen Nutzen bei der Weitergabe eigener Erfahrungen an Andere zu erzielen, sollte man daher immer auch Beobachtung und Emotion mitliefern, nicht nur die Folgerung.

3. **Information besteht nur aus Folgerung:** Nur die Folgerung weiterzugeben (z.B. in Form einer Best Practice), ohne Begründung wie es dazu gekommen ist (Beobachtung), und warum es sinnvoll ist es so zu machen (Emotion), ist die schwächste Variante der Erfahrungsweitergabe, da die Folgerung alleine nur Faktenwissen ist, das man sich merkt oder nicht merkt. Der Abruf ist schwieriger, da weniger Zugriffspfade bestehen. Die Wahrscheinlichkeit, dass man sich in einer Situation, in der die erlernte Folgerung hilfreich wäre, daran erinnert, ist bei dieser Variante am geringsten.

Auf Grund dieser Überlegungen sollte Erfahrungsinformation aus den drei Teilen einer Erfahrung, wie oben beschrieben, bestehen. Erfahrung wird daher für FLOW wie folgt definiert (in Anlehnung an [Schneider 2009, S. 14]):

Definition 6.8: **Erfahrung**

Erfahrung ist Information über eine *Beobachtung*, über eine zur Beobachtung gehörenden *Emotion* und über eine aus der Beobachtung und der Emotion abgeleiteten *Folgerung*.

6.1.3 Grafische Notation

Eine der Grundannahmen von FLOW ist, dass grafische Modelle die Analyse von Informationsflüssen erleichtern. Für die Erstellung grafischer Modelle bedarf es einer Notation. Ausgehend von den Grundkonzepten und der Metapher der Aggregatzustände wurde eine Notation entwickelt, die es erlaubt, die wesentlichen Eigenschaften von FLOW-Modellen, nämlich die explizite Unterscheidung von festen und flüssigen Informationsflüssen, intuitiv darzustellen. Im Wesentlichen gibt es je ein Symbol für feste und flüssige Informationsspeicher und entsprechend je einen Pfeiltyp für Informationsflüsse. Ergänzt wird das Ganze durch die Möglichkeit FLOW-Modelle über das Aktivitäts-Element zu strukturieren und zu hierarchisieren. Insgesamt ergeben sich aus

den Grundannahmen von FLOW folgende Anforderungen an die grafische FLOW-Notation:

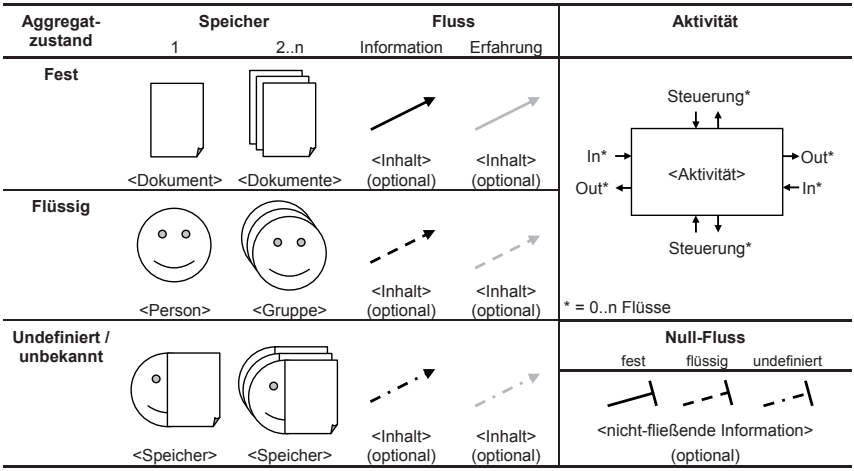
- Sie soll einfach zu erstellen sein (für jmd. der die FLOW-Methode anwendet, vgl. S. 303)
- Sie soll einfach zu verstehen sein (für jmd. der die FLOW-Methode anwendet, vgl. S. 303)
- Feste und flüssige Informationsspeicher *und* -flüsse sollen leicht unterschieden werden können
- Erfahrungen sollen als besondere Art von Informationen erkennbar sein
- Möglichkeit darzustellen, dass Informationen *nicht* fließen
- Möglichkeit darzustellen, dass der Aggregatzustand eines Informationsspeichers und der zugehörigen Informationsflüsse noch nicht bekannt ist (bei Ist-Modellen) bzw. noch nicht festgelegt wurde (bei Soll-Modellen).
- Möglichkeit große Modelle (viele Informationsflüsse) durch hierarchische Strukturierung übersichtlich darzustellen

Aus diesen Anforderungen wurde die grafische FLOW-Notation mit den in Tabelle 6.2 abgebildeten Elemente entwickelt.

Fest Das Symbol für einen festen Informationsspeicher ist das Dokument. Es wurde gewählt, weil es in der Softwareentwicklung ein typischer Vertreter eines festen Informationsspeichers ist. Sollen mehrere feste Informationsspeicher eines Typs modelliert werden, so stellt man das mit dem dreifach hinterlegten Dokumentensymbol dar. Alle Informationsflüsse, die aus einem festen Speicher hervorgehen, sind wiederum fest. Sie werden mit einem Pfeil mit durchgehender Linie dargestellt.

Flüssig Das Symbol für einen flüssigen Informationsspeicher ist ein Smiley. Es wurde gewählt, weil flüssige Informationen immer an Personen gebunden sind. Eine Gruppe von Personen wird mit dem dreifachen Smiley dargestellt. Falls die Unterscheidung zwischen Rollen und Individuen bei Personen erforderlich ist, kann man dies durch verschiedene Bezeichner darstellen. So kann zum Beispiel durch Unterstreichen des Namens verdeutlicht werden, dass es

Tabelle 6.2: Syntax der grafischen FLOW-Notation



sich um eine Rolle handelt (z. B.: „Müller“ im Vergleich zu „Entwickler“). Informationsflüsse, die von einem flüssigen Speicher ausgehen, sind flüssig. Sie werden mit einem Pfeil mit gestrichelter Linie dargestellt.

Undefiniert oder unbekannt Das Symbol für einen undefinierten bzw. einen unbekannten Informationsspeicher ist eine Kombination aus Smiley und Dokument. Mehrere unbekannte bzw. undefinierte Informationsspeicher werden mit dem dreifach hinterlegten Smiley-Dokument-Symbol dargestellt. Alle Informationsflüsse, die aus einem undefinierten/unbekannten Speicher hervorgehen, sind wiederum undefiniert/unbekannt. Sie werden mit einem Pfeil mit einer strichpunktierten Linie dargestellt.

Erfahrung Da Erfahrung eine in der Softwareentwicklung besonders wertvolle Art von Information ist, kann man sie in FLOW explizit darstellen. Dies geschieht durch eine andere Farbe der Speicher und Flüsse. Meist wird Grau als Kontrast zu den sonst schwarzen Symbolen und Bezeichnern verwendet, so dass eine Unterscheidung auch noch auf Schwarz-Weiß-Ausdrucken möglich ist.

Aktivität In FLOW-Modellen hat das Aktivitätssymbol zwei Aufgaben:

Hierarchisierung: Das Aktivitätssymbol fasst Informationsflüsse und zugehörige Speicher (und ggf. andere Aktivitäten) zusammen. Damit ermöglichen Aktivitäten eine hierarchische Strukturierung von FLOW-Modellen. Die eingehenden und ausgehenden Flüsse einer Aktivität, das so genannte FLOW-Interface, müssen konsistent mit den ein- und ausgehenden Flüssen im darunter liegenden Detailmodell sein.

Prozessanbindung: Das Aktivitätssymbol dient als Verbindung zu bestehenden Prozessnotationen. Aspekte von Informationsflüssen, insbesondere die Unterscheidung zwischen fest und flüssig, können so in bestehende Prozessdarstellungen integriert werden.

Es gibt die Möglichkeit Informationsflüsse von unterschiedlichen Seiten mit dem Aktivitätssymbol zu verbinden, um den Unterschied zwischen steuernden und inhaltlichen Informationsflüssen darstellen zu können. Inhaltliche Informationsflüsse enthalten die eigentlichen produktrelevanten Informationen wie Anforderungen, Entwurfsentscheidungen, Quellcode oder Testfälle, also meist Informationen darüber, wie das Problem aus der Anwendungsdomäne gelöst werden soll. Steuernde Informationsflüsse enthalten Informationen, die zur Durchführung der inhaltlichen Flüsse notwendig oder zumindest dafür hilfreich sind, also meist Informationen darüber, wie das Problem der Softwareentwicklung gelöst werden soll (vgl. Kapitel 3.4.2 und Abb. 3.10). Informationen, die steuernd an einer Aktivität mitwirken, werden von oben oder unten an das Aktivitätssymbol gezeichnet. Steuernde Informationen sind oft Erfahrungen. Informationen, die inhaltlich in der Aktivität verarbeitet werden, werden von links oder rechts an das Aktivitätssymbol gezeichnet.

Abbildung 6.1 zeigt ein Beispiel eines FLOW-Modells, in dem alle Modellelemente aus Tabelle 6.2 vorkommen.

6.1.3.1 Modellierungsregeln

Folgende Regeln sollen die Erstellung von Informationsflussmodellen mit der FLOW-Notation unterstützen. Die Hinweise stammen aus Erfahrungen bei der Erstellung von FLOW-Modellen (vgl. u.a. [Schneider 2004, Schneider 2005, Schneider 2005a, Schneider 2005c, Schneider 2006b, Stapel 2006, Stapel 2007, Stapel 2008a, Schneider 2008, Stapel 2009, Stapel 2011a]).

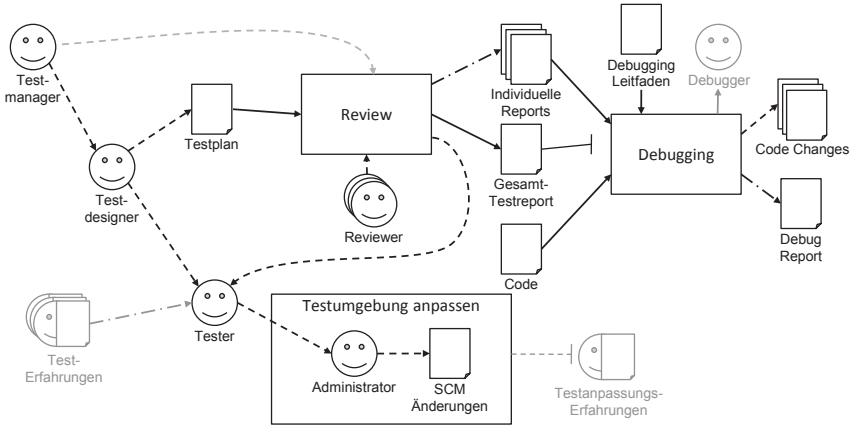


Abb. 6.1: Beispiel FLOW-Modell

- Zunächst nur grob modellieren und das Ziel (vgl. Kapitel im Auge behalten 6.2.1). Ziel ist meist Verständnis und nicht Vollständigkeit. Bei Bedarf kann das Modell später noch verfeinert werden.
- Flüssige Speicher resultieren immer in flüssigen Flüssen (gestrichelter Pfeil).
- Feste Speicher resultieren immer in festen Flüssen (durchgängiger Pfeil).
- Projektinformationen werden von links oder rechts an die Aktivität gezeichnet, wenn man zwischen steuernden und inhaltlichen Flüssen unterscheiden möchte.
- Steuernde Informationen (meist Erfahrung) werden von oben oder unten an die Aktivität gezeichnet, wenn man zwischen steuernden und inhaltlichen Flüssen unterscheiden möchte.
- Wenn die genauen Flüsse innerhalb einer Aktivität nicht bekannt sind (Black-Box), haben ausgehende Flüsse den Aggregatzustand unbekannt / undefiniert, d.h. sie werden mit einer strichpunktierten Linie gezeichnet.
- Die FLOW-Aktivität kann zur Strukturierung und Hierarchisierung von FLOW-Modellen genutzt werden, da sie Detailflüsse verbirgt.

- Üblicherweise werden entweder nur Rollennamen oder nur Personennamen als Bezeichner für flüssige Speicher in einem FLOW-Modell verwendet. Für den Fall, dass Beides in einem Modell vorkommt, kann man z.B. durch Unterstreichen die Rollen von den Individuen unterscheiden.
- FLOW-Modelle sind zeitlos, wie Datenfluss-Diagramme (DFDs) [DeMarco 1979]. Der zu modellierende Zeitrahmen muss vorher festgelegt werden (vgl. Def. 6.3).
- Dennoch können zeitliche Abhängigkeiten aus der Reihenfolge von Flüssen abgeleitet werden, wenn keine Zyklen bestehen.
- Da die Unterscheidung zwischen fest und flüssig neben dem betrachteten Zeitrahmen noch von der betrachteten Personengruppe abhängt (vgl. Def. 6.3), sollte diese bei jedem FLOW-Modell genannt werden.

6.1.4 Metamodell

Das Metamodell bildet die Grundlage für die Modellierung. Es kann u.a. dazu genutzt werden, Werkzeuge zu entwickeln, die die Modellierung mit FLOW ermöglichen. Ein solches Werkzeug ist z.B. ProFLOW[2]. Weiterhin ist das Metamodell Grundlage für die Prüfung, ob ein FLOW-Modell valide in Bezug auf die FLOW-Notation ist.

Nach dem Metamodell aus Abbildung 6.2 ist ein FLOW-Modell ein gerichteter Graph mit Informationsspeichern als Knoten und Informationsflüssen als Kanten. Jeder Knoten hat einen Namen. Jeder Informationsfluss kann einen Namen haben. Es gibt elementare Informationsflüsse und Informationsflussaktivitäten. Ein elementarer Informationsfluss hat genau einen Quell- und genau einen Zielinformationsspeicher, einen Aggregatzustand und einen Inhaltstyp.

Es gibt drei Inhaltstypen von Informationsflüssen: (1) Produktinformationsflüsse, die alle produktrelevanten Informationen transportieren, (2) Steuerungsinformationsflüsse, die Informationen zur Steuerung der Produktflüsse transportieren und (3) Null-Flüsse, die explizit ausdrücken, dass zwischen zwei Informationsspeichern keine Information fließt. Erfahrungsflüsse sind spezielle Informationsflüsse. Der Aggregatzustand eines elementaren Informationsflusses ergibt sich aus dem Aggregatzustand seines Quellinformationsspeichers.

[2] http://www.se.uni-hannover.de/proflow

Eine Informationsflussaktivität (kurz Aktivität) ist ein Container, der 0 bis n Flüsse enthalten kann. Eine Aktivität kann somit weitere elementare Informationsflüsse und Aktivitäten enthalten. Informationsflüsse (Component), elementare Informationsflüsse und Aktivitäten bilden ein Composite-Pattern. Baumstrukturen bzw. Hierarchien sind somit modellierbar. Darüber hinaus hat eine Aktivität eingehende und ausgehende Informationsflüsse.

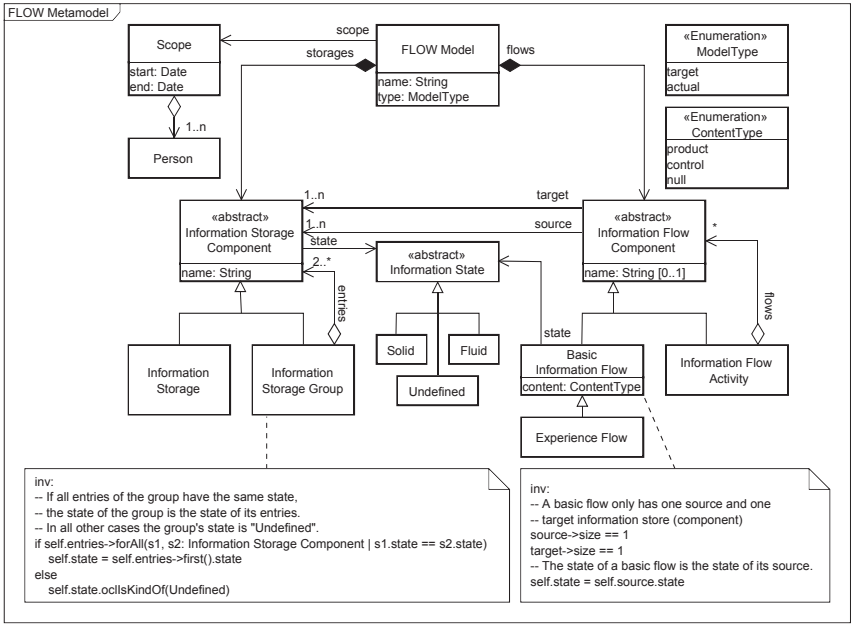


Abb. 6.2: FLOW-Metamodell

Das FLOW-Metamodell enthält ein weiteres Composite-Pattern, bestehend aus Informationsspeicher-Komponenten, Informationsspeichern und Informationsspeichergruppen. Damit lassen sich Hierarchien von Informationsspeichern modellieren. Eine Informationsspeicherkomponente hat einen Aggregatzustand. Der Aggregatzustand einer Informationsspeichergruppe ergibt sich aus dem Aggregatzustand seiner Kindelemente, wenn diese alle den gleichen Aggregatzustand haben, oder er ist undefiniert.

Das FLOW-Modell selbst hat neben Speichern und Flüssen noch einen Namen, einen Betrachtungsbereich und einen Typ. Der Typ des FLOW-Modells

gibt an, ob es sich um ein Soll- oder Ist-Modell handelt. Der Betrachtungsbereich gibt den Zeitraum und die Personengruppe an, für die das Modell erstellt wurde und auf dessen Basis die Unterscheidung zwischen fest und flüssig stattfindet.

6.2 Die FLOW-Methode

Basierend auf den Grundlagen von FLOW, insbesondere der Unterscheidung zwischen fester und flüssiger Information, und mit Hilfe der FLOW-Notation wurde die FLOW-Methode entwickelt. Die FLOW-Methode soll Informationsfluss-basierte Softwareprozessverbesserung praktisch nutzbar machen. Bevor die FLOW-Methode beschrieben wird, soll zunächst geklärt werden, was hier mit einer Methode gemeint ist. Es wird der Methodenbegriff nach Bjørner benutzt und von seiner ursprünglichen Bedeutung als Softwareentwicklungsmethode auf eine Methode zur Softwareentwicklungsverbesserung verallgemeinert.

> *Method: By a method we understand a set of principles for selecting and applying a number of analyses and synthesis (construction) techniques and tools in order efficiently to construct an efficient artifact, here software (i.e. a computing system).* [Bjoerner 2006, S. 32]

Demnach ist eine Methode eine Menge von Prinzipien zur Auswahl und Anwendung von Techniken und Werkzeugen für die effizienten Analyse und Konstruktion von effizienter Software. Dieser Begriff kann leicht auf Methoden zur Analyse und Verbesserung (Konstruktion) von Softwareentwicklungsprojekten verallgemeinert werden.

Die FLOW-Methode ist demnach eine Menge von Prinzipien zur Erstellung, Auswahl und Anwendung von FLOW-Techniken und zugehörigen Werkzeugen für die Analyse und Verbesserung von Softwareentwicklungsprojekten. D.h. die FLOW-Methode ist sowohl eine abstrakte Beschreibung, wie Softwareprojektverbesserung mit FLOW funktioniert, als auch ein Framework zur Erstellung und Einordnung von FLOW-Techniken. Eine FLOW-Technik beschreibt konkrete Aktivitäten und Schritte, die ausgeführt werden müssen, um ein bestimmtes Verbesserungsziel zu erreichen. Der FLOW-Ansatz wird durch FLOW-Techniken praktisch anwendbar. Die FLOW-Methode beschreibt prinzipielle Phasen, Strategien und Verfahren einer FLOW-Technik und gibt damit FLOW-Techniken eine Struktur.

Kern der FLOW-Methode ist ein dreiphasiger Verbesserungsprozess (vgl. Abb. 6.3 rechts), der Strategien und Verfahren zur (1) Erhebung, (2) Analyse und (3) Verbesserung von Informationsflüssen beschreibt. Im Forschungsprojekt InfoFLOW wurden verschiedene Techniken und Werkzeuge entwickelt (vgl.

Kapitel 6.3), die Aktivitäten in den einzelnen Phasen des Verbesserungsprozesses beschreiben bzw. unterstützen. Um passende Techniken und Werkzeuge auswählen zu können, sollte im Rahmen eines Verbesserungsvorhabens das verfolgte FLOW-Ziel und die Randbedingungen klar sein. Die Festlegung des FLOW-Ziels und das Festhalten der Randbedingungen des Verbesserungsvorhabens geschehen in der Vorbereitungsphase.

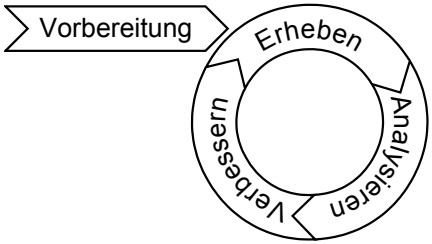


Abb. 6.3: FLOW-Verbesserungsprozess

Die FLOW-Methode hat zwei Aufgaben:

1. Die FLOW-Methode leitet einen Praktiker bei der Informationsfluss-basierten Verbesserung der Softwareentwicklung an, indem sie den prinzipiellen Ablauf eines FLOW-Verbesserungsvorhabens beschreibt. Zudem beschreibt die FLOW-Methode, wie bei einem FLOW-Verbesserungsvorhaben geeignete FLOW-Techniken für die Umsetzung gewählt werden.

2. Die FLOW-Methode leitet einen Praktiker oder Forscher bei der Erstellung neuer FLOW-Techniken an, indem sie den Aufbau und Bestandteile von FLOW-Techniken beschreibt.

6.2.1 Vorbereitung

Die Vorbereitungsphase besteht aus den folgenden drei Schritten:

1. Festlegung des im Verbesserungsprozess zu verfolgenden FLOW-Ziels:

Bei der Festlegung des FLOW-Ziels werden Absicht, Umfang und Zeit des Verbesserungsvorhabens festgelegt. Also z.B., ob verbessert oder nur

analysiert werden soll, ob eine Aktivität oder ein ganzes Projekt betrachtet werden soll oder ob der Verbesserungsprozess vor oder zur Projektlaufzeit durchgeführt werden soll.

2. Projektparameter festhalten (bzw. Aktivitäts- oder Organisationsparameter): Für die Auswahl von FLOW-Techniken relevante Projektparameter müssen festgehalten werden. Relevante Parameter sind z.B. die Anzhal der beteiligten Personen, die Rollenbesetzungen, das eingesetzte Vorgehensmodell und zu beachtende Randbedingungen (Einschränkungen, Verteiltheit, gesetzliche Vorgaben, vorhandene Ressourcen).

3. Auswahl passender FLOW-Techniken: Schließlich werden basierend auf dem FLOW-Ziel und den Projektparametern passende FLOW-Techniken ausgewählt.

Diese Vorbereitungsmaßnahmen sind notwendig, um später geeignete Techniken und Werkzeuge auswählen zu können.

Schritt 1. FLOW-Ziel festlegen Das Hauptziel der FLOW-Methode ist es Informationsflüsse in der Softwareentwicklung zu verbessern. Dieses Ziel kann erreicht werden, indem mindestens eines der Unterziele verfolgt wird, die aus den folgenenden Aspekten eines FLOW-Ziels gebildet werden können.

Absicht: Absicht eines FLOW-Ziels ist es entweder Informationsflüsse zu *erheben*, sie zu *verstehen* oder sie zu *verbessern*. Normalerweise erfüllt eine FLOW-Technik nacheinander alle drei Absichten: Zuerst müssen Informationsflüsse erhoben werden. Diese Informationsflüsse können dann analysiert und damit erst verstanden werden. Erst dann können die Informationsflüsse verbessert werden.

Umfang: Der Umfang eines FLOW-Ziels legt die Reichweite eines Verbesserungsvorhabens fest. Sie kann entweder eine *Aktivität*, ein *Projekt* oder eine ganze *Organisation* umfassen. Ist der Umfang auf eine Aktivität festgelegt, werden nur die Informationsflüsse einer einzelnen Softwareentwicklungsaktivität berücksichtigt, z.B. die Anforderungs-Elicitation. Auf Projektebene werden alle Informationsflüsse eines gesamten Softwareentwicklungsprojekts betrachtet. Wohingegen auf der Organisationsebene projektübergreifende Informationsflüsse im Mittelpunkt stehen.

Tabelle 6.3: Beispiele von FLOW-Zielen

FLOW-Ziel-Aspekte			**Konkretes FLOW-Ziel**
Absicht	Zeit	Umfang	
Verstehe	während	Aktivität	Verstehe die Anforderungserhebung während eines Anforderungsinterviews.
Verstehe	nach	Aktivität	Verstehe eine Review-Session durch eine Post-Mortem-Analyse.
Verbessere	vor	Aktivität	Spezifikationsreviewaktivität vorher tailorn. (vgl. [Schneider 2005])
Verbessere	während	Projekt	Anforderungsfluss während eines Projekts verbessern.

Zeit: Die Hauptaktivitäten einer FLOW-Technik werden entweder *vor*, *nach* oder *während* der Durchführung der im Fokus stehenden Aktivität (bzw. des Projekts oder der Organisation) durchgeführt. Dieser zeitliche Aspekt ist Teil des FLOW-Ziels.

Theoretisch gibt es 27 Permutationen dieser drei FLOW-Ziel-Aspekte. Einige Kombinationen sind aber nicht sinnvoll, z.B. Verbesserungsvorhaben, die nach der Lebensdauer einer Organisation durchgeführt werden sollen. Beispiele typischer FLOW-Ziele sind in Tabelle 6.3 aufgelistet.

Schritt 2: Projektparameter sammeln Zur Wahl geeigneter FLOW-Techniken ist neben dem verfolgten Ziel noch die Kenntnis einiger Projektparameter wichtig. Relevante Parameter sind:

Projektgröße: Die Projektgröße kann über die Teamgröße oder die Größe des verfügbaren Budgets angegeben werden. Beides ist wichtig, um abschätzen zu können, wie komplex die FLOW-Modelle wahrscheinlich werden und wie viele Ressourcen zur Anwendung der FLOW-Methode bereit stehen. Außerdem hat die Projektgröße Einfluss auf Kommunikations- und Dokumentationsaufwand. Daher können FLOW-Techniken für bestimmte Projektgrößen spezialisiert sein.

Domäne: FLOW-Techniken können auf bestimmte Anwendungsdomänen oder bestimmte Softwaredomänen (Web, Embedded, etc.) spezialisiert sein (vgl. Kapitel 3.4.2)

Vorgehensmodell: Die Art wie entwickelt wird, ob agil oder Prozess-getrieben, hat großen Einfluss auf Art und Häufigkeit von Informationsflüssen. Daher können FLOW-Techniken für bestimmte Vorgehensmodelle spezialisiert sein.

Verteiltheit: Informationsflüsse in lokalen Projekten sind meist anders als Informationsflüsse in verteilten Projekten. Einige FLOW-Techniken wurden speziell für verteilte Projekte entwickelt. Eine weitere wichtige Information für die Wahl einer geeigneten FLOW-Technik ist die Art der Verteilung. Informationsflüsse in Projekten, die entlang des Entwicklungsprozesses verteilt sind (z.B. Anforderungsphase in D, Implementierung in CZ), sind anders als Informationsflüsse in Projekten, die innerhalb einer Phase verteilt sind (Z.B. Implementierung auf drei Standorte verteilt, vgl. u.a. [Stapel 2009]), da die parallele Arbeit an einem Artefakt mehr koordinierende Kommunikation erfordert als die sequentielle.

Sonstige Randbedingungen: Andere Randbedingungen, die bei der Wahl einer geeigneten FLOW-Technik eine Rolle spielen können.

Schritt 3: FLOW-Techniken auswählen Der letzte Schritt der Vorbereitungsphase ist die Wahl der FLOW-Techniken, die zum gesetzten Ziel und den Projektparametern passen. Um dies zu vereinfachen sollten alle verfügbaren FLOW-Techniken mit Hilfe des in Tabelle 6.4 dargestellten Templates klassifiziert werden. Jede FLOW-Technik hat einen beschreibenden Namen. Eine FLOW-Technik bietet Unterstützung für verschiedene Aspekte eines FLOW-Ziels. Welche Aspekte das sind wird in der nächsten Zeile des Templates markiert. Hier ist eine Mehrfachauswahl möglich, da es Techniken gibt, die zur Erreichung verschiedener FLOW-Ziel-Aspekte genutzt werden können, z.B. zur Verbesserung einer Entwicklungsaktivität oder zur Verbesserung eines ganzen Projekts. Im nächsten Absatz des Templates werden die Phasen des FLOW-Verbesserungsprozesses notiert, für die die FLOW-Technik konkrete Unterstützung anbietet. Der letzte Teil des Templates sind die Projektparameter, für die eine Technik entwickelt wurde bzw. die den Anwendungsbereich einer Technik einschränken.

Tabelle 6.4: Template zur Klassifikation von FLOW-Techniken

Name der Technik		_____		
Ziel[1]	**Absicht**	□ Verstehen	□ Verbessern	
	Zeit	□ vorher	□ während dessen	□ nachher
	Umfang	□ Aktivität	□ Projekt	□ Organisation
Phase[2]		□ Erheben	□ Analysieren	□ Verbessern
Projektparameter				
Projektgröße		□ Teamgröße: _____ □ Budget: _____ □ Benötigte Ressourcen: _____		
Domäne		□ Anwendungsdomäne: _____ □ Softwaredomäne: _____		
Vorgehensmodell		□ Prozess: _____ □ Agil: _____ □ Andere: _____		
Verteiltheit		□ Lokal	□ Verteilt (Art: _____)	
Sonstiges		_____		

[1] FLOW-Ziel-Aspekte, die mit dieser FLOW-Technik verfolgt werden können
[2] Diese Technik bietet Unterstützung in diesen Phasen des FLOW-Verbesserungsprozesses

Für ein gegebenes FLOW-Ziel und gegebene Randbedingungen können dann geeignete Techniken durch Abgleich der klassifizierten FLOW-Techniken mit dem Ergebnis der Vorbereitungsphase gefunden werden.

Vorbedingungen und Rollen Damit die FLOW-Methode eingesetzt werden kann, müssen einige Vorbedingungen erfüllt sein. Vorbedingungen für die Durchführung FLOW-basierter Softwareprozessverbesserung sind:

- Das zu verbessernde Projekt (die Aktivität, oder die Organisation) muss bereit sein, ihre Informationsflüsse für die Analyse preiszugeben.

- Das analysierte Projekt (Aktivität, Organisation) muss offen für Verbesserungsvorschläge sein und Änderungen flexibel umsetzen können.
- Es müssen für die Problemstellung (FLOW-Ziel und Projektparameter) passende FLOW-Techniken inkl. zugehöriger Werkzeuge zur Verfügung stehen.

Die FLOW-Methode wird üblicherweise von externen FLOW-Experten, Mitarbeitern interner Prozessabteilungen oder der Projektleitung durchgeführt. Wenn alle Vorbedingungen erfüllt sind und die Vorbereitungsphase durchgeführt wurde, kann der FLOW-Verbesserungsprozess (vgl. Abb. 6.3) gestartet werden.

6.2.2 Phase 1: Erheben

Die erste Phase des FLOW-Verbesserungsprozesses hat die Erhebung von Informationsflüssen zum Ziel. Es gilt, eine Basis für spätere Analysen und Verbesserungen zu schaffen. Ergebnis der ersten Phase ist ein FLOW-Modell, das in den folgenden Phasen zur Analyse und als Ausgangspunkt für eine Verbesserung genutzt werden kann. Als Erheben gilt sowohl das Erstellen eines FLOW-Modells ausgehend von vorhandenen Informationsflüssen (Ist-Modell), zum Beispiel
durch Beobachten, als auch das konstruktive planende Erstellen eines FLOW-Modells (Soll-Modell). Techniken und Werkzeuge der ersten Phase lassen sich nach Erhebungsstrategie und nach dem Verfahren zur Datenerfassung unterscheiden.

6.2.2.1 Erhebungsstrategien

Die Erhebungsstrategie bestimmt die prinzipielle Herangehensweise an die Erhebung von Informationsflüssen. Es gibt die *Top-Down-* und die *Bottom-Up-*Strategie. Bei der Top-Down-Strategie werden ausgehend von einem Informationsflussmodell auf hoher Abstraktionsebene Schritt für Schritt Informationsflussdetails hinzugefügt. Dies führt zu einem immer ausführlicheren Modell auf niedrigerer Abstraktionsebene. Bei der Bottom-Up-Strategie wird durch schrittweise Zusammenführung mehrerer unabhängiger Einzelmodelle auf niedriger Abstraktionsebene nach und nach ein Gesamtmodell höherer Abstraktion erstellt.

Tabelle 6.5: Vor- und Nachteile der Strategien der Phase 1: Erheben

	Vorteile	Nachteile
Bottom-Up	- Zu Beginn kein Gesamtüberblick notwendig - Eignet sich gut für Ist-Modelle	- Zusammenführen der Einzelmodelle oft nicht trivial - Lokale Sichten können stark differieren
Top-Down	- Bietet Möglichkeit nur bei relevanten Stellen ins Detail zu gehen - Eignet sich gut für Soll-Modelle	- Gesamtüberblick ist häufig nicht vorhanden - Zu große Modelle können unübersichtlich sein

Die Entscheidung, welche Strategie gewählt werden sollte, wird u.a. vom Aspekt "Umfang" des FLOW-Ziels beeinflusst. So ist es zum Beispiel sinnvoll, bei einer organisationsweiten Betrachtung der Informationsflüsse den Top-Down-Ansatz zu wählen und ausgehend von einem abstrakten Überblicksdiagramm, das die Informationsflüsse der gesamten Organisation zeigt, an interessanten Stellen Details hinzuzufügen, um diese genauer Analysieren zu können. Eine weitere Entscheidungshilfe zur Wahl einer geeigneten Erhebungsstrategie bieten die Vor- und Nachteile in Tabelle 6.5.

6.2.2.2 Erhebungsverfahren

Verfahren für die Erhebung von Informationsflüssen beschreiben typische Vorgehensweisen, wie man die Daten, die zur Erstellung eines Informationsflussmodells notwendig sind, sammelt und umwandelt. Erhebungsverfahren lassen sich nach der Datenquelle unterscheiden (vgl. Abb. 6.4): Elicitation-Verfahren nutzen das Wissen von Personen über die zu erhebenden Informationsflüsse. Ableitende Verfahren nutzen bereits bestehende Modelle, die Aspekte von Informationsflüssen enthalten, z.B. Prozessmodelle. Eine FLOW-Technik kann

auch mehrere Erhebungsverfahren kombinieren, z.B. eine Modellableitung und eine anschließende Elicitation.

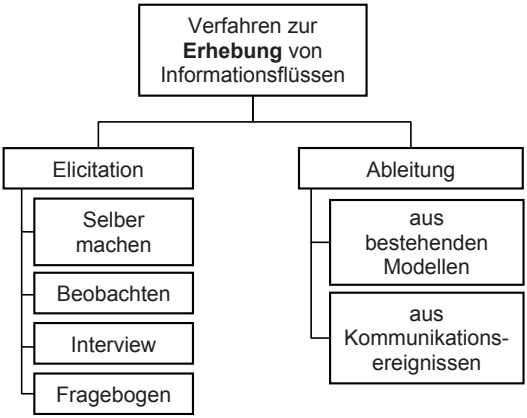


Abb. 6.4: Verfahren zur Erhebung von Informationsflüssen

Verfahren der Elicitation sind:

Selber machen: Ein FLOW-Experte begibt sich selbst in die Situation, zu der Informationsflüsse erhoben werden sollen, und führt diese selbst aus. Dabei notiert er alle relevanten Informationsflüsse und erstellt das Informationsflussmodell.

Beobachten: Ein FLOW-Experte beobachtet die Situation, zu der Informationsflüsse erhoben werden sollen. Dabei notiert er alle relevanten Informationsflüsse und erstellt das Informationsflussmodell.

Interview: Ein FLOW-Experte führt mit allen – oder zumindest den wichtigsten – Personen ein Interview, die an der Situation beteiligt sind, zu der Informationsflüsse erhoben werden sollen. Dabei notiert er alle relevanten Informationsflüsse und erstellt das Informationsflussmodell.

Fragebogen: Ein FLOW-Experte erstellt einen Fragebogen, der alle Informationsfluss-relevanten Fragen enthält, und verteilt diese an möglichst alle Personen, die an der Situation beteiligt sind, zu der Informationsflüsse erhoben werden sollen. Dabei notiert er alle relevanten Informationsflüsse und erstellt das Informationsflussmodell.

Verfahren der Ableitung sind:

Modellableitung: Ausgehend von einem bestehenden Modell, das Informationsflussaspekte der zu untersuchenden Situation enthält (z.B. ein Prozessmodell), werden relevante Teile extrahiert und in einem neuen Informationsflussmodell zusammen gefasst.

Kommunikationsereignisableitung: Aus Kommunikationsereignissen wie E-Mails, Telefonaten, Meetings etc. werden Informationsflüsse abgeleitet.

Bei allen Erhebungsverfahren ist zu beachten, dass die gewonnenen Daten nur indirekt Aufschluss über tatsächliche Informationsflüsse geben. Z.B. können Interviewdaten subjektiv verfälscht und modellierte Prozessmodelle veraltet sein. Kommunikationsereignisse sagen nicht direkt etwas über den kommunizierten Inhalt aus. Aus einem Kommunikationsereignis, wie zum Beispiel einer Skype-Telefonkonferenz, das aus Start-, Endzeitpunkt und einer Liste von Teilnehmern besteht, kann nicht direkt ein Informationsfluss abgeleitet werden. Es kann entweder gar keine Information fließen, zum Beispiel wenn jeder Teilnehmer eine andere Sprache spricht, oder es fließt nicht projektrelevante Information, zum Beispiel wenn über das letzte Freizeiterlebnis gesprochen wird. Eine Ableitung von Informationsflüssen aus Kommunikationsereignissen funktioniert also nur unter gewissen Annahmen.

- Es wird angenommen, dass während einer Kommunikation auch Informationen fließen.

- Es wird angenommen, dass während einer professionellen Kommunikation hauptsächlich über projektrelevante Dinge gesprochen wird. Davon ist im Softwareentwicklungsumfeld auszugehen. So konnte in einer Studie über das Kommunikationsverhalten beim Pair-Programming gezeigt werden, dass nur 7 % der Kommunikation privaten Inhalt haben [Stapel 2010].

Die folgende Tabelle 6.6 listet einige Vor- und Nachteile der einzelnen Verfahren auf. Diese gilt es bei der Auswahl des anzuwendenden Verfahrens zu berücksichtigen.

Tabelle 6.7 fasst Strategien und Verfahren der ersten Phase des FLOW-Verbesserungprozesses zusammen.

Tabelle 6.6: Vor- und Nachteile der Verfahren der Phase 1: Erheben

	Vorteile	Nachteile
Selber machen	- Sehr effektiv. Kein Stille-Post-Effekt verfälscht Ergebnis - Ermöglicht auch implizites Wissen zu erfassen	- Teuer - Oft nicht möglich, da Domänenwissen fehlt oder nicht erlaubt
Beobachtung	- Effektiv. Kein Stille-Post-Effekt verfälscht Ergebnis	- Teuer - Oft nicht möglich, da Domänenwissen nicht erlaubt
Interview	- Flexibel. Es kann auf vorher unbekannte Sachverhalte eingegangen werden	- Hoher Vorbereitungsaufwand
Fragebogen	- Günstig umzusetzen - Erleichtert Auswertung	- Hoher Vorbereitungsaufwand - Starr, da nicht möglich auf vorher nicht bekannte Sachverhalte einzugehen
Modellableitung	- Wiederverwendung bestehender Modelle - Vergleichsweise günstig - Kann teilweise automatisiert werden	- Deckt meist nur Teilaspekte von Informationsflüssen ab - Kann je nach Ausgangsmodell große unübersichtliche Informationsflussmodelle zur Folge haben
Kommunikations-ereignisableitung	- Kann teilweise automatisiert werden - Nutzung bestehender Kommunikationsinfra-struktur möglich - Bei Automatisierung vergleichsweise günstig	- Es ist schwierig tatsächliche Informationsflüsse aus Kommunikationsereignissen herzuleiten - Automatische Erfassung kann viel Vorbereitungsaufwand erfordern - Automatische Erfassung kann zu Datenschutzproblemen führen

Tabelle 6.7: Zusammenfassung der phasenspezifischen Aspekte für die Phase 1: Erheben

Strategie	□ Bottom-Up	□ Top-Down		
Verfahren	□ Selber machen	□ Beobachten	□ Interview	□ Fragebogen
	□ Modellableitung	□ Kommunikationsereignisableitung		

6.2.3 Phase 2: Analysieren

Bei der Analyse geht es darum, die Informationsflüsse zu verstehen und Probleme zu finden. Das wichtigste Werkzeug der Analyse ist die Visualisierung der Informationsflüsse. Techniken und Werkzeuge der zweiten Phase lassen sich nach Analysestrategie und nach Analyseverfahren unterscheiden.

6.2.3.1 Analysestrategien

Die Analysestrategie bestimmt die prinzipielle Herangehensweise an die Analyse von Informationsflüssen. Es gibt die Strategie der manuellen, automatischen und teilautomatischen Analyse von Informationsflüssen. Bei der manuellen Strategie analysiert ein FLOW-Experte das Informationsflussmodell. Bei der (teil-)automatischen Analyse werden Probleme in Informationsflussmodellen (teil-)automatisch gesucht. Tabelle 6.8 zeigt die Vor- und Nachteile der beiden Pole der manuellen und automatischen Analysestrategien. Bei der teilautomatischen Analyse vermischen sich Vor- und Nachteile der anderen Strategien.

6.2.3.2 Analyseverfahren

Verfahren zur Analyse von Informationsflüssen beschreiben typische Vorgehensweisen, wie Probleme und andere Besonderheiten in Informationsflussmodellen identifiziert werden können. Für die Analyse von Informationsflussmodellen stehen die folgenden drei Verfahren zur Verfügung (vgl. Abb. 6.5):

Visualisierung: Die Visualisierung ist das grundlegendste Analyseverfahren. Dabei wird ein FLOW-Modell mit Hilfe der FLOW-Notation visuell

Tabelle 6.8: Vor- und Nachteile der Strategien der Phase 2: Analysieren

	Vorteile	Nachteile
manuell	- Expertenwissen leichter nutzbar - Weniger Vorbereitungsaufwand - Zusatzwissen, welches nicht im Modell abgebildet ist, kann zur besseren Analyse mit genutzt werden	- Ergebnisse abhängig von FLOW-Experten - Ergebnisse können subjektiv beeinflusst sein - Nur für relativ kleine Modelle geeignet
automatisch	- Objektive Analysen möglich - Ergebnisse leicht reproduzierbar - Auch für große Modelle geeignet	- FLOW-Modelle müssen vollständig und in gewisser Form vorliegen - Hoher Vorbereitungsaufwand zur Erstellung von Analysewerkzeugen

dargestellt und meist manuell (siehe Strategie) untersucht. Auffälligkeiten wie überwiegend feste oder flüssige Flüsse oder unnötige Dokumentation können so identifiziert werden (vgl. auch Mustersuche unten).

Simulation: Bei der Simulation werden mit Hilfe von zusätzlichen Elementen im Informationsflussmodell, z.B. Informationseinheiten (vgl. u.a. [Knauss 2008e]), Informationsflüsse simuliert. Mit Hilfe von Simulationen können dynamische Probleme, z.B. das Vergessen wichtiger Informationen, leichter erkannt werden.

Mustersuche: Bei der Mustersuche werden FLOW-Muster (vgl. [Ge 2008]) im FLOW-Modell gesucht. Je nach Strategie kann dies manuell, z.B. mit Hilfe des FLOW-Musterkatalogs [Ge 2008a], oder automatisch, z.B. mit Hilfe von ProFLOW [Gross 2009], geschehen.

Visualisierung, Simulation und Mustersuche können in einer FLOW-Technik auch kombiniert werden.

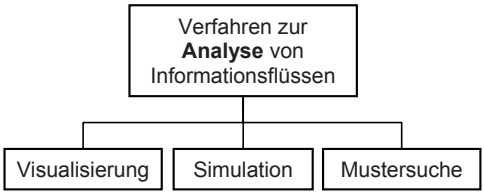


Abb. 6.5: Verfahren zur Analyse von Informationsflüssen

Tabelle 6.9 zeigt einige Vor- und Nachteile der unterschiedlichen Analyseverfahren.

Tabelle 6.10 fasst Strategien und Verfahren der zweiten Phase des FLOW-Verbesserungsprozesses zusammen.

6.2.4 Phase 3: Verbessern

Die letzte Phase im FLOW-Verbesserungsprozess ist die Verbesserung der Informationsflüsse auf Basis der Erkenntnisse der beiden vorangegangenen Phasen. Als Verbesserung im Sinne der dritten Phase gilt sowohl ein Soll-FLOW-Modell, welches die verbesserten aber noch nicht umgesetzten Informations-

Tabelle 6.9: Vor- und Nachteile der Verfahren der Phase 2: Analysieren

	Vorteile	Nachteile
Visualisierung	- Erfahrung des Analysten wird genutzt	- Teuer / hoher Personalaufwand (manuell) - Ineffizient bei großen Modellen
Simulation	- Erfahrung des Analysten wird genutzt	- Teuer / hoher Personalaufwand (manuell) - Simulation muss erstellt, vorbereitet und durchgeführt werden
Mustersuche	- Günstig und effektiv durch Automatisierung	- Abhängig von Quantität und Qualität der Muster - Abhängig von Matching-Algorithmus

Tabelle 6.10: Zusammenfassung der phasenspezifischen Aspekte für die Phase 2: Analysieren

Strategien	□ manuelle Analyse	□ teilautomatische Analyse	□ automatische Analyse
Verfahren	□ Visualisierung	□ Simulation	□ Mustersuche

flüsse enthält, als auch tatsächlich verbesserte Informationsflüsse in realen Projekten, unabhängig davon ob diese modelliert sind oder nicht. Techniken der dritten Phase lassen sich nach Verbesserungsstrategie und Verbesserungsverfahren unterscheiden.

6.2.4.1 Verbesserungsstrategien

Die Strategie zur Verbesserung von Informationsflüssen bestimmt den prinzipiellen Ansatz, der von einer Technik verfolgt wird, um Informationsflüsse zu verbessern. Es gibt vier Verbesserungsstrategien, von denen sich jeweils zwei gegenseitig ausschließen: Hauptprodukt- und Nebenprodukt-Strategie, sowie schwergewichtige und leichtgewichtige Strategie. Bei der *Hauptprodukt*-Strategie wird eine Verbesserung durch zusätzlich neu zu erstellende Informationen angestrebt. Mit der *Nebenprodukt*-Strategie (vgl. [Schneider 2006]) wird versucht, ausgehend von vorhandenen Informationsflüssen, eine Verbesserung durch Ableitung von Informationsflüssen als Nebenprodukt zu erreichen, d.h. vorhandene Informationen wiederzuverwenden. Ein weitere Strategie ist die leichtgewichtige Strategie. Bei der leichtgewichtigen Strategie sollen Verbesserungen im Informationsfluss durch einen möglichst geringen Zusatzaufwand erreicht werden. Das Gegenteil ist die schwergewichtige Strategie. Dabei werden Verbesserungen mit hohem Aufwand angestrebt. Meist werden Nebenprodukt und leichtgewichtige Strategie zusammen verwendet, da Änderungen im Prozess oft nur dann durchsetzbar sind, wenn für diejenigen, die den Prozess ausführen, möglichst wenig Zusatzaufwand zur eigentlichen Arbeit entsteht. Tabelle 6.11 zeigt einige Vor- und Nachteile der beiden Verbesserungsstrategien.

6.2.4.2 Verbesserungsverfahren

Verfahren zur Verbesserung von Informationsflüssen beschreiben typische Flussveränderungen, die einen übergeordneten Informationsfluss verbessern können. Prinzipiell lassen sich Verfahren zur Verbesserung in vier Kategorien aufteilen, je nach FLOW-Modellelement Speicher, Fluss und Aktivität (vgl. Tabelle 6.2 und Abb. 6.2), oder Informationsflussmuster (vgl. Abb. 6.6).

Tabelle 6.11: Vor- und Nachteile der Strategien der Phase 3: Verbessern

	Vorteile	Nachteile
Nebenprodukt	- Wiederverwendung vorhandener Informationen	- Einmalige Vorbereitungsaufwände (z.B. für Nebenprodukt-Werkzeuge) können teils sehr hoch sein
Hauptprodukt	- Neue bzw. andere Informationen können berücksichtigt werden	- Zusätzliche Informationen führen meist zu mehr Arbeitsaufwand
Leichtgewichtig	- Wenig zusätzlicher Arbeitsaufwand - Leicht zu lernen - Wenig Anpassungen notwendig - Einfach durchzuführen	- Effekte können zu gering sein
Schwergewichtig	- Es können große Verbesserungseffekte erzielt werden	- Schwergewichtige Ansätze werden oft nicht angenommen, da hoher Zusatzaufwand abschreckt - Hohe Einführungshürden bei Management und Entwicklern

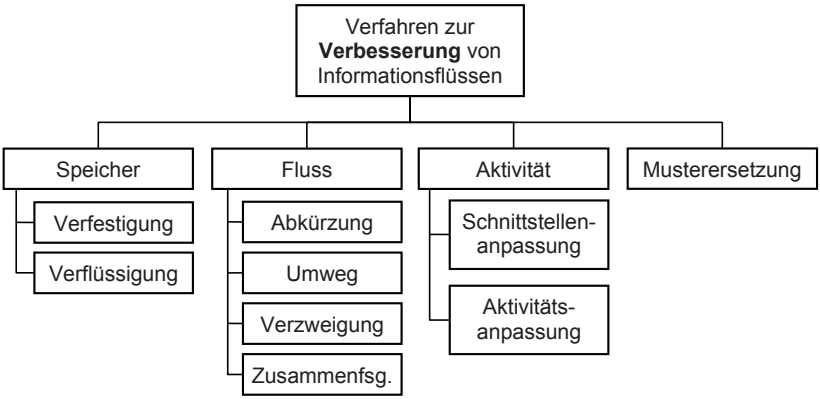


Abb. 6.6: Verfahren zur Verbesserung von Informationsflüssen

1. Aggregatzustandsänderung

   Verfestigung: Bei der Verfestigung wird eine Verbesserung angestrebt, indem flüssige Informationen in eine feste Form gebracht werden. Dabei sollen die Vorteile des festen Aggregatzustands, wie wiederholter und langfristiger Abruf oder Verständlichkeit für Dritte, ausgenutzt werden.

   Verflüssigung: Bei der Verflüssigung wird eine Verbesserung angestrebt, indem feste Informationen in eine flüssige Form gebracht werden. Dabei sollen die Vorteile des flüssigen Aggregatzustands, wie effiziente und schnelle Weitergabe, ausgenutzt werden.

2. Flussveränderung

   Abkürzung: Bei der Abkürzung wird eine Verbesserung angestrebt, indem Informationsflüsse über weniger Zwischenschritte, d.h. weniger Zwischenspeicher, ans Ziel geleitet werden. Dabei soll eine schnellere Informationsweitergabe erreicht und Fehler vermieden werden.

   Umweg: Bei einem Umweg wird eine Verbesserung angestrebt, indem Informationsflüsse über mehr Zwischenschritte, d.h. mehr Zwischenspeicher, ans Ziel geleitet wer-den. Dabei soll eine breitere

Informationsverteilung und zusätzlicher Input (z.B. Reviews) erreicht werden.

Verzweigung: Bei der Verzweigung wird eine Verbesserung angestrebt, indem Informationsflüsse an mehr Informationsspeicher verteilt werden. Dabei soll eine breitere Informationsverteilung erreicht werden.

Zusammenfassung: Bei der Zusammenfassung von Informationsflüssen wird eine Verbesserung angestrebt, indem Informationsflüsse an weniger Informationsspeicher verteilt werden. Dabei soll eine gezieltere und damit Ressourcen-schonendere Informationsverteilung erreicht werden.

3. Aktivitätsänderung

   Schnittstellenanpassung: Bei der Schnittstellenanpassung wird eine Verbesserung angestrebt, indem Aggregatzustand und Anzahl eingehender, ausgehender, steuernder und unterstützender Informationsflüsse angepasst werden. Dabei soll eine gezieltere, d.h. der Aufgabe angemessene, Informationsbereitstellung bzw. -erstellung erreicht werden. Eine geänderte Schnittstelle zieht meist auch eine Änderung der Aktivität selbst nach sich (vgl. nächster Punkt).

   Aktivitätsanpassung: Bei der Aktivitätsanpassung wird eine Verbesserung durch Veränderung der Interna einer Aktivität angestrebt. Die Schnittstelle nach außen bleibt gleich. Es werden nur interne Informationsflüsse verändert.

4. **Musterersetzung:** Bei der Musterersetzung werden Teilinformationsflüsse gezielt durch FLOW-Muster ersetzt, die gewünschte Informationsflusseigenschaften aufweisen (vgl. auch Mustersuche bei den Analyseverfahren).

Eine Verbesserungstechnik kann mehrere Verfahren kombinieren und so z.B. eine Verbesserung durch Verflüssigung und Abkürzung erzielen.

Tabelle 6.12 listet einige Vor- und Nachteile der einzelnen Verfahren auf. Diese gilt es bei der Auswahl des anzuwendenden Verfahrens zu berücksichtigen.

Tabelle 6.13 fasst Strategien und Verfahren der dritten Phase des FLOW-Verbesserungsprozesses zusammen.

Tabelle 6.12: Vor- und Nachteile der Verfahren der Phase 3: Verbessern

	Vorteile	Nachteile
Verfestigung	- Wiederholter Zugriff möglich - Langfristiger Zugriff möglich - Für alle Beteiligten im Betrachtungsbereich zugänglich - Für alle Beteiligten im Betrachtungsbereich verständlich	- Erstellung ist aufwendig - Abruf ist aufwendig
Verflüssigung	- Effektive und effiziente Informationsweitergabe möglich	- Informationen gehen leicht verloren - Nur für wenige im betrachteten Bereich verständlich
Abkürzung	- Schnelle Informationsweitergabe - Fehlervermeidung durch Verfälschung	- Weniger Informationsverteilung
Umweg	- Redundanz (mehr Sicherheit) - Bessere Informationsverbreitung - Zusätzlicher Input / Validierungsmöglichkeit	- Redundanz (mehr Synchronisierungsaufwand) - Langsamere Informationsweitergabe - Fehler durch Verfälschung der Information
Verzweigung	- Redundanz (mehr Sicherheit) - Bessere Informationsverbreitung - Zusätzlicher Input / Validierungsmöglichkeit	- Redundanz (mehr Synchronisierungsaufwand) - Fehler durch Verfälschung der Information
Zusammenfassung	- Redundanz (weniger Synchronisierungsaufwand) - Gezieltere Informationsverteilung	- Redundanz (weniger Sicherheit) - Fehlende Validierungsmöglichkeiten
Schnittstellenanpassung	- Individuelle Informationsbereitstellung	- Hoher Vorbereitungsaufwand

Tabelle 6.13: Zusammenfassung der phasenspezifischen Aspekte für die Phase 3: Verbessern

Strategie	○ Nebenprodukt	○ Hauptprodukt
	○ Leichtgewichtig	○ Schwergewichtig
Verfahren	□ Aggregatzustand	(○ verfestigen ○ verflüssigen)
	□ Fluss	(○ Abkürzung ○ Umweg ○ Verzweigung ○ Zusammenführung)
	□ Aktivität	(○ Schnittstellenanpassung ○ Aktivitätsanpassung)
	□ Musterersetzung	

6.3 Ausgewählte FLOW-Techniken

Zur Durchführung und Unterstützung des FLOW-Verbesserungsprozesses gibt es verschiedene Techniken. Eine FLOW-Technik beschreibt Aktivitäten für eine oder mehrere Phasen des FLOW-Verbesserungsprozesses. Verbesserungsaktivitäten können durch Werkzeugunterstützung (analog oder digital) operationalisiert sein.

Techniken beschreiben für mindestens eine der drei Phasen des Verbesserungsprozesses ein Vorgehen. Im Forschungsprojekt FLOW[3] werden Techniken und Werkzeuge erarbeitet und evaluiert. Im Folgenden werden ausgewählte FLOW-Techniken und zugehörige Werkzeuge dargestellt, die im Rahmen dieser Arbeit neu erstellt bzw. erweitert wurden. Dies sind:

- FLOW-Mapping
- Verfestigung als Nebenprodukt
- Integration von SCRUM in V-Modell XT Projekte

Die folgenden Beschreibungen der Techniken beginnen jeweils mit einer allgemeinen Beschreibung. Anschließend werden unterstützte Aktivitäten in den Phasen der FLOW-Methode erläutert. Es werden Werkzeuge vorgestellt, die

[3] http://www.se.uni-hannover.de/infoflow

die Durchführung der Aktivitäten unterstützen. Weiterhin werden erste Evaluationsergebnisse präsentiert, falls diese vorhanden sind. Eine Zusammenfassung beendet die Vorstellung jeder Technik. Dabei wird jede FLOW-Technik mit Hilfe des Templates aus Tabelle 6.4 klassifiziert.

6.3.1 FLOW-Mapping

FLOW-Mapping ist eine FLOW-Technik zur Steigerung der Awareness[4] in verteilten Teams. Nach Endsley [Endsley 1995] ist Awareness das Wissen einer Person über seine sich ständig ändernde Umgebung. Hat eine Person gute Awareness, dann ist die Wahrscheinlichkeit größer, dass sie angemessene Entscheidungen in dynamischen Umgebungen trifft. Betrachtet man verteilte Softwareentwicklung als dynamische Umgebung, dann sind die beteiligten Standorte, Teammitglieder, Aufgaben und Kommunikationsverhalten wichtige Bestandteile in Bezug auf Informationsflüsse in dieser Umgebung. Die mit der FLOW-Mapping-Technik angestrebte Verbesserung der Awareness soll gezieltere Informationsflüsse in verteilten Projekten ermöglichen. Gezielte Informationsflüsse verbessern ein Softwareprojekt, da die Wahrscheinlichkeit sinkt, dass wichtige Informationen fehlen oder durch den Umweg über viele Zwischenspeicher missverstanden werden.

6.3.1.1 FLOW-Map

Kern und namensgebender Bestandteil der FLOW-Mapping-Technik ist die so genannte FLOW-Map. Eine FLOW-Map ist ein FLOW-Modell, welches eine erweiterte FLOW-Notation nutzt, um Spezifika verteilter Softwareprojekte darstellen zu können. Eine FLOW-Map hat folgende zusätzliche Bestandteile:

- Zuordnung von Informationsspeichern zu Entwicklungsstandorten. Der Ort eines flüssigen Speichers repräsentiert den physischen Standort der Person, an die die flüssige Information gebunden ist. Der Ort eines festen Speichers repräsentiert den Standort welcher für den Inhalt des Speichers *verantwortlich* ist, d.h. insbesondere nicht seinen physischen Standort.

[4] Hier wird bewusst das englische Wort Awareness benutzt, da die deutschen Übersetzungen Bewusstsein, Wahrnehmung und Erkenntnis im deutschen Sprachgebrauch nicht die gleiche Bedeutung haben und daher irreführend wären.

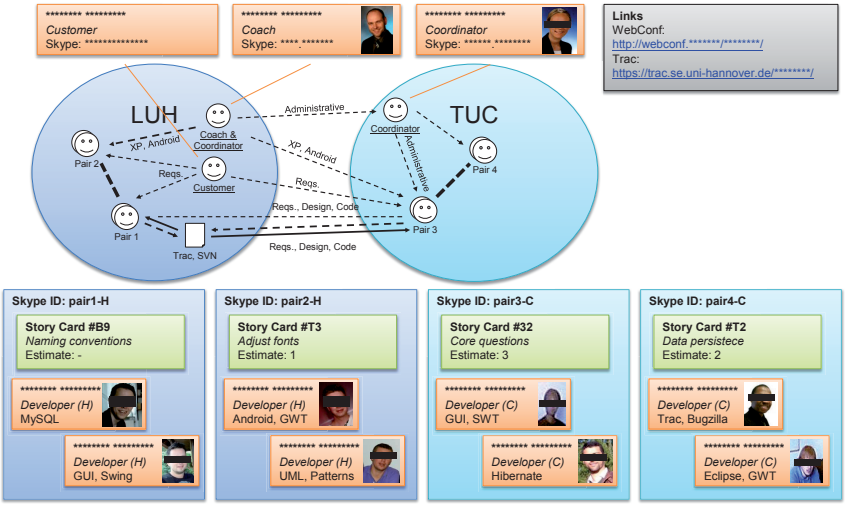


Abb. 6.7: Beispiel einer FLOW-Map (aus [Stapel 2011a])

- Unterschiedliche Liniendicken zur Unterscheidung unterschiedlich intensiver Informationsflüsse.
- Ungerichtete Flüsse zur Darstellung von Informationsflüssen in beide Richtungen.
- Optional: Piktogramme an standortübergreifenden Informationsflüssen zur Darstellung der genutzten Kommunikationsmedien.
- Zusätzliche Meta-Daten für jeden Speicher, die so genannten "Gelben Seiten": Kontaktinformationen (Name, E-Mail-Adresse, etc.), Bilder der Entwickler, lokale Zeit, Statusinformationen (beschäftigt, verfügbar), Rolle im Projekt, projektrelevante Fähigkeiten, aktuelle Aufgabe (z.B. bestimmtes Bug-Ticket oder User Story) und aktueller Arbeitsgegenstand (z.B. bestimmte Teile des Quellcodes), usw.
- Erweiterung für agile Projekte: Ein doppeltes Smiley-Symbol zur Darstellung von Pair-Programmierern.
- Erweiterung für agile Projekte: Erweiterte Gelbe-Seiten-Informationen zur Darstellung der Zusammensetzung von Programmierpaaren.

Eine FLOW-Map visualisiert Projektbeteiligte als flüssige Speicher, ihren Entwicklungsstandort, zentrale standortübergreifende Dokumente als feste Speicher, Gelbe-Seiten-Informationen und geplante bzw. tatsächliche Informationsflüsse zwischen allen Speichern. Ein Beispiel einer FLOW-Map mit geplanten Informationsflüssen und Gelbe-Seiten-Informationen ist in Abbildung 6.7 dargestellt.

Eine FLOW-Map kann neben der Awarenesssteigerung auch als kognitive Hilfe sowohl bei der Planung als auch bei der Durchführung der Kommunikation in verteilten Projekten dienen. Im Folgenden wird anhand der drei Phasen des FLOW-Verbesserungsprozesses beschrieben, wie mit Hilfe der FLOW-Map die Kommunikation in verteilten Projekten geplant und durchgeführt werden kann.

6.3.1.2 Phase 1: Erheben

FLOW-Mapping bietet in der ersten Phase des FLOW-Verbesserungsprozesses Unterstützung für die Planung der Kommunikation in einem verteilten Projekt und Unterstützung zur Ableitung der Informationsflüsse mit Hilfe von Kommunikationsereignissen. Abbildung 6.8 zeigt einen Überblick über die Aktivitäten der ersten Phase in FLOW-Notation. Zunächst werden die zentralen Rollen in FLOW-Mapping beschrieben.

Rollen in FLOW-Mapping Bei der Durchführung der FLOW-Mapping-Technik werden drei Rollen unterschieden:

1. **Koordinator:** Die koordinierende Rolle, z.B. ein Projektleiter oder Kommunikations-Coach, übernimmt planende, überwachende und steuernde Aufgaben bei der Durchführung von FLOW-Mapping. Der Koordinator ist für die Informationsflüsse des gesamten Projekts verantwortlich.
2. **Entwickler:** Entwickler nutzen die FLOW-Map zur Steigerung der Team-Awareness und zur Initiierung gezielter Kommunikation. Sie liefern individuelle Kommunikationsereignisse zur Steigerung der Awareness ihrer Kollegen und damit der Koordinator einen Überblick gewinnen kann.
3. **Administrator:** Die administrative Rolle übernimmt Aufgaben zur Bereitstellung und Aktualisierung der FLOW-Map.

Eine Person kann in einem Projekt auch mehrere Rollen übernehmen, z.B. die Koordination und die Administration.

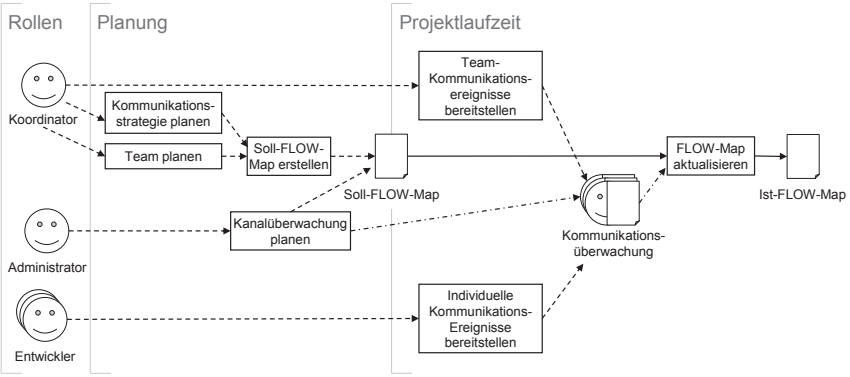


Abb. 6.8: FLOW-Mapping Aktivitäten der Phase 1: Erheben

Planung der Kommunikation Wie in Abbildung 6.8 gezeigt, besteht die Planung der Kommunikation aus vier Aktivitäten. Der Koordinator plant zunächst das Team und die Kommunikationsstrategie (vgl. auch [Stapel 2011a]). Beides ist Voraussetzung für die Erstellung der Soll-FLOW-Map für das Projekt. Für die in der Kommunikationsstrategie geplanten Medien bereitet der Administrator die Kanalüberwachung vor. Nach der Planung werden die Soll-FLOW-Map und Informationen der Vorbereitung der Kanalüberwachung im laufenden Projekt benutzt (siehe unten).

Bei der Planung des Teams werden die beteiligten Standorte, Entwickler im verteilten Team und andere für die Kommunikation wichtige Dinge, wie die für die standortübergreifende Kommunikation zu nutzende Sprache und Mindestanfoderungen an Programmierfähigkeiten, festgelegt. Schließlich müssen Kontaktdaten aller Projektbeteiligten gesammelt werden.

Eine Kommunikationsstrategie ist eine Menge von geplanten oder ereignisbasierten Kommunikationsaktivitäten mit zugehörigen zu nutzenden Kommunikationsmedien. Typische Kommunikationsaktivitäten in verteilten Projekten sind zum Beispiel tägliche verteilte Meetings zur Abstimmung oder ad-hoc Abstimmung zwischen zwei Entwicklern, die gemeinsam an einem Problem arbeiten. Die Kommunikationsstrategie wird in drei Schritten erstellt:

1. **Kommunikationsaktivitäten planen:** In diesem Schritt werden die Kommunikationsaktivitäten festgelegt, die für die weitere Betrachtung (Medienplanung und Überwachung) wichtig sind. Dabei werden zwei Arten von Aktivitäten unterschieden: (1) Aktivitäten, die regelmäßig, z.B. jeden Morgen oder zum Ende einer Woche, durchgeführt werden sollen, wie tägliche Stand-Ups, und (2) Aktivitäten, die zu einem bestimmten Ereignis ausgeführt werden sollen, wie Ad-hoc-Kommunikation, wenn eine Frage aufkommt, oder das Weiterleiten von Änderungswünschen des Kunden. Kommunikationsaktivitäten können aus dem verwendeten Softwareentwicklungsprozess oder aus Erfahrung aus ähnlichen Projekten abgeleitet werden.

2. **Mediennutzung planen:** Für jede Kommunikationsaktivität werden passende Kommunikationsmedien festgelegt, damit die Kanalüberwachung vorbereitet werden kann und während des Projekts der Aufgabe angemessene Medien genutzt werden. Für die Wahl geeigneter Medien können die Theoreme aus Kapitel 4.3 oder andere Theorien zur Medienwahl wie die Media Synchronicity Theory [Dennis 2008] genutzt werden. Bei verteilter Entwicklung ist es besonders wichtig, dass entsprechend Hypothese 4.1 gezielt ein Kanal für die Inhaltsübermittlung *und* ein Kanal für die Steuerung gewählt wird, statt nur einen Kanal für beides einzuplanen.

3. **Kommunikationsaktivitäten-spezifische FLOW-Maps erstellen:** Der letzte Schritt bei der Planung der Kommunikationsstrategie ist die Erstellung spezifischer FLOW-Maps für jede Kommunikationsaktivität. Diese spezifischen FLOW-Maps helfen Teilnehmer (Personen und Standorte), Informationsflussbesonderheiten der jeweiligen Aktivität und zwischen den Standorten zu nutzende Medien darzustellen. Wie eine FLOW-Map erstellt werden kann wird im folgenden Abschnitt beschrieben.

Mit Hilfe der Ergebnisse der vorangegangenen Aktivitäten, d.h. mit den Informationen über das Team und der Kommunikationsstrategie, kann die Soll-FLOW-Map erstellt werden. Die Soll-FLOW-Map gibt die geplanten Informationsflüsse für ein Projekt wieder. Es wird festgelegt welche Informationsspeicher (Personen und Dokumente), über welche Medien, wie intensiv kommunizieren sollen. Die FLOW-Map kann mit den folgenden sechs Schritten erstellt werden:

1. Erstelle eigene Bereiche für jeden beteiligten Standort.
2. Erstelle einen flüssigen Speicher für jeden Projektbeteiligten. Ordne die Speicher ihren Standorten zu.
3. Erstelle einen festen Speicher für jedes Dokument bzw. jeden Datenspeicher, der standortübergreifend-relevante Informationen enthält.
4. Erstelle flüssige Informationsflüsse zwischen flüssigen Speichern, die Informationen regelmäßig austauschen sollen. Erstelle flüssige Informationsflüsse zwischen flüssigen Speichern und festen Speichern, wenn Informationen in Dokumenten verfestigt werden sollen. Wenn Informationen vorwiegend in eine Richtung fließen sollen, dann kann man das mit einer Pfeilspitze in die entsprechende Richtung markieren. Ansonsten werden ungerichtete Informationsflüsse notiert. Die Intensität des gewünschten Flusses kann durch die Liniendicke verdeutlicht werden. Diese ist auch abhängig vom verwendeten Kommunikationskanal. Zum Beispiel kann in einem lokalen Gespräch von Angesicht zu Angesicht viel mehr Information ausgetauscht werden, als in der gleichen Zeit über einen Sofortnachrichtendienst (vgl. z.B. gesprochen vs. geschrieben in Tabelle 3.6).
5. Erstelle feste Informationsflüsse, wenn Dokumente regelmäßig gelesen werden sollen.
6. Abschließend wird die FLOW-Map um folgende Zusatzinformationen zu den Projektbeteiligten ergänzt:
    - Porträt-Bilder
    - Kontaktinformationen, wie E-Mail-Adresse, Telefonnummer oder Skype-ID entsprechend den geplanten Kommunikationsmedien
    - Geplante Rolle im Projekt
    - Projektrelevante Fähigkeiten
    - Zu bearbeitende Aufgabe(n)

Die Soll-FLOW-Map kann während der Planung genutzt werden, um den aktuellen Stand der Planung zwischen Koordinatoren und Entwicklern zu kommunizieren und während des Projekts, um den Entwicklern eines Standorts zu zeigen, wer noch am Projekt beteiligt ist und wie diese erreicht werden können. Zudem geben die Liniendicken der Informationsflüsse die Intensität der

geplanten Kommunikation an, sodass die Entwickler immer sehen können, mit wem sie regelmäßig in Kontakt sein sollten.

Die letzte Aktivität der Planung ist die Vorbereitung der Kanalüberwachung. Dies ist notwendig, damit später in der Ist-FLOW-Map ohne großen Aufwand, d.h. möglichst automatisch, aktuelle Kommunikationsereignisse visualisiert und Abweichungen vom Plan festgestellt werden können. Ein Vorteil verteilter Entwicklung ist, dass meist elektronische Medien zur standortübergreifenden Kommunikation genutzt werden. Diese können leichter automatisch überwacht werden als analoge Medien wie Briefpost oder lokale Meetings von Angesicht zu Angesicht. Für die Kanalüberwachung muss je Kanal ein Mechanismus geschaffen werden, der Kommunikationsereignisse erfassen und für die Visualisierung in der Ist-FLOW-Map verfügbar machen kann. Das kann z.B. eine Software sein, die Start, Ende und Teilnehmer einer Skype-Telefonkonferenz aufzeichnet und weiterreicht. Für Kanäle, die gar nicht oder nur mit sehr hohem Aufwand automatisch überwacht werden könnten, können auch Personen die Aufgabe der Erfassung und Weiterleitung von Kommunikationsereignissen übernehmen. Einige Beispiele für Werkzeuge zur Kanalüberwachung sind in Kapitel 6.3.1.4 weiter unten und in [Stapel 2011a] beschrieben. Als Hilfsmittel zur Vorbereitung für den späteren Soll-Ist-Vergleich sei auf die für Kommunikationsaktivitäten angepassten Conformance-Templates nach Zazworka verwiesen [Zazworka 2010, Stapel 2011a].

Projektdurchführung Nachdem die Kommunikation mit Hilfe von FLOW-Mapping geplant ist und die Soll-FLOW-Map erstellt wurde, kann das Projekt starten. Für eine effektive Awareness-Steigerung sollten die Projektbeteiligten nicht nur die geplanten, sondern auch die tatsächlichen Informationsflüsse sehen können. Daher werden während des Projekts mit Hilfe der Mechanismen der Kommunikationsüberwachung aktuelle Kommunikationsereignisse erfasst und in einer Ist-FLOW-Map dargestellt (vgl. Abb. 6.8, rechts).

Die Kommunikationsüberwachung sammelt Kommunikationsereignisse automatisch oder manuell, bei denen das gesamte Team oder nur ein Teil des Teams beteiligt sind, und stellt sie für die Aktualisierung der FLOW-Map zur Verfügung. Bei der Aktualisierung der Ist-FLOW-Map sollten folgende Änderungen berücksichtigt werden:

- Aktuelle Kommunikationsereignisse, wie Team-Meetings, Telefonkonferenzen, Dokumentenänderungen (z.B. Wiki-Änderungen) oder Quellcodeänderungen (z.B. via Versionsverwaltungssystem).

- Änderungen der Gelbe-Seiten-Informationen, z.B. wenn sich Rollen, Aufgaben oder Teamzusammensetzungen ändern.

Insgesamt wird für die Erhebung der Informationsflüsse beim FLOW-Mapping eine Bottom-Up-Strategie und das Verfahren der Kommunikationsereignisableitung genutzt (vgl. Tabelle 6.16 und Kapitel 6.2.2). Weitere Ausführungen, wie FLOW-Mapping zur Planung der Kommunikation in verteilten Projekten genutzt werden kann, finden sich in [Stapel 2011a].

6.3.1.3 Phase 2 & 3: Analysieren & Verbessern

Zur Analyse der Kommunikation nutzen die Entwickler und der Koordinator die Ist-FLOW-Map (vgl. Abb. 6.9). Wenn die Ist-FLOW-Map die aktuellen Informationsflüsse enthält, können die Entwickler sehen, wer mit wem kommuniziert, wer im Moment für Kommunikation verfügbar ist und wer gerade an welcher Aufgabe arbeitet. Mit einem Vergleich von Soll- und Ist-FLOW-Map können Unterschiede zur geplanten Kommunikation festgestellt werden. Auf Basis dieser Informationen können die Entwickler gezielter Kommunikation initiieren und der Koordinator die Kommunikation des gesamten Teams besser steuern. Beides führt zu einem verbesserten Informationsfluss im Projekt.

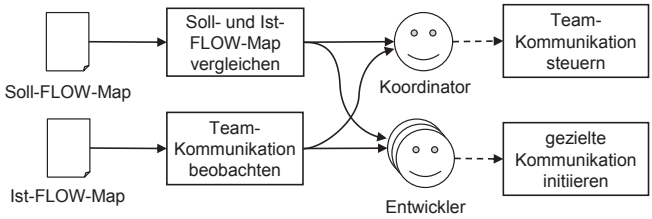


Abb. 6.9: FLOW-Mapping Aktivitäten der Phasen 2 & 3: Analysieren und Verbessern

6.3.1.4 Werkzeugunterstützung: Web-FLOW-Map

Die Web-FLOW-Map ist ein webbasiertes Werkzeug, das Unterstützung für folgende Aktivitäten der FLOW-Mapping-Technik bietet:

- Ist-FLOW-Map als Webseite darstellen
- Team-Kommunikationsereignisse entgegennehmen
- Individuelle Kommunikationsereignisse entgegennehmen
- Automatische Kanalüberwachung für Skype (Text, Telefon und Video), Subversion und Trac (Wiki- und Ticket-Änderungen).
- Aktuelle Ereignisse auf FLOW-Map visualisieren (Live-View)
- Unterstützung zur Initiierung gezielter Kommunikation
- Vergangene Ereignisse auf FLOW-Map visualisieren (History-View)
- Unterstützung bei Analyse des Kommunikationsverhaltens

Abbildung 6.10 zeigt einen Screenshot der Weboberfläche des Werkzeugs. Zentraler Bestandteil ist die Darstellung der FLOW-Map. Neben Standorten und wichtigen Informationsspeichern zeigt sie abhängig vom gewählten Modus entweder aktuelle Kommunikationsereignisse (Live-View) oder eine Zusammenfassung vergangener Kommunikationsereignisse (History-View). In der Live-View werden aktive Kommunikationsaktivitäten, die mehr als zwei Personen involvieren und aktive Gruppenchats unterhalb der FLOW-Map dargestellt. Dies geschieht außerhalb der FLOW-Map, um diese nicht visuell zu überladen. Auf der linken Seite gibt es Gelbe-Seiten-Informationen über den aktuell ausgewählten Entwickler und die Möglichkeit direkte Kommunikation mit diesem zu initiieren. Web-FLOW-Map wurde im Rahmen der Bachelorarbeit von Cieśnik [Ciesnik 2010] entwickelt. Dort finden sich auch technische Details zur Implementierung.

6.3.1.5 Evaluation der FLOW-Mapping-Technik

Ziel der Evaluation ist zu überprüfen, ob sich die FLOW-Mapping-Technik dazu eignet, in verteilten Projekten Kommunikation zu planen, Kommunikation zu steuern, Awareness zu steigern und gezielte Kommunikation zu initiieren. Um das zu erreichen, wurde eine Fallstudie [Wohlin 2000, S. 12] in einem verteilten studentischen Projekt durchgeführt. Mit einer Fallstudie kann geprüft werden, ob die FLOW-Mapping-Technik für den Einsatz in realistischen Projektsituationen geeignet ist.

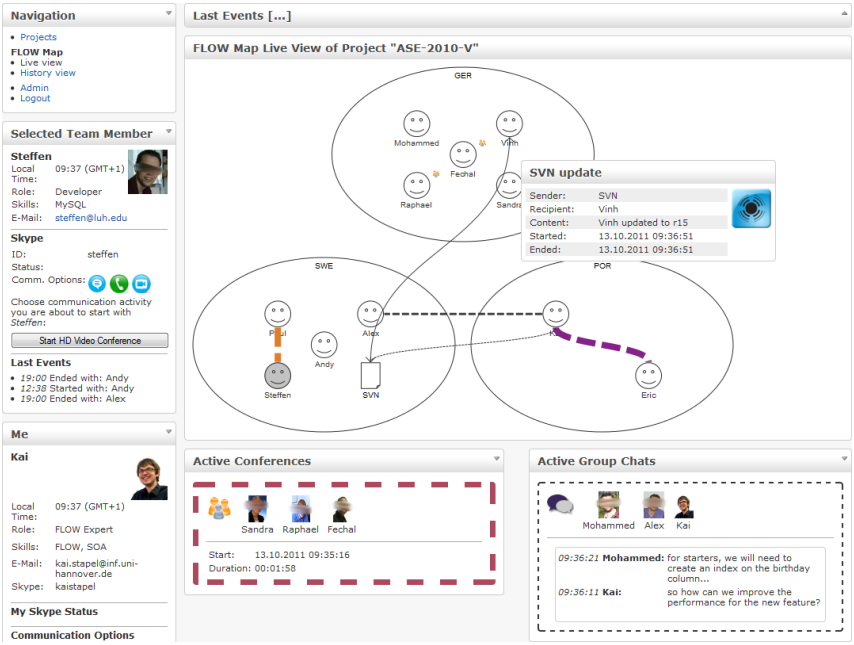


Abb. 6.10: Screenshot der Web-FLOW-Map

Kontext der Studie Die Studie wurde im Rahmen eines studentischen, agilen und verteilten Programmierprojekts im Sommersemester 2010 durchgeführt. Das agile Programmierprojekt ist Teil des Informatiklehrplans der Leibniz Universität Hannover (LUH) und hat hauptsächlich das Ziel, den Studenten agile Entwicklung am Beispiel von Extreme Programming in einer möglichst realistischen Umgebung beizubringen (vgl. [Stapel 2008]). 2010 wurde das Projekt zum ersten Mal verteilt durchgeführt. Zusätzlich zu den vier Informatikstudenten am Standort Hannover wurden vier Informatikstudenten der Technischen Universität Clausthal (TUC) integriert, um ein verteiltes Team zu schaffen. Der Hauptteil des Projekts ist eine einwöchige Entwicklungsphase, die der 40-Stunden-Wochen-Praktik nach [Beck 2000] folgt (XP-Woche). Zur Vorbereitung der XP-Woche gab es zwei vier-stündige Veranstaltungen, um den Studenten die XP-Grundlagen beizubringen. Zum zweiten Termin waren die Studenten aus Clausthal vor Ort in Hannover. Die Studenten konnten sich also vor der XP-Woche persönlich kennenlernen. Weiterhin gab es einen vier-

stündigen Kundentermin (verteilt) und zwei vier-stündige Termine für explorative Prototypen (verteilt) zur Vorbereitung auf die XP-Woche. Nach der XP-Woche gab es vier weitere Termine für abschließende Programmierarbeiten, Post-Mortem-Analysen und Diskussionen. Die Evaluation beschränkt sich im Wesentlichen auf die Ereignisse der XP-Woche, da die XP-Praktiken bis dahin üblicherweise ausreichend gut umgesetzt werden [Stapel 2008].

Planung und Studiendesign: Zur Evaluation der FLOW-Mapping-Technik wurde eine Fallstudie mit Studenten durchgeführt (vgl. [Stapel 2011a, Stapel 2011]). Mit der Studie sollten die folgenden Fragen beantwortet werden:

Eignung für die Planung: Eignet sich die FLOW-Mapping-Technik zur Planung der Kommunikation in verteilten Projekten? Zur Beantwortung dieser Frage wurde die Kommunikation des Projekts der Studie mit FLOW-Mapping geplant.

Eignung für das Management: Eignet sich die FLOW-Mapping-Technik zur Steuerung der Kommunikation in verteilten Projekten? Zur Beantwortung dieser Frage hat der Projektleiter des Projekts der Studie die Rolle des FLOW-Mapping-Koordinators übernommen. Zudem haben wir ausgewählte Kommunikationsmedien überwacht und die Kommunikationsereignisse während der Durchführung des Projekts berücksichtigt.

Awarenesssteigerung: Hilft die FLOW-Map die Awareness im Team zu steigern? Zur Beantwortung dieser Frage wurde die Soll-FLOW-Map im Entwicklerraum visualisiert und am Ende jedes Entwicklungstages nach der Nützlichkeit der Map gefragt.

Steigerung gezielter Kommunikation: Steigert die FLOW-Map gezielte Kommunikation? Zur Beantwortung dieser Frage wurden standortübergreifende Kommunikationsereignisse überwacht und die Entwickler befragt.

Kosten: Wie hoch sind einmalige und regelmäßige Kosten? Die Kosten einer Technik haben großen Einfluss auf ihren potenziellen Einsatz. Daher wurden Zeitaufwände für Planung und Durchführung der FLOW-Mapping-Technik in der Studie ermittelt.

Metriken Zur Beantwortung der obigen Fragen wurden die folgenden Metriken benutzt:

Koordinator-Befragung: Am Ende des Projekts wurde der Koordinator des Projekts in einem informellen Interview zu seiner Meinung zur FLOW-Mapping-Technik in Bezug auf die Planung und Steuerung der Kommunikation in verteilten Projekten befragt.

Entwickler-Fragebogen: Ein täglich von den Entwicklern nach ihrer Arbeit auszufüllender Fragebogen, u.a. mit folgenden Fragen (vgl. kompletten Fragebogen im Anhang A.2):

- Haben Sie heute die FLOW-Map benutzt bzw. bewusst angesehen?
- Empfanden Sie heute die FLOW-Map als hilfreich?
- War die FLOW-Map heute immer aktuell?
- Wussten Sie heute immer, an was die Kollegen am anderen Standort aktuell arbeiten?

Kommunikationsüberwachung: Auswertung der Log-Dateien der elektronischen Kommunikationsmedien, die für die standortübergreifende Kommunikation genutzt wurden. Dabei wurden die folgenden Eigenschaften eines Kommunikationsereignisses erfasst: Medium, Startzeit und -datum, Dauer (wenn zutreffend), Richtung des Informationsflusses, Sender, Empfänger, Inhalt (wenn vorhanden).

Validitätsdiskussion In der Studie wurde nur ein verteiltes Team betrachtet. Ein Vergleich zwischen Projekten, die FLOW-Mapping anwenden, und Projekten, die keine oder eine andere Methode anwenden, ist daher nicht möglich. Die Metriken der täglichen Befragung sind qualitativ und können daher subjektiv beeinflusst sein. Diesem Einfluss wurde teilweise durch objektivere Zweitmetriken, z.B. Messung des tatsächlichen Kommunikationsverhaltens, entgegengewirkt. Um Einflüssen auf Grund von Bewertungsangst entgegenzuwirken, wurde den Studenten versichert, dass ihre Antworten auf den Fragebögen vertraulich behandelt werden und sie keinen Einfluss auf das Bestehen der Lehrveranstaltung haben. An der Studie haben vier Bachelor- und vier Masterstudenten über zwei Standorte verteilt teilgenommen. Beide Standorte waren in Deutschland, d.h. auch in derselben Zeitzone. Das Team war

kulturell durchmischt (Westeuropa, Asien, Afrika, Mittlerer Osten). Alle Studenten kannten aber die deutsche Kultur, da jeder mindestens ein paar Monate in Deutschland gelebt hatte. Die Projektsprache war Deutsch. Drei der acht Studenten waren keine Muttersprachler. Die beiden Sub-Teams hatten einen unterschiedlichen Ausbildungshintergrund (TUC vs. LUH) und waren unterschiedlich weit im Informatikstudium fortgeschritten (Bachelor vs. Master). Die Hauptmotivation der Studenten an dem Projekt teilzunehmen war es Extreme Programming zu lernen. Daher kann es sein, dass die Studienteilnehmer kommunikativer waren als ein durchschnittlicher Informatikstudent. Weiterhin kann die Validität unserer Ergebnisse zunächst nur für agile Softwareentwicklung beansprucht werden. Aber gerade bei agiler Entwicklung ist gut funktionierende standortübergreifende Kommunikation besonders wichtig. Insgesamt sind die gewonnenen Ergebnisse besonders relevant für kleine, national verteilte, kulturell homogene und agile Teams.

Durchführung Das Projekt der Studie wurde unter Anwendung der FLOW-Mapping-Technik geplant und durchgeführt. Entsprechend Abbildung 6.8 wurden zunächst das Team und die in Tabelle 6.14 zusammengefasste Kommunikationsstrategie geplant. So wurden z.B. tägliche Stand-Up- und Wrap-Up-Meetings per Videokonferenz mit dem gesamten Team eingeplant. Kommunikation zwischen zwei Entwicklern oder Entwicklerpaaren sollte über Skype (Text oder Telefon) direkt vom Arbeitsplatz aus durchgeführt werden, wann immer ein Entwickler dies für nötig hielt. Zusätzlich wurde die Kanalüberwachung vorbereitet, indem spezielle Skype-Accounts angelegt wurden und ein Werkzeug zur Überwachung der Skype-Statusmeldungen entwickelt wurde. Anschließend wurde eine Soll-FLOW-Map erstellt (vgl. Abb. 6.7) und in weiteren Planungsiterationen zur Kommunikation des Plans genutzt und dabei stetig weiter entwickelt.

Für das Projekt konnte ein echter Kunde mit einem echten Produktwunsch gewonnen werden. Es sollte eine Smartphone-Anwendung für Jugendliche entwickelt werden, die durch spielerische und lehrreiche Elemente an das Thema Suchtprävention herangeführt werden sollen. Mit Hilfe der Anwendung sollen Jugendliche lernen, wie man sich in einer Notsituation richtig verhalten soll, z.B. bei Alkoholmissbrauch eines Freundes. Wie Abbildung 6.7 zeigt war die Kundin vor Ort am Standort Hannover. Dort hat sie die Rolle des On-Site Customers eingenommen. Die Projektleitung war auch in Hannover angesiedelt. Der Projektleiter hat neben der Rolle des FLOW-Koordinators auch die Rol-

Tabelle 6.14: Kommunikationsstrategie der Studie zur Evaluation von FLOW-Mapping (aus [Stapel 2011a])

Kommunikations-aktivität	**Zeitplan / Ereignis**	**Kommunikations-medien**	**Teilnehmer**
Stand-Up- / Wrap-Up-Meetings	jeden Morgen / Abend	Videokonferenz	alle, exkl. Kunde
Planning Game	Iterationsbe-ginn	Videokonferenz mit geteilter Mind-Map	alle, inkl. Kunde
Akzeptanztests je Iteration	Iterationsen-de	Videokonferenz mit geteiltem Deskop	alle, inkl. Kunde
Akzeptanztests je User Story	Fertigstel-lung User Story	Skype-Telefonkonferenz mit geteiltem Desktop	Entwicklerpaar und Kunde
Informelle Zusammenarbeit	bei Bedarf	Skype-Anruf oder -Chat mit geteiltem Desktop	zwei Entwick-lerpaare
Informelle Abstimmung	bei Bedarf	Skype-Anruf oder -Chat	zwei Entwick-lerpaare
Statusaktualisie-rung	bei Statusän-derung	Skype-Statusnachricht	Entwickler

le des XP- und technischen Coaches für beide Standorte übernommen. Eine Mitarbeiterin des Lehrstuhls in Clausthal hat dort organisatorische Aufgaben übernommen. Den Entwicklern stand neben den in der Strategie genannten Kommunikationsmedien (vgl. Tabelle 6.14) ein Ticket- und Versionsverwaltungssystem zur Verfügung.

Während der XP-Woche haben die acht Studenten gemeinsam an der Entwicklung des Lernteils der Smartphone-Anwendung gearbeitet. Dabei haben sie die Extreme Programming Praktiken angewandt. Die Iterationslänge war zwei Tage. Die Entwickler haben lokal in Paaren programmiert. Die Paare mussten nach Fertigstellung jeder User Story und zu Beginn jedes neuen Tages wechseln, um die Wissensaustauscheffekte dieser Praktik für jedes Sub-Team zu nutzen. Zwischen den Standorten wurden keine Paare gebildet. Bei jeder Änderung der Paarzusammensetzung und der bearbeiteten User Story wurden die Gelbe-Seiten-Informationen der Ist-FLOW-Map angepasst und den Entwicklern zugänglich gemacht. Die FLOW-Map selbst hat stets die Soll-Informationsflüsse gezeigt, da zum Zeitpunkt der Studien noch keine Werkzeugunterstützung für die automatische Aktualisierung von Kommunikationsereignissen vorhanden war. Das Fehlen eines solchen Werkzeugs hat auch dazu geführt, dass wir am 4. Tag auf Grund der häufig aufgetretenen Änderungen die Gelbe-Seiten-Informationen nicht mehr aktuell halten konnten. Am Ende jedes Tages haben die Entwickler einen Fragebogen ausgefüllt (vgl. Anhang A.2).

Ergebnisse Die Koordinator-Befragung hat die folgenden wichtigen Punkte für die Nützlichkeit von FLOW-Mapping bei der Planung von verteilter Kommunikation ergeben:

- Die visuelle Darstellung aller Teilnehmer in den verschiedenen Standorten hat bei der Rollenverteilung geholfen.
- Die visuelle Darstellung der Soll-Informationsflüsse hat geholfen, Kommunikationsaktivitäten einzuplanen und zu priorisieren.

Weiterhin wurden folgende Punkte über die Nützlichkeit bei der Durchführung von verteilten Projekten genannt:

- Es war hilfreich, immer die aktuelle Paar-Zusammensetzung und die aktuell bearbeitete User Story zu sehen.

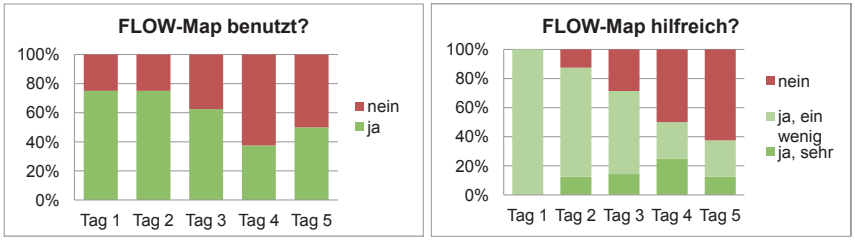


Abb. 6.11: Nutzung (links) und wahrgenommene Nützlichkeit (rechts) der FLOW-Map im ASE10-Projekt ($n = 8$)

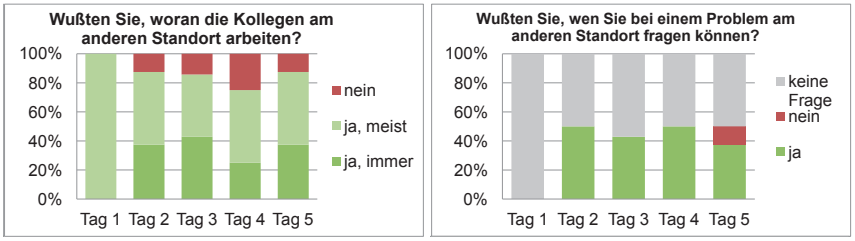


Abb. 6.12: Awareness über bearbeitete Aufgaben (links) und Ansprechpartner (rechts) am anderen Standort im ASE10-Projekt ($n = 8$)

- Die Darstellung von Namen und Bildern der Projektbeteiligten hat geholfen, direkt zu Beginn des Projekts jeden beim Namen nennen zu können.

Die Auswertung des Entwickler-Fragebogens hat das in den Abbildungen 6.11 und 6.12 dargestellte Bild ergeben.

Abbildung 6.11 (links) zeigt, dass die FLOW-Map zu Beginn des Projekts von 75 % der Entwickler genutzt wurde. Am 4. Tag ist die Nutzungsrate auf 38 % zurückgegangen. Nach dem Wochenende, am 5. Tag, ist sie wieder leicht auf 50 % gestiegen. Abbildung 6.11 (rechts) zeigt die von den Entwicklern wahrgenommene Nützlichkeit der FLOW-Map. Die Nützlichkeit ist während der XP-Woche monoton von 100 % an Tag 1 auf 38 % an Tag 5 gefallen. Um dieses Ergebnis korrekt interpretieren zu können, sei angemerkt, dass die FLOW-Map an Tag 4 und 5 nicht mehr aktuell war, weil der manuelle Aktualisierungsprozess mit den vielen Änderungen nicht mehr Schritt halten konnte.

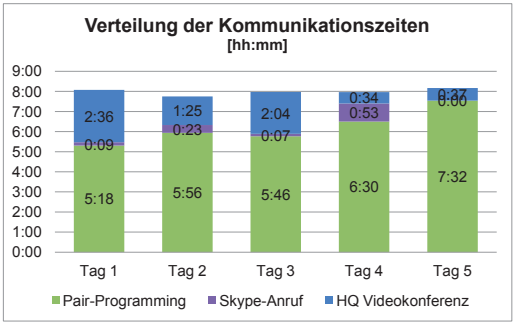


Abb. 6.13: Verteilung der Kommunikationszeiten in der Hauptentwicklungsphase der Studie je Entwickler (aus [Stapel 2011a])

Abbildung 6.12 (links) zeigt, dass im Durchschnitt weniger als 15 % der Entwickler nicht wussten, woran ihre Kollegen am anderen Standort gerade arbeiten. Abbildung 6.12 (rechts) zeigt, dass während der XP-Woche nur einmal der Fall eingetreten ist, dass ein Entwickler eine Frage hatte, die nur am anderen Standort beantwortet werden konnte, er aber nicht wusste, wen er dort fragen kann.

Die Auswertung der Kommunikationsüberwachung hat die beiden in Abbildung 6.13 und Abbildung 6.14 dargestellten Übersichten über den Kommunikationsverlauf während der XP-Woche ergeben. Abbildung 6.13 zeigt die durchschnittlichen Nutzungszeiten der wichtigsten Kommunikationsmedien je Entwickler pro Tag der XP-Woche. Skype-Telefonate und Videokonferenzen wurden für die standortübergreifende Kommunikation genutzt. Die Zeiten für Pair-Programming geben die ausschließlich lokal stattfindende Kommunikation an. Abbildung 6.14 gibt einen detaillierteren Überblick über die Verteilung der standortübergreifenden Kommunikation während der XP-Woche. Es ist zu erkennen, dass an Tag 1 und an Tag 3 ein Planning Game stattgefunden hat. Die Planning Games repräsentieren Iterationsgrenzen. An Tag 5 ist das Planning Game ausgefallen, obwohl es dafür geplant war, weil die Kundin nicht vor Ort sein konnte. Zudem gab es an Tag 5 kein Stand-Up-Meeting. Weiterhin sind zwei verteilte Mittagspausen zu erkennen, bei denen beide Teams zusammen bei laufender Videokonferenz Mittag gegessen haben. Informelle Kommunikation via Skype ist relativ gleichmäßig über die Entwicklungstage verteilt.

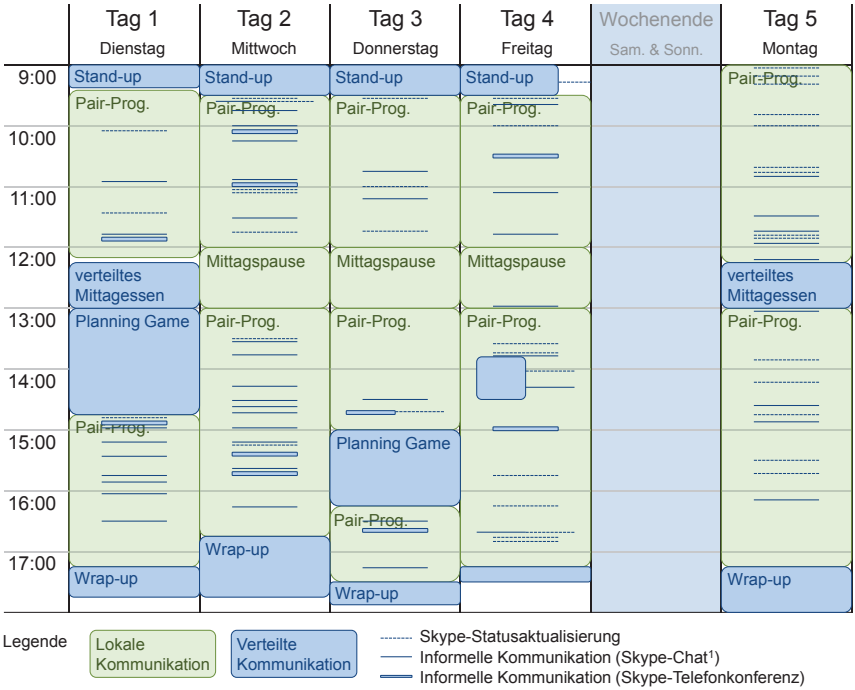


Abb. 6.14: Überblick über Kommunikation in der Hauptentwicklungsphase der Studie (aus [Stapel 2011a])

Tabelle 6.15 zeigt die während der Anwendung von FLOW-Mapping entstandenen zeitlichen Aufwände, klassifiziert nach Planung und Projektlaufzeit. Die Planungsphase wird nur dann sehr aufwendig, wenn Werkzeuge zur Unterstützung der Kanalüberwachung neu entwickelt werden müssen (vgl. Skype-Statusmonitor). Ansonsten konnte die Kommunikation für das Projekt der Studie in ca. einem Arbeitstag geplant werden. Während der Projektlaufzeit fallen relativ wenig Aufwände an, je nachdem, wie gut die Kommunikationsüberwachung vorbereitet wurde. Für die in der hier vorgestellten Evaluation genutzten Metriken konnten alle Daten automatisch erhoben werden, sodass während des Projekts kein zusätzlicher Aufwand entstand. Allerdings war die Aktualisierung der FLOW-Map zum Zeitpunkt der Studie noch nicht auto-

Tabelle 6.15: Aufwände für Anwendung der FLOW-Mapping-Technik in der Studie (aus [Stapel 2011a])

FLOW-Mapping-Aktivität	Aufwand	Häufigkeit
Planung		
Team planen	4 h für koordinierende Anrufe und Mails	1
Kommunikationsstrategie erstellen	2 h Brainstorming-Meeting	1
Soll-FLOW-Map erstellen	15 min Skizze, 1 h poliert	1
Kanalüberwachung vorbereiten	1 d Entwicklung Skype-Statusmonitor, 1 h Skype-Benutzerkonten einrichten, 2 h Erhebungsbögen vorbereiten, 15 min SVN-Commit-Template vorbereiten	1
Projektlaufzeit		
Kommunikationsereignisse sammeln und auswerten	0, da automatisiert	täglich, je Ereignis
FLOW-Map aktualisieren	10 min je Änderung	täglich, je Änderung

matisiert, was bei vielen Änderungen der Paarzusammensetzungen zu einem hohen Arbeitsaufwand geführt hat.

Interpretation Im Folgenden werden die Ergebnisse der Studie entsprechend den eingangs genannten Fragen interpretiert.

Eignung für die Planung: Durch Anwendung der FLOW-Mapping-Technik in dem Projekt der Fallstudie konnte gezeigt werden, dass sie sich für die Planung der Kommunikation in verteilten Projekten eignet. Die Befragung des Koordinators hat ergeben, dass vor allem die visuelle Darstellung der FLOW-Map hilfreich für die Planung war. Das erfolgreiche Projekt[5] und die tatsächlich stattfindende standortübergreifende Kom-

[5] Erfolgreiche Abnahme und zufriedene Kundin, vgl. Anhang A.1

munikation, bei der sich weitestgehend an die Kommunikationsstrategie gehalten wurde (vgl. [Stapel 2011a]), zeigen auch, dass der Plan seinen Zweck erfüllt hat und damit praktikabel war.

Eignung für das Management: Die Befragung des Koordinators hat ergeben, dass die FLOW-Map auch während des Projekts nützlich für die Steuerung der Kommunikation war. Informationen aus der Kommunikationsüberwachung konnten direkt in die Steuerung des laufenden Projekts eingehen. Zudem wird davon ausgegangen, dass der positive Effekt der FLOW-Map bei der Unterstützung der Steuerung der Kommunikation in größeren Teams, die weniger und in größeren Abständen geplante Meetings haben, noch stärker zum Tragen kommt.

Awarenesssteigerung: Abbildung 6.11 (links) zeigt, dass die FLOW-Map während der XP-Woche von der Mehrheit der Entwickler genutzt wurde. Je aktueller die FLOW-Map ist, desto nützlicher ist sie (vgl. Abb. 6.11 (rechts)). Insbesondere am 4. Tag, als die FLOW-Map nicht mehr aktuell gehalten werden konnte, d.h. als sie nicht mehr die aktuellen Aufgaben und Paarzusammensetzungen gezeigt hat, wurde sie am wenigsten benutzt und als wenig hilfreich angesehen. Der leichte Anstieg der Nutzung der FLOW-Map am 5. Tag kann evtl. mit dem vorangegangenen Wochenende erklärt werden. Während des Wochenendes ist die Awareness etwas zurückgegangen, sodass am Wochenanfang auch die nicht ganz aktuelle FLOW-Map als teilweise nützlich angesehen wurde. Da eine aktuelle FLOW-Map sehr wichtig, aber der manuelle Aktualisierungsaufwand sehr hoch ist, wurde im Anschluss an die Studie das oben (S. 326) beschriebene Werkzeug zur (semi-)automatischen Erfassung und Darstellung von Kommunikationsereignissen und sonstigen Änderungen der FLOW-Map entwickelt. Abbildung 6.12 (links) zeigt, dass fast alle Entwickler durchgängig wussten, woran die Entwickler am anderen Standort gerade arbeiten. Abbildung 6.12 (rechts) zeigt, dass nur einmal ein Entwickler nicht wusste, wen er am anderen Standort zur Lösung seines Problems fragen kann. D.h., die Awareness schien während der XP-Woche auf einem hohen Niveau gewesen zu sein. Insgesamt hat die Auswertung des Entwickler-Fragebogens gezeigt, dass die Informationen auf der FLOW-Map insbesondere zu Beginn eines verteilten Projekts hilfreich für die Awarenesssteigerung sein können.

Steigerung gezielter Kommunikation: Während der XP-Woche gab es 57 gezielte Kommunikationsereignisse zwischen Entwicklerpaaren der bei-

den Standorte (vgl. Abb. 6.14). D.h., trotz der geplanten Meetings zu Beginn und zum Ende jedes Entwicklungstages haben die Entwickler auch während der Pair-Programming-Phasen regelmäßig gezielt standortübergreifend kommuniziert. Zu welchem Teil das der FLOW-Mapping-Technik zuzuordnen ist, lässt sich nicht genau sagen, da ein Vergleichsprojekt fehlt. Dennoch kann festgestellt werden, dass im Projekt eine kommunikationsintensive agile Entwicklung möglich war, auch über die Standortgrenzen hinweg.

Kosten: Der Einsatz einer Technik ist nur dann sinnvoll, wenn ihr Nutzen die Kosten ihres Einsatzes übersteigt. Um die Kosten abschätzen zu können, wurden die durch Nutzung von FLOW-Mapping entstandenen Aufwände für Planung und Durchführung der Kommunikation ermittelt (vgl. Tabelle 6.15). FLOW-Mapping kann als leichtgewichtiger Ansatz für die Planung der Kommunikation in verteilten Projekten gesehen werden, da nur ca. ein Arbeitstag Planungsaufwand entsteht, sofern das Team nicht viel größer als das in der Studie ist und keine Werkzeuge für die Kanalüberwachung neu entwickelt werden müssen. Der zusätzliche Planungsaufwand für die Neuentwicklung von Kanalüberwachungswerkzeugen sollte aber in Betracht gezogen werden, wenn der Aufwand während des Projekts gering gehalten werden soll. Hier gilt es einmalige gegen ständig auftretende Aufwände abzuwägen.

Insgesamt konnten Hinweise für die Nützlichkeit der FLOW-Mapping-Technik für Planung und Durchführung verteilter Kommunikation gefunden werden. Die Ergebnisse der hier präsentierten Studie zeigen, dass die FLOW-Map die Awareness in verteilten Teams steigern, bzw. auf einem hohen Niveau halten kann, und dass die FLOW-Map gezielte Kommunikation erleichtern kann.

6.3.1.6 Zusammenfassung und Einordnung in FLOW-Methode

FLOW-Mapping ist eine Technik zur Verbesserung der Informationsflüsse in verteilten Projekten. Erste Evaluationsergebnisse haben gezeigt, dass die Technik erfolgreich zur Planung und Durchführung von verteilten Softwareprojekten eingesetzt werden kann (vgl. oben und [Stapel 2011a, Stapel 2011].

Die Visualisierung geplanter, laufender und vergangener Informationsflüsse hilft die Kommunikation in verteilten Projekten zu planen, zu verstehen und schließlich zu verbessern. FLOW-Mapping bietet daher für alle drei Phasen der

FLOW-Methode (vgl. Kapitel 6.2) konkrete Unterstützung. Tabelle 6.16 fasst die wesentlichen Eigenschaften von FLOW-Mapping mit Hilfe des Templates aus Tabelle 6.4 zusammen.

6.3.2 Verfestigung als Nebenprodukt

Fehlende Dokumentation ist ein typisches Problem der Softwareentwicklung, insbesondere in global verteilten Projekten [Stapel 2009]. Die FLOW-Technik zur Verfestigung von Kommunikation als Nebenprodukt hilft, wichtige Informationen, z.B. aus verteilten Meetings, so zu dokumentieren, dass sie später leicht wieder gefunden werden können. Um die Teilnehmer des Meetings bei der Dokumentationserstellung möglichst wenig bei ihrer eigentlichen Arbeit zu stören und trotzdem nützliche, gut strukturierte Dokumentation zu bekommen, wird der Nebenprodukt-Ansatz von Schneider [Schneider 2006] verwendet. Dieser sieht vor, dass Informationen während einer Tätigkeit automatisch aufgezeichnet werden und dabei so viele Zusatzinformationen wie möglich, aber ohne dabei die eigentliche Aufgabe zu stören, für die Erstellung eines Index mit aufgezeichnet werden. Falls die Aufzeichnung noch weiter strukturiert und indexiert werden soll, soll dies in einem separaten Arbeitsschritt passieren. Der Index ist notwendig, um später schneller die relevanten Informationen in einer potenziell sehr großen Datenmenge (z.B. Audiomitschnitt eines drei stündigen Meetings) wiederfinden zu können.

Abbildung 6.15 zeigt die wesentlichen Informationsflüsse der Verfestigungstechnik. Während ihrer regulären Arbeit kommunizieren die Entwickler. Ein speziell auf eine bestimmte Kommunikationsaktivität ausgelegtes Verfestigungswerkzeug erstellt eine Dokumentation inklusive Index nach [Schneider 2006]. Diese Dokumentation kann zu einem späteren Zeitpunkt von den Entwicklern genutzt werden, um wichtige Informationen wiederzufinden. Der Index hilft dabei, diese Informationen schneller wiederzufinden.

6.3.2.1 Evaluation

Ziel der Verfestigung-als-Nebenprodukt-Technik ist die Verbesserung des Informationsflusses in der Softwareentwicklung durch Verfestigung wichtiger Informationen. Der Nebenprodukt-Ansatz [Schneider 2006] schlägt vor, zusätzlich zum bloßen Mitschneiden einer Kommunikation, Zusatzinformationen zur Erstellung eines Index zu berücksichtigen, damit bei der späteren Suche

Tabelle 6.16: Klassifikation von FLOW-Mapping als FLOW-Technik

Name der Technik		FLOW-Mapping
Ziel	**Absicht**	☒ Verstehen ☒ Verbessern
	Zeit	☒ vorher ☒ während dessen ☐ nachher
	Umfang	☐ Aktivität ☒ Projekt ☐ Organisation
Phase		☒ Erheben ☒ Analysieren ☒ Verbessern
Projektparameter		
Verteiltheit		☐ Lokal ☒ Verteilt (Art: horizontal)
Phasenspezifische Aspekte		
Erhebungsstrategie		☒ Bottom-Up ☐ Top-Down
Erhebungsverfahren		☐ Selber machen ☐ Beobachten ☐ Interview ☐ Fragebogen
		☐ Modellableitung ☒ Kommunikationsereignisableitung
Analysestrategie		☒ manuelle Analyse ☒ teilautomatische Analyse ☐ automatische Analyse
Analyseverfahren		☒ Visualisierung ☐ Simulation ☐ Mustersuche
Verbesserungs-strategie		○ Nebenprodukt ⊗ Hauptprodukt ○ Leichtgewichtig ○ Schwergewichtig
Verbesserungs-verfahren		☐ Aggregatzustand (○ verfestigen ○ verflüssigen)
		☒ Fluss (⊗ Abkürzung ○ Umweg ○ Verzweigung ○ Zusammenführung)
		☐ Aktivität (○ Schnittstellenanpassung ○ Aktivitätsanpassung)
		☐ Musterersetzung

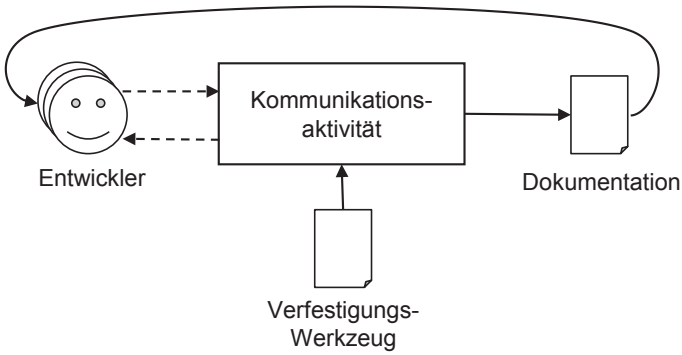


Abb. 6.15: FLOW-Technik: Verfestigung als Nebenprodukt

in der Aufzeichnung relevante Informationen schneller gefunden werden können. In der hier präsentierten Evaluation soll geprüft werden, ob Dokumentation mit Index tatsächlich schneller bei der Suche wichtiger Informationen ist als Dokumentation ohne Index.

Die Prüfung findet am Beispiel einer 20-minütigen Skype-Audioaufzeichnung eines verteilten Stand-Up-Meetings eines global verteilten Softwareprojekts (vgl. [Stapel 2009]) statt. Folgende Hypothese wird in der Evaluation untersucht:

Alternativhypothese: Es gibt einen Unterschied zwischen der Dauer der Suche einer bestimmten Information in einer Audioaufzeichnung mit Index und der Suche in der selben Audioaufzeichnung ohne Index.

Nullhypothese: Es gibt keinen Unterschied zwischen der Dauer der Suche einer bestimmten Information in einer Audioaufzeichnung mit Index und der Suche in der selben Audioaufzeichnung ohne Index.

Studiendesign und Durchführung Zur Prüfung der Hypothese wurde ein Online-Fragebogen-basiertes Experiment benutzt. Es wurde ein Online-Fragebogen gewählt, um eine möglichst große Stichprobe erreichen zu können. Der Fragebogen bestand aus einer einleitenden Szenariobeschreibung, Fragen zu Metadaten der Teilnehmer, und drei Fragen, die mit Hilfe des Audiomitschnitts beantwortet werden sollten. Vor, zwischen und nach der Beantwortung der Fragen sollten die Teilnehmer die Zeit notieren. Zusätzlich hat das

Tabelle 6.17: Deskriptive Statistik des Effizienzvergleichs zwischen Audiodokumentation mit und ohne Index

$n = 15$, in [mm:ss]	**Mit Index**	**Ohne Index**
Maximum	12:00	15:41
Oberes Quartil	8:38	14:30
Median	6:30	12:00
Unteres Quartil	5:28	7:24
Minimum	5:00	5:35
Mittelwert	7:17	11:00
Standardabweichung	2:25	4:13
Relativer Unterschied der Mittelwerte	34 % schneller	51 % langsamer
Signifikanz des Unterschieds[1]	$p = 6{,}9\,\% < \alpha = 10\,\%$	

[1] p-Wert des generalisierten t-test nach Welch zwischen den Stichproben mit und ohne Index [Welch 1947]

verwendete Online-Fragebogensystem[6] die Gesamtzeit zur Beantwortung aller Fragen protokolliert. Es gab einen Fragebogen mit Audiodokumentation inklusive Index und einen Fragebogen mit Audiodokumentation ohne zusätzliche Informationen. Die Audiodokumentation wurde als MP3-Download bereitgestellt. Der Index wurde in Form eines Cuesheet[7] bereitgestellt. Ein Cuesheet erlaubt es, mehrere Sprungmarken innerhalb einer Audiodatei zu setzen und wird von vielen Werkzeugen zum Abspielen von Audiodateien unterstützt. Der Index wurde von uns in der Rolle des Erfahrungsingenieurs (vgl. Abb. 6.17) nachträglich manuell erstellt. Um den Einfluss eines Index auf Suchzeit in Audiomitschnitten isoliert betrachten zu können, wurden folgende Variablen im Experiment kontrolliert: Alle Teilnehmer haben die gleiche Szenariobeschreibung, die gleiche Audiodatei und die gleichen Fragen bekommen.

Die in der Studie verwendeten Fragebögen für die Befragung mit und ohne In-

[6] https://www.soscisurvey.de/
[7] http://de.wikipedia.org/wiki/Cuesheet

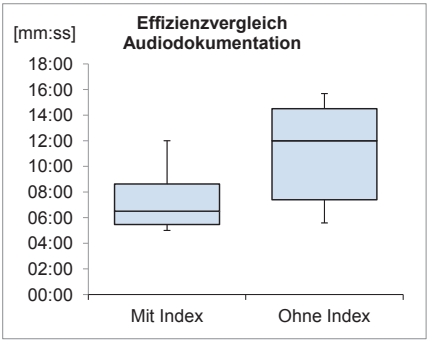


Abb. 6.16: Effizienzvergleich zwischen Audiodokumentation mit und ohne Index ($n = 15$)

dex finden sich im Anhang A.3 ab Seite 376. Der Fragebogen war während der Befragung über einen Link erreichbar. Das Fragebogensystem hat beim Öffnen des Links zufällig den Fragebogen mit oder ohne Index ausgegeben, sodass eine zufällige und gleichverteilte Aufteilung der beiden Varianten auf die Teilnehmer gewährleistet war. Die Teilnehmer wurden unter den Mitarbeitern und wissenschaftlichen Hilfskräften des Fachgebiets Software Engineering sowie den Informatikstudenten der Leibniz Universität Hannover akquiriert.

Ergebnis 16 Personen haben den Fragebogen vollständig beantwortet. Ein Fragebogen wurde von der Auswertung ausgeschlossen, weil ein Teilnehmer die Fragen nur zu ca. 50 % korrekt beantwortet und bei Frage 3 offensichtlich geraten hatte. Daher war nicht davon auszugehen, dass dieser Teilnehmer sich ernsthaft mit den Fragen beschäftigt hat. In der Auswertung wurden schließlich 8 Fragebögen mit und 7 Fragebögen ohne Index berücksichtigt. Tabelle 6.17 zeigt deskriptive Statistiken in den beiden Kategorien. Abbildung 6.16 zeigt Median, Maximum, Minimum, oberes und unteres Quartil der Bearbeitungszeiten nochmal als Boxplot. Im Mittel haben die Teilnehmer mit Index 7:17 Minuten und die Teilnehmer ohne Index 11:00 Minuten zur Bearbeitung der Fragen gebraucht. Im Durchschnitt hat die Beantwortung der Fragen ohne Hilfe des Index also 51 % mehr Zeit in Anspruch genommen als mit Hilfe des Index. Dieser Unterschied ist schwach signifikant. Die Nullhypothese kann mit einer Fehlerwahrscheinlichkeit von $\alpha = 10\,\%$ verworfen werden.

Interpretation und Validitätsdiskussion Der Index für die in dieser Studie benutzte Audiodokumentation wurde nachträglich manuell erstellt. Das entspricht dem Szenario der nachträglichen Erfahrungsaufbereitung durch einen Erfahrungsingenieur nach Nebenproduktansatz [Schneider 2006]. Es wurde gezeigt, dass ein Index, der wichtige Entscheidungspunkte markiert, zur schnelleren Navigation in Audioaufzeichnungen hilfreich ist.

Das Ergebnis, dass ein Index hilft, relevante Informationen schneller zu finden, scheint trivial, untermauert aber nochmal, dass es sinnvoll ist, zusätzlichen Aufwand in die Erstellung eines solchen Index zu stecken. Der in der Studie benutzte Audiomitschnitt war nur 20 Minuten lang. Es wird davon ausgegangen, dass die positiven Effekte eines Index in längeren Mitschnitten stärker zum Tragen kommen. Die Technik der Verfestigung als Nebenprodukt scheint für die Dokumentation langer Meetings also besonders wertvoll zu sein.

Um den Zusatzaufwand für die Erstellung eines Index für einen Audiomitschnitt möglichst gering zu halten, wurde ein Werkzeug entwickelt, welches einen parallel zu einer Skype-basierten Telefonkonferenz stattfindenden Skype-Chat nutzt, um einen Index zu erstellen. Das Werkzeug kann so helfen, ohne Nachbearbeitungsaufwand einen Index zu erstellen, der wichtige Entscheidungen in der Meetingdokumentation markiert.

6.3.2.2 Werkzeugunterstützung

In der Bachelorarbeit von Song [Song 2011] wurde das Werkzeug Skype-Solidifier entwickelt, welches zur automatischen Verfestigung und Indexierung von Skype-basierten Meetings genutzt werden kann. Dieses Werkzeug ist speziell für Meetings in verteilten Teams geeignet. Es wurde die Tatsache genutzt, dass in verteilten Meetings meist elektronische Medien für die Kommunikation genutzt werden. Diese können leichter zur (teil-)automatischen Dokumentationserstellung genutzt werden, als es z.B. bei einem Gespräch von Angesicht zu Angesicht möglich wäre.

Abbildung 6.17 zeigt die beiden Aktivitäten, die durch Skype-Solidifier unterstützt werden: (1) die automatische Dokumentation von Skype-basierten Meetings inklusive Indexerstellung aus den im Meeting gesendeten Sofortnachrichten und (2) die spätere Nutzung der Dokumentation. Darüber hinaus bietet Skype-Solidifier die Möglichkeit sowohl während der Aufzeichnung, als auch

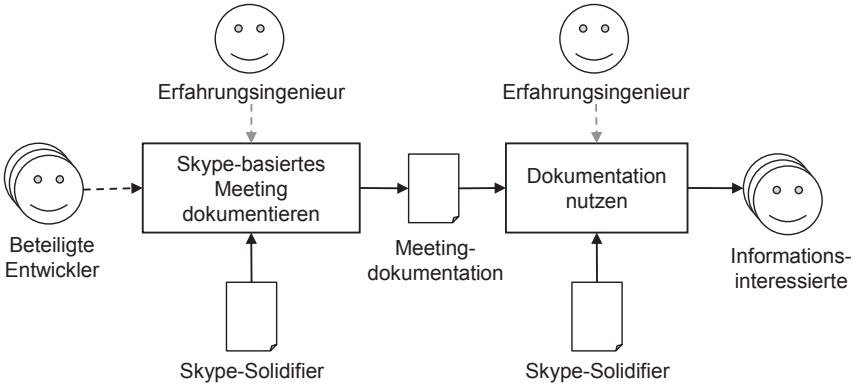


Abb. 6.17: Informationsflüsse in Skype-Solidifier, aus [Song 2011]

später bei der Nutzung, den Index zu bearbeiten. Dies kann laut [Schneider 2006] z.B. durch einen Erfahrungsingenieur geschehen.

6.3.2.3 Zusammenfassung und Einordnung in FLOW-Methode

Ziel der Technik der Dokumentation als Nebenprodukt ist die Verbesserung des Informationsflusses durch beiläufige Verfestigung flüssiger Informationsflüsse und zusätzlicher Strukturierung dieser verfestigten Informationen durch einen Index. Die Technik wird während eines Meetings oder anderen Kommunikationsaktivitäten eingesetzt. Es wurde ein Werkzeug geschaffen, dass die Verfestigungstechnik für Skype-Telefonkonferenzen umsetzt. Damit ist es möglich strukturierte Dokumentation als Nebenprodukt einer Telefonkonferenz zu erhalten. Die während einer Konferenz verschickten Textnachrichten werden automatisch zur Erstellung eines Index genutzt. Weiterhin kann mit Hilfe des Werkzeugs der Index während oder nach dem Meeting bearbeitet werden. Die Nützlichkeit des Index und damit die Nützlichkeit der Verfestigungstechnik wurde in einer Studie evaluiert. Es wurde gezeigt, dass Informationen mit Hilfe eines Index signifikant schneller gefunden werden können.

Tabelle 6.18 fasst die wesentlichen Eigenschaften der Technik Verfestigung als Nebenprodukt mit Hilfe des Templates aus Tabelle 6.4 zusammen.

Tabelle 6.18: Klassifikation der FLOW-Technik Verfestigung als Nebenprodukt

Name der Technik		Verfestigung als Nebenprodukt
Ziel	**Absicht**	□ Verstehen ⊠ Verbessern
	Zeit	□ vorher ⊠ während dessen □ nachher
	Umfang	⊠ Aktivität □ Projekt □ Organisation
Phase		□ Erheben □ Analysieren ⊠ Verbessern
Projektparameter		keine Einschränkungen
Phasenspezifische Aspekte		
Verbesserungs-strategie		⊗ Nebenprodukt ○ Hauptprodukt
		⊗ Leichtgewichtig ○ Schwergewichtig
Verbesserungs-verfahren		⊠ Aggregatzustand (⊗ verfestigen ○ verflüssigen)
		⊠ Fluss (○ Abkürzung ○ Umweg ⊗ Verzweigung ○ Zusammenführung)
		□ Aktivität (○ Schnittstellenanpassung ○ Aktivitätsanpassung)
		□ Musterersetzung

6.3.3 Integration von SCRUM in das V-Modell XT

Die Integrationstechnik beschreibt, wie SCRUM (vgl. [Schwaber 2002]) in auf V-Modell XT [VModell 2009] basierte Projekte integriert werden kann. Um geeignete Integrationspunkte zu finden und um herzuleiten, was bei einer Integration zu beachten ist, werden Informationsflüsse in SCRUM und im V-Modell betrachtet. Das Ziel der Technik ist es, Informationsflüsse in V-Modell-Projekten durch punktuelle Nutzung von SCRUM an geeigneten Stellen zu verbessern. Agile Vorteile wie ein frühzeitig lauffähiges Produkt, regelmäßiges Kundenfeedback, daraus resultierende Risikominimierung, und weniger sowie effizientere Dokumentation sollen dadurch auch V-Modell-Projekten zugänglich gemacht werden.

Abbildung 6.18 zeigt die Aktivitäten und Informationsflüsse der Integrationstechnik, wie sie Kiesling in seiner Masterarbeit entwickelt hat [Kiesling 2011]. Zunächst werden ausgehend von den Beschreibungen des V-Modells und von SCRUM Informationsflüsse bzw. Informationsflussschnittstellen abgeleitet. Diese werden im nächsten Schritt auf geeignete Integrationspunkte untersucht. Dabei ergeben sich Integrationsvarianten und Bedingungen, die es bei einer Integration zu beachten gilt. Die eigentliche Verbesserung findet im letzten Schritt durch die Integration von SCRUM in das V-Modell-XT-Projekt statt. Im Folgenden werden die drei Aktivitäten genauer beschrieben.

6.3.3.1 Phase 1: Informationsflüsse Ableiten

Die Grundidee der Integrationstechnik ist, die beiden Vorgehensmodelle der Softwareentwicklung V-Modell XT und SCRUM auf einen gemeinsamen Nenner zu bringen, um SCRUM-basierte Entwicklungsschritte in ein übergeordnetes V-Modell-Projekt integrieren zu können. Als gemeinsamer Nenner dienen Informationsflüsse. Nach den Grundannahmen von FLOW (vgl. Kapitel 6.1) eignen sich Informationsflüsse sowohl zur Analyse von Dokumentenzentrierten Entwicklungsprozessen als auch zur Analyse von agilen Entwicklungsmethoden.

Um auf einen gemeinsamen Nenner der beiden Ansätze zu kommen und damit die Grundlage für die Integration zu schaffen, werden in der ersten Phase der Integrationstechnik Informationsflüsse aus den Beschreibungen des V-Modells XT [VModell 2009] und von SCRUM [Schwaber 2002] hergeleitet.

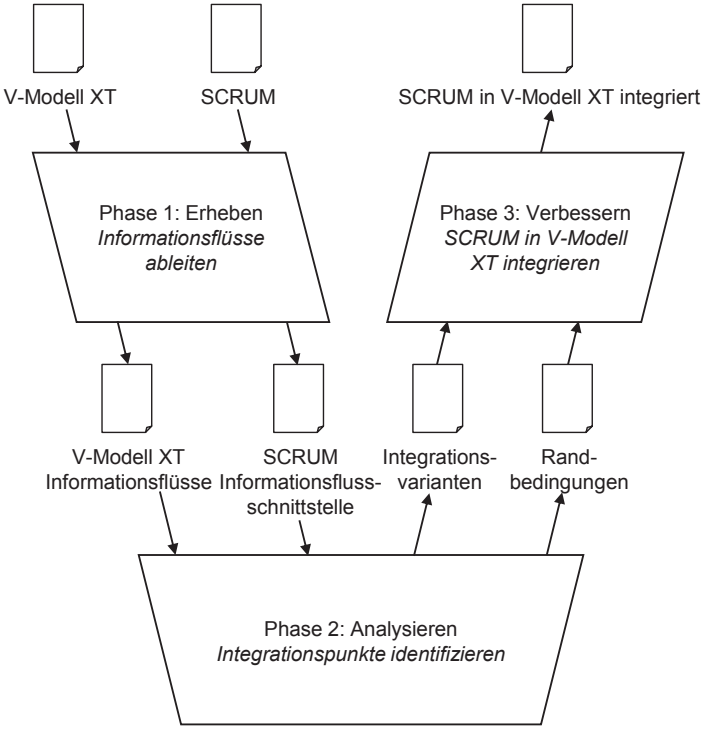


Abb. 6.18: Aktivitäten der SCRUM-Integration für alle drei Phasen des FLOW-Verbesserungsprozesses (in Anlehnung an [Kiesling 2011])

Informationsflüsse im V-Modell XT Der wichtigste Informationsträger im V-Modell sind Dokumente, oder Produkte in V-Modell-Terminologie. Produkte werden in Aktivitäten durch vorgegebene Rollen erzeugt. Jedem Produkt ist genau eine Aktivität zugeordnet, in der das Produkt erstellt wird. Neben dem Zusammenhang zwischen Produkt und Aktivität enthält das V-Modell noch Abhängigkeiten zwischen Produkten. Das V-Modell XT [VModell 2009] beschreibt vier verschiedene Arten von Produktabhängigkeiten [VModell 2009, Teil 1, Kapitel 4.5]:

Strukturelle Produktabhängigkeiten: Strukturelle Produktabhängigkeiten beschreiben die hierarchische Struktur des im V-Modell-Projekt zu ent-

wickelnden Systems. "Ein System kann dabei aus beliebig vielen ineinander geschachtelten Segmenten bestehen. Segmente, die nicht in weitere Segmente unterteilt sind, bestehen aus SW-Einheiten, HW-Einheiten und Externen Einheiten. Externe Einheiten sind nicht weiter untergliedert. SW-Einheiten und HW-Einheiten unterteilen sich in beliebig viele, ineinander geschachtelte SW- beziehungsweise HW-Komponenten. SW- und HW-Komponenten, die ihrerseits nicht weiter unterteilt sind, bestehen aus Produkten des Typs SW-Modul und HW-Modulen. Daneben gibt es auf HW- und SW-Ebene Produkte der Typen Externes HW-Modul bzw. Externes SW-Modul, die ebenfalls nicht weiter untergliedert sind." [VModell 2009, Teil 5, Kapitel 2.2]

Erzeugende Produktabhängigkeiten: "Eine erzeugende Produktabhängigkeit beschreibt, in welchen Produktexemplaren der Ausgangsprodukte die Bedingungen für die Erstellung von Produktexemplaren der Zielprodukte festgelegt werden." [VModell 2009, Teil 5, Kapitel 2.3]. Über die erzeugende Produktabhängigkeit wird somit auch der Inhalt des erzeugten Produkts festgelegt.

Inhaltliche Produktabhängigkeiten: Inhaltliche Produktabhängigkeiten beschreiben Konsistenzbeziehungen zwischen Produkten, d.h., wenn sich der Inhalt eines Produkts ändert, müssen die Inhalte aller von diesem Produkt inhaltlich abhängenden Produkte angepasst werden. (vgl. [VModell 2009, Teil 5, Kapitel 5])

Tailoring-Produktabhängigkeiten: "Tailoring-Produktabhängigkeiten beschreiben die für das Tailoring relevanten Relationen von Produkten zu Vorgehensbausteinen." [VModell 2009, Teil 3, Kapitel 3.8] Über diese Abhängigkeit werden beim Tailoring Vorgehensbausteine identifiziert, die bei der Erstellung bestimmter Produkte notwendig sind.

Für die Herleitung von Informationsflüssen zwischen den Produkten sind neben den Produkt- und Aktivitätsbeschreibungen der V-Modell-Dokumentation noch strukturelle und erzeugende Produktabhängigkeiten relevant. Die erzeugenden Produktabhängigkeiten beschreiben den Informationsfluss, der bei der initialen Erstellung eines Produkts stattfindet. Die strukturellen Produktabhängigkeiten beschreiben inhaltliche Zusammenhänge zwischen Produkten über Teil-Ganzes-Beziehungen. Z.B. besteht eine Software-Komponente (SW-Komponente) nach V-Modell aus mehreren Software-Modulen (SW-Modul). Damit sind die Informationen eines SW-Moduls auch in der übergeordneten

SW-Komponente enthalten. Inhaltliche Produktabhängigkeiten werden für die SCRUM-Integration nicht betrachtet, da diese nur die Änderungen von Teilen von Produkten während eines laufenden Projekts und nicht die initiale Erstellung der Produkte beschreiben. Neben Dokumenten sind auch Personen Informationsträger, die in V-Modell-Projekten wichtige Informationen liefern. Personen sind im V-Modell über verantwortliche und mitwirkende Rollen abgebildet. Für die Herleitung von Informationsflüssen werden hier nur die verantwortlichen Rollen betrachtet, da die mitwirkenden Rollen bei der Erstellung eines Produkts zwar einbezogen werden sollen, aber nicht genau festgelegt ist, ob sie tatsächlich Informationen liefern. Insgesamt lassen sich folgende Elemente des V-Modells zur Herleitung von Informationsflüssen nutzen:

- Aktivitätsbeschreibungen
- Produktbeschreibungen
- Erzeugende Produktabhängigkeiten
- Strukturelle Produktabhängigkeiten
- Verantwortliche Rollen

Abbildung 6.19 zeigt ein Beispiel einer Herleitung von Informationsflüssen aus dem V-Modell XT, bei dem nur Produkte und Aktivitäten der Softwareentwicklung berücksichtigt wurden.

Informationsflüsse in SCRUM Ziel der Integration von SCRUM in V-Modell-Projekte ist die Nutzung der agilen Vorteile von SCRUM. SCRUM soll daher bei einer Integration so wenig wie möglich verändert werden. Zur Herleitung der Informationsflüsse wird deshalb nur die Informationsfluss-Schnittstelle von SCRUM betrachtet. Die Informationsfluss-Schnittstelle einer Aktivität sind alle eingehenden und ausgehenden Informationsspeicher der Aktivität (vgl. [Schneider 2005]), d.h. alle Informationen, die zur Durchführung und Steuerung der Aktivität notwendig sind und alle Informationen, die in der Aktivität erstellt werden.

Mit Hilfe von SCRUM wird üblicherweise Software erstellt. Hier sollen die Prinzipien von SCRUM aber auch auf die Erstellung von anderen Dokumenten angewendet werden, sodass SCRUM für die Entwicklung beliebiger Produkte des V-Modells genutzt werden kann. Nach Schwaber und Beedle [Schwaber 2002] enthält das Product Backlog alle technischen und fachlichen An-

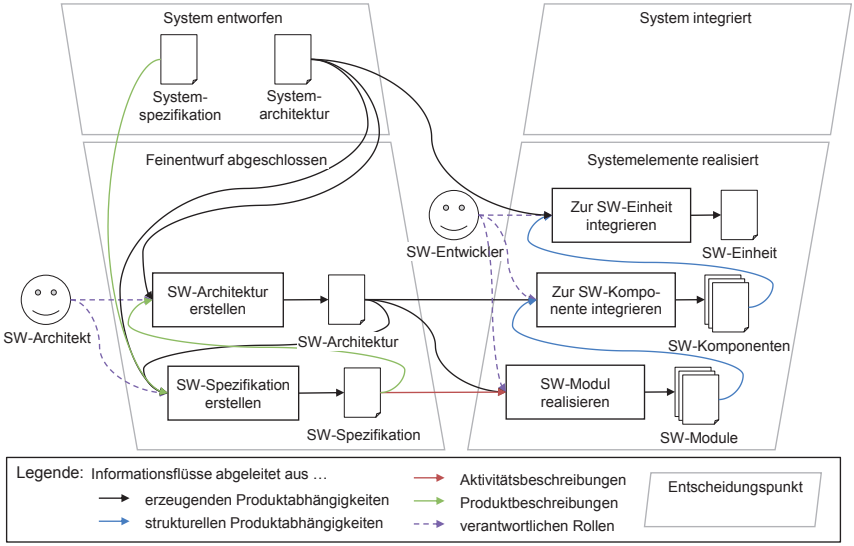


Abb. 6.19: Aus dem V-Modell XT abgeleitete Informationsflüsse inkl. Entscheidungspunkte am Beispiel der Softwareentwicklung

forderungen an das zu entwickelnde Produkt. Der so genannte Product Owner ist verantwortlich für die Erstellung, Pflege und Priorisierung des Product Backlogs. Für eine Integration müssen also zunächst die relevanten Anforderungen in das Product Backlog übernommen werden. Diese sollten, so weit wie möglich, aus den bis zum Zeitpunkt der Integration fertiggestellten Produkten übernommen werden. Der Product Owner priorisiert diese Anforderungen und fügt gegebenenfalls bisher noch nicht dokumentierte Anforderungen hinzu. Eine weitere für die SCRUM-Integration wichtige Rolle übernimmt der SCRUM Master. Ein SCRUM Master hat aus Informationsflusssicht eher steuernde Aufgaben. Er muss organisatorische und technische Probleme erkennen und helfen, diese auszuräumen. Er stellt die Schnittstelle zwischen SCRUM-Team und dem Management dar. Insgesamt ergibt sich die in Abbildung 6.20 dargestellte, für die Integration von SCRUM relevante Informationsfluss-Schnittstelle. Dabei sind die Quell-Produkte, der Product Owner und der SCRUM Master die Schnittstelle nach außen. Der eigentliche SCRUM (Abb. 6.20, innere Aktivität) kann bei Vorhandensein aller relevanten Informationen unverändert stattfinden. D.h. Sprint Planning, die Sprints

selbst und Sprint Reviews finden wie bei [Schwaber 2002] beschrieben statt.

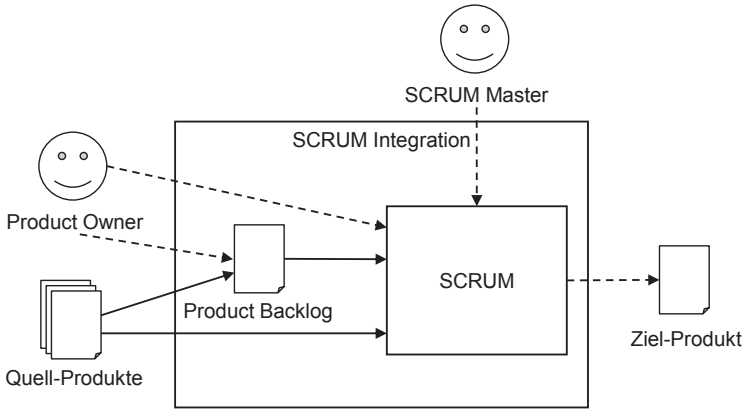


Abb. 6.20: Informationsfluss-Schnittstelle von SCRUM (angepasst aus [Kiesling 2011, Abb. 3.3])

Eine ausführlichere Beschreibung der Herleitung von Informationsflüssen aus dem V-Modell XT und aus SCRUM findet sich in der Masterarbeit von Kiesling [Kiesling 2011, Kapitel 3].

6.3.3.2 Phase 2: Integrationspunkte Identifizieren

Ergebnis der ersten Phase des FLOW-Verbesserungsprozesses sind Informationsflussmodelle der Informationsflüsse des V-Modells XT und der Informationsfluss-Schnittstelle von SCRUM. In der zweiten Phase werden die Informationsflüsse des V-Modells auf mögliche Integrationspunkte hin analysiert. "*Die Herausforderung hierbei ist, den im V-Modell vorhandenen Informationsfluss an der richtigen Stelle zu schneiden, so dass er zu dem Informationsfluss in SCRUM passt und somit ein Übergang möglich ist.*" [Kiesling 2011, S. 34] Dabei werden auch für die Integration wichtige Randbedingungen festgehalten (vgl. Abb. 6.18, Mitte rechts).

Abbildung 6.21 zeigt die prinzipielle Vorgehensweise bei der Integration von SCRUM. Zunächst wird im Informationsflussmodell des V-Modells ein Bereich ausgewählt (vgl. Abb. 6.21, links), der durch SCRUM ersetzt werden soll (vgl. Abb. 6.21, rechts). Durch die Ersetzung werden die Zwischen-Produkte

des V-Modell-Prozesses eingespart und stattdessen das Ziel-Produkt agil entwickelt.

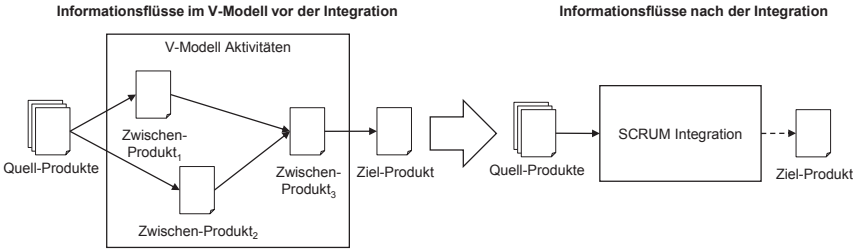


Abb. 6.21: Prinzipielle Vorgehensweise der SCRUM-Integration (angepasst aus [Kiesling 2011, Abb. 4.1])

Allerdings können nicht beliebige Produkte des V-Modells einfach weggelassen werden. Es können nur die Dokumente als Zwischen-Produkt weggelassen werden, die neben dem Ziel-Produkt keine weiteren inhaltlichen Abhängigkeiten zu Produkten außerhalb der SCRUM-Integration haben, d.h. die für die Erstellung von Produkten im weiteren Projektverlauf nicht weiter benötigt werden. Das bedeutet wiederum, dass, wenn man ein Ziel-Produkt und eine Menge von Zwischen-Produkten, die durch eine Integration eingespart werden sollen, identifiziert hat, alle von dieser Produktmenge abhängigen Produkte im SCRUM erstellt werden müssen. Eine Nachdokumentation ist zwar auch vorstellbar, würde aber einige durch die Integration gewonnene Vorteile wieder zunichtemachen.

Abbildung 6.22 zeigt ein Beispiel einer Integration, bei der die Produkte SW-Architektur und SW-Spezifikation eingespart werden können und mit Hilfe von SCRUM ausgehend von Systemarchitektur und Systemspezifikation direkt SW-Module und SW-Komponenten erstellt werden. Das Beispiel bezieht sich auf die in Abbildung 6.19 hergeleiteten Informationsflüsse.

Um den Verlauf von V-Modell-Projekten durch eine SCRUM-Integration nicht zu stark zu verändern, sollte eine Integration entlang von Entscheidungspunkten in Betracht gezogen werden. Entscheidungspunkte bieten sich als Schnittpunkt für eine Integration an, weil darin auf Basis fertig gestellter Produkte eine Fortschrittskontrolle stattfindet. Das Beispiel in Abbildung 6.22 zeigt eine Integration entlang von Entscheidungspunkt-Grenzen. Die im Entscheidungspunkt "Feinentwurf abgeschlossen" geforderten Dokumente SW-Architektur

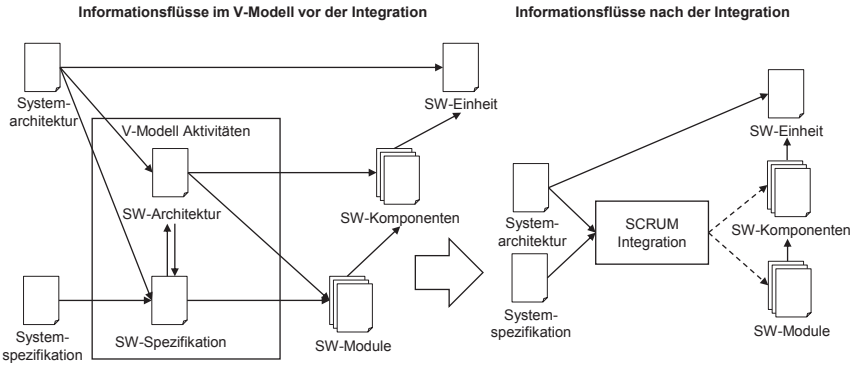


Abb. 6.22: SCRUM-Integration am Beispiel der Informationsflüsse aus Abbildung 6.19

und SW-Spezifikation werden durch SCRUM ersetzt, sodass dieser Entscheidungspunkt wegfallen kann (vgl. Abb. 6.19. Das V-Modell-Projekt geht mit Hilfe der SCRUM-Integration vom Entscheidungspunkt "System entworfen" zum Entscheidungspunkt "Systemelemente realisiert" über.

Eine Ausführlichere Betrachtung der Identifikation von Integrationspunkten und weiteren auf Entscheidungspunkten basierenden Integrationsvarianten findet sich in [Kiesling 2011, Kapitel 4 & 5]. Dort werden u.a. die folgenden Integrationsvarianten besprochen:

1. SCRUM für Realisierung, Integration und Test
2. SCRUM für die Systementwicklung
3. SCRUM für die Softwareentwicklung

Abbildung 6.23 zeigt schematisch, welche Entscheidungspunkte bei diesen drei Varianten der SCRUM-Integration wegfallen können.

6.3.3.3 Phase 3: SCRUM in V-Modell XT Integrieren

Nach der Analyse der Informationsflüsse und Identifikation geeigneter Integrationspunkte kann die eigentliche Integration vorgenommen werden. Durch

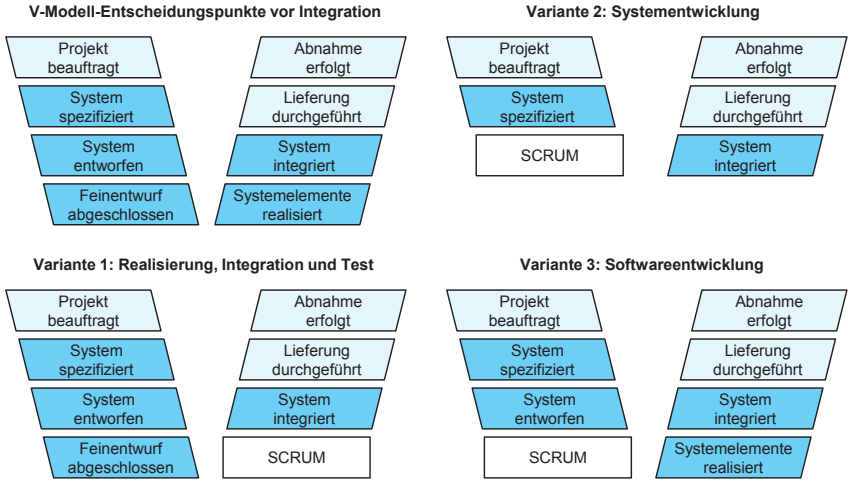


Abb. 6.23: Vergleich ausgewählter SCRUM-Integrationsvarianten aus [Kiesling 2011, Kapitel 5]

eine SCRUM-Integration werden aus festen flüssige Informationsflüsse. Es werden weniger Informationen in Dokumenten gespeichert. Das Ziel-Produkt kann inkrementell und im Vergleich zum V-Modell in kürzeren Iterationen erstellt werden. Dies ermöglicht frühes und häufiges Feedback, was wiederum zur Risikominimierung genutzt werden kann. D.h. die SCRUM-Integration ist besonders für unbekannte, risikoreiche Entwicklungsaufgaben geeignet.

Die Integration von SCRUM bringt viele Vorteile, aber auch einige Nachteile, die es zu berücksichtigen gilt. Folgende Vorteile und Nachteile ergeben sich zum Teil aus den FLOW-Verbesserungsverfahren Verflüssigung, Abkürzung und Zusammenfassung (vgl. Tabelle 6.12), oder direkt aus SCRUM (vgl. [Schwaber 2002]).

Vorteile:

- Risikominimierung durch kürzere Iterationen
- Schnellere erste Ergebnisse
- Schnelleres finales Ergebnis

- Aus Sicht des Kunden wertvollere Ergebnisse, da immer die hoch priorisierten Anforderungen zuerst umgesetzt werden
- Höhere Flexibilität bei Änderungen
- Weniger Dokumentationsaufwand

Nachteile:

- Getroffene Fehlentscheidungen sind durch agiles Vorgehen im Nachhinein schwerer nachvollziehbar, was insbesondere bei sicherheitskritischen Produkten problematisch ist
- Nachdokumentation kann notwendig werden
- Für Nachdokumentation notwendige Informationen können nach SCRUM evtl. bereits verloren sein
- Zusätzlicher Vorbereitungsaufwand, weil Product Backlog erst erstellt werden muss

6.3.3.4 Werkzeugunterstützung

Kiesling hat in seiner Masterarbeit [Kiesling 2011] ein Werkzeug entwickelt, welches automatisch aus der V-Modell XT Dokumentation Informationsflüsse ableitet. Damit können folgende im V-Modell beschriebene Abhängigkeiten automatisch in Informationsflüsse umgewandelt werden:

- Erzeugende Produktabhängigkeiten
- Strukturelle Produktabhängigkeiten
- Verantwortliche Rollen

Dazu parst das Werkzeug die HTML-Beschreibung des V-Modell XT[8]. Dies hat den Vorteil, dass mit Hilfe des V-Modell XT Projektassistenten das V-Modell zunächst für ein Projekt getailort werden kann, und anschließend die Informationsflüsse aus dem getailorten Modell extrahiert werden können. Weiterhin bietet das Werkzeug zur Informationsflussableitung die Möglichkeit die Menge der zu berücksichtigenden Produkte einzuschränken, sodass z.B. nur

[8] http://v-modell.iabg.de/v-modell-xt-html/

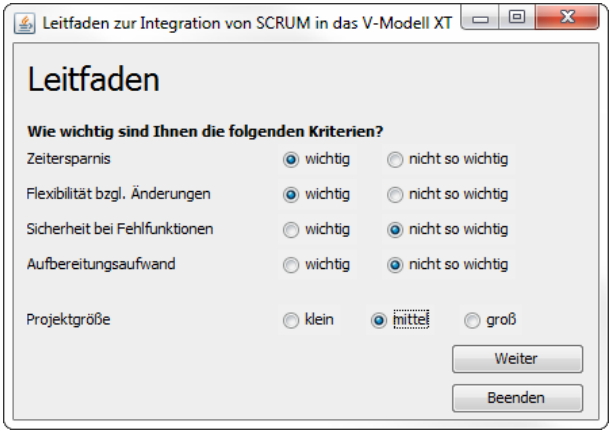


Abb. 6.24: Screenshot des Leitfadens zur SCRUM-Integration aus [Kiesling 2011, Kapitel 6.2]

die für die Softwareentwicklung relevanten Dokumente bei der Informationsflussherleitung berücksichtigt werden können.

Ein weiteres von Kiesling in seiner Masterarbeit entwickeltes Werkzeug zur Unterstützung der Integrationstechnik ist ein Leitfaden. Der Leitfaden hilft einem Projektleiter, der eine SCRUM-Integration durchführen möchte, ausgehend von Zielen, die durch eine Integration erreicht werden sollen, geeignete Integrationsvarianten zu wählen. Abbildung 6.24 zeigt die Eingabeparameter des Leitfadens. Die vier Parameter und ihr Einfluss auf die Wahl sollen hier kurz erläutert werden. Eine ausführlichere Beschreibung findet sich in [Kiesling 2011].

Zeitersparnis: Wenn Zeitersparnis ein wichtiger Faktor ist, dann sollten so viele Produkte wie möglich bei der Integration von SCRUM eingespart werden.

Flexibilität: Wenn es wichtig ist, flexibel auf Anforderungsänderungen reagieren zu können, dann sollte die Systementwicklung oder zumindest die Softwareentwicklung mit SCRUM durchgeführt werden.

Sicherheit: Wenn das zu entwickelnde System sicherheitskritisch ist, d.h. Nachverfolgbarkeit von Entscheidungen wichtig ist, dann sollten nur risiko-

reiche Teile der Systementwicklung prototypisch mit SCRUM umgesetzt werden. Kritische Systemteile sollten weiterhin nur mit V-Modell entwickelt werden, oder es sollte während des SCRUM die notwendige Dokumentation parallel erstellt werden.

Vorbereitungsaufwand: Wenn zur Vorbereitung einer Integration nur wenig Zeit verfügbar ist, dann eignen sich diejenigen Integrationsvarianten besser, bei denen nur ein geringer Vorbereitungsaufwand, z.B. für die initiale Erstellung des Product Backlog, erforderlich ist.

6.3.3.5 Zusammenfassung und Einordnung in die FLOW-Methode

Die Integrationstechnik zeigt, dass sich der FLOW-Ansatz eignet, die beiden Welten der agilen und Dokumenten-zentrierten Softwareentwicklung aus einem einheitlichen Blickwinkel zu betrachten und die jeweiligen Vorteile gezielt zu vereinen. Durch Erhebung, Analyse und schließlich Verbesserung der Informationsflüsse in V-Modell-Projekten bietet die Technik Unterstützung für alle drei Phasen der FLOW-Methode.

Zusammenfassend lässt sich das allgemeine Vorgehen zur Integration von SCRUM in das V-Modell XT wie folgt beschreiben:

Phase 1: Informationsflüsse herleiten:

1. Alle V-Modell-Informationsflüsse für einen getailorten V-Modell-Prozess herleiten.

Phase 2: Integrationspunkte identifizieren:

2. A: Freie Integration:
   a) Das mit SCRUM zu erstellende Ziel-Produkt festlegen.
   b) Gewünschte Quell-Produkte festlegen.
2. B: Integration entlang von Entscheidungspunkten:
   a) Den mit SCRUM zu erreichenden Ziel-Entscheidungspunkt festlegen
   b) Gewünschten Start-Entscheidungspunkt festlegen
3. Im V-Modell-Informationsfluss alle Zwischenprodukte zwischen Quell- und Ziel-Produkten identifizieren.

4. Alle Produkte identifizieren, die von Zwischenprodukten abhängig sind und weder selbst Zwischenprodukt noch Ziel-Produkt sind. Diese Produkte werden zu weiteren Ziel-Produkten der SCRUM-Integration.

Phase 3: SCRUM integrieren:

5. Vorteile gegen Nachteile abwägen.
6. Entsprechend der gewählten Integrationsvariante integrieren.

Tabelle 6.19 zeigt die Einordnung der Integrationstechnik in die FLOW-Methode mit Hilfe des Templates aus Tabelle 6.4.

Tabelle 6.19: Klassifikation der Integration von SCRUM in das V-Modell XT als FLOW-Technik

Name der Technik		Integration von SCRUM in das V-Modell XT
Ziel	**Absicht**	□ Verstehen ⊠ Verbessern
	Zeit	⊠ vorher ⊠ während dessen □ nachher
	Umfang	□ Aktivität ⊠ Projekt □ Organisation
Phase		⊠ Erheben ⊠ Analysieren ⊠ Verbessern
Projektparameter		
Vorgehensmodell		⊠ Prozess: V-Modell XT ⊠ Agil: SCRUM
Verteiltheit		⊠ Lokal □ Verteilt
Phasenspezifische Aspekte		
Erhebungsstrategie		⊠ Bottom-Up ⊠ Top-Down
Erhebungsverfahren		□ Selber machen □ Beobachten □ Interview □ Fragebogen ⊠ Modellableitung □ Kommunikationsereignisableitung
Analysestrategie		⊠ manuelle Analyse ⊠ teilautomatische Analyse ⊠ automatische Analyse
Analyseverfahren		⊠ Visualisierung □ Simulation ⊠ Mustersuche
Verbesserungs-strategie		○ Nebenprodukt ○ Hauptprodukt ○ Leichtgewichtig ⊗ Schwergewichtig
Verbesserungs-verfahren		⊠ Aggregatzustand (○ verfestigen ⊗ verflüssigen) ⊠ Fluss (⊗ Abkürzung ○ Umweg ○ Verzweigung ⊗ Zusammenführung) ⊠ Aktivität (○ Schnittstellenanpassung ⊗ Aktivitätsanpassung) ⊠ Musterersetzung

7 Zusammenfassung

Die vorliegende Arbeit liefert einen neuen Kandidaten der häufig geforderten Theorien im Software Engineering (vgl. u.a. [Wohlin 2000, Juristo 2001, Kruchten 2002, Herbsleb 2003, Easterbrook 2008, Jacobson 2009a, Cockburn 2010, Broy 2011]). Die vorgestellte Theorie charakterisiert Softwareentwicklung als Informationsfluss. So können der Mensch und die Interaktionen zwischen den Menschen – insbesondere die Kommunikation – als charakteristische Eigenschaften der Softwareentwicklung in den Vordergrund gestellt werden. Die Informationsflusstheorie kann in der Software-Engineering-Forschung helfen, typische Phänomene zu erklären und besser zu verstehen. Weiterhin liefert die Arbeit eine auf der Theorie basierende Methode zur praktischen Verbesserung der Softwareentwicklung. Sowohl für den theoretischen als auch für den praktischen Teil dieser Arbeit werden empirische Belege präsentiert.

Die wesentlichen Beiträge der vorliegenden Arbeit für Wissenschaft und Praxis sind die folgenden:

Theorie: Präzise, zusammenhängende Terminologie zur grundlegenden Beschreibung der Softwareentwicklung (vgl. Kapitel 3); daraus abgeleitete Sätze und Hypothesen zur Erklärung und Vorhersage typischer Softwareentwicklungsphänomene (vgl. Kapitel 4). Prüfung der Informationsflusstheorie durch Vergleich mit verwandten Theorien und empirischer Validierung (vgl. Kapitel 5).

Praxis: Auf der Theorie aufbauende Methode zur praktischen Softwareprozessverbesserung und empirische Evaluation der Nützlichkeit zweier Verbesserungstechniken (vgl. Kapitel 6).

Im Folgenden werden die Beiträge der Arbeit ausführlicher diskutiert.

7.1 Diskussion

Dieser Abschnitt fasst den Beitrag dieser Arbeit zusammen und diskutiert, in wie weit die in der Einleitung genannten Ziele erreicht werden (vgl. Kapitel 1.2.3).

Das Ziel der Arbeit war die Entwicklung einer grundlegenden Theorie der Softwareentwicklung, die Informationsflüsse als zentrales Element enthält. Dazu wurde die Idee, Softwareentwicklung aus der Perspektive von Informationsflüssen zu betrachten, von Schneider (vgl. u.a. [Schneider 2004, Schneider 2005, Schneider 2005a]) übernommen und sowohl für die Theorie (vgl. Kapitel 3 und 4) als auch für den FLOW-Ansatz (vgl. Kapitel 6) theoretisch fundiert.

Die Informationsflusstheorie soll – soweit möglich und sinnvoll – auf bestehenden Erkenntnissen aufbauen und nicht im Widerspruch mit etabliertem Software-Engineering-Wissen stehen. Zudem soll die Informationsflusstheorie sowohl für die Forschung als auch für die Praxis nützlich sein. Im Folgenden werden zunächst die Beiträge für die Wissenschaft, anschließend die Beiträge für die Praxis diskutiert.

7.1.1 Beitrag für die Wissenschaft

Die Informationsflusstheorie besteht aus zwei Teilen.

1. In Kapitel 3 wird eine Terminologie definiert, mit der grundlegende und charakteristische Zusammenhänge in der Softwareentwicklung beschrieben werden können.

2. Mit Hilfe der grundlegenden Definitionen werden in Kapitel 4 Sätze und Hypothesen hergeleitet, die zur Analyse und Vorhersage von Phänomenen aus der Softwareentwicklung genutzt werden können.

In Kapitel 5 wird die Informationsflusstheorie einer umfassenden Prüfung unterzogen. Im Folgenden werden Ergebnisse dieser Prüfung anhand der allgemeinen Qualitätskriterien einer Theorie, der SE-spezifischen Kriterien und der selbst gesteckten Ziele präsentiert und diskutiert.

Allgemeine Qualitätskriterien einer Theorie nach Popper [Popper 2002, S. 7-8]:

Prüfung des wissenschaftlichen Fortschritts: In Kapitel 5.1 wird die Informationsflusstheorie mit zehn verwandten Theorien verglichen. Der Vergleich ergibt, dass die Informationsflusstheorie grundlegender und umfangreicher als die meisten anderen Theorien des Software Engineering ist und nicht im Widerspruch mit etabliertem Software-Engineering-Wissen steht. Darüber hinaus wird gezeigt, dass mit Hilfe der Informationsflusstheorie neue Erkenntnisse hergeleitet werden können, z.B. die Erkenntnis, dass es gut für die Softwareentwicklung ist, wenn hierarchische Organisationsstrukturen entsprechend Conway's Law in Softwarearchitekturen übernommen werden, da sie zu linearer Abstimmungskomplexität führen können (vgl. S. 224). Die Informationsflusstheorie stellt daher auch einen wissenschaftlichen Fortschritt dar.

Empirische Anwendung: In Kapitel 5.2 wird eine eigene empirische Studie zur Prüfung von sechs Hypothesen der Informationsflusstheorie präsentiert. Für vier Hypothesen der Informationsflusstheorie werden statistisch signifikante Belege geliefert. Auch der von Anderen (vgl. u.a. [Curtis 1988, Herbsleb 2003a, Silva 2010]) oft berichtete große Einfluss von Kommunikation in der Implementierungsphase auf den Softwareprojekterfolg wird in der Arbeit durch die eigene empirische Studie nachgewiesen. Zudem werden durch das verwendete Sudiendesign gezielt Vorteile von fallstudienbasierten und experimentellen Untersuchungen genutzt (vgl. Kapitel 5.2.2).

SE-spezifische Kriterien (vgl. S. 195):

Präzise Terminologie: Eine präzise Terminologie ist wichtig für die Entwicklung von Verbesserungsmethoden (vgl. Kapitel 6) und für die Entwicklung und Auswertung von empirischen Studien. Die in Kapitel 5.2 vorgestellte Studie basiert auf den Definitionen aus Kapitel 3 und den Hypothesen aus Kapitel 4. Es werden testbare Hypothesen und Metriken abgeleitet. Die Ergebnisse werden in Bezug auf die testbaren Hypothesen und die dahinter liegende Informationsflusstheorie interpretiert. Zudem ermöglichen die Definitionen und Theoreme den Entwurf weiterer Studien zur Überprüfung der Informationsflusstheorie. Die Interpretation von Software-Engineering-Studien aus der Litertatur (vgl. Kapitel 5.3) ergibt, dass mit Hilfe der Definitionen und Theoreme der Informationsflusstheorie empirische Ergebnisse erklärt werden können. Damit kann die Informationsflusstheorie auch als Ausgangspunkt zur Kumulation der Ergebnisse empirischer Studien genutzt werden.

Bereiche Management und Mensch: Nach Boehm sollte eine Software-Engineering-Theorie die Bereiche Informatik, Management sowie persönliche, kulturelle und ökonomische Werte, die bei der Entwicklung von Software eine Rolle spielen, abdecken [Boehm 2006]. Die Sicht auf Softwareentwicklung als Informationsfluss (vgl. Def. 3.41) oder die Klassifikation der möglichen Wissensunterschiede zwischen den verschiedenen Rollen eines Softwareprojekts (vgl. Kapitel 3.6.1 und 3.7.4) gibt dem Management von Softwareprojekten neue Ansatzpunkte zur Verbesserung (vgl. u.a. Kapitel 6.3). Der Mensch spielt in der Informationsflusstheorie eine zentrale Rolle, ohne den keine Softwareentwicklung möglich ist. Dies zeigt sich z.B. an der Definition von Wissen (vgl. Def. 3.2) und der Definition und dem großen Stellenwert von Kommunikation (vgl. Def. 3.24 sowie die Kapitel 3.6 und 4.2). Ökonomische Betrachtungen werden z.B. durch den Effizienzanteil an Kommunikations- und Softwareentwicklungserfolg (vgl. Defs. 3.38 und 3.42) und den Theoremen zur Problemgröße (vgl. Kapitel 4.4) ermöglicht.

Methodenunabhängigkeit: Nach Jacobson und Meyer sollte eine Software-Engineering-Theorie Probleme und nicht deren Lösung beschreiben [Jacobson 2009]. Die Neu-Interpretation der Ergebnisse von Studien aus der SE-Literatur in Kapitel 5.3 zeigt, dass die Fokussierung auf Informationsflüsse – insbesondere die Klassifikation von Wissensunterschieden (vgl. Kapitel 3.6.1) und ihr Einfluss auf den Kommunikationserfolg (vgl. Korollare 4.1 und 4.3) – es ermöglicht, die Ursachen von Problemen zu erklären. Damit ist die Informationsflusstheorie methodenunabhängig.

Fundamentale Sätze der Softwareentwicklung: Wie in Kapitel 4 beschrieben, liefert die Informationsflusstheorie auch Sätze, die grundlegende Prinzipien der Softwareentwicklung beschreiben, z.B. den Zusammenhang von Anzahl an Kommunikationsteilnehmern und Wissensunterschieden (vgl. Satz 4.2), die Schwierigkeit der Kommunikation zwischen Fachleuten und Softwareentwicklern (vgl. Korollar 4.5), die Auswirkungen der Medienwahl auf den Kommunikationserfolg (vgl. Satz 4.7) oder den Zusammenhang zwischen Problemkomplexität und Abstimmungskomplexität (vgl. Satz 4.10).

Eigene Ziele für die Software-Engineering-Forschung (vgl. Kapitel 1.2.3):

Auf bestehenden Erkenntnissen aufbauen: Die in Kapitel 2 beschriebenen Grundlagen und Erkenntnisse aus der Software-Engineering-Literatur

(vgl. Einzelnachweise in Kapitel 3) werden zur Herleitung und Beschreibung der Definitionen in Kapitel 3 genutzt. Die Informationsflusstheorie erklärt und formalisiert bereits vorhandene Ansätze und bringt sie in einen Gesamtzusammenhang, so dass sie zur Beschreibung von grundlegenden Phänomenen der Softwareentwicklung genutzt werden können. Der Vergleich mit verwandten Theorien in Kapitel 5.1 und die Neu-Interpretation von Ergebnissen anderer empirischer Studien in Kapitel 5.3 zeigen, dass dies gelungen ist.

Kein Widerspruch zu etablierten Erkenntnissen: Der Vergleich mit verwandten Theorien des Software Engineering in Kapitel 5.1 zeigt, dass die Informationsflusstheorie nicht im Widerspruch mit den Aussagen der zehn untersuchten Theorien steht (vgl. Tab. 5.1).

SE-Forschung – Probleme erklären: Aus dem Vergleich mit Studien aus der Literatur in Kapitel 5.3 ergibt sich, dass viele Probleme mit Hilfe der Informationsflusstheorie erklärt werden können.

SE-Forschung – Probleme vorhersagen: Die Theoreme aus Kapitel 4 können genutzt werden, um in konkreten Entwicklungssituationen Vorhersagen über den Verlauf machen zu können. Z.B. können mit den Hypothesen zur Medienwahl (vgl. Kapitel 4.3) Vorhersagen über den Kommunikationserfolg basierend auf Eigenschaften der Kommunikationsteilnehmer und der genutzten Kommunikationsmedien gemacht werden.

SE-Forschung – Verbesserungsmethoden entwickeln: Dass die Informationsflusstheorie zur Entwicklung von Verbesserungsmethoden genutzt werden kann, wird beispielhaft mit der FLOW-Methode und den FLOW-Techniken in den Kapiteln 6.2 und 6.3 gezeigt und empirisch belegt.

Zur Verdeutlichung der obigen Aussage, dass die Informationsflusstheorie bestehende SE-Erkenntnisse in einen Gesamtzusammenhang bringt, werden im Folgenden einige typische Charakterisierungen von Softwareentwicklung aufgelistet und anschließend mit der Informationsflusssicht in Verbindung gebracht.

Softwareentwicklung und Dokumentation: Für Bjørner [Bjoerner 2006-1], Ludewig und Lichter [Ludewig 2010] ist Dokumentation die wesentliche Eigenschaft der Softwareentwicklung:

> *There is nothing else emanating from steps, stages and phases* [in software development] *than documents* [Bjoerner 2006-1, S. 14]

> *Damit lassen sich Software-Entwickeln und Dokumentieren nicht trennen, das Resultat der Entwicklung* ist *die Dokumentation.* [Ludewig 2010, S. 259]

Softwareentwicklung und Lernen: Für Armour ist Softwareentwicklung Wissenserwerb, d.h. Lernen (vgl. Def. 3.26) in Informationsflussterminologie:

> *Software development is knowledge acquisition.* [Armour 2000, Armour 2003]

Auch für Cockburn ist Wissenserwerb ein wichtiger Faktor der Softwareentwicklung (vgl. Kapitel 5.1.4) [Cockburn 2009].

Softwareentwicklung und Kommunikation: Cockburn beschreibt Softwareentwicklung als kooperatives Spiel, bei dem die Entwickler kommunizieren müssen, um erfolgreich zu sein [Cockburn 2007]. Kommunikation wird von vielen Softwareingenieuren als essenzieller Teil der Softwareentwicklung angesehen (vgl u.a. [Herbsleb 2003, Brooks 1974, Curtis 1988, Cataldo 2008, Aranda 2010]).

Dokumentation, Lernen und Kommunikation sind also wichtige Bestandteile der Softwareentwicklung. Die oben genannten Charakterisierungen beschreiben teilweise ähnliche Dinge, nur aus einem anderen Blickwinkel. Die Definition von Softwareentwicklung als Informationsfluss (vgl. Def. 3.41) bringt diese Sichten in einen Gesamtzusammenhang. Das ist ein weiterer Beleg dafür, dass die Informationsflusstheorie eine grundlegende Software-Engineering-Theorie ist.

7.1.2 Beitrag für die Praxis

Der in Kapitel 6 vorgestellte FLOW-Ansatz bietet eine spezielle Informationsflusssicht, eine Methode und spezielle Techniken, die zur praktischen Verbesserung von Softwareentwicklungsprojekten eingesetzt werden können.

Die in FLOW benutzte Metapher des Aggregatzustands von Informationen hilft, diejenigen Eigenschaften von Informationen in verschiedenen Informationsspeichern gezielt zu identifizieren, die großen Einfluss auf den Softwareentwicklungserfolg haben, und macht damit die Informationsflusstheorie praktisch nutzbar (vgl. Kapitel 6.1). Die grafische FLOW-Notation ist dabei ein

Hilfsmittel, welches die Analyse und Verbesserung von Softwareentwicklungsprojekten durch die Visualisierung von Informationsflüssen erleichtert.

Die FLOW-Methode (vgl. Kapitel 6.2) leitet einen Praktiker bei der Informationsfluss-basierten Verbesserung der Softwareentwicklung an. Die FLOW-Methode beschreibt, wie bei einem FLOW-Verbesserungsvorhaben geeignete FLOW-Techniken für die Umsetzung gewählt werden. Zudem ist die FLOW-Methode ein Framework zur Erstellung von neuen FLOW-Techniken.

Mit drei ausgewählten FLOW-Techniken (vgl. Kapitel 6.3), von denen zwei empirisch geprüft werden, wird gezeigt, dass die FLOW-Methode anwendbar und nützlich ist. Die FLOW-Mapping-Technik kann zur Planung und Steuerung von Kommunikation in verteilten Softwareentwicklungsprojekten eingesetzt werden (vgl. Kapitel 6.3.1). Die FLOW-Technik zur Verfestigung von Information als Nebenprodukt hilft, nützliche Dokumentation von verteilten Meetings ohne viel Zusatzaufwand zu schaffen (vgl. Kapitel 6.3.2). Die SCRUM-Integrations-Technik kann helfen, agile Elemente in ansonsten unflexible V-Modell-XT-Projekte zu integrieren, und so z.B. Entwicklungsrisiken zu minimieren (vgl. Kapitel 6.3.3).

Damit ist das in der Einleitung gesteckte Ziel erreicht (vgl. Kapitel 1.2.3), eine Verbesserungsmethode für die Software-Engineering-Praxis zu entwickeln, die Softwareentwicklung tatsächlich verbessern kann.

Zusammenfassend kann festgestellt werden, dass die Informationsflusstheorie alle gesteckten Ziele erfüllt und somit eine valide Theorie der Softwareentwicklung darstellt.

7.2 Ausblick

Da in dieser Arbeit eine grundlegende Theorie für die Softwareentwicklung geschaffen wurde, bieten sich vielfältige Anknüpfpunkte.

Zum Einen kann die Theorie selbst noch ausgebaut werden. Auf Basis der Definitionen in Kapitel 3 können weitere Sätze und Hypothesen hergeleitet werden. So sind z.B. weitere Theoreme für die Informationsflusstypen Dokumentation und Lernen vorstellbar (vgl. Kapitel 3.5). Eine andere Möglichkeit ist die Herleitung von neuen Theoremen für spezielle Anwendungsbereiche der Softwareentwicklung. Vor allem im Bereich der global verteilten Softwareentwicklung können weitere Theoreme der Informationsflusstheorie einen wertvollen Beitrag liefern, z.B. Hypothesen und Sätze über Zusammenhänge von Wissensunterschieden auf Kulturebene und Kommunikations- oder Softwareentwicklungserfolg.

Zum Anderen muss die Informationsflusstheorie natürlich noch weiter evaluiert werden. Das kann durch weitere empirische Studien oder indirekt durch Anwendung und Evaluation der FLOW-Methode geschehen. Z.B. sind weitere Studien im Bereich der Medienwahl denkbar. Beispielsweise kann geprüft werden, ob die auf Basis der Theoreme in Kapitel 4.3 gewählten Medien die Kommunikation in global verteilter Softwareentwicklung verbessern und ob die Vorschläge der Informationsflusstheorie zu einem größeren Kommunikationserfolg führen als bei der Wahl von Kommunikationsmedien entsprechend den Vorgaben einer anderen Theorie, z.B. der Media Synchronicity Theory [Dennis 2008]. Für eine solche Studie wurden bereits Vorarbeiten geleistet. Nünke hat in seiner Bachelorarbeit ein Werkzeug entwickelt, das basierend auf Eingaben, die eine Kommunikationssituation beschreiben, Vorschläge zur Medienwahl entsprechend Media Synchronicity Theory und Informationsflusstheorie generiert [Nuenke 2011].

Auch die FLOW-Methode kann weiterentwickelt werden. Es können neue FLOW-Techniken erstellt oder bestehende verbessert werden. Z.B. könnte die in Kapitel 6.3.3 beschriebene FLOW-Technik zur Integration von SCRUM in V-Modell-XT-Projekte in realen Projekten validiert werden.

Viele weitere Einsatzbereiche der Informationsflusstheorie sind vorstellbar. So könnte die Theorie bei der Bearbeitung existierender Forschungsfragen im Bereich der Softwareentwicklung hilfreich sein. Oder sie wird als Grundlage für

die Erarbeitung weiterer Methoden und Techniken zur Verbesserung der Softwareentwicklung genutzt. Schließlich könnte mit Hilfe der Informationsflusstheorie der von Jacobson und Meyer geforderte Vergleich existierender Methoden durchgeführt werden [Jacobson 2009].

> *All methods* [. . .] *share many properties. Using the theory as a basis, we should describe the properties that any method should satisfy if it is going to be effective for developing quality software.* [. . .] *By doing this we can clearly see the real differences between different methods* [. . .] [and we can] *provide the industry with what it is entitled to expect from the experts: scientifically sound, practically useful methods and tools.* [Jacobson 2009]

Die Informationsflusstheorie könnte also helfen, *"die stabile Grundlage zu entwickeln, die die Softwareentwicklungsindustrie benötigt."* [Jacobson 2009]

A Fragebögen

A.1 Fragebogen Projekterfolg

Fragebogen zur Bestimmung des Projekterfolgs im SWP

Definition Projekterfolg:

JA — Die Software kann wie beabsichtigt verwendet werden. Die Abnahme war erfolgreich.

JA- — Die Abnahme war erfolgreich. Der Kunde glaubt, dass er die Software nach einem Monat Nacharbeit (unter SWP-Bedingungen) wie beabsichtigt verwenden würde.

NEIN++ — Projekte dieser Kategorie verfehlten die Ziele des Kunden. Zwar wurde die Quality Gate 3 im ersten Versuch bestanden aber die Abnahme war nicht erfolgreich. Der Kunde glaubt nicht, dass er die Software nach einem Monat Nacharbeit (unter SWP-Bedingungen) wie beabsichtigt verwenden könnte.

NEIN+ — Projekte dieser Kategorie verfehlten die Ziele des Kunden. Zwar wurden das Quality Gate 3 mit Nacharbeit bestanden, aber die Abnahme war nicht erfolgreich. Der Kunde glaubt nicht, dass diese Ziele mit Nacharbeit innerhalb eines Monats (unter SWP-Bedingungen) erreichbar sind.

NEIN — Projekte dieser Kategorie verfehlten die Ziele des Kunden. Das Quality Gate 3 wurde endgültig nicht bestanden und die Abnahme war nicht erfolgreich.

Fragen an den Kunden zum Projekterfolg:

War die Abnahme erfolgreich?

- ☐ ja: Verwenden Sie die Software wie beabsichtigt?
  - ☐ ja → JA
  - ☐ nein, aber nach einem Monat Nacharbeit (unter SWP-Bed.) würde ich das → JA-
  - ☐ nein und daran würde auch ein Monat Nacharbeit nichts ändern: Warum nicht?
    - ☐ die Ziele wurden nicht ausreichend umgesetzt → weiter m. QG-Frage
    - ☐ es gibt eine bessere Software: Wenn es die bessere Software nicht gäbe, könnten Sie die Software sofort wie beabsichtigt verwenden?
      - ☐ ja → JA
      - ☐ nein, aber mit einem Monat Nacharbeit → JA-
      - ☐ nein, die Ziele wurden nicht ausreichend umgesetzt → weiter m. QG-Frage
    - ☐ anderer Grund: _____
- ☐ nein: "Wie" wurde das Quality Gate 3 bestanden?
  - ☐ im ersten Versuch → NEIN++
  - ☐ im zweiten Versuch → NEIN+
  - ☐ gar nicht → NEIN

A.2 Fragebogen verteiltes XP Programmierprojekt

Fragebogen für Entwickler

- täglich, nach der Arbeit auszufüllen -

Allgemein

Team **Global** ☐ TUC ☐ LUH	Datum: ___.___.2010	Name: _____

Die Angaben aus diesem Fragebogen werden vertraulich behandelt und nur anonymisiert ausgewertet.

1. Stimmung

1.1 Auf einer Noten-Skala von 1 (sehr wohl) bis 5 (unwohl), wie **wohl** haben Sie sich heute im Team **gefühlt**?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
sehr wohl unwohl

1.2 Auf einer Noten-Skala von 1 (sehr gut) bis 5 (mangelhaft), wie beurteilen Sie die heutige **Stimmung** im Team?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
sehr gut mangelhaft

1.3 Auf einer Noten-Skala von 1 (sehr viel) bis 5 (gar keinen Spaß), wie viel **Spaß** hat Ihnen heute die Arbeit bereitet?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
sehr viel gar keinen

2. Produktivität

2.1 Wie viele Story Cards, die Sie bearbeitet haben, wurden heute vom Kunden abgenommen?

2.2 Wie viele Gummibärchen haben Sie nach eigener Einschätzung zusammen mit Ihrem Pair-Partner(n) heute etwa geschafft?

2.3 Auf einer Noten-Skala von 1 (sehr gut) bis 5 (mangelhaft), wie würden Sie Ihre heutige Produktivität beurteilen?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
sehr gut mangelhaft

Zu 2.3: Wenn Sie mit 4 oder 5 geantwortet haben: Woran lag es?

3. eXtreme Programming

3.1. Haben Sie heute den „XP Programmer Guide“ aus dem Wiki benutzt bzw. angesehen?

☐ ja ☐ nein

3.2. Haben Sie heute durch das Pair Programming etwas von Ihrem Pair-Partner gelernt?

☐ ja ☐ nein

3.3. Wie oft haben Sie heute einen Test vor der Implementierung geschrieben?

☐ immer ☐ meist ☐ ab und zu ☐ gar nicht

3.4. Wie oft haben Sie heute ein Refactoring durchgeführt?

☐ oft ☐ ab und zu ☐ einmal ☐ gar nicht

3.5. Was denken Sie, wieviel Prozent des gesamten Quellcodes kennen Sie (Stand heute, gesamtes Projekt bis heute)? Mit kennen ist gemeint: Haben Sie schonmal dran gearbeitet oder Sie wissen, wo was zu finden ist und was der Quellcode macht.

☐ <25% ☐ 25%-50% ☐ 50%-75%
☐ 75%-99% ☐ 100%

4. Kommunikation

4.1. Haben Sie heute den „XP Communication Guide“ (inkl. der FLOW-Map) aus dem Wiki benutzt bzw. bewusst angesehen?

☐ ja ☐ nein

4.2. Empfanden Sie heute die **FLOW-Map** als hilfreich?

☐ ja, sehr ☐ ja, ein wenig ☐ nein ☐ N/A

4.3. War die **FLOW-Map** heute immer aktuell?

☐ ja ☐ nein ☐ N/A: _____

4.4. Wussten Sie heute immer, an was (z.B. welcher Story Card) die Kollegen am anderen Standort aktuell arbeiten?

☐ ja, immer ☐ ja, meist ☐ nein

4.5. Wussten Sie heute bei einem Problem (bzw. bei einer Frage), das nicht im lokalen Team gelöst werden konnte, wen Sie am anderen Standort dazu fragen könnten?

☐ ja ☐ nein ☐ kein Problem bzw. Frage

Wenn es ein Problem/eine Frage gab, haben Sie dann gefragt?

☐ ja ☐ nein

Wenn nein, warum nicht?

4.6. Haben Sie heute **Skype-Text-Chat** benutzt, um mit den Kollegen am anderen Standort zu kommunizieren?

☐ ja, mehrmals ☐ ja, einmal ☐ nein

4.7. Haben Sie heute **Skype-Telefonkonferenz** benutzt, um mit den Kollegen am anderen Standort zu kommunizieren?

☐ ja, mehrmals ☐ ja, einmal ☐ nein

4.8. Haben Sie heute **Application- bzw. Desktop-Sharing** (über WebConf) benutzt, um mit den Kollegen am anderen Standort zu kommunizieren?

☐ ja, mehrmals ☐ ja, einmal ☐ nein

4.9. Auf einer Noten-Skala von 1 (sehr gut) bis 5 (mangelhaft), wie beurteilen Sie die **Standort-übergreifende Kommunikation** heute insgesamt?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5
sehr gut mangelhaft

4.10. Haben Sie heute mit der Kundin gesprochen bzw. gechattet?

☐ ja, mehrmals ☐ ja, einmal ☐ nein

4.11. Hat das heutige gemeinsame Mittagsessen dazu beigetragen, die anderen im Team besser kennen zu lernen?

☐ ja, sehr ☐ ja, ein wenig ☐ nein
☐ heute gab es kein gemeinsames Mittag

5. Anmerkungen

Was ist Ihnen sonst noch aufgefallen? Gab es Probleme (technische, organisatorische, bezüglich Kommunikation, bezüglich Anforderungen)? Sonstiges?

A.3 Fragebogen Effizienzvergleich Audiodokumentation

Fragebogen Einleitung: *ohne* Index

Effizienz von Audiodokumentation

Hallo und vielen Dank, dass Sie sich die Zeit für diesen Fragebogen nehmen.

Achtung: Zur Teilnahmer an dieser Studie benötigen Sie ein Programm, das Audiodateien im MP3-Format abspielen kann. Funktionen zum Spulen, Springen und Pausieren sind hilfreich. Wir empfehlen VLC.

In dieser Studie wollen wir herausfinden, wie gut sich Audioaufzeichnungen von verteilten Meetings als Dokumentation eignen. Dazu messen wir (bzw. Sie) die Zeit, die Sie benötigen, um bestimmte Informationen in einem Audiomitschnitt zu finden.

> *Stellen Sie sich vor, Sie sind Softwareentwickler in einem global verteilten Softwareprojekt. Teams aus Russland, Finnland und Deutschland sind beteiligt. Die Teams stimmen sich in regelmäßigen Skype-Meetings ab. Sie haben das letzte Meeting verpasst und suchen nun wichtige Informationen in einem Mitschnitt vom letzten Skype-Meeting. Dieses liegt ihnen als MP3 vor. Aus früheren Meetings ist Ihnen bekannt, dass die Agenda üblicherweise wie folgt aussieht:*
>
> - *Bericht über aktuellen Status des Projekts*
> - *Besprechung der Ziele für heute*
> - *Planung des nächsten Releases*

Download des Skype-Meeting-Mitschnitts (Rechtsklick -> Ziel speichern unter...)

Hinweis: Die Aufzeichnung des Meetings ist 20 min. lang. Wir gehen aber davon aus, dass Sie nicht die komplette Aufnahme hören müssen, um die 3 Fragen dazu beantworten zu können.

Weiter

Kai Stapel, FG Software Engineering, Leibniz Universität Hannover

0% ausgefüllt

Effizienz von Audiodokumentation

Hallo und vielen Dank, dass Sie sich die Zeit für diesen Fragebogen nehmen.

Achtung: Zur Teilnahmer an dieser Studie benötigen Sie ein Programm, das Audiodateien im MP3-Format abspielen, sowie CUE-Dateien öffnen kann. Funktionen zum Spulen, Springen und Pausieren sind hilfreich. Wir empfehlen VLC.

In dieser Studie wollen wir herausfinden, wie gut sich Audioaufzeichnungen von verteilten Meetings als Dokumentation eignen. Dazu messen wir (bzw. Sie) die Zeit, die Sie benötigen, um bestimmte Informationen in einem Audiomitschnitt mit einem CUE-Sheet als Index zu finden.

> *Stellen Sie sich vor, Sie sind Softwareentwickler in einem global verteilten Softwareprojekt. Teams aus Russland, Finnland und Deutschland sind beteiligt. Die Teams stimmen sich in regelmäßigen Skype-Meetings ab. Sie haben das letzte Meeting verpasst und suchen nun wichtige Informationen in einem Mitschnitt vom letzten Skype-Meeting. Dieses liegt ihnen als MP3 vor. Um **leichter in der Aufzeichnung navigieren zu können**, wurde zusätzlich ein Inhaltsverzeichnis des Audiomitschnitts in Form eines CUE-Sheets mit "Lesezeichen" erstellt. Aus früheren Meetings ist Ihnen bekannt, dass die Agenda üblicherweise wie folgt aussieht:*
>
> - *Bericht über aktuellen Status des Projekts*
> - *Besprechung der Ziele für heute*
> - *Planung des nächsten Releases*

Download des Skype-Meeting-Mitschnitts (Rechtsklick -> Ziel speichern unter...)

Download des Skype-Meeting-Index (Rechtsklick -> Ziel speichern unter...)

Hinweise:

- Laden Sie beide Dateien herunter und speichern Sie sie gemeinsam in einem Verzeichnis.
- Öffnen Sie dann mit einem geeigneten Audioplayer (z.B. VLC) das CUE-Sheet, um über den darin enthaltenen Index in der Audioaufzeichnung navigieren zu können.
- Im VLC können Sie sich in der Menüleiste über Ansicht -> Wiedergabeliste die Wiedergabliste mit den Indexeinträgen anzeigen lassen.
- Dabei ist der "Titel" der Inhalt, über den gerade gesprochen wird, und der "Künstler" der gerade sprechende Meetingteilnehmer.
- Im VLC wird der Künstler mit den Default-Einstellungen in der Wiedergabeliste nicht angezeigt. Er kann durch Rechtsklick auf die Überschriften in der Wiedergabeliste eingeblendet werden.

Weiter

Kai Stapel, FG Software Engineering, Leibniz Universität Hannover

0% ausgefüllt

Metainformationen

Zunächst ein paar Fragen über Sie, die wir ausschließlich zur genaueren Auswertung und Interpretation Ihrer Antworten benötigen.

Wie beurteilen Sie Ihre Englischkenntnisse?

	sehr gut	gut	befrie-digend	ausrei-chend	schlecht bzw. gar nicht
	1	2	3	4	5
Selber Sprechen	○	○	○	○	○
Verstehendes Hören	○	○	○	○	○

Welche Erfahrung haben sie in der Softwareentwicklung?

An welcher Art von Softwareprojekten haben Sie schon mitgewirkt?

	keine Erfahrung bzw. Ein-Mann-Projekte	2-6 Entwickler	7-14 Entwickler	15-20 Entwickler	mehr als 20 Entwickler
Privat (non-profit)	○	○	○	○	○
Studium/Ausbildung/Praktikum	○	○	○	○	○
Unternehmen (eigenes oder angestellt)	○	○	○	○	○

Zurück Weiter

Kai Stapel, FG Software Engineering, Leibniz Universität Hannover

33% ausgefüllt

Fragen zum Mitschnitt des verpassten Meetings

Versuchen Sie nun mit Hilfe der Audioaufzeichnung die folgenden Fragen zu beantworten. Notieren Sie jeweils Start- und Endzeit, damit wir den Aufwand bei der Suche der jeweiligen Information abschätzen können.

Startzeit vor der Suche im Audiomitschnitt?

Notieren Sie hier bitte die aktuelle Uhrzeit, bevor sie mit der Beantwortung der Fragen beginnen.

Startzeit

Format [hh:mm:ss], Beispiel: 17:43:22

Frage 1: Wie ist der Status des Projekts aus finnischer Sicht? Was wurde bis zu dem heutigen Meeting vom finnischen Team fertiggestellt?

Mehrfachauswahl: Nennen Sie die Ergebnisse, die das Team Finnland zu Beginn des Meetings als fertiggestellt nennt.

- [ ] Fixed bug (this morning)
- [ ] Story tree view (yesterday)
- [ ] Roadmap view (this morning)
- [ ] Fixed bug (yesterday)
- [ ] Story tree view (this morning)
- [ ] Roadmap view (yesterday)

Zwischenzeit nach Beantwortung der ersten Frage.

Zwischenzeit

Format [hh:mm:ss], Beispiel: 17:43:22

Startzeit 2. Frage (optional, falls Sie eine Pause gemacht haben)

Frage 2: Welche Ziele hat das deutsche Team für den heutigen Tag?

Mehrfachauswahl: Nennen Sie alle Ziele, die Eric (Team Deutschland) nennt.

- [ ] Create database structure for burndown view
- [ ] Create database structure for iteration view
- [ ] Create database structure for increment view
- [ ] Close a test on burndown bar calculations
- [ ] Create testing database outside test
- [ ] Cleanup code
- [ ] Cleanup tests
- [ ] Refactor code

- [ ] Refactor tests
- [ ] Open a test on burndown bar calculations
- [ ] Create testing database inside test

Zwischenzeit nach Beantwortung der zweiten Frage.

Zwischenzeit

Format [hh:mm:ss], Beispiel: 17:43:22

Startzeit 3. Frage (optional, falls Sie eine Pause gemacht haben)

Frage 3: Für wann wird das nächste Release festgesetzt?

Nennen Sie die Zeit (und die Zeitzone bzw. das Land), für die das nächste Release geplant wird.

Zeit und Zeitzone bzw. Land

Endzeit nach der Suche im Audiomitschnitt und Beantwortung der Fragen?

Notieren Sie hier bitte die aktuelle Uhrzeit, nachdem sie die Fragen beantwortet haben.

Endzeit

Format [hh:mm:ss], Beispiel: 17:43:22

Abschließende Bemerkungen

Anmerkungen zum Fragebogen oder zur Studie?

Was sollten wir Ihrer Meinung nach noch beachten?

Zurück Weiter

Kai Stapel, FG Software Engineering, Leibniz Universität Hannover

67% ausgefüllt

Lebenslauf

Name	Kai Stapel, geboren am 28.08.1981 in Erfurt
Ausbildung	
02/2007 - 03/2012	Promotion am Fachgebiet Software Engineering der Leibniz Universität Hannover
10/2004 - 11/2006	Studium der Informatik an der Leibniz Universität Hannover Abschluss: Master of Science (M. Sc.) Informatik *Informationsflussoptimierung eines Softwareentwicklungsprozesses aus der Bankenbranche*
10/2002 - 09/2004	Studium der Angewandten Informatik an der Universität Hannover Abschluss: Bachelor of Science (B. Sc.) Angewandte Informatik *Transformation objektorientierter Programmtraces in Animationsskripte eines 3D-Renderers*
8/1999 - 07/2001	Sekundarstufe II am Gottfried-Wilhelm-Leibniz-Gymnasium Leinefelde, Abschluss: Allgemeine Hochschulreife
7/1998 - 06/1999	Schüleraustausch an der Colony High School, Wasilla, AK, USA
Berufserfahrung	
11/2006 - 03/2012	Wissenschaftlicher Mitarbeiter am Fachgebiet Software Engineering der Leibniz Universität Hannover
09/2006 - 11/2006	Wissenschaftliche Hilfskraft am Fachgebiet Software Engineering der Leibniz Universität Hannover
01/2005 - 09/2006	Wissenschaftliche Hilfskraft am Institut für Mikroelektronische Systeme der Leibniz Universität Hannover
01/2005 - 12/2005	Studentische Hilfskraft in der Forschungsgruppe Software Engineering der Universität Kassel
11/2003 - 12/2004	Studentische Hilfskraft am Institut für Mikroelektronische Systeme der Universität Hannover

Definitionen und Sätze

Literaturverzeichnis

[Anderson 2000] Anderson, John R. *Learning and Memory: An Integrated Approach*. Wiley, second edition Auflage. 2000. → S. 29, 30, 31, 32, 34, 35, 37

[Anderson 2001] Anderson, John R. *Kognitive Psychologie*. Spektrum, Akad. Verlag, 3. aufl. Auflage. 2001. → S. 11, 30, 32, 34

[Aranda 2010] Aranda, Jorge. *A Theory of Shared Understanding for Software Organizations*. Doktorarbeit, Graduate Department of Computer Science, University of Toronto. Nov 2010. → S. 198, 236, 237, 240, 368

[Armel 2009] Armel, Kate. *Technology Can Only Do So Much: Why the Human Factor Continues to Frustrate Software Developers*. Forschungsbericht, Quantitative Software Management, Inc. 2009.
URL http://www.qsm.com/blog/2009/empirical-examination-brooks-law/index.html → S. 3, 8, 130

[Armel 2011a] Armel, Kate. *Has Software Productivity Declined Over Time?* Forschungsbericht, Quantitative Software Management, Inc. 2011.
URL http://www.qsm.com/blog/2011/has-software-productivity-declined-over-time → S. 3, 8

[Armel 2011b] Armel, Kate. *Simpson's Paradox*. Forschungsbericht, Quantitative Software Management, Inc. 2011.
URL http://www.qsm.com/blog/2011/simpsons-paradox → S. 3, 8

[Armel 2011] Armel, Kate. *Technology Can Only Do So Much: Using History to Manage the Human Factor in Software Development*. Forschungsbericht, Quantitative Software Management, Inc. 2011.
URL http://www.qsm.com/sites/www.qsm.com/themes/diamond/docs/TechnologyCanOnlyDoSoMuch.pdf → S. 3

[Armour 2000] Armour, Phillip Glen. *The Five Orders of Ignorance: Viewing software development as knowledge acquisition and ignorance reduction*. Communications of the ACM, 43(10):17–20. Oct 2000. → S. 12, 13, 217, 219, 220, 368

[Armour 2003] Armour, Phillip Glen. *The Laws of Software Process: A New Model for the Production and Management of Software*. Auerbach Publications, 1 Auflage. Sep 2003. → S. 198, 216, 217, 218, 220, 221, 227, 240, 368

[Armour 2006] Armour, Phillip Glen. *Software: Hard Data - Seeking solid information on the behavior of modern projects*. Communications of the ACM, 49(9):15–17. Sep 2006. → S. 2, 8, 130

[Barnard 1994] Barnard, Jack und Price, Art. *Managing code inspection information*. IEEE Software, 11(2):59–69. Mar 1994. → S. 111

[Basili 2007] Basili, Victor R. *Empirical Software Engineering Issues - Critical Assessment and Future Directions*, Kapitel The Role of Controlled Experiments in Software Engineering Research, S. 33–37. Nummer 4336 in LNCS. Springer-Verlag Berlin Heidelberg. 2007. → S. 18

[Bazire 2005] Bazire, Mary und Brézillon, Patrick. *Understanding Context Before Using It*. In *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT 2005*, Band 3554/2005 von *Lecture Notes in Computer Science*, S. 29–40. Springer. 2005. → S. 46

[Beck 2000] Beck, Kent. *Extreme Programming Explained*. Addison-Wesley. 2000. → S. 1, 142, 168, 220, 328

[Bjoerner 2006-1] Bjørner, Dines. *Software Engineering 1: Abstraction and Modelling*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg. 2006. → S. 230, 231, 232, 233, 367

[Bjoerner 2006-2] Bjørner, Dines. *Software Engineering 2: Specification of Systems and Languages*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg. 2006. → S. 230, 233

[Bjoerner 2006-3] Bjørner, Dines. *Software Engineering 3: Domains, Requirements, and Software Design*. Texts in Theoretical Computer Science. An EATCS Series. Springer Berlin Heidelberg. 2006. → S. 230, 233

[Bjoerner 2006] Bjørner, Dines. *The Triptych Process Model - Process Assessment and Improvement*. online. 2006. From a seminar at Tokyo University in Honour of Kouichi Kishida's 70th Anniversary. URL http://www.semat.org/pub/Main/PubsandRefs/believable-software-management.pdf → S. 143, 198, 230, 231, 232, 240, 298

[Boehm 2002] Boehm, Barry. *Software Engineering Is a Value-Based Contact Sport*. IEEE Software, 19(5):95–97. Sep/Oct 2002. → S. 3, 11, 115, 242

[Boehm 2006] Boehm, Barry W. und Jain, Apurva. *Value-Based Software Engineering*, Kapitel An Initial Theory of Value-Based Software Engineering, S. 15–37. Springer Berlin Heidelberg. 2006. → S. 196, 198, 228, 229, 240, 280, 281, 366

[Boehm 1989] Boehm, Barry W. und Ross, Rony. *Theory-W Software Project Management: Principles and Examples*. IEEE Transactions on Software Engineering, 15(7):902–916. Jul 1989. → S. 10, 17, 198, 204, 205, 206, 240

[Bordia 1997] Bordia, Prashant. *Face-to-Face Versus Computer-Mediated Communication: A Synthesis of the Experimental Literature*. Journal of Business Communication, 34(1):99–120. 1997. → S. 272, 276

[Brezillon 2005] Brézillon, Patrick. *Task-Realization Models in Contextual Graphs*. In *Modeling and Using Context, 5th International and Interdisciplinary Conference, CONTEXT*, Band 3554 von *Lecture Notes in Computer Science*, S. 55–68. Springer, Paris, France. Jul 2005. → S. 47

[Briet 1951] Briet, Suzanne. *Qu'est-ce que la documentation?* Éditions documentaires, industrielles et techniques, Paris. 1951. → S. 40, 41

[Brooks 1974] Brooks, Fred P. *The Mythical Man-Month*. Datamation, S. 44–52. 1974. → S. 198, 242, 368

[Brooks 1987] Brooks, Fred P. *No Silver Bullet — Essence and Accident in Software Engineering*. IEEE Computer, 20(4):10–19. 1987. → S. 65, 188, 189

[Brooks 1995] Brooks, Frederick P. *The Mythical Man-Month. Essays on Software Engineering*, Band Anniversary Edition. Addison-Wesley Longman. Aug 1995. → S. 132, 137, 185, 186, 188, 192, 194, 198, 199, 200, 201, 202, 203, 240

[Broy 2011] Broy, Manfred. *Can Practitioners Neglect Theory and Theoreticians Neglect Practice?* IEEE Computer, 44(10):19–24. Oct 2011. → S. 6, 7, 9, 13, 195, 196, 280, 281, 363

[Buckland 1997] Buckland, Michael K. *What Is a "Document"?* Journal of the American Society for Information Science, 48(9):804–809. Sep 1997. → S. 40, 41

[Capurro 2000] Capurro, Rafael. *Einführung in den Informationsbegriff*. Vorlesungsskript. 2000. Letzter Abruf: 22.06.2011.
URL http://www.capurro.de/infovorl-index.htm → S. 43

[Carmel 1999] Carmel, Erran. *Global Software Teams: Collaborating Across Borders and Time Zones*. Prentice Hall. 1999. → S. 4, 8, 242

[Cataldo 2008] Cataldo, Marcelo, Herbsleb, James D. und Carley, Kathleen M. *Socio-Technical Congruence: A Framework for Assessing the Impact of Technical and Work Dependencies on Software Development Productivity*. In *ESEM '08: Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement*. 2008. → S. 10, 199, 234, 235, 240, 368

[Chapanis 1972] Chapanis, Alphonse, Ochsman, Robert B., Parrish, Robert N. und Weeks, Gerald D. *Studies in Interactive Communication: I. The Effects of Four Communication Modes on the Behavior of Teams During Cooperative Problem-Solving*. Human Factors: The Journal of the Human Factors and Ergonomics Society, 14(6):487–509. December 1972. → S. 55

[Chapanis 1977] Chapanis, Alphonse, Parrish, Robert N., Ochsman, Robert B. und Weeks, Gerald D. *Studies in Interactive Communication: II. The Effects of Four Communication Modes on the Linguistic Performance of Teams during Cooperative Problem Solving*. Human Factors: The Journal of the Human Factors and Ergonomics Society, 19(2):101–126. April 1977. → S. 55

[Ciesnik 2010] Cieśnik, Robert. *Dashboard zur Verbesserung der Kommunikation bei verteilter Softwareentwicklung*. Bachelorarbeit, Leibniz Universität Hannover, Fachgebiet Software Engineering. 9 2010. → S. 327

[Clark 1993] Clark, Herbert H. *Arenas of Language Use*. University of Chicago Press. May 1993. → S. 83, 116, 171

[Clark 1991] Clark, Herbert H. und Brennan, Susan E. *Perspectives on Socially Shared Cognition*, Kapitel Grounding in Communication, S. 127–149. American Psychological Association. 1991. → S. 58, 60, 62

[Cockburn 2001] Cockburn, Alistair. *Agile Software Development*. Addison-Wesley Professional, 1 Auflage. 2001. → S. 212, 213

[Cockburn 2007] Cockburn, Alistair. *Agile Software Development. The Cooperative Game*. The Agile Software Development Series. Pearson Education, Inc., 2 Auflage. 2007. → S. 10, 55, 126, 199, 212, 214, 227, 240, 242, 368

[Cockburn 2009] Cockburn, Alistair. *"I Come to Bury Agile, Not to Praise It" - Effective Software Development in the 21st Century*. 2009. Keynote at Agile 2009, Chicago, IL, USA.
URL http://alistair.cockburn.us/Alistairs+keynote+at+Agile2009 → S. 211, 212, 215, 216, 368

[Cockburn 2010] Cockburn, Alistair. *A Detailed Critique of the SEMAT Initiative*. Forschungsbericht, Humans and Technology, inc. 2010.
URL http://alistair.cockburn.us/A+Detailed+Critique+of+the+SEMAT+Initiative → S. 7, 9, 13, 363

[Cockburn 2010a] Cockburn, Alistair. *Short Theory of Designing in Teams*. Forschungsbericht, Humans and Technology, inc. 2010.
URL http://alistair.cockburn.us/A+short+theory+of+designing+in+teams.pps → S. 214

[Cody 1985] Cody, Michael J. und McLaughlin, Margaret L. *The Situation as a Construct in Interpersonal Communication Research*. In Mark L.Knapp und Gerald R. Miller, Herausgeber, *Handbook of Interpersonal Communication*, S. 263–312. Sage Publishers, Beverly Hills. 1985. → S. 237

[Conway 1968] Conway, Melvin Edward. *How Do Committees Invent?* Datamation, 14(4):28–31. Apr 1968. → S. 188, 222, 223, 225

[Cosgrove 1971] Cosgrove, J. *Needed: A New Planning Framework*. Datamation, 17(23):37–39. Dez. 1971. → S. 201

[Curtis 1984] Curtis, Bill. *Fifteen Years of Psychology in Software Engineering: Individual Differences and Cognitive Science*. In *7th International Conference on Software Engineering*, S. 97–106. 1984. → S. 30, 92

[Curtis 1988] Curtis, Bill, Krasner, Herb und Iscoe, Neil. *A Field Study of the Software design Process for Large Systems*. Communications of the ACM, 31(11):1268–1287. Nov 1988. → S. 3, 15, 92, 115, 122, 220, 227, 242, 365, 368

[Daft 1984] Daft, Richard L. und Lengel, Robert H. *Information Richness: A New Approach to Managerial Behavior and Organization Design*. Research in Organizational Behavior, 6:191–233. May 1984. → S. 56, 62

[Daft 1986] Daft, Richard L. und Lengel, Robert H. *Organizational information requirements, media richness and structural design*. Management Science, 32(5):554–571. May 1986. → S. 56, 59

[Daft 1987] Daft, Richard L., Lengel, Robert H. und Trevino, Linda Klebe. *Message Equivocality, Media Selection, and Manager Performance: Implications for Information Systems*. MIS Quarterly, 11(3):355–366. Sep 1987. → S. 56, 274

[DeLuca 2006] DeLuca, Dorrie und Valacich, Joseph S. *Virtual teams in and out of synchronicity*. Information Technology & People, 19(4):323–344. 2006. → S. 210

[DeLuca 2006a] DeLuca, Dorrie, Gasson, Susan und Kock, Ned. *Adaptations that Virtual Teams Make so that Complex Tasks Can Be Performed Using Simple E-Collaboration Technologies*. International Journal of e-Collaboration (IJeC), 2(3):65–91. 2006. → S. 61

[DeMarco 1979] DeMarco, Tom. *Structured Analysis and System Specification*. Prentice-Hall. 1979. → S. 295

[Dennis 1999] Dennis, Alan R. und Valacich, Joseph S. *Rethinking Media Richness: Towards a Theory of Media Synchronicity*. In *32nd Annual Hawaii International Conference on System Sciences (HICSS-32)*, S. 1–10. Maui, HI , USA. Jan 1999. → S. 59, 60, 198, 208, 209

[Dennis 2008] Dennis, Alan R., Fuller, Robert M. und Valacich, Joseph S. *Media, Tasks, and Communication Processes: A Theory of Media Synchronicity*. MIS Quarterly, 32(3):575–600. 2008. → S. 59, 60, 61, 62, 175, 183, 198, 208, 210, 240, 323, 370

[Devlin 2003] Devlin, Keith. *Why Universities Require Computer Science Students to Take Math*. Communications of the ACM, 46(9):36–39. Sep 2003. → S. 91

[Dey 2001] Dey, Anind K. *Understanding and Using Context*. Personal and Ubiquitous Computing, 5(1):4–7. Feb 2001. → S. 45, 46

[Duden 1996] Scholze-Stubenrecht, Werner und Wermke, Matthias, Herausgeber. *Duden - Die deutsche Rechtschreibung*, Band 1. Dudenverlag, Mannheim - Leipzig - Wien - Zürich, 21. auflage Auflage. 1996. → S. 54

[Easterbrook 2008] Easterbrook, Steve, Singer, Janice, Storey, Margaret-Anne und Damian, Daniela. *Guide to Advanced Empirical Software Engineering*, Kapitel 11, Selecting Empirical Methods for Software Engineering Research, S. 285–311. Springer, London. 2008. → S. 6, 8, 9, 17, 195, 196, 280, 281, 363

[Emam 2008] Emam, Khaled El und Koru, A. Güneş. *A Replicated Survey of IT Software Project Failures*. IEEE Software, 25(5):84–90. Aug 2008. → S. 3, 8, 257

[Endsley 1995] Endsley, Mica R. *Toward a Theory of Situation Awareness in Dynamic Systems*. Human Factors: The Journal of the Human Factors and Ergonomics Society, 37(1):32–64. Mar 1995. → S. 319

[Erra 2010] Erra, U. und Scanniello, G. *Assessing communication media richness in requirements negotiation*. IET Software, 4(2):134–148. Apr 2010. → S. 272, 275

[Flohr 2008] Flohr, Thomas. *Ein Framework als Grundlage der Ausgestaltung von Quality-Gate-Referenzprozessen für die Software-Entwicklung*. Doktorarbeit, Leibniz Universität Hannover, Welfengarten 1, 30167 Hannover. 2008. → S. 246

[Floridi 2005] Floridi, Luciano. *Is Information Meaningful Data?* Philosophy and Phenomenological Research, LXX(2):351–370. Nov 2005. → S. 48

[Floridi 2006] Floridi, Luciano. *Semantic Conceptions of Information*. The Stanford Encyclopedia of Philosophy (Summer 2006 Edition). 2006. → S. 39, 40, 48, 75

[Foulke 1968] Foulke, Emerson. *Listening Comprehension as a Function of Word Rate*. Journal of Communication, 18(3):198–206. Sep 1968. → S. 111

[Gamma 1995] Gamma, Erich, Helm, Richard, Johnson, Ralph und Vlissides, John. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company. 1995. → S. 101, 112

[Ge 2008] Ge, Xiaoxuan. *FLOW Patterns: Beschreibung und Diskussion von Informationsflussmustern in der Softwareentwicklung*. Masterarbeit, Leibniz Universität Hannover, Fachgebiet Software Engineering. 2008. → S. 311

[Ge 2008a] Ge, Xiaoxuan. *FLOW-Patterns-Katalog*. Forschungsbericht, Leibniz Universität Hannover, Fachgebiet Software Engineering. 2008. URL http://static.se.uni-hannover.de/documents/flow/Xiaoxuan_Ge-FLOW-Patterns-Katalog.pdf → S. 311

[Glickstein 2008] Glickstein, Ira. *Quantifying Brooks Mythical Man-Month*. online - Google knol. Aug 2008. Letzter Abruf: 23.11.2011, Version 94. URL http://knol.google.com/k/ira-glickstein/quantifying-brooks-mythical-man-month/3ncxde0rz8dtk/3 → S. 193

[Glinz 2009] Glinz, Martin. *Informatik II: Modellierung, Kapitel 12: Abstraktion*. Vorlesungsskript. 2009. URL http://www.ifi.uzh.ch/rerg/fileadmin/downloads/teaching/courses/inf_II_fs09/inf_II_kapitel_12.pdf → S. 65

[Gross 2009] Gross, Michael. *Semantische Erweiterung eines Frameworks für graphische Editoren*. Bachelorarbeit, Leibniz Universität Hannover, Fachgebiet Software Engineering. 2009. → S. 311

[CHAOS 1995] Group, The Standish. *Chaos Report*. 1995. → S. 248

[Hartley 1928] Hartley, Ralph V. L. *Transmission of Information*. Bell System Technical Journal, S. 535–563. Jul 1928. → S. 44

[Henderson 1990] Henderson, Rebecca M. und Clark, Kim B. *Architectural Innovation: The Reconfiguration of Existing Product Technologies and the Failure of Established Firms*. Administrative Science Quarterly, 35(1):9–30. Mar 1990. → S. 223

[Herbsleb 2003a] Herbsleb, James D. und Mockus, Audris. *An Empirical Study of Speed and Communication in Globally Distributed Software Development*. IEEE Transactions on Software Engineering, 29(6):481–494. June 2003. → S. 365

[Herbsleb 2003] Herbsleb, James D. und Mockus, Audris. *Formulation and Preliminary Test of an Empirical Theory of Coordination in Software Engineering*. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, S. 138–147. 2003. → S. 5, 7, 8, 9, 13, 188, 196, 198, 221, 222, 223, 228, 234, 240, 280, 363, 368

[Herbsleb 2006] Herbsleb, James D., Mockus, Audris und Roberts, Jeffrey A. *Collaboration in Software Engineering Projects: A Theory of Coordination*. In *ICIS 2006: Proceedings of the 27th International Conference on Information Systems*. Milwaukee, WI, USA. Dec 2006. → S. 223, 224, 227, 228

[Hofstadter 1999] Hofstadter, Douglas R. *Gödel, Escher, Bach: Ein Endloses Geflochtenes Band*. Klett-Cotta, Stuttgart, 15 Auflage. 1999. → S. 94

[IEEE 1990] IEEE. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std 610.12-1990. 1990. → S. 11, 12, 39, 40, 132, 133

[ISO 1995] ISO. *Information technology - Specification and standardization of data elements - Part 4: Rules and guidelines for the formulation of data definitions*. Jul 1995. URL http://metadata-stds.org/11179/ → S. 47

[Jacobson 2009] Jacobson, Ivar und Meyer, Bertrand. *Methods Need Theory*. Dr. Dobbs - The World of Software Development. 06 Aug 2009. Letzter Abruf: 11.01.2011. → S. 6, 9, 13, 196, 197, 281, 366, 371

[Jacobson 2009a] Jacobson, Ivar und Spence, Ian. *Why We Need a Theory for Software Engineering*. Dr. Dobbs - The World of Software Development. 02 Oct 2009. Letzter Abruf: 11.01.2011. → S. 5, 6, 9, 13, 363

[Jacobson 2009b] Jacobson, Ivar, Meyer, Bertrand und Soley, Richard. *The SEMAT Initiative: A Call for Action*. Dr. Dobbs - The World of Software Development. 09 Dec 2009. Letzter Abruf: 11.01.2011. → S. 7, 13

[Jedlitschka 2005] Jedlitschka, Andreas und Pfahl, Dietmar. *Reporting Guidelines for Controlled Experiments in Software Engineering*. In *ISESE '05: International Symposium on Empirical Software Engineering*, S. 95–104. 2005. → S. 257

[Jorgensen 2007] Jorgensen, Magne und Shepperd, Martin. *A Systematic Review of Software Development Cost Estimation Studies*. IEEE Transactions on Software Engineering, 33(1):33 –53. Jan 2007. → S. 278, 279

[Joergensen 2004] Jørgensen, Magne und Sjøberg, Dag. *Generalization and Theory-Building in Software Engineering Research*. In *EASE '04: The 8th International Conference on Empirical Assessment in Software Engineering, co-located at ICSE '04*, S. 29–35. 2004. → S. 196

[Juristo 2001] Juristo, Natalia und Moreno, Ana M. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers. 2001. → S. 7, 9, 196, 267, 280, 363

[Karim 2005] Karim, Nor Shahriza Abdul und Heckman, Robert. *Group communication media choice and the use of information and communication technology to support learning: a case study*. Campus-Wide Information Systems, 22(1):28–42. 2005. → S. 61, 210

[Keil 1998] Keil, Mark, Cule, Paul E., Lyytinen, Kalle und Schmidt, Roy C. *A Framework for Identifying Software Project Risks*. Communications of the ACM, 41(11):76–83. 1998. → S. 162

[Kersten 2007] Kersten, Mik. *Focusing Knowledge Work with Task Context*. Doktorarbeit, The Software Practices Lab, The University of British Columbia, Canada. 2007. → S. 38

[Kersten 2006] Kersten, Mik und Murphy, Gail C. *Using Task Context to Improve Programmer Productivity*. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE-14)*, S. 1–11. ACM, New York, NY, USA, Portland, Oregon, USA. 2006. → S. 38

[Kiesling 2011] Kiesling, Stephan. *Integration agiler Entwicklungszyklen in das V-Modell auf Basis von Informationsflüssen*. Masterarbeit, Leibniz Universität Hannover, Fachgebiet Software Engineering. 2011. → S. 348, 349, 353, 354, 355, 356, 357, 358

[Kirsner 1998] Kirsner, Kim, Speelman, Craig, Maybery, Murray, O'Brien-Malone, Angela und Anderson, Mike, Herausgeber. *Implicit and Explicit Mental Processes*. Psychology Press, 1 Auflage. 1998. → S. 36

[Kitchenham 2007] Kitchenham, Barbara. *Empirical Software Engineering Issues - Critical Assessment and Future Directions*, Kapitel Empirical Paradigm - The Role of Experiments, S. 25–32. Nummer 4336 in LNCS. Springer-Verlag Berlin Heidelberg. 2007. → S. 18

[Knauss 2008e] Knauss, Eric, Schneider, Kurt und Stapel, Kai. *A Game for Taking Requirements Engineering More Seriously*. In *Proceedings of Third International Workshop on Multimedia and Enjoyable Requirements Engineering (MERE '08)*. Barcelona, Spain. November 2008. → S. 311

[Kock 2005] Kock, Ned. *Media Richness or Media Naturalness? The Evolution of Our Biological Communication Apparatus and Its Influence on Our Behavior Toward E-Communication Tools*. IEEE Transactions on Professional Communication, 48(2):117–130. Jun 2005. → S. 61, 62, 176, 183

[Kramer 2007] Kramer, Jeff. *Is Abstraction the Key to Computing?* Communications of the ACM, 50(4):36–42. Apr 2007. → S. 91

[Kraut 1995] Kraut, Robert E. und Streeter, Lynn A. *Coordination in Software Development*. Communications of the ACM, 38(3):69–81. Mar 1995. → S. 4, 115, 242

[Kritzenberger 2004] Kritzenberger, Huberta. *Multimediale und interaktive Lernräume*. Oldenbourg Wissenschaftsverlag. 2004. → S. 55

[Kruchten 2002] Kruchten, Philippe. *The Nature of Software - What's so special about Software Engineering?* In *Proceedings of International Conference on The Sciences of Design*. Lyon, France. Mar 2002. → S. 6, 9, 13, 363

[Kruchten 2007] Kruchten, Philippe. *Software Project Management with OpenUP*, Kapitel A Conceptual Model of Software Development. Apr 2007. Draft. → S. 65

[Kurtzberg 2005] Kurtzberg, Terri R. *Feeling Creative, Being Creative: An Empirical Study of Diversity and Creativity in Teams*. Creativity Research Journal, 17(1):51–65. 2005. → S. 127, 140

[Kwan 2011] Kwan, Irwin, Schröter, Adrian und Damian, Daniela. *Does Socio-Technical Congruence Have an Effect on Software Build Success? A Study of Coordination in a Software Project*. IEEE Transactions on Software Engineering, 37(3):307–324. May/Jun 2011. → S. 235

[Lengel 1989] Lengel, Robert H. und Daft, Richard L. *The Selection of Communication Media as an Executive Skill*. The Academy of Management Executive (1987-1989), 2(3):225–232. Aug 1989. → S. 56

[Littlejohn 1983] Littlejohn, Stephen W. *Theories of Human Communication*. Wadsworth Publishing Company, 2 Auflage. 1983. → S. 59

[Lubars 1993] Lubars, Mitch, Potts, Colin und Richter, Charles. *A Review of the State of the Practice in Requirements Modeling*. In *IEEE International Symposium on Requirements Engineering*, S. 2–14. 1993. → S. 162

[Luebke 2005b] Lübke, Daniel und Flohr, Thomas. *Experiences from the Conducation of a Simulated Software Project Driven by Quality Gates*. In *TESI 2005*. 2005. → S. 247

[Ludewig 2010] Ludewig, Jochen und Lichter, Horst. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*, Band 2. überarb. u. akt. Aufl. dpunkt Verlag. 2010. → S. 2, 4, 5, 6, 9, 39, 41, 42, 64, 65, 79, 195, 280, 367, 368

[Lutz 1992] Lutz, Robyn R. *Analyzing Software Requirements Errors in Safety-Critical, Embedded Systems*. In *IEEE International Symposium on Requirements Engineering*, S. 126–133. 1992. → S. 162

[McGrath 1984] McGrath, Joseph E. *Groups: Interaction and Performance*. Prentice-Hall, New Jersey. 1984. → S. 56, 57, 62

[Mednick 1962] Mednick, Sarnoff A. *The Associative Basis of the Creative Process*. Psychological Review, 69(3):220–232. 1962. → S. 127

[Mill 1868] Mill, John Stuart. *System der deduktiven und induktiven Logik*, Band 1. Braunschweig, 5. erweiterte auflage Auflage. 1868. → S. 24

[Mill 1868b] Mill, John Stuart. *System der deduktiven und induktiven Logik*, Band 2. Braunschweig, 5. erweiterte auflage Auflage. 1868. → S. 64

[Mittelstrass 1980] Mittelstrass, Jürgen, Herausgeber. *Enzyklopädie Philosophie und Wissenschaftstheorie*, Band 1: A-G. Wissenschaftsverlag Bibliographisches Institut. 1980. → S. 23

[Mohapatra 2010] Mohapatra, Prateeti, Björndal, Petra und Smiley, Karen. *Causal Analysis of Factors Governing Collaboration in Global Software Development Teams*. In *IEEE Int'l Conf. on Global Software Engineering*, S. 128–132. Princeton, NJ, USA. 2010. → S. 5

[NATO 1968] Naur, Peter und Randell, B., Herausgeber. *Software Engineering: Report of a conference sponsored by the NATO Science Committee*. NATO, Garmisch, Germany. Oct 1968. → S. 1

[NATO 1969] Buxton, J. N. und Randell, B., Herausgeber. *Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee*. NATO, Rome, Italy. Oct 1969. → S. 93

[Niinimaeki 2009] Niinimäki, Tuomas, Piri, Arttu und Lassenius, Casper. *Factors Affecting Audio and Text-Based Communication Media Choice in Global Software Development Projects*. In *IEEE International Conference on Global Software Engineering*, S. 153–162. 2009. → S. 272

[Niinimaeki 2010] Niinimäki, Tuomas, Piri, Arttu, Lassenius, Casper und Paasivaara, Maria. *Reflecting the Choice and Usage of Communication Tools in GSD Projects with Media Synchronicity Theory*. In *IEEE International Conference on Global Software Engineering*, S. 3–12. 2010. → S. 208, 272, 274

[Nuenke 2011] Nünke, Gerrit. *Ein Empfehlungssystem für die Wahl geeigneter Kommunikationsmedien in verteilten Teams*. Bachelorarbeit, Leibniz Universität Hannover, Fachgebiet Software Engineering. 2011. → S. 370

[Nonaka 1991] Nonaka, Ikujiro. *The Knowledge-Creating Company*. Harvard Business Review, S. 96–104. Nov-Dec 1991. → S. 28, 71, 99

[Nonaka 2007] Nonaka, Ikujiro. *The Knowledge-Creating Company (HBR Classic)*. Harvard Business Review, S. 162–171. Jul-Aug 2007. Reprint of article from 1991. → S. 28

[North 1998] North, Klaus. *Wissensorientierte Unternehmensführung: Wertschöpfung durch Wissen*. Gabler Verlag, Wiesbaden, 3. auflage Auflage. 1998. → S. 48

[Oberauer 2008] Oberauer, Klaus und Lewandowsky, Stephan. *Forgetting in Immediate Serial Recall: Decay, Temporal Distinctiveness, or Interference?* Psychological Review, 115(3):544–576. 2008. → S. 31

[Parnas 1972] Parnas, David Lorge. *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, 15(12):1053–1058. Dec 1972. → S. 147, 222, 227

[PMBOK 1996] Duncan, William R., Herausgeber. *A Guide to the Project Management Body of Knowledge*. PMI Standards Committee, 2 Auflage. 1996. → S. 12

[Pohl 2008] Pohl, Klaus. *Requirements Engineering: Grundlagen, Prinzipien, Techniken*. dpunkt.verlag GmbH, Heidelberg, Germany, 2., korrigierte auflage Auflage. 2008. → S. 65, 140

[Polanyi 1962] Polanyi, Michael. *Tacit Knowing: Its Bearing on Some Problems of Philosophy*. Reviews of Modern Physics, 34(4):601–616. 1962. → S. 28

[Polanyi 1966] Polanyi, Michael. *The Tacit Dimension*. Doubleday, Garden City, NY. 1966. → S. 28

[Popper 2002] Popper, Karl R. *Logik der Forschung*. Mohr Siebeck, Tübingen, nachdr. der 10., verb. und vermehrten aufl., jub.-ausg. Auflage. 2002. ISBN 3-16-147837-1. → S. 195, 280, 364

[Prechelt 2000] Prechelt, Lutz und Unger, Barbara. *An Experiment Measuring the Effects of Personal Software Process (PSP) Training*. IEEE Transactions on Software Engineering, 27(5):465–472. May 2000. → S. 111

[Probst 1997] Probst, Gilbert J. B., Raub, Steffen und Romhardt, Kai. *Wissen managen. Wie Unternehmen ihre wertvollste Ressource optimal nutzen*. Gabler Verlag, Wiesbaden. 1997. → S. 27, 45, 48

[Rao 2006] Rao, Prabhakhar P. *Knowledge: What is Knowledge?* Wikipedia, The Free Encyclopedia. 2006. Artikel vom 03.10.2006. URL http://en.wikipedia.org/w/index.php?title=Knowledge&oldid=79327240 → S. 27, 70

[Reinsch 1990] Reinsch, N. L. und Beswick, Raymond W. *VOICE MAIL VERSUS CONVENTIONAL CHANNELS: A COST MINIMIZATION ANALYSIS OF INDIVIDUALS' PREFERENCES*. Academy of Management Journal, 33(4):801–816. 1990. → S. 60

[Rensink 2000] Rensink, R. A., Deubel, H. und Schneider, W. X. *The Incredible Shrinking Span: Estimates Of Memory Capacity Depend On Interstimulus Interval*. Investigative Ophthalmology & Visual Science, 41:425. 2000. ARVO 2000; Ft. Lauderdale, FL. → S. 31

[Rittel 1984] Rittel, Horst. *Developments in Design Methodology*, Kapitel Second-Generation Design Methods, S. 317–327. John Wiley & Sons. 1984. → S. 2, 98

[Roeber 2003] Roeber, Helena, Bacus, John und Tomasi, Carlo. *Typing in thin air: the canesta projection keyboard - a new method of interaction with electronic devices*. In *CHI '03 extended abstracts on Human factors in computing systems*, S. 712–713. Fort Lauderdale, FL, USA. Apr 2003. → S. 110, 111

[Royce 1970] Royce, Winston W. *Managing the Development of Large Software Systems*. In *Proceedings of IEEE WESCON*, S. 328 – 338. 1970. → S. 141

[RUP 2003] Essigkrug, Andreas und Mey, Thomas. *Rational Unified Process kompakt*. Spektrum Akademischer Verlag, Heidelberg - Berlin, 1 Auflage. 2003. → S. 1, 39, 141

[Schneider 2000] Schneider, Kurt. *LIDs: A Light-Weight Approach to Experience Elicitation and Reuse*. In Frank Bomarius und Markku Oivo, Herausgeber, *Product Focused Software Process Improvement*, Band 1840/2000 von *Lecture Notes in Computer Science*, S. 407–424. Springer Berlin / Heidelberg. 2000. → S. 251

[Schneider 2004] Schneider, Kurt. *A Descriptive Model of Software Development to Guide Process Improvement*. In *Conquest*. ASQF, Nürnberg, Germany. 2004. → S. 283, 284, 288, 293, 364

[Schneider 2005c] Schneider, Kurt. *Software Process Improvement from a FLOW Perspective*. In *Learning Software Organizations Workshop*. 2005. → S. 293

[Schneider 2005b] Schneider, Kurt. *Vier Formen der Erfahrungsvermittlung im Studium*. In *Software Engineering im Unterricht der Hochschulen (SEUH 2005)*. 2005. → S. 247

[Schneider 2006b] Schneider, Kurt. *Aggregatzustände von Anforderungen erkennen und nutzen*. GI Softwaretechnik-Trends. 2006. → S. 293

[Schneider 2006] Schneider, Kurt. *Rationale Management in Software Engineering*, Kapitel Rationale as a By-Product, S. 91–109. Springer, Berlin, Heidelberg. 2006. → S. 313, 340, 345, 346

[Schneider 2009] Schneider, Kurt. *Experience and Knowledge Management in Software Engineering*. Springer-Verlag. 2009. → S. 90, 91, 289, 290

[Schneider 2005] Schneider, Kurt und Lübke, Daniel. *Systematic Tailoring of Quality Techniques*. In *World Congress of Software Quality 2005*, Band 3. 2005. → S. 283, 284, 288, 293, 301, 351, 364

[Schneider 2005a] Schneider, Kurt, Lübke, Daniel und Flohr, Thomas. *Softwareentwicklung zwischen Disziplin und Schnelligkeit*. TeleKommunikationAktuell, 59(05-06):1–21. 2005. → S. 283, 284, 293, 364

[Schneider 2008] Schneider, Kurt, Stapel, Kai und Knauss, Eric. *Beyond Documents: Visualizing Informal Communication*. In *Proceedings of Third International Workshop on Requirements Engineering Visualization (REV '08)*. Barcelona, Spain. November 2008. → S. 293

[Schuenemann 2004] Schünemann, Ulf. *Model vs. Abstraction (levels of meta)*. online. 2004. Letzter Abruf: 04.08.2011. URL http://www.cs.mun.ca/~ulf/mod/rel.html → S. 64

[Schuenemann 2005] Schünemann, Ulf. *Composite Objects: Dynamic Representation and Encapsulation by Static Classification of Object References*. PhD thesis, Memorial University of Newfoundland, Canada. 2005. → S. 65, 92

[Schwaber 2002] Schwaber, Ken und Beedle, Mike. *Agile Software Development with Scrum*. Prentice Hall. 2002. → S. 1, 348, 351, 353, 356

[Seiffert 1989] Seiffert, Helmut und Radnitzky, Gerard, Herausgeber. *Handlexikon zur Wissenschaftstheorie*. Ehrenwirth, München. 1989. ISBN 3-431-02616-8. → S. 21, 22, 23, 24, 383

[Shannon 1948] Shannon, Claude Elwood. *A Mathematical Theory of Communication*. Bell System Technical Journal, 27:379–423 & 623–656. Jul & Oct 1948. → S. 43, 50, 51, 55, 88, 115, 117

[Shannon 1953] Shannon, Claude Elwood. *The Lattice Theory of Information*. Transactions of the Professional Group on Information Theory, 1(1):105–107. Feb 1953. → S. 44

[Short 1976] Short, John, Williams, Ederyn und Christie, Bruce. *The Social Psychology of Telecommunications*. John Wiley & Sons Ltd. 1976. → S. 55, 62

[Silva 2010] da Silva, Fabio Q. B., Costa, Catarina, França, A. César C. und Prikladnicki, Rafael. *Challenges and Solutions in Distributed Software Development Project Management: a Systematic Literature Review*. In *IEEE Int'l Conf. on Global Software Engineering*, S. 87–96. Princeton, NJ, USA. 2010. → S. 4, 8, 115, 242, 268, 269, 270, 272, 365

[Song 2011] Song, Hui. *Erstellung eines Werkzeugs zur Unterstützung der Dokumentation von Skype-basierten Meetings*. Bachelorarbeit, Leibniz Universität Hannover, Fachgebiet Software Engineering. 2011. → S. 345, 346

[Sosa 2004] Sosa, Manuel E., Eppinger, Steven D. und Rowles, Craig M. *The Misalignment of Product Architecture and Organizational Structure in Complex Product Development*. Management Science, 50(12):1674–1689. Dec 2004. → S. 223

[Stapel 2006] Stapel, Kai. *Informationsflussoptimierung eines Softwareentwicklungsprozesses aus der Bankenbranche*. Master, FG Software Engineering, Leibniz Universität Hannover. 11 2006. → S. 39, 43, 49, 79, 81, 82, 83, 293

[Stapel 2011] Stapel, Kai und Schneider, Kurt. *Managing Knowledge on Communication and Information Flow in Global Software Projects*. Expert Systems - Special Issue on Knowledge Engineering in Global Software Engineering (to appear). 2011. To appear. → S. 329, 339

[Stapel 2007] Stapel, Kai, Schneider, Kurt, Lübke, Daniel und Flohr, Thomas. *Improving an Industrial Reference Process by Information Flow Analysis: A Case Study*. In *Proceedings of PROFES 2007*, Band 4589 von *LNCS*, S. 147–159. Springer-Verlag Berlin Heidelberg, Riga, Latvia. 2007. → S. 4, 293

[Stapel 2008a] Stapel, Kai, Knauss, Eric und Allmann, Christian. *Lightweight Process Documentation: Just Enough Structure in Automotive Pre-Development*. In Rory V. O'Connor, Nathan Baddoo, Kari Smolander und Richard Messnarz, Herausgeber, *Proceedings of the 15th European Conference (EuroSPI '08)*, Communications in Computer and Information Science, S. 142–151. Springer, Dublin, Ireland. Sept. 2008. → S. 293

[Stapel 2008] Stapel, Kai, Lübke, Daniel und Knauss, Eric. *Best Practices in eXtreme Programming Course Design*. In *Proceedings of the 30th International Conference on Software Engineering (ICSE '08)*, S. 769–776. Association for Computer Machinery, ACM Press, Leipzig, Germany. May 2008. → S. 328, 329

[Stapel 2009] Stapel, Kai, Knauss, Eric und Schneider, Kurt. *Using FLOW to Improve Communication of Requirements in Globally Distributed Software Projects*. In *Workshop on Collaboration and Intercultural Issues on Requirements: Communication, Understanding and Softskills (CIRCUS '09)*. Atlanta, USA. November 2009. → S. 293, 302, 340, 342

[Stapel 2010] Stapel, Kai, Knauss, Eric, Schneider, Kurt und Becker, Matthias. *Towards Understanding Communication Structure in Pair Programming*. In Alberto Sillitti, Herausgeber, *Proceedings of the 11th International Conference on Agile Software Development (XP '10)*, Band 48 von *LNBIP*, S. 117–131. Springer, Trondheim, Norway. June 2010. → S. 307

[Stapel 2011a] Stapel, Kai, Knauss, Eric, Schneider, Kurt und Zazworka, Nico. *FLOW Mapping: Planning and Managing Communication in Distributed Teams*. In *Proceedings of 6th IEEE International Conference on Global Software Engineering (ICGSE '11)*, S. 190–199. Helsinki, Finland. 2011. → S. 293, 320, 322, 325, 326, 329, 332, 335, 336, 337, 338, 339

[Straus 1994] Straus, Susan G. und McGrath, Joseph E. *Does the medium matter? The interaction of task type and technology on group performance and member reactions.* Journal of Applied Psychology, 79(1):87–97. Feb 1994. → S. 56, 125

[Summers 2003] Summers, Janet und Catarro, Fatima. *Assessment of handwriting speed and factors influencing written output of university students in examinations*. Australian Occupational Therapy Journal, 50(3):148–157. Sep 2003. → S. 111

[Sunassee 2002] Sunassee, Nakkiran N. und Sewry, David A. *A Theoretical Framework for Knowledge Management Implementation*. In *Proceedings of the 2002 Annual Research Conference of the South African Institute for Computer Scientists and Information Technologists on Enablement Through Technology*, SAICSIT '02, S. 235–245. South African Institute for Computer Scientists and Information Technologists. 2002. ISBN 1-58113-596-3. → S. 27

[SWEBOK 2004] IEEE Computer Society. *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE CSPress, 2004 version Auflage. 2004. → S. 91

[Tauroza 1990] Tauroza, Steve und Allison, Desmond. *Speech Rates in British English*. Applied Linguistics, 11(1):90–105. Mar 1990. → S. 110, 111

[Thun 1981] von Thun, Friedemann Schulz. *Miteinander reden 1: Störungen und Klärungen. Allgemeine Psychologie der Kommunikation*. Rowohlt Tb. 1981. → S. 53

[Tulving 2009] Tulving, Endel und Szpunar, Karl K. *Episodic memory*. Scholarpedia, 4(8):3332. 2009. → S. 37, 73

[UMLStruct 2011] *UML Superstructure*. online. Mar 2011. Letzter Abruf: 03.08.2011. URL http://www.omg.org/spec/UML/2.4/Superstructure/Beta2/PDF → S. 39, 40

[VModell 2009] *V-Modell XT 1.3*. Feb 2009. URL http://v-modell.iabg.de/dmdocuments/V-Modell-XT-Gesamt-Deutsch-V1.3.pdf → S. 1, 39, 141, 348, 349, 350

[Watson 2007] Watson-Manheim, Mary Beth und Bélanger, France. *Communication Media Repertoires: Dealing with the Multiplicity of Media Choices*. MIS Quarterly, 31(2):267–293. 2007. → S. 61, 210

[Watzlawick 1969] Watzlawick, Paul, Beavin, Janet H. und Jackson, Don D. *Menschliche Kommunikation - Formen, Störungen, Paradoxien*. Huber, Bern. 1969. → S. 51

[Welch 1947] WELCH, B. L. *The Generalization of 'Student's' Problem when Several Different Population Variances are Involved*. Biometrika, 34(1-2):28–35. 1947. → S. 343

[Whittaker 2003] Whittaker, Steve. *Theories and Methods in Mediated Communication*. In *In*, S. 243–286. Erlbaum. 2003. → S. 56

[Wiig 1996] Wiig, Karl M. *On the Management of Knowledge - Position Statement*. online. Feb 1996. Letzte Aktualisierung: 08.03.2002, letzter Abruf: 16.03.2011. URL http://www.km-forum.org/what_is.htm → S. 27

[WikiCommMedia] Wikipedia, The Free Encyclopedia. *Kommunikationsmittel*. online. 2010. Artikel vom 25.10.2010. URL http://de.wikipedia.org/w/index.php?title=Kommunikationsmittel&oldid=78261436 → S. 54

[Williams 1977] Williams, Ederyn. *Experimental Comparisons of Face-to-Face and Mediated Communication: A Review*. Psychological Bulletin, 84(5):963–976. Sep 1977. → S. 60

[Wilson 2001] Wilson, E. Vance und Connolly, James R. *Effects of Group Task Pressure on Perceptions of Email and Face-to-Face Communication Effectiveness*. In *GROUP '01: Proceedings of the 2001 International ACM SIGGROUP Conference on Supporting Group Work*. Boulder, Colorado, USA. Sep-Oct 2001. → S. 57

[Wohlin 2000] Wohlin, Claes, Runeson, Per, Höst, Martin, Ohlsson, Magnus C., Regnell, Björn und Wesslén, Anders. *Experimentation in Software Engineering: An Introduction*. The Kluwer International Series in Software Engineering. Kluwer Academic Publishers, Boston/Dordrecht/London. 2000. → S. 7, 9, 18, 241, 244, 245, 264, 327, 363

[Zazworka 2010] Zazworka, Nico, Stapel, Kai, Knauss, Eric, Shull, Forrest, Basili, Victor R. und Schneider, Kurt. *Are Developers Complying with the Process: An XP Study*. In *Proceedings of the 4th International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*. IEEE Computer Society, Bolzano-Bozen, Italy. Sept. 2010. Best Paper. → S. 325

[Ziefle 1998] Ziefle, Martina. *Effects of Display Resolution on Visual Performance*. Human Factors: The Journal of the Human Factors and Ergonomics Society, 40(4):554–568. Dec 1998. → S. 110, 111

[Zimbardo 2003] Zimbardo, Philip G., Gerrig, Richard J., Hoppe-Graff, Siegfried, Engel, Irma und Hoppe-Graff, Siegfried. *Psychologie*. Springer, 7 Auflage. 2003. → S. 32

[Zimmermann 2007] Zimmermann, Andreas und und Reinhard Oppermann, Andreas Lorenz. *An Operational Definition of Context*. In *Modeling and Using Context, 6th International and Interdisciplinary Conference, CONTEXT*, Band 4635 von *Lecture Notes in Computer Science*, S. 558–571. Springer, Roskilde, Denmark. Aug 2007. → S. 46