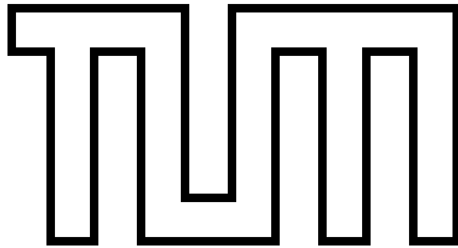
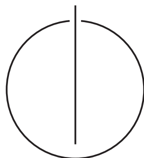


Technische Universität München

Implizite Entwurfsregeln in Softwaresystemen: Entstehung, Erfassung und Konformitätsprüfung

Martin Feilkas



Fakultät für Informatik
Software & Systems Engineering

TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK
Lehrstuhl für Software & Systems Engineering

Implizite Entwurfsregeln in Softwaresystemen: Entstehung, Erfassung und Konformitätsprüfung

Martin Feilkas

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Hans-Joachim Bungartz

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Dr. h.c. Manfred Broy
2. Univ.-Prof. Dr. Helmut Seidl

Die Dissertation wurde am 01.02.2010 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 16.06.2010 angenommen.

Kurzzusammenfassung

Implizite Entwurfsregeln in Softwaresystemen. Das Verstehen eines Programms, um daran Änderungen vornehmen zu können, ist eine sehr anspruchsvolle Aufgabe, mit der sich Softwareentwickler häufig auseinandersetzen müssen. In der Wartungphase ist der Quellcode eines Softwaresystems für Entwickler meist die zuverlässigste Informationsquelle, um dessen Funktionsweise zu verstehen. Problematisch ist hierbei, dass der Quellcode meist einer hohen Anzahl an Entwurfsregeln der Ersteller unterliegt, die nicht direkt im Code ablesbar sind. Diese im Code lediglich implizit vorhandenen Entwurfsregeln umfassen beispielsweise die gewünschte statische Struktur des Systems, den Einsatz bestimmter Entwurfsmuster oder das Zusammenspiel von Methoden einer Schnittstelle (API - **A**pplication **P**rogramming **I**nterface). Im Code lassen sich diese Regeln lediglich über die Wahl der Bezeichner der implementierten Deklarationen erschließen, sind aber weder explizit ablesbar, noch werden sie durch den Compiler oder andere Werkzeuge überprüft. Für einen Wartungsingenieur werden diese impliziten Regeln zu Vorgaben, nach denen er sich richten muss, um weder die Integrität der Architektur und des Entwurfs zu verletzen noch (latente) Fehler zu verursachen. Wenn überhaupt, liegt das Wissen über diese impliziten Vorgaben im Programmcode meist lediglich in Form zusätzlicher Dokumentation vor. Allerdings ist häufig zu beobachten, dass diese unpräzise und veraltet ist. Zudem kann sich ein Wartungsingenieur niemals sicher sein, dass er alle Vorgaben der ursprünglichen Entwickler richtig interpretiert und in seinem Code richtig umgesetzt hat. Oftmals sind implizite Entwurfsregeln nur in den Köpfen der ursprünglichen Entwickler vorhanden. Durch die Fluktuation von Entwicklern gehen diese Informationen im Lauf der Evolution sukzessive verloren.

Explizitmachung und Überprüfung impliziter Entwurfsregeln. Um nicht-adäquaten Änderungen an Softwaresystemen vorzubeugen, ist es notwendig, das Wissen der ursprünglichen Entwickler über die von ihnen definierten Entwurfsregeln über den Lauf der Softwareevolution hinweg in einem Projekt zu erhalten. Es stellt für ein Projekt eine enorme Herausforderung dar, trotz Entwicklerfluktuation das Wissen über diese Regeln durch direkte und indirekte Kommunikation im Projekt zu bewahren. In dieser Arbeit werden Methoden und Techniken vorgestellt, die es ermöglichen, implizite Entwurfsregeln in Softwaresystemen explizit und in automatisch überprüfbarer Form zu erfassen. Dies wird durch die Spezifikation von Constraints in Form von syntaktischen Mustern bewerkstelligt, die festlegen, welche Strukturen in Programmen eingehalten werden müssen. Entwickler können Entwurfsregeln in Form von Modellen oder durch Annotationen im Quellcode ausdrücken. Constraints definieren schließlich deren Abbildung auf den Quellcode, so dass dadurch das Programm auf Konformität mit den Regeln überprüft werden kann. Dies stellt einen Beitrag zum Wissensmanagement von Entwurfsregeln

in Softwareentwicklungsprojekten dar. Ziel ist es dabei, das individuelle Wissen der Entwickler über Entwurfsregeln derart zu erfassen, dass andere Entwickler durch statische Analyse überprüfen können, ob ihr Code konform zu diesen Regeln ist. Nach der Durchführung von Änderungen an einem Programm, kann somit geprüft werden, ob es immer noch den Vorgaben der ursprünglichen Entwickler gerecht wird. Werden derartige Überprüfungen kontinuierlich durchgeführt, kann auf diese Weise die Konformität der Implementierung mit den Entwurfsregeln langfristig sichergestellt werden. Durch das Explizitmachen und die Möglichkeit der automatischen Konformitätsüberprüfung wird die Übergabe von Softwaresystemen an andere Entwickler (etwa Wartungsingenieure) zur Weiterentwicklung dahingehend erleichtert, dass unbeabsichtige Verstöße gegen Architektur- und Entwurfsvorgaben vermieden werden können. Im Gegensatz zur Dokumentation in Textdokumenten werden die Entwickler durch die statische Überprüfung auf Inkonformitäten zwischen den Vorgaben und dem Programm aufmerksam gemacht. Damit kann die Aktualität der Dokumentation sichergestellt werden. Das Explizitmachen impliziter Entwurfsregeln stellt somit eine präventive Maßnahme zur Verbesserung der Verständlichkeit und Wartbarkeit von Softwaresystemen dar und beugt aufwändigem Reverse Engineering zur Wiedergewinnung dieser verborgenen Informationen vor.

Validierung und praktische Anwendung. Im Rahmen von zwei Fallstudien werden die zentralen Thesen dieser Arbeit evaluiert. In der ersten Fallstudie wird aufgezeigt, in welcher Form und welcher Quantität implizite Vorgaben in der bekannten Java-Standard-API verborgen sind und wie diese Informationen explizit gemacht werden können. Darüber hinaus werden Beispiele nicht-adäquater Verwendungen dieser API in realen Programmen (Eclipse) gezeigt, die bestätigen, dass die rein prosaische Dokumentation nicht ausgereicht hat, um die Vorgaben zu den API-Nutzern zu kommunizieren. In der zweiten Fallstudie wird auf Basis von drei industriellen Projekten die Abweichung zwischen der Dokumentation und der Implementierung der Architektur (statische Struktur) untersucht, die im Lauf der Softwareevolution aufgetreten ist. Es wird quantifiziert, in welchem Umfang die Dokumentation und der Code zueinander konform gehalten wurden. Darüber hinaus wird durch Interviews mit den Entwicklern herausgearbeitet, inwieweit die Inkonsistenzen zwischen der Dokumentation und dem Code Verletzungen der vorgesehenen Architektur im Code darstellen oder als Unzulänglichkeiten der Dokumentation einzuordnen sind. Anhand der Ergebnisse dieser Studie wird gezeigt, dass sich die Dokumentation in Form von Textdokumenten, die nicht mit dem Code auf Konformität abgeglichen werden, als unzureichend erweist, um architektonische Vorgaben dauerhaft durchzusetzen.

Danksagung

An erster Stelle möchte ich Herrn Prof. Dr. Dr. h.c. Manfred Broy dafür danken, dass er es mir ermöglicht hat, diese Dissertation in seiner Gruppe anzufertigen, für die fruchtbaren Diskussionen und die positive Arbeitsatmosphäre. Auch Herrn Prof. Dr. Helmut Seidl gebührt mein Dank für die Übernahme des Zweitgutachtens und seine nützlichen Hinweise zu meiner Arbeit.

Besonderer Dank gebührt meinem Kollegen Daniel Ratiu, der meine Arbeit durch zahlreiche Diskussionen und interessante Gespräche enorm befruchtet hat. In gleicher Weise möchte ich mich bei Elmar Jürgens für die gute Zusammenarbeit in mehreren Projekten und auch für viele interessante Diskussionen und zahlreiche Denkanstöße bedanken, die sich auch in dieser Arbeit niedergeschlagen haben. Auch Florian Deißenböck und Benjamin Hummel möchte ich ganz herzlich für die immer wieder interessante Zusammenarbeit danken. Stefan Wagner, Daniel Ratiu und Wolfgang Schwitzer möchte ich für das Korrekturlesen meiner Arbeit und deren wertvolle Hinweise herzlich danken.

Auch bei den Mitarbeitern der Münchener Rück, insbesondere Herrn Dr. Rainer Janßen und Herrn Rudolf Vaas, gebührt mein Dank für die hervorragende Zusammenarbeit. Die Möglichkeit, Fallstudien in professionellem Umfeld in der Praxis durchführen und die Ergebnisse im Rahmen dieser Arbeit publizieren zu dürfen, hat meine Forschungsarbeit große Schritte vorangebracht.

Nicht weniger bin ich meinen Kollegen Bernhard Schätz und Florian Hölzl für die immer angenehme Zusammenarbeit und die vielen dadurch gewonnenen Einsichten zu Dank verpflichtet. Auch meinen Kollegen Martin Fritzsche, Andreas Fleischmann, Markus Herrmannsdörfer, Christian Pfaller, Wassiou Sitou, Maria Spichkova, Sabine Rittmann, Judith Thyssen, David Trachtenherz möchte ich herzlich für die gute Zusammenarbeit in den Projekten DENTUM und SPES danken. Dank gebührt auch Silke Müller, Sebastian Winter und vielen anderen Kollegen, die enorm zur sehr guten Atmosphäre am Lehrstuhl beitragen.

Ganz besonders möchte ich meiner Familie, meiner Frau Anika, meinen Eltern Johann und Johanna, sowie meiner Schwester Andrea und deren Ehemann Christian danken. Euer Zuspruch war gerade in den schwierigeren Phasen dieser Arbeit eine sehr wertvolle Hilfe.

Inhaltsverzeichnis

1. Einführung und Überblick **9**
1.1. Forschungshypothesen 11
1.2. Ansatz und Beiträge 11
1.3. Überblick 15

I. Implizite Entwurfsregeln in der Softwareevolution **17**

2. Motivation: Auswirkungen impliziter Entwurfsregeln in Programmen **19**
2.1. Intuition: Entstehung impliziter Entwurfsregeln in der Softwareentwicklung . . 20
2.1.1. Die Psychologie der Programmentwicklung 21
2.1.2. Adäquate Verwendung von Deklarationen 24
2.2. Störungen des Informationsflusses in Softwareentwicklungsprojekten 27

3. Grundlegende Techniken und Methoden **31**
3.1. Klassisches Wissensmanagement 31
3.2. Explizitmachung impliziter Entwurfsregeln durch Dokumentation 34
3.2.1. Arten von Dokumentation 34
3.2.2. Zweck von Dokumentation 37
3.3. Statische Programmanalyse 39
3.4. Entwicklung von Sprachen 41

4. Entstehung impliziter Entwurfsregeln **45**
4.1. Modellbildung: Von der Konzeption zum System 45
4.1.1. Konzeptmodelle 45
4.1.2. Implementierungsmodelle 48
4.1.3. Von Konzept- zu Implementierungsmodellen 50
4.2. Entstehung impliziter Entwurfsregeln in der Modellbildung 56
4.2.1. Datenmodellierung 58
4.2.2. Verhaltensmodellierung 60
4.2.3. Strukturmodelle 63
4.3. Reverse Engineering von Konzeptmodellen 65

II. Formalisierung impliziter Entwurfsregeln als Constraints **69**

5. Sicherstellung der adäquaten Verwendung von Deklarationen durch Constraints **71**
5.1. Anatomie von Sprachen 71

5.2. Constraints über der Verwendung von implementierten Konzepten 74

6. Methodische Anwendung von Constraints in Softwareprojekten 79
6.1. Repräsentation von Constraints in Entwicklungsartefakten 79
6.1.1. Modelle als graphische Repräsentation von Constraints 79
6.1.2. Deklaration von Constraints durch Programmannotationen 81
6.2. Einsatz von Constraints im Entwicklungsprozess 82
6.2.1. Der Ursprung von (impliziten) Vorgaben in einem Projekt 82
6.2.2. Lebenszyklus von Constraints 84
6.3. Die Grenze zwischen syntaktischer und semantischer Überprüfung impliziter Vorgaben 86

7. Werkzeugunterstützung zur effizienten Überprüfung von Constraints 91
7.1. Anforderungen an die Werkzeugunterstützung 91
7.2. Architektur und Funktionsweise 92
7.3. Wiederverwendung von Constraints 95

8. Verwandte Methoden und Techniken 97
8.1. Konservierung von Wissen 97
8.2. Dynamische Ansätze 97
8.3. Constraint-Sprachen 98
8.4. Statische Analysen 98
8.5. Erweiterbare Typsysteme 99

III. Praktische Anwendung von Constraints 101

9. Anwendungsbereich 1: Implizite Vorgaben in objektorientierten APIs 103
9.1. Typische Vorgaben in der Java API 104
9.1.1. Vorgaben über den Systemzustand 105
9.1.2. Vorgaben über die Kommunikation/Parametrierung 106
9.1.3. Vorgaben über Unterklassen 107
9.1.4. Vorgaben über den Geltungsbereich/Sichtbarkeit 108
9.2. Klassifikation von Constraints 109
9.2.1. Kontrollfluss-Constraints 109
9.2.2. Datenfluss-Constraints 110
9.2.3. Vererbungs-Constraints 112
9.2.4. Strukturelle Constraints 113
9.3. Quantifizierung der Ergebnisse 115
9.3.1. Verbreitung von impliziten Vorgaben 115
9.3.2. Restriktivität von Constraints 116

10. Anwendungsbereich 2: Implizites Architekturwissen 119
10.1. Motivation 119
10.2. Implizites Architekturwissen 121
10.3. Technik und Methodik der Architekturanalyse 122

10.3.1. Architekturkonformitätsanalyse 122
10.3.2. Methodisches Vorgehen 125
10.4. Die Fallstudie 127
10.4.1. Forschungsfragen 127
10.4.2. Experimentieller Aufbau 128
10.4.3. Quantitative Ergebnisse 130
10.4.4. Beantwortung der Forschungsfragen und Interpretation der Ergebnisse . 132
10.5. Erfahrungen 134
10.6. Einschränkungen der Aussagekraft 136
10.7. Verwandte Arbeiten 137
10.8. Zusammenfassung und Fazit 138

11. Anwendungsbereich 3: Implizite Vorgaben im Softwareentwurf 141
11.1. Das Observer-Pattern 142
11.2. Das Visitor-Pattern 144
11.3. Systemspezifische Muster 147

IV. Zusammenfassung und Ausblick 149

12. Zusammenfassung 151

13. Ausblick: Erweiterbare Sprachen 153

Glossar 161

1. Einführung und Überblick

Implizite Entwurfsregeln. Der Wortschatz eines Programms wird durch die verwendete Programmiersprache und die Deklarationen festgelegt, die durch die Entwickler in der Programmiersprache verfasst werden. Im Fall der Sprache Java bilden Deklarationen von Methoden, Klassen, Paketen etc. die wesentlichen Bausteine eines Programms. Deklarationen sollen meist nicht auf jede durch die Programmiersprache zugelassene Art und Weise genutzt werden. Sie unterliegen meist Beschränkungen, die durch Architektur- und Entwurfsentscheidungen sowie durch Konventionen bedingt sind. Dadurch werden Regeln festgelegt, die den Nutzern einer Deklaration Vorgaben auferlegen, die im Nutzercode eingehalten werden müssen. Nach Art der Deklaration kann man diese Entwurfsregeln wie folgt klassifizieren:

- Entwurfsregeln bzgl. der statischen Struktur: Diese Art von Regeln gibt dem Nutzer einer Deklaration vor, von welchen Teilen des Systems eine Deklaration genutzt werden sollte (z.B. um die Unabhängigkeit bestimmter Komponenten durchzusetzen).
- Entwurfsregeln bzgl. des Datenflusses: Diese Kategorie entspricht Einschränkungen der Argumente, die einer Methode übergeben werden sollten (z.B. muss ein Argument aus einer Menge von Konstanten ausgewählt werden).
- Entwurfsregeln bzgl. des Kontrollflusses: Diese Entwurfsregeln bestimmen, dass etwa ein Aufruf einer Methode nur in bestimmten Konstellationen des Kontrollflusses stattfinden sollten (z.B. das System muss in einem bestimmten Zustand sein).
- Entwurfsregeln bzgl. Vererbungsbeziehungen (Implementierung abstrakter Datentypen): Diese Art von Regeln sind Vorgaben an den Entwickler einer Unterklasse, bestimmte Restriktionen zu beachten (z.B. eine bestimmte Methode der Oberklasse aufzurufen).

Derartige Vorgaben entstehen durch architektonische Entscheidungen (vgl. [HRY95, vGB02]), den Einsatz von Entwurfsmustern (vgl. [HHHL03, vGBB05, Bro96]) oder durch die Art und Weise, wie eine Schnittstelle gestaltet wird. Die Nutzung einer Deklaration wird als adäquat bezeichnet, wenn sie diesen Vorgaben gerecht wird.

Um Änderungen und Erweiterungen an einem existierenden Softwaresystem durchführen zu können, ohne eine Verschlechterung des Designs zu verursachen und ohne Fehler in das System einzubringen, muss ein Entwickler über Informationen über die dem System zugrunde liegenden Entwurfsregeln verfügen. Nur ein Teil dieser Regeln sind jedoch im Code sichtbar, viele liegen lediglich implizit vor und sind nicht direkt ablesbar. Anquetil [AdOdSD07] bezeichnet die Softwarewartung aus diesem Grund primär als eine Herausforderung des Wissensmanagements. Obwohl der Code scheinbar eine vollständige Beschreibung eines Programms darstellt, sind darin nicht alle für einen Entwickler relevanten Informationen zu finden. Implizite Entwurfsregeln können lediglich über die Wahl bestimmter Bezeichner erkannt werden (vgl. [JM97]). Dadurch

muss die adäquate Verwendung (Verwendung im Sinne der Entwurfsregeln) einer Deklaration gewissermaßen 'erraten' werden.

Kommunikation impliziter Entwurfsregeln. Innerhalb eines Projekts wird das Wissen über implizite Entwurfsregeln zwischen den Entwicklern durch direkte Kommunikation oder durch Dokumentation ausgetauscht. Die Aufrechterhaltung dieses Informationsflusses und die Erhaltung der Informationen über implizite Entwurfsregeln im Verlauf der Evolution der Software, stellt eine große Kommunikationsaufgabe dar [RL04, AdOdSD07]. Es gibt verschiedenste Gründe, die den Informationsfluss zwischen Entwicklern in einem Projekt beeinträchtigen:

- Große Projekte: Der Kommunikationsbedarf zwischen den Entwicklern nimmt zu, je größer die Anzahl an Entwicklern ist.
- Verteilte Entwicklung: Aufgrund einer Aufteilung in Unterprojekte (Teams) oder sogar einer geographischen Verteilung wird die Kommunikation erschwert.
- Langfristige Wartung und Evolution: In der Initialentwicklung ist es oftmals noch möglich, Kollegen nach bestimmten Informationen zu fragen. In der Wartung ist man meist auf die in Artefakten integrierten Informationen angewiesen.
- Bibliotheken und Frameworks: Bei der Nutzung von Komponenten, die durch Drittanbieter bereitgestellt werden, ist meist keine direkte Kommunikation mit den Erstellern möglich. In vielen Fällen sind die Ersteller sogar vollkommen unbekannt (z.B. bei Open Source Software).

Auswirkungen auf die Softwareevolution. In der Literatur wird häufig darauf verwiesen, dass Programme während der Wartungsphase einem sukzessiven Verfall unterliegen, der ihre Wartbarkeit und Verständlichkeit verschlechtert [Leh80, BL76, Par94, EGK+01, LST78], da Wartungsingenieure im Zuge von Änderungen an den Programmen verschiedene Defizite in das Softwaresystem einführen: Diese Defizite reichen von unpräzisen und inhomogen verwendeten Bezeichnern [DP06] oder einer Verschlechterung der Programmstruktur (vgl. [Dvo94]), bis hin zu (latenten) internen Fehlern. In [DL99] wurde im Rahmen eines Experiments gezeigt, dass 34% aller Änderungen an einem System, mit dem Wartungsingenieure nicht vertraut waren, mit (latenten) Fehlern behaftet waren. Selbst kleine und einfache Änderungen zogen eine alarmierende Anzahl an Fehlern nach sich. Der Grund für diese erheblichen Verschlechterungen von Programmen im Zuge von Wartungsarbeiten liegt oftmals an der Nichtbeachtung von impliziten Entwurfsregeln über die ursprünglichen Strukturen und Konventionen in den Programmen.

Brooks schreibt in [Bro01], dass gerade der enorme Aufwand des Erlernens und Verstehens von Softwareprojekten personelle Fluktuationen zu einem großen Problem werden lassen. Boehm [Boe91] schreibt, dass eine geringe Rate an personeller Fluktuation eine Voraussetzung ist, um die Risiken eines Softwareprojekts in den Griff zu bekommen. Im Fall von langlebiger Software ist dies jedoch oftmals nicht möglich. Es ist gängige Praxis, dass Entwicklerteams nach der initialen Fertigstellung eines Systems ausgetauscht werden (insbesondere bei beauftragten Entwicklungsprojekten) und somit andere Entwickler für die Wartung und Weiterentwicklung

des Softwaresystems eingesetzt werden. Aus diesem Grund werden Methoden und Techniken benötigt, die es ermöglichen, die Informationen über die Strukturen in der Implementierung eines Softwaresystems in expliziter Form zu erfassen, so dass auch Entwickler, die nicht mit dem System vertraut sind, nicht gegen Entwurfsregeln verstoßen. Der Bedarf an expliziter Information umfasst sowohl die Entwurfs- und Architekturstrukturen im System als auch die adäquate Nutzung von systeminternen Schnittstellen. Zur Konservierung dieser Regeln ist es notwendig, diese Information explizit zu erfassen und geeignet zu repräsentieren. Implizite Entwurfsregeln lediglich in Form von losen Textdokumenten und Diagrammen zu dokumentieren ist jedoch meist nicht ausreichend, um zu verhindern, dass Entwickler Verstöße gegen diese Vorgaben im Code verursachen [LVD06, OR92]. Aus diesem Grund sollte das Wissen der Initialentwickler über implizite Entwurfsregeln in nachprüfbarer Form vorliegen, um künftige Wartungsingenieure in die Lage zu versetzen, ihre Modifikationen auf Konformität mit den ursprünglichen Designprinzipien und der Intention der Initialentwickler zu prüfen.

1.1. Forschungshypothesen

In dieser Arbeit wird der Verlust an Information über implizite Entwurfsregeln in Softwaresystemen im Lauf der Evolution untersucht. Es werden Methoden und Techniken präsentiert, die eine Konservierung der Konformität eines Softwaresystems bezüglich der enthaltenen impliziten Entwurfsregeln ermöglichen. Die gleichen Techniken können auch für Bibliotheken eingesetzt werden, um die in der Schnittstelle versteckten impliziten Vorgaben explizit zu machen und Nutzerprogramme auf Einhaltung der impliziten Vorgaben zu überprüfen. Dies wird durch die Formulierung von Einschränkungen der Nutzung (Constraints) von in Programmiersprachen implementierten Deklarationen erreicht. Die wichtigsten Hypothesen, die in dieser Arbeit untersucht werden, lauten:

1. In Programmen existieren implizite Vorgaben bezüglich deren Entwurfs- und Architekturstruktur sowie bezüglich der adäquaten Verwendung von Schnittstellen.
2. Ein Teil des Wissens der Initialentwickler über implizite Entwurfsregeln geht im Laufe der Evolution eines Softwaresystems verloren, wenn dem nicht entgegen gewirkt wird.
3. Entwickler, die nicht mit den impliziten Entwurfsregeln in einem System vertraut sind, führen Verletzungen der Architektur (statische Struktur) in Programme ein und verursachen (latente) Fehler aufgrund von nicht-adäquater Nutzung von Schnittstellen.
4. Das Explizitmachen impliziter Entwurfsregeln durch Constraints und die kontinuierliche Überprüfung der Konformität zwischen den Constraints und dem Code hilft, Verstößen gegen implizite Entwurfsregeln vorzubeugen.

1.2. Ansatz und Beiträge

Lösungsansatz. Die im Rahmen dieser Arbeit betrachteten Regeln in Programmen sind Informationen, welcher Entwurfsstruktur ein System folgt, wie dessen Architektur aussehen sollte

und auf welche Art und Weise Schnittstellen in einem System oder einer Bibliothek genutzt werden sollten. Diese Entwurfsregeln werden als implizit bezeichnet, falls Entwickler aufgrund mangelnder Überprüfungsmechanismen unwissentlich Verstöße hervorrufen können.

Entstehung impliziter Entwurfsregeln. Im Rahmen dieser Arbeit wird untersucht, wie Informationen, die etwa in Modellen des Softwareentwurfs enthalten sind, lediglich in Form impliziter Entwurfsregeln in eine entsprechende Implementierung Einzug finden. Die Ursache dafür, dass Deklarationen oftmals mit impliziten Entwurfsregeln behaftet sind, liegt darin, dass die Abstraktionsmechanismen von Programmiersprachen nur eine schwache Approximation der Vorstellungen des Erstellers repräsentieren. Die Deklarationen in einem Programm und deren Relationen modellieren die darin ausgedrückten Vorstellungen des Erstellers nur unzureichend. Dieses Phänomen wird in dieser Arbeit als Übergang zwischen einem mentalen Konzeptmodell zu Implementierungsmodellen (Artefakte des Entwicklungsprozesses) definiert und daran untersucht, wie implizite Entwurfsregeln entstehen.

Erfassung impliziter Entwurfsregeln durch Constraints. Implizite Entwurfsregeln müssen vom Ersteller einer Deklaration zu den Entwicklern kommuniziert werden, die diese Deklaration nutzen, um Verstößen gegen die adäquate Nutzung vorzubeugen. Der grundlegende Ansatz, der in dieser Arbeit vorgestellt wird, basiert darauf, bereits im Forward Engineering implizite Entwurfsregeln explizit zu erfassen. Hierfür werden implizite Regeln in Form von Constraints (Einschränkungen der Syntax von Nutzercode) ausgedrückt. Diese Constraints beschreiben Vorgaben, die eine Nutzung einer Deklaration in einem Programm (Deklaration einer Methode, eines Namensraums, einer Klasse etc. – je nach verwendeter Programmiersprache) einhalten muss. Diese Constraints sind automatisiert überprüfbar, so dass Entwickler gewarnt werden, falls sie gegen die Vorgaben seitens des Erstellers einer Deklaration verstoßen (nicht-adäquate Verwendung).

Überprüfung der Konformität. Die Explizitmachung impliziter Entwurfsregeln ermöglicht es Entwicklern, wichtige Informationen über ein System nachzulesen. Dennoch kann alleine durch die Explizitmachung eine unbewusste Verletzung von Entwurfsregeln im Code nicht verhindert werden. Aus diesem Grund werden in dieser Arbeit Techniken vorgestellt, um die Nutzung einer Deklaration durch Nutzercode auf Konformität zu den durch Constraints explizit gemachten Entwurfsregeln zu überprüfen. Dadurch werden Entwickler, die eine Deklaration auf nicht-adäquate Weise nutzen, durch unterstützende Werkzeuge darauf aufmerksam gemacht, dass sie Verstöße gegen Entwurfsregeln des Ersteller verursachen. Damit wird dem Ersteller einer Deklaration durch Constraints ein zusätzliches Ausdrucksmittel gegeben, mit den Nutzern (indirekt) zu kommunizieren und damit der vorgesehenen Nutzung Ausdruck zu verleihen.

Wissenschaftliche Beiträge. Die wichtigsten Beiträge dieser Arbeit sind:

- *Entstehung impliziter Entwurfsregeln.* Anhand einer Formalisierung wird die Ursache für die Entstehung impliziter Entwurfsregeln untersucht. Es wird gezeigt, wie die Vorstellungen eines Entwicklers in Modellierungssprachen und in Programmiersprachen ausgedrückt werden, welche Eigenschaften bei diesem Übergang wünschenswert sind und inwiefern dabei ein Verlust an Explizitheit verursacht wird.

- *Definition eines formalen Frameworks zur Spezifikation von Constraints.* Es wird vorgestellt, inwiefern implizite Entwurfsregeln in einem Softwaresystem als Einschränkungen der Syntax des Nutzercodes von Deklarationen verstanden werden können. Es wird ein formales Framework eingeführt, das es ermöglicht, Constraints als kontextsensitive Bedingungen in Softwaresystemen und APIs zu spezifizieren. Dadurch können implizite Entwurfsregeln explizit formuliert werden.

- *Methodische Anwendung von Constraints und Werkzeugunterstützung.* Es wird dargestellt, wie Constraints in Projekten eingesetzt werden können und aufgezeigt, wo die Grenzen dieser Technik liegen. Darüber hinaus werden sowohl Anforderungen an ein Werkzeug zur statischen Überprüfung von Constraints als auch eine prototypische Realisierung eines derartigen Werkzeugs präsentiert.

Diese Techniken und Methoden werden im Rahmen dieser Arbeit angewandt, um den Verlust an Explizitheit an realen Softwaresystemen nachzuweisen. Es werden drei Anwendungsbereiche präsentiert, an denen die praktische Anwendung der Methoden und Techniken demonstriert wird:

- *Anwendungsbereich 1: Implizite Vorgaben in objektorientierten APIs.* Anhand der Java-Standard-API wird quantifiziert, inwiefern darin implizite Vorgaben existieren. Es wird aufgezeigt, wie diese impliziten Vorgaben durch Constraints explizit gemacht werden können. Es wird überprüft, inwieweit diese Vorgaben im Open-Source-Projekt Eclipse und in der Implementierung der Java-Standard-API selbst eingehalten werden. Dies verdeutlicht die Gefahr der inadäquaten Verwendung von Deklarationen in Bibliotheken.

- *Anwendungsbereich 2: Implizites Architekturwissen.* Auf Basis von drei industriellen Projekten wird dargestellt, zu welchem Grad die Implementierung der Systeme konform zu den Architekturvorgaben ist. Daran wird aufgezeigt, dass die Architekturdokumentation nicht in der Lage war, das Architekturwissen über den Lauf der Systemevolution hinweg im Projekt zu halten. Zudem wird erläutert, wie durch eine kontinuierliche Architekturkonformitätsanalyse sowohl die Aktualität der Dokumentation als auch die Konformität des Codes mit den Architekturvorgaben sichergestellt werden kann.

- *Anwendungsbereich 3: Implizite Regeln im Softwareentwurf.* Sowohl anhand von bekannten Entwurfsmustern als auch anhand von systemspezifischen Mustern wird dargestellt, inwiefern deren Einsatz mit impliziten Regeln einhergehen. Darüber hinaus wird gezeigt, wie diese impliziten Entwurfsregeln durch Constraints ausgedrückt werden können.

Teile dieser Beiträge wurden bereits in folgenden wissenschaftlichen Publikationen veröffentlicht: [FR08, FRJ09, RFJ08, RFD+08, Fei07, BFH+10, HF10, BFG+08, FFH+09, Fei06]

Validierung der Hypothesen. Um die in Abschnitt 1.1 vorgestellten Hypothesen zu validieren, werden folgende Nachweise erbracht:

- Auf Basis der Fallstudie zum Verlust an Architekturwissen in der Softwareevolution (Anwendungsgebiet 2) wird nachgewiesen, dass in der Praxis implizite Entwurfsregeln (bzgl.

der Architektur des Softwaresystems) existieren und ein Verlust an Wissen über diese Vorgaben stattfindet. Es wird aufgezeigt, dass die Konformität zwischen Architekturvorgaben und dem Code durch kontinuierliche Überprüfung konserviert werden kann (Hypothesen 1, 2, 3 und 4).

- Anhand einer Formalisierung und Beispielen verschiedenster Modellierungstechniken wird aufgezeigt, wodurch implizite Entwurfsregeln entstehen (Hypothesen 1 und 2).

- Durch eine Fallstudie auf Basis der Java-Standard-APIs (Anwendungsgebiet 1) wird gezeigt, dass in diesen Bibliotheken implizite Vorgaben zu finden sind und dass diese durch Constraints ausgedrückt werden können (Hypothesen 1, 2 und 4). Desweiteren wird im Rahmen dieser Fallstudie dargelegt, dass diese impliziten Vorgaben zu (latenten) Fehlern bei der Nutzung dieser APIs führen (Hypothesen 3).

- Anhand von Beispielen von Entwurfsmustern (Anwendungsgebiet 3) wird gezeigt, dass die adäquate Implementierung und Erweiterung von Entwurfsmustern durch Constraints ausgedrückt und überprüft werden kann (Hypothese 4).

Wissenschaftliche Einordnung. Diese Arbeit leistet Beiträge zur Forschungsrichtung des Programmverstehens (engl. Program Comprehension). Viele Arbeiten in diesem Bereich gehen von der Problemstellung aus, dass ein vorliegendes Programm verstanden werden muss, um Änderungen daran durchführen oder Teile davon in neuem Kontext wiederverwenden zu können. Dadurch wird diese Disziplin meist von Reverse Engineering Fragestellungen dominiert. Die zentrale Herausforderung im Bereich des Reverse Engineerings ist die Extraktion von in Programmen nur noch bruchstückhaft vorhandenen Informationen [CCI90]. Im Rahmen dieser Arbeit wird bereits der Verlust dieser Informationen während der Entwicklung eines Programms betrachtet. Es werden Methoden und Techniken vorgestellt, welche die Konservierung impliziter Entwurfsregeln über die Softwareevolution hinweg unterstützen sowie die Sicherstellung der Konformität zwischen diesen Vorgaben und der Implementierung ermöglichen. Damit soll einer Reverse Engineering Problematik bereits im Vorfeld durch eine verbesserte Konservierung des Wissens der Initialentwickler über Entwurfsregeln im System vorgebeugt werden.

Die in [BR00] aufgezeigte Roadmap für den Bereich der Softwarewartung und -evolution verweist auf die Wichtigkeit des Managements von Wissen und Informationen über ein Softwaresystem. Diese Aspekte jedoch ausschließlich im Kontext der Wartung und Evolution zu sehen, wäre zu kurz gegriffen. Bereits während der Initialentwicklung sollten geeignete Maßnahmen ergriffen werden, um die Evolutionierbarkeit eines Softwaresystems sicherzustellen. Es ist akzeptiert, dass man das Design und die Architektur dahingehend ausrichten sollte, ein möglichst erweiterbares und anpassbares System zu erzeugen [PCW84]. Darüber hinaus ist es jedoch notwendig, das Wissen über die adäquate Verwendung der verschiedenen Elemente einer Implementierung auch in geeigneter Form im Prozess der Wartung und Weiterentwicklung zu konservieren, um zu vermeiden, dass etwa durch Unwissenheit Brüche des ursprünglichen Designs hervorgerufen werden.

1.3. Überblick

Die Arbeit ist in drei Teilen gegliedert. Im ersten Teil wird die Problemstellung des Verlusts der Explizitheit bestimmter Informationen in einem Softwaresystem vorgestellt und dessen Ursachen beleuchtet. Teil I umfasst folgende Kapitel:

- Kapitel 2 führt in die Aufgabenstellung ein, erklärt die zentralen Problemstellungen und schildert deren Auswirkungen. Dabei wird die der Arbeit zugrunde liegende Terminologie definiert.
- Kapitel 3 ordnet die Arbeit in das Wissenschaftsfeld des Wissensmanagements ein und stellt grundlegende Methoden und Techniken vor.
- Kapitel 4 führt ein formales Framework ein, das dazu dient, den Explizitheitsverlust in der Softwareentwicklung präzise zu definieren. Zudem wird anhand von unterschiedlichen Modellierungstechniken der Verlust von Information während der Transition zur Implementierung veranschaulicht.

Teil II stellt ein leichtgewichtiges Verfahren zur Konservierung von impliziten Entwurfsregeln im Code vor. Dieser Ansatz ermöglicht es Entwicklern, implizite Entwurfsregeln über die adäquate Verwendung von Schnittstellen oder die Strukturierung von Programmen explizit auszudrücken. Nutzer von Schnittstellen und Wartungsingenieure können diese Informationen verwenden, um zu validieren, ob der von ihnen geschriebene Code diesen Regeln gerecht wird. Der zweite Teil dieser Arbeit ist in folgende Kapitel untergliedert:

- Kapitel 5 stellt einen Ansatz zur Spezifikation von Constraints vor, welche die Einhaltung eines syntaktischen Kontexts bei der Nutzung von Deklarationen sicherstellt. Weiterhin wird der methodische Umgang mit Constraints sowie ein Werkzeug zur Überprüfung von Constraints vorgestellt.
- Kapitel 6 führt in die methodische Anwendung von Constraints ein.
- Kapitel 7 präsentiert einen Ansatz zur werkzeuggestützten Überprüfung von Constraints.
- Kapitel 8 setzt den in Teil II dieser Arbeit vorgestellten Ansatz zu verwandten Arbeiten in Bezug.

In Teil III werden wichtige Hypothesen dieser Arbeit auf realen Systemen und Bibliotheken überprüft und die im Rahmen der Arbeit eingeführten Techniken evaluiert. Zudem werden die praktische Relevanz der Problemstellung gezeigt und Gründe für den Explizitheitsverlust im Lauf der Softwareentwicklung und -evolution präsentiert.

- Kapitel 9 nutzt den im vorangehenden Kapitel vorgestellten Ansatz, um implizite Vorgaben in der Java-Standard-API zu formulieren und zu überprüfen, inwieweit diese Vorgaben in der Implementierung der Java-Standard-API selbst eingehalten werden. Diese Fallstudie zeigt, inwieweit implizite Regeln in APIs vorherrschen und verdeutlicht, dass die daraus resultierenden impliziten Vorgaben in Entwicklerteams oftmals nicht ausreichend kommuniziert werden.

- Kapitel 10 präsentiert eine industrielle Fallstudie, in welcher der Verlust von Architekturwissen quantifiziert wird. Hiermit wird die praktische Relevanz der Aufgabenstellung verdeutlicht.
- Kapitel 11 überträgt diese Erkenntnisse auf Regeln aus dem Systementwurf. Es wird aufgezeigt, inwieweit Entwurfsmuster implizite Regeln in sich tragen und wie die Konformität zu diesen Mustern durch Constraints abgesichert werden kann.

Teil IV schließt die Arbeit und präsentiert einen Ausblick auf weiterführende wissenschaftliche Forschungsrichtungen:

- Kapitel 12 fasst die Ergebnisse der Arbeit zusammen.
- Kapitel 13 schließt die Arbeit mit einem Ausblick auf zukünftige weiterführende Forschungsthemen. Es wird erläutert, wie die Erweiterung von Sprachen durch Komposition von domänenspezifischen Sprachbausteinen künftig einen schwergewichtigen alternativen Ansatz zur Nutzung von Bibliotheken und Frameworks darstellen und wie dadurch eine explizitere Beschreibung von Systemen ermöglicht werden könnte.

Teil I.

Implizite Entwurfsregeln in der Softwareevolution

2. Motivation: Auswirkungen impliziter Entwurfsregeln in Programmen

"Die Wahrheit ist nützlicher für den, der sie erfährt, als für den, der sie sagt."
- Blaise Pascal

Evolution von Softwaresystemen. In vielen Unternehmen ist Software zu einem wichtigen Faktor geworden, um den effizienten Ablauf von Geschäftsprozessen zu gewährleisten. Die für die individuellen Aufgaben in einem Unternehmen entwickelten Softwaresysteme stellen häufig einen hohen Wert dar, der über den Lebenszyklus der Systeme erhalten werden sollte. Softwaresysteme werden oftmals über einige Jahrzehnte hinweg genutzt. Über ihre Lebensdauer hinweg müssen Systeme häufig um neue Funktionalitäten erweitert und an neue technische Umgebungen angepasst werden. Dieser Prozess wird als die Evolution eines Softwaresystems bezeichnet.

Implizite Entwurfsregeln. Entwickler eines Softwaresystems treffen eine Vielzahl an Festlegungen bezüglich der Gestalt des zu realisierenden Systems. Sie stehen dabei oftmals einem großen Lösungsraum an potentiellen Architektur-, Entwurfs- und Implementierungsalternativen gegenüber. Die Auswahl von geeigneten Strukturen für ein zu erstellendes Softwaresystem stellt eine große intellektuelle Herausforderung dar und erfordert damit meist einen hohen Aufwand. Derartige Entscheidungen legen wesentliche Kriterien fest, denen der Code des zu erstellenden Systems gerecht werden muss. Diese Entwurfsregeln bestimmen die statische Struktur eines Systems, die Nutzung von Entwurfsmustern und die Art und Weise, in der bestimmte Schnittstellen eingesetzt werden. Sie dienen der Bewältigung der Komplexität der Programmentwicklung, wenn sie auch im Programmcode entsprechend strikt umgesetzt werden. Durch derartige Festlegungen werden die systeminternen Strukturen homogener und damit einfacher verständlich. Da ein großer Teil der Aufwände in der Softwarewartung auf das Programmverstehen entfällt [WTMS95, LST78], ist die Erhaltung der Konformität des Codes mit diesen Entwurfsregeln von großer Bedeutung. Die Information über Regeln in Programmen sind ein schützenswertes Gut, das in der Wartungsphase dringend benötigt wird [BL76]. Die Wiedergewinnung verlorener Information über Vorgaben der Architektur und des Entwurfs durch Reverse Engineering Aktivitäten ist sehr aufwändig [WTMS95]. Im Code werden die Festlegungen leider nur sehr unzureichend ausgedrückt, da in heutigen Programmiersprachen nur sehr primitive Mechanismen (z.B. Sichtbarkeiten) existieren, um Architektur- und Entwurfsvorgaben Ausdruck zu verleihen. Dadurch können diese Regeln nicht direkt im Code abgelesen werden. Falls diese nicht zusätzlich dokumentiert werden, die Dokumentation nach

Änderungen nicht aktuell gehalten wird oder die darin enthaltenen Informationen für die Entwickler schlecht verfügbar sind, werden die Entwurfsregeln implizit und sind schließlich (wenn überhaupt) nur noch in den Köpfen der Entwickler vorhanden. Ein Mangel an Explizitheit und Verfügbarkeit der Informationen über die in einem System enthaltenen Entwurfsregeln führt dazu, dass Entwickler unwissentlich Verletzungen der Architektur und des Entwurfs des Systems sowie (latente) Fehler verursachen. Dies bedroht die langfristige Wartbarkeit und Verständlichkeit sowie die Korrektheit eines Systems.

Kommunikation impliziter Entwurfsregeln. Modifikationen und Erweiterungen an einem bestehenden Softwaresystem durchzuführen bedarf eines guten Verständnisses von dessen internem Aufbau und damit über die Entwurfsregeln. Ein Entwickler benötigt tiefgehende Kenntnisse über die für ein System vorgesehene Architektur, die Datenstrukturen und deren Beziehungen, den Kontroll- und Datenfluss. Im Lauf der Evolution eines Softwaresystems treten meist immer wieder personelle Fluktuationen im Entwicklerteam auf. Häufig sind keine oder nur noch wenige der ursprünglichen Entwickler, die zur initialen Entwicklung eines Systems beigetragen haben, während der Wartungsphase des Projekts beteiligt. Da Entwurfsregeln nicht direkt im Code abgelesen werden können und meist nicht automatisch überprüft werden, ist es notwendig, diese Informationen auf anderen Wegen an immer wieder neue Entwickler weiterzugeben. Aus diesem Grund stellt die Softwarewartung eine Herausforderung des Wissensmanagements dar [RL04, AdOdSD07].

Ansatz. Diese Arbeit befasst sich mit der geeigneten Repräsentation von Entwurfsregeln in Softwaresystemen und der langfristigen Sicherung der Konformität der Implementierung mit diesen Vorgaben. Im Rahmen dieser Arbeit wird ein Ansatz präsentiert, wie Entwurfsregeln durch Nutzung bestimmter Modelle oder Annotationen im Code explizit dargestellt werden können. Es wird ein formales Framework präsentiert, das die Definition sogenannter Constraints ermöglicht, die spezifizieren, welche Zusammenhänge im Programmcode gelten müssen, damit dieser konform zu den Entwurfsvorgaben ist. Durch statische Analyse kann schließlich die Konformität des Quellcodes mit den Vorgaben überprüft werden und somit Entwickler vor unwissentlichen Verstößen gewarnt werden.

In den folgenden Abschnitten dieses Kapitels wird zunächst erklärt, wie Entwurfsregeln in Softwaresysteme einfließen. Anschließend wird aufgezeigt, wie derzeit mit Entwurfsvorgaben in Systemen umgegangen wird und welche Auswirkungen dies nach sich zieht.

2.1. Intuition: Entstehung impliziter Entwurfsregeln in der Softwareentwicklung

Im Folgenden wird die Entstehung impliziter Entwurfsregeln in der Programmentwicklung genauer beleuchtet und erklärt, inwiefern diese Regeln als implizite Vorgaben bezüglich der adäquaten Verwendung von Deklarationen in Programmen betrachtet werden können. Damit

soll ein erster intuitiver Einstieg in diese Thematik ermöglicht werden, der in den folgenden Kapiteln vertieft wird.

2.1.1. Die Psychologie der Programmentwicklung

In verbreiteten Vorgehensmodellen zur Durchführung von Softwareentwicklungsprojekten wird zwischen den Phasen der Analyse, des Entwurfs und der Implementierung unterschieden. In der Analysephase werden zunächst Informationen über Anforderungen und die Anwendungsdomäne erfasst. Während des Entwurfs werden diese Erkenntnisse genutzt, um die Funktionsweise des künftigen Systems zu entwickeln. Basierend auf dem in diesen Phasen gewonnenen Wissen über das System wird schließlich während der Implementierungsphase der Code verfasst. Abbildung 2.1 illustriert den Zugewinn an Wissen über ein System im Lauf der frühen

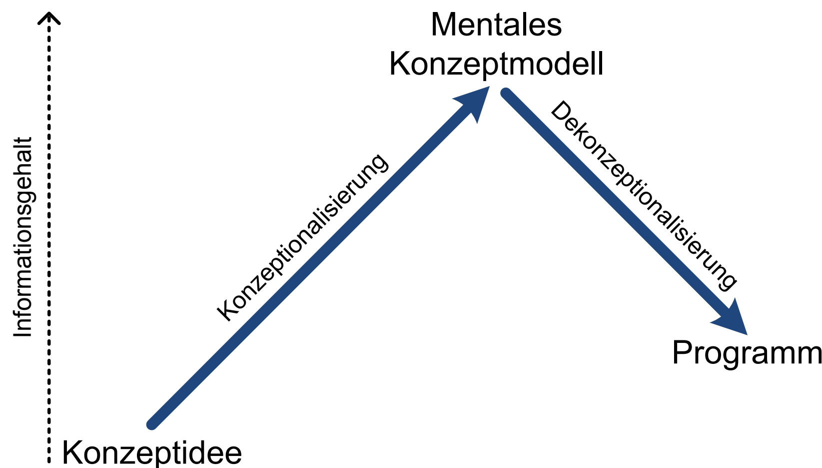


Abbildung 2.1.: Die Psychologie des Programmierens: Von der Konzeptidee zum Programm

Phasen der Softwareentwicklung (idealisiertes Wasserfallmodell, in dem das gesamte Wissen über ein System bereits vor der Implementierung feststeht). Ausgehend von einer Konzeptidee, welche die grundlegenden Ziele beinhaltet, die mit der Erstellung des Systems verbunden sind, muss sukzessiv Wissen hinzugewonnen werden, bis die Entwickler eine gemeinsame Vorstellung von dem zu erstellenden System haben. Dieses Wissen über das zu realisierende System umfasst Festlegungen bezüglich der Architektur und des Entwurfs über die Gestaltung von Schnittstellen bis hin zu Implementierungsentscheidungen, welche die Gestalt der benötigten Deklarationen festlegen. Im Rahmen dieser Arbeit wird dieses Wissen als Konzeptmodell bezeichnet. Ein Konzeptmodell ist ein mentales Modell, das die Vorstellungen der Entwickler repräsentiert. Jedes persistierte Artefakt (Modelle, Dokumentation, Code etc.) wird als Implementierungsmodell bezeichnet, da es ein Abbild eines Teils des Wissens der Entwickler über das System (Konzeptmodell) widergibt. Die rechte Seite von Abbildung 2.1 zeigt, dass der Informationsgehalt des Programmcodes unter dem des Konzeptmodells liegt. Der Grund hierfür

liegt darin, dass nur ein Teil der Informationen über das System im Code ablesbar ist. Etwa Architektur- und Designvorgaben gehen nicht daraus hervor. Die Schaffung jedes Artefakts kann als Übergang von einem Konzept- zu einem Implementierungsmodell verstanden werden. Da in der Wartungsphase der Code eines Systems die zentrale Rolle spielt [LVD06], steht im Rahmen dieser Arbeit der Verlust an Wissen beim Übergang zum Programmcode im Vordergrund.

Konzeptmodelle. Im Rahmen dieser Arbeit wird die Repräsentation von Konzepten in Programmiersprachen untersucht, um die Entstehung impliziter Entwurfsregeln in Softwaresystemen aufzeigen zu können. Konzeptmodelle werden hierfür in Form von Multigraphen formalisiert, welche das Wissen eines Entwicklers über ein System darstellen. Die Knoten und Kanten dieses Graphen werden hierbei mit natürlich-sprachlichen Bezeichnern beschriftet. Die Knoten werden im Rahmen dieser Arbeit als Konzepte bezeichnet. Konzepte repräsentieren hierbei die Vorstellung eines Entwickler von einem bestimmten atomaren Aspekt des zu erstellenden Systems. Der Begriff des Konzepts wird in der Literatur sehr unterschiedlich eingesetzt. Die Definition eines intellektuellen/mentalen Konzepts entspricht im Rahmen dieser Arbeit der von Bjørner [Bjø06], die in sehr ähnlicher Form auch im Bereich der Konzeptmodellierung (engl. Conceptual Modeling) anzutreffen ist [Myl92]:

Unter einem intellektuellen Konzept versteht man eine Abstraktion, die normalerweise nicht greifbar ist, wie eine Idee, eine Vorstellung, ein Gedankenkonstrukt.

Original: By an intellectual concept we understand an abstraction, something usually not manifest, an idea, a notion, a thought construct.

In [RW02] geben Rajlich und Wilde eine Definition des Konzept-Begriffs, die konform zur Definition von Bjørner ist:

Konzepte sind Einheiten von menschlichem Wissen, die durch den menschlichen Verstand (Kurzzeitgedächtnis) in einer Instanz verarbeitet werden können.

Original: Concepts are units of human knowledge that can be processed by the human mind (short-term memory) in one instance.

Der Begriff des Konzepts ist gerade im Bereich der ‘Concept Location’ (dem Auffinden von Konzepten im Code) sehr zentral, einem wichtigen Teilbereich der Forschungsrichtung des Programmverstehens. Das Ziel der Arbeiten in diesem Gebiet ist das Erkennen von Regionen im Code, die von einer Anforderung oder einer Änderungsanfrage betroffen sind [BMW93]. Konzepte werden in diesem Bereich (wie auch in dieser Arbeit) meist durch Begriffe in natürlicher Sprache definiert, wie sie etwa in der informellen Beschreibung einer Anforderung oder einer Änderungsanfrage auftreten [MSRM04, MRB+05].

Eine stark vereinfachende Definition, die jedoch weit verbreitet ist, setzt Konzepte mit Objekten im Sinne der objektorientieren Programmierung gleich. Während Klassendeklarationen in gut strukturierten Programmen Konzepte darstellen können, gilt der Umkehrschluss jedoch nicht, dass Konzepte ausschließlich in Form von Klassen implementiert werden [RW02]. Für den in dieser Arbeit verfolgten Zweck wäre eine derartige Definition nicht geeignet, da sie

keine Trennung von der gedanklichen Welt eines Entwicklers (dessen Vorstellung vom finalen System) und der Syntax einer (objektorientierten) Programmiersprache schafft. Um diese Trennung herzustellen, wird in dieser Arbeit der Prozess der Implementierung als Übergang von einem *Konzeptmodell* zu einem oder mehreren *Implementierungsmodell(en)* definiert.

Das Konzeptmodell stellt eine Repräsentation der Vorstellungen der Entwickler dar. Ein Implementierungsmodell entspricht einem Artefakt, das im Lauf eines Entwicklungsprozesses entsteht. Durch eine Gegenüberstellung der in einem Implementierungsmodell enthaltenen Informationen mit dem Konzeptmodell können Rückschlüsse darüber getroffen werden, welche Informationen im Implementiertungsmodell lediglich implizit vorliegen (im Konzeptmodell enthaltene Konten und Kanten, die im Implementierungsmodell nicht erkennbar sind). Im Rahmen dieser Arbeit werden hauptsächlich Programme als Implementierungsmodelle betrachtet. In einem Programm werden die Konzepte und ihre Relationen in einer Programmiersprache in Form von Deklarationen und Nutzungen dieser Deklarationen dargestellt. Deklarationen schaffen Einheiten in einem Programm, die über Bezeichner von mehreren Stellen im Programm sichtbar und nutzbar sind. Dadurch definieren sie das Vokabular des Programms, das als Implementierungsmodell formalisiert werden kann.

Unter dem Begriff 'Schema' wurden Konzeptmodelle auch schon in anderen Wissenschaftsbereichen, wie der Künstlichen Intelligenz (z.B. 1974 von Minsky [Min74]) und der kognitiven Psychologie (z.B. 1979 von Bower et al. [BDW79] oder auch [Tve93]), untersucht und genutzt [Rob99].

Obwohl Konzeptmodelle eines Softwaresystems während des Entwicklungsprozesses erarbeitet werden, ist zu beachten, dass sie kein physisches Artefakt des Entwicklungsprozesses darstellen. Teile der in ihnen enthaltenen Informationen sind jedoch in vielen Artefakten auffindbar. Im Rahmen dieser Arbeit werden Konzeptmodelle ausschließlich zur Illustration der Entstehung und zur Definition von impliziten Entwurfsregeln verwendet.

Dekonzeptionalisierung. Abbildung 2.2 stellt die Ursache des Problems des Verlusts an Explizitheit dar. Während der Analyse und des Entwurfs erarbeiten sich Entwickler eine immer genauere Vorstellung von der Funktionsweise, der Struktur und der Konzeption eines Systems. Im Zuge der Implementierung muss dieses gemeinsame Konzeptmodell durch Konstrukte der gewählten Programmiersprache ausgedrückt werden. Deklarationen spielen hierbei eine zentrale Rolle: Sie ermöglichen es, die Konzepte durch einen symbolischen Bezeichner in einem Programm zu repräsentieren. So werden Konzepte über Klassen, Methoden, Funktionen, Variable etc. dargestellt und sind meist (wenn überhaupt) nur durch deren Bezeichner dem Konzept zuordenbar (vgl. [RD07]). Dieser Prozess wird im Rahmen dieser Arbeit als *Dekonzeptionalisierung* bezeichnet. Während dieses Übergangs von einem Konzeptmodell zu einem Programm fließen einige Informationen nicht in den Programmcode ein. Die ursprünglichen Konzepte und deren Beziehungen sind ohne Vorwissen nicht mehr direkt, ohne Kenntnis über den Kodierungsprozess, erkennbar. Ein Grund hierfür ist etwa die "Tyrannei der dominanten Dekomposition" [TOHS99]: Viele Deklarationen müssen aufgrund des Aufbaus heutiger Programmiersprachen in eine Baumstruktur gepackt werden, obwohl die durch sie modellierten Konzepte oftmals in komplexeren Beziehungen zueinander stehen. Diese Beziehungen sind jedoch nicht explizit

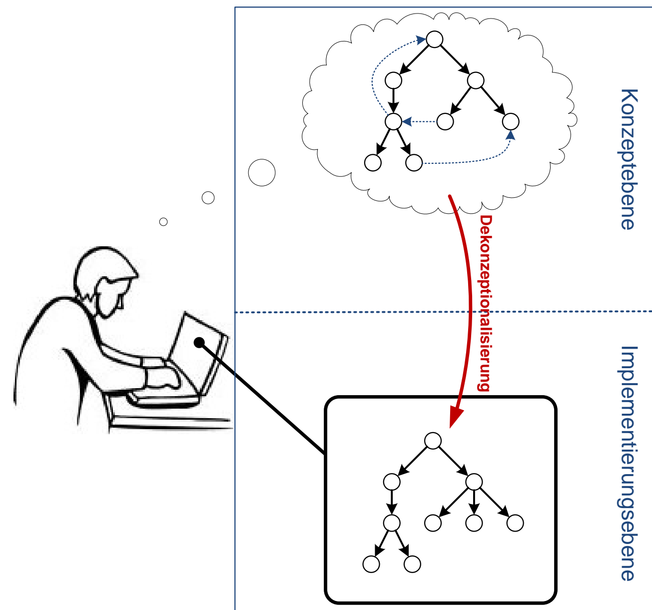


Abbildung 2.2.: Dekonzeptionalisierung von Konzepten in eine Programmiersprache

im Code dargestellt. Dadurch ergibt sich eine Menge an *impliziten Entwurfsregeln* bezüglich der Verwendung der Deklarationen. Im Code sind schließlich folgende Information nicht mehr ersichtlich, die meist lediglich dem Ersteller einer Deklaration bekannt sind:

- Von welchen Stellen im Programm sollte eine Deklaration verwendet werden (z.B. strukturelle Architektur)?
- Wie sollte eine Deklaration richtig verwendet werden (z.B. implizite Abhängigkeiten zwischen Schnittstellen, Einhaltung von Mustern)?
- Was sollte bei der Erweiterung einer Deklaration beachtet werden (z.B. Vererbung bei Klassen, Überschreiben von Methoden)?

2.1.2. Adäquate Verwendung von Deklarationen

Implizite Entwurfsregeln in Softwaresystemen. Die durch Programmiersprachen angebotenen Mechanismen zur Deklaration von Methoden, Klassen, Paketen etc. erlauben eine Modellierung des Softwaresystems. Die Deklarationen stehen in bestimmten über die Grammatik der Programmiersprache festgelegten Relationen zueinander (Vererbung zwischen Klassen, Enthaltensein von Methoden in Klassen etc.). Jedoch ist diese Modellierung nur eine Approximation an die tatsächlich vorliegenden Relationen der durch die Deklarationen repräsentierten Konzepte. Dadurch entstehen vielerlei implizite Abhängigkeiten zwischen Deklarationen, die nicht direkt im Code ablesbar sind.

Rekonzeptionalisierung. Bevor ein Wartungsingenieur Änderungen an einem Softwaresystem durchführen kann, muss er die Bezeichner der Deklarationen in einem Programm interpretieren (mentale Extraktion und Interpretation des Konzeptmodells aus dem Code) und dadurch Rückschlüsse bezüglich deren *adäquater Verwendung* ziehen. Eine nicht-adäquate Verwendung von Deklarationen wäre beispielsweise die Nutzung von Schnittstellen, die den Architekturvorgaben widerspricht, die Übergabe von unzulässigen Argumenten an eine Methode oder die Missachtung von im System implementierten Entwurfsmustern. Eine nicht-adäquate Verwendung führt sukzessive zu monolithischen Programmstrukturen und (latenten) Fehlern im System. Der Verfall eines Systems wird dadurch beschleunigt. Zur Rückgewinnung des Wissens über die adäquate Verwendung muss die Informationslücke geschlossen werden, die durch implizite Entwurfsregeln hervorgerufen wird. Dieser Rückgewinnungsprozess wird als *Rekonzeptionalisierung* bezeichnet. Abbildung 2.3 illustriert den Informationsfluss von einem Konzeptmodell über die Konzeptionalisierung zur Implementierung bis hin zur Informationsrückgewinnung aus der Implementierung durch Rekonzeptionalisierung. Nicht nur in der Softwarewartung ist Rekonzeptionalisierung notwendig. Auch bei der Verwendung von Bibliotheken muss der Nutzer einer API umfangreiches Wissen über die adäquate Verwendung erwerben, das im Lauf der Dekonzeptionalisierung durch den Ersteller während der Implementierung der API im Code verloren ging. Diese Situation ist der Auslöser dafür, dass ein signifikanter Teil des Aufwands, der während der Evolution eines Softwaresystems anfällt, auf das Programmverstehen entfällt. In der Literatur wird dieser Aufwand oftmals auf bis zu 50% des Gesamtarbeitsaufwands beziffert [WTMS95, LST78].

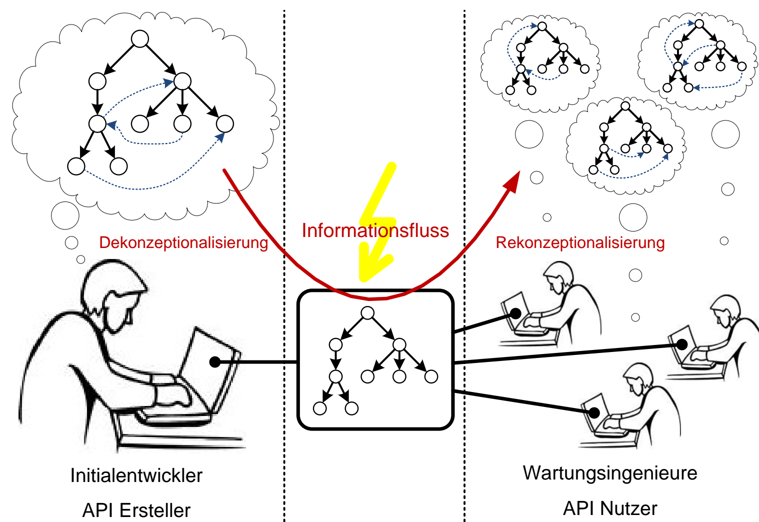


Abbildung 2.3.: Wissenstransfer in Softwareentwicklungsprojekten

Gerade im Bereich der Wartung gilt der Code meist als einzige zuverlässige Informationsquelle, Dokumentation gilt meist als nur wenig vertrauenswürdig [Sin98]. Die Kommunikation zwischen den Entwicklern spielt eine wichtige Rolle bei der Übermittlung des notwendigen Wissens über implizite Entwurfsregeln an neue Teamkollegen. Aus diesem Grund ist der Wechsel eines kompletten Entwicklerteams besonders schwierig und aufwändig. Dies ist beispielswei-

se bei der Wartung von Altsystemen der Fall, wenn die Initialentwickler oftmals nicht mehr verfügbar sind. Bei der Auftragsvergabe zur Entwicklung von Softwaresystemen an externe Unternehmen führt die Problematik der aufwändigen Wissensreproduktion aus Systemen zu einer hohen Lieferantenmacht. Der Auftraggeber kommt leicht in eine Abhängigkeitssituation gegenüber dem Auftragnehmer, da gegebenenfalls nur dieser Änderungen und Wartungsarbeiten mit vergleichsweise geringem Aufwand durchführen kann.

Konformitätsprüfung. Das Fatale an der Aufgabe der Rekonzeptionalisierung ist, dass sich ein Entwickler nie sicher sein kann, ob er wirklich alle impliziten Entwurfsregeln der Initialentwickler vollständig erkannt und eingehalten hat. Bei der Durchführung von Änderungen an fremden Programmen oder auch bei der Programmierung gegen eine unbekannte API kann sich ein Entwickler nie vollkommen sicher sein, dass er die Arbeiten auch adäquat bezüglich aller im System verborgener impliziter Entwurfsregeln durchgeführt hat. Entwickler sind oftmals dazu gezwungen, aufgrund eines Mangels an expliziter Information über die implementierten Deklarationen, deren Funktionsweise durch Interpretation der Bezeichner gewissermaßen zu erraten. Auf Basis von ‘Versuch und Irrtum’ muss ausprobiert werden, ob eine Änderung an einem System oder die Nutzung einer API zu unerwünschten Nebenwirkungen führt. Die Durchführung von Tests kann zwar dabei helfen, Implementierungsfehler aufzudecken, Verstöße gegen Entwurfs- und Architekturprinzipien können dadurch jedoch nicht identifiziert werden. Auch latente Fehler [HP04], die sich nicht unmittelbar auf das Laufzeitverhalten auswirken, sondern nur in bestimmten Situationen Fehler hervorrufen, stellen Fallstricke für künftige Änderungen dar. Gerade Fehler, die durch vormals latente Fehler induziert werden, sind oft besonders schwer zu identifizieren und zu beheben, da diese nicht von den gerade veränderten Codestellen hervorgerufen werden, sondern an anderen Stellen im System verborgen liegen [LPS00, Lut93].

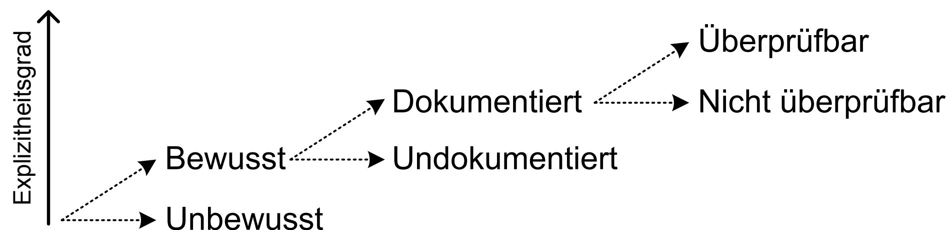


Abbildung 2.4.: Unterschiedliche Niveaus an Explizitheit

Explizitheitsniveau. Implizite Vorgaben entstehen aufgrund von Architektur-, Entwurfs- und Implementierungsentscheidungen. Diese Entscheidungen werden nicht immer bewusst getroffen. Abbildung 2.4 zeigt die unterschiedlichen Niveaus auf denen derartige Entwurfsregeln in Projekten explizit gemacht werden. Die Ersteller von Deklarationen erkennen möglicherweise selbst nicht, dass die Nutzung der Deklarationen bestimmten Einschränkungen unterliegt. Unbewusst eingeführte Entwurfsregeln verbleiben somit zunächst immer *implizit*. Sollten im weiteren Verlauf eines Projekts jedoch Inkonformitäten auffallen, können weitere Schritte veranlasst

werden, die Regeln explizit zu machen. Sobald sich ein Entwickler einer Entwurfsregel bewusst ist, ist diese Information grundsätzlich verfügbar. Durch Dokumentation kann die Verfügbarkeit weiter erhöht und die Entwurfsregel dauerhaft festgehalten werden. Jedoch gewährleistet die Dokumentation in Form von losen, nicht kontinuierlich mit dem Quellcode abgeglichenen Dokumenten nicht die Konformität des Codes zu den Entwurfsregeln. Die Regeln sind somit zwar *explizit dokumentiert*, jedoch verbleiben die dadurch festgelegten einschränkenden Bedingungen an die adäquate Nutzung von Deklarationen im Quellcode implizit. Es wird weder sichergestellt, dass die Dokumentation nicht veraltet und die darin festgehaltenen Regeln aktuell bleiben noch, dass im Code unwissentlich Verletzungen der Entwurfsregeln hervorgerufen werden. Erst eine Form der Dokumentation, die eine Abbildung der Entwurfsregeln auf den Quellcode definiert, welche die Grundlage für eine (automatische) Konformitätsüberprüfung der Regeln bildet, wird im Rahmen dieser Arbeit als vollkommen *explizit* bezeichnet.

2.2. Störungen des Informationsflusses in Softwareentwicklungsprojekten

Der wissensintensive Prozess der Softwareentwicklung setzt sich aus verschiedenen Informationsflüssen zusammen, die im Lauf der Entwicklung bewältigt werden müssen. Abbildung 2.5 illustriert die Dimensionen der Informationsflüsse in einem Entwicklungsprojekt.

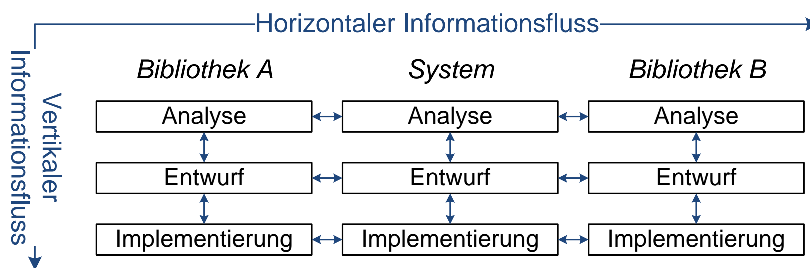


Abbildung 2.5.: Informationsfluss in einem Softwareentwicklungsprojekt

In einem Entwicklungsprozess lassen sich zwei Dimensionen des Informationsflusses unterscheiden:

- **Vertikaler Informationsfluss**: Von der Analysephase, die sich hauptsächlich mit dem Sammeln und Aufbereiten der Anforderungen beschäftigt, über die Phase des Entwurfs, in der verschiedenste Modelle des zu erstellenden Softwaresystems angefertigt werden, bis hin zur Implementierungsphase, in der das System in einer Programmiersprache ausformuliert wird, werden immer wieder Beschreibungen verschiedener Aspekte des Systems in andere Repräsentationsformen übersetzt. Zunächst liegen diese Informationen häufig in Form von Beschreibungen in natürlicher Sprache vor, die sukzessive in Modellen formalisiert und mit syntaktischer Struktur versehen werden. Die Implementierung unterliegt schließlich vollständig der Syntax der verwendeten Programmiersprache(n). Implizite Entwurfsregeln

in den Artefakten können beim Übergang zwischen den Phasen verloren gehen oder zu Fehlinterpretationen führen.

- **Horizontaler Informationsfluss**: Die Systementwicklung erfolgt ab einer bestimmten Systemgröße nicht mehr nur durch eine Person. Meist handelt es sich um ein Team von Personen oder sogar mehrere Teams. Daraus ergibt sich innerhalb einer Phase des Entwicklungsprozesses zwischen den Entwicklern Kommunikationsbedarf, um ein gemeinsames Verständnis des zu entwickelnden Systems zu schaffen. Im Rahmen der Analyse müssen beispielsweise Anforderungen, die durch verschiedene Requirementsingenieure erhoben wurden, zueinander konsistent und widerspruchsfrei integriert werden. Während des Entwurfs und der Implementierung müssen die Dekomposition des Systems, die Definition von Aufgaben der einzelnen Systemkomponenten sowie Schnittstellen und deren adäquate Verwendung abgestimmt werden. Aber die horizontale Kommunikation ist nicht nur auf die Mitglieder eines Entwicklungsprojekts beschränkt. Sobald extern entwickelte Systemkomponenten oder Bibliotheken verwendet werden, ergibt sich Kommunikationsbedarf über die Grenzen des Projekts hinaus. Meist wird diese externe Kommunikation nicht direkt, sondern über Artefakte wie Dokumentation oder Code vorgenommen. Durch implizite Entwurfsregeln in den Artefakten wird zusätzlicher Kommunikations- und Abstimmungsaufwand notwendig, um Fehlinterpretationen zu vermeiden.

Die Bewältigung dieser Informationsströme ist eine enorme Herausforderung in einem Entwicklungsprojekt. Nach [KCJ94] verbringt ein Entwickler zwischen 40% und 66% seiner Arbeitszeit mit Kommunikation, um an die für seine Arbeit benötigten Informationen zu gelangen und die Ergebnisse seiner Arbeit an andere weiterzugeben. Dieser Bedarf an Informationsaustausch kann nur durch Kommunikation in unterschiedlichen Formen bewältigt werden. In einem Entwicklungsprojekt werden folgende Arten der Kommunikation eingesetzt, um den Informationsfluss sicherzustellen:

- Direkte Kommunikation (synchron): Persönliche Kommunikation durch Zwiegespräche, Besprechungen oder Telefonate.
- Indirekte Kommunikation (asynchron): Kommunikation durch den Austausch von Artefakten, wie Dokumentation oder Quellcode. Auch Kommunikation über E-Mails fallen in diese Kategorie.

Folgende Faktoren führen zu Störungen im Informationsfluss während der Evolution eines Systems:

Direkte Kommunikation. Studien zeigen, dass die Wahl der Informationsquelle durch die Einfachheit der Verwendung und die Zeit, die benötigt wird, um an Informationen zu gelangen, bestimmt wird [KCJ94]. Der einfachste und damit der am häufigsten genutzte Weg [PSV94], um an Informationen zu gelangen, ist die direkte Befragung eines Teamkollegen [HP00]. Kollegen sind jedoch nicht nur eine reine Informationsquelle, sie können sich auch die Arbeiten ansehen, bei denen sich unerfahrene Entwickler noch unsicher sind, und Feedback geben. Problematisch ist dabei jedoch, dass durch direkte Kommunikation immer wieder Teammitglieder unterbrochen und damit aus dem aktuellen Arbeitskontext herausgerissen werden. Dabei vergisst der

Unterbrochene oftmals selbst, die Ziele, Entscheidungen und Interpretationen bezüglich der Aufgabe, der er sich ursprünglich gewidmet hatte. Im Rahmen einer ausführlichen Studie wird in [LVD06] dargestellt, dass sich viele Entwickler zu häufig unterbrochen fühlen. Entwickler beschreiben, dass durch die vielen Unterbrechungen ihre eigene Produktivität auf der Strecke bleibt. Die Teilnehmer an der Befragung äußerten sogar, dass die Unterbrechungen häufiger wurden, als ein agiles Vorgehen eingeführt und damit die Erstellung von Dokumentation reduziert wurde. Trotz der scheinbaren Effizienz der schnellen direkten Kommunikation ergibt sich dadurch die Gefahr, dass die langfristige persistente Sicherung von Informationen unterlassen wird. Einzelne erfahrene Entwickler werden zu zentralen Anlaufstellen und es entsteht sukzessive eine Abhängigkeit. Verlässt ein derartiger Entwickler das Projekt, geht ein signifikantes Maß an Wissen, Information und Know-How verloren. Die Kompensation dieses Verlusts ist unter Umständen mit hohen Aufwänden verbunden. In [CKI88] wird auf Basis einer Feldstudie beschrieben, dass Entwickler oftmals sehr unzufrieden mit Dokumentation als (indirektes) Kommunikationsmedium sind. Zusätzlich wurde jedoch festgestellt, dass gerade in größeren Teams die direkte Kommunikation nicht mehr skaliert.

Implizite Entwurfsregeln in einem Softwaresystem werden häufig durch direkte Kommunikation zwischen den Entwicklern ausgetauscht. In der Wartung ist die direkte Kommunikation jedoch nur eingeschränkt oder sogar überhaupt nicht möglich, weil etwa die Initialentwickler nicht mehr verfügbar sind. Auch die adäquate Nutzung von Schnittstellen ist oftmals schwer durch direkte Kommunikation zu klären, wenn Module von verschiedenen Teams oder sogar verschiedenen Unternehmen realisiert werden. Bei der Nutzung von Bibliotheken und Frameworks sind deren Urheber oftmals generell nicht kontaktierbar.

Indirekte Kommunikation. Die Autoren von [LVD06] beschreiben den Quellcode als wichtigste indirekte Informationsquelle. Insgesamt ca. 62% der zum Verstehen benötigten Zeit wird direkt am Code gearbeitet, in Form von reinem Code-Lesen oder der Nutzung des Debuggers. Informationen bezüglich der vorgesehenen Architektur und des Entwurfs werden meist in eigenen Dokumenten festgehalten. Gerade für Entwickler, die neu in ein Projekt kommen, ist diese Art der Dokumentation sehr hilfreich [LVD06]. Dokumentation durch prosaischen Text ist jedoch oftmals ungeeignet, implizite Entwurfsregeln langfristig über den Lauf der Softwareevolution hinweg in einem Projekt zu halten. Problematisch ist, dass natürlich-sprachliche Beschreibungen zu Redundanz neigen [DJS09], oftmals sehr unpräzise sind und relevante Stellen nur schwer auffindbar sein können. Damit wird die enthaltene Information nur schwer verfügbar und veraltet im Lauf der Zeit, da sich Entwickler, die Änderungen am System durchführen, oftmals nicht bewusst sind, dass Abschnitte in den Dokumentationen existieren, die entsprechend angepasst werden müssten. Falls überhaupt zusätzliche Dokumentation existiert, kann die in diesen Dokumenten enthaltene Information oftmals nur noch bruchstückhaft in der Implementierung nachvollzogen werden. Oftmals ist unklar, ob Abweichungen zwischen Dokumentation und Implementierung auf Defizite der Dokumentation (etwa aufgrund von Veraltung) oder auf Mängel in der Implementierung zurück zu führen sind. Aus dieser Unwissenheit heraus schleichen sich sukzessive Wartbarkeitsdefizite in ein Softwaresystem ein und die internen Strukturen tendieren zu monolithischen Gebilden, welche das System im Lauf der Zeit immer schwerer verständlich werden lassen [Par94, Leh80, EGK+01].

Kommunikation über Teamgrenzen hinweg. Eine der wichtigsten Prinzipien, um Abhängigkeiten sowohl zwischen Systemteilen als auch zwischen Entwicklerteams zu reduzieren, ist die Idee des "Information Hiding" von David Parnas [Par72]. Nach diesem Prinzip gestaltete Module sollten sowohl *offen* für Erweiterungen und Anpassungen als auch *geschlossen* bezüglich Änderungen sein, welche die Nutzer betreffen würden [Lar01]. Eine besondere Rolle kommen dabei Schnittstellen zu: Sie sollten das Zusammenspiel der Module definieren und diese entkoppeln. Diese Entkopplung kann schließlich auf den Entwicklungsprozess übertragen werden und die Gesamtentwicklung anhand der Modulstruktur auf einzelne Entwicklerteams aufgeteilt werden. Innerhalb dieser Teams findet die Kommunikation über die Realisierung der Modulfunktionalität statt, zwischen diesen Teams bildet die Schnittstelle einen Vertrag, der beiderseitig eingehalten werden muss. In [dSRC+04] wird jedoch anhand einer industriellen Fallstudie dargestellt, dass genau diese Schnittstellen erhöhten Bedarf an Kommunikation benötigen. Die Gründe hierfür liegen überwiegend darin, dass die Schnittstellen (APIs) häufig Änderungen unterworfen sind. In [LPS00] wurde eine Studie zur Ursache von Softwarefehlern und deren Beseitigungsaufwand durchgeführt. Dabei wurde herausgefunden, dass ca. 25% der analysierten Fehler als Schnittstellenfehler klassifiziert werden können. Zudem ruft diese Fehlerklasse den höchsten Beseitigungsaufwand hervor.

3. Grundlegende Techniken und Methoden

"Zuverlässige Informationen sind unbedingt nötig für das Gelingen eines Unternehmens."
- Christoph Kolumbus

Dieses Kapitel führt in für diese Arbeit grundlegende Themen ein. Zunächst werden wichtige Arbeiten aus dem Forschungsbereich des Wissensmanagements eingeführt und deren Übertragbarkeit auf die Herausforderungen der Softwareentwicklung aufgezeigt. Hierbei werden wesentliche Begrifflichkeiten dieser Arbeit definiert. Anschließend folgt ein Überblick über wichtige Arbeiten aus dem Bereich der Dokumentation von Software. Gängige Methoden zur Softwaredokumentation werden hierbei bezüglich ihres Einsatzes in der Softwareevolution kategorisiert. In Abschnitt 3.3 werden wichtige Techniken der statischen Analyse vorgestellt und die zentralen Herausforderungen diskutiert. Dadurch wird in grundlegende technische Fragestellungen eingeführt, die zur Bewertung der in den folgenden Kapiteln vorgestellten Techniken notwendig sind. Abschließend werden die für diese Arbeit wichtigsten Begrifflichkeiten aus dem Bereich der Entwicklung von Programmier- und Modellierungssprachen vorgestellt.

3.1. Klassisches Wissensmanagement

Implizites Wissen. Den Begriff des "impliziten Wissens" (engl. "tacit knowledge") wurde bereits 1966 von Michael Polanyi [Pol66] geprägt. Er bezeichnet damit Wissen, das nicht explizit formalisiert ist, ggf. nicht einmal explizit formalisierbar ist. Auch in der Softwareentwicklung findet man derartiges implizites Wissen, beispielsweise Wissen über Kompetenzen bestimmter Personen, Abläufe im Entwicklungsprozess etc. Neben diesem stark prozessorientierten implizitem Wissen kann man auch produktorientiertes implizites Wissen identifizieren. Im Rahmen dieser Arbeit werden speziell implizite Entwurfsregeln betrachtet, die Wissen über systeminterne Strukturen darstellen, das in gewisser Weise 'zwischen den Codezeilen' verschwindet. Dabei handelt es sich beispielsweise um Informationen über Architektur- und Entwurfsvorgaben oder auch Informationen über die adäquate Nutzung von APIs. Dieses Wissen hat immer dann impliziten Charakter, wenn es in der genutzten Programmiersprache nicht ausdrückbar ist (bzw. nicht ausgedrückt wird). Um derartige Informationen explizit zu machen, behilft man sich meist durch Nutzung der natürlichen Sprache, die eine flexiblere Beschreibung ermöglicht. Diese Dokumentation ist jedoch oftmals nicht präzise genug und neigt dazu, im Lauf der Evolution zu veralten. Auch die Verfügbarkeit der Dokumentation ist oftmals eingeschränkt.

Mentale Modelle. Erklärung "mentaler Modelle" aus kognitionspsychologischer Sicht von Johnson-Laird [JL83]: "Ein mentales Modell ist das Abbild der Wirklichkeit in der menschlichen Wahrnehmung. Gedächtnis, Wirklichkeitswahrnehmung, Problemlösung und alle anderen Denkleistungen beruhen auf der Anwendung dieser Abbilder (sog. kognitiver Artefakte). Vermutlich beruht auch das Textverständnis auf dem Entstehen mentaler Modelle der beschriebenen Situation und nicht auf einem semantischen Abbild (d.h. Speicherung und Verarbeitung der Wörter)."

Diese mentalen Modelle spielen auch in der Softwareentwicklung eine zentrale Rolle. Softwaresysteme sind eine reine Schöpfung des menschlichen Geists. Die Modelle, die einem System zugrunde liegen, haben meist kein physisches Ebenbild. Somit sind die in Systemen implementierten Konzepte sehr abstrakter Natur und provozieren leicht Missverständnisse unter den Entwicklern. Entwickler, die gemeinsam ein System implementieren, müssen ihre mentalen Modelle, welche die in einem System implementierten Konzepte miteinander in Bezug setzen, synchronisieren, um auf Basis eines gemeinsamen Verständnisses das Softwaresystem realisieren zu können.

Das SECI-Modell. Basierend auf dem Begriff des impliziten Wissens von Polanyi entwarfen Nonaka und Takeuchi das sogenannte SECI-Modell, das in Abbildung 3.1 dargestellt ist [NT95]. Dieses Modell beschreibt eine kontinuierliche Transformation von implizitem zu explizitem Wissen, aus dem sukzessive eine Verbreitung des Wissens und neues Wissen entsteht. Der in der Abbildung skizzierte kontinuierliche Prozess unterteilt sich in vier Phasen:

- *Socialization.* Implizites Wissen besteht in Form von Fähigkeiten und Erfahrungen der Beteiligten. Durch direkte Kommunikation, Interaktion oder gemeinsame Erfahrungen wird implizites Wissen gewonnen und geteilt (implizit zu implizit).
- *Externalization.* Artikulation des impliziten Wissens.
- *Combination.* Kombination verschiedener Elemente expliziten Wissens.
- *Internalization.* Integration des expliziten Wissens in das mentale Modell einer Person. Dies entspricht einem Prozess des Erlernens.

Das SECI-Modell lässt sich auf die Softwareentwicklung und -evolution übertragen:

Socialization: Fähigkeiten und Wissen der Entwickler und Projektbeteiligen, die den Ausgangspunkt eines Projekts bilden.

Externalization: Austausch durch Diskussionen und Modellierungsaktivitäten.

Combination: Bildung eines gemeinsamen mentalen Modells von dem zu erstellenden Softwaresystem.

Internalization: Erlernen des gemeinsam geschaffenen Wissens durch die Individuen.

Dieser spiralförmig ablaufende Wissensgewinnungsprozess Bedarf sehr viel Kommunikation. Er wird durch Entwicklerfluktuation erschwert. Durch diesen Einfluss erfolgt eine Auftrennung des Zyklusses nach der Phase der Combination. Der Internalisierungsprozess wird erschwert, da

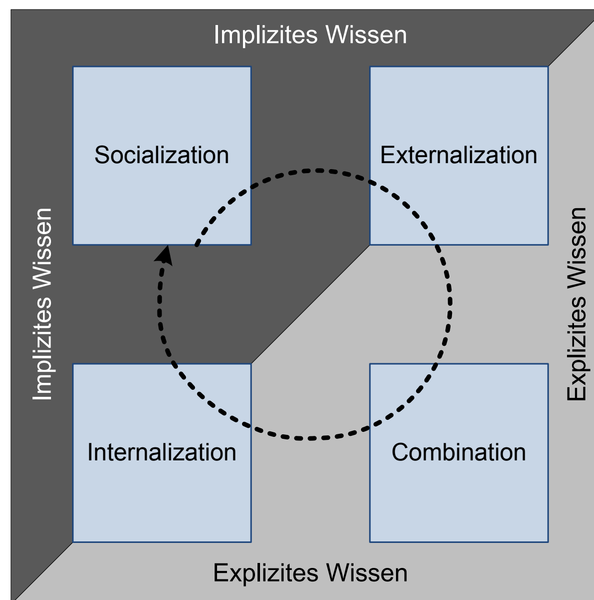


Abbildung 3.1.: Das SECI-Modell [NT95]

viele der Informationen zwar während der Konzeption und des Entwurfs explizit artikuliert werden, aber nicht in expliziter Form konserviert werden. Dies macht es Entwicklern, die nicht mit einem System vertraut sind, schwer den Wissensvorsprung aufzuholen und das mentale Modell der Initialentwickler zu erarbeiten, ohne großen Kommunikationsaufwand hervorzurufen. Auch [WA04] beschreibt die Übertragung dieses Prozesses auf das Wissensmanagement im Softwareentwicklungsprozess. Anhand zweier Fallstudien, in denen Entwickler über ihre Erfahrungen befragt wurden, kommen die Autoren zu dem Schluss, dass heutige Methoden, Techniken und Werkzeuge unzureichend sind, um das Wissensmanagement effizient zu unterstützen.

Strategien des Wissensmanagements. Die Autoren von [HNT99] unterscheiden zwischen der *Personalisierungsstrategie* und der *Kodifizierungsstrategie* des Wissensmanagements als zwei orthogonale strategische Vorgehensweisen, Wissen in einer Organisation (einem Projekt) zu verbreiten. Nach [BdBDF07] wird die Personalisierungsstrategie, die auf der Förderung des Wissensaustauschs zwischen Entwicklern beruht, vorwiegend in der Industrie eingesetzt. Die Kodifizierungsstrategie hingegen ist meist die Basis derzeitiger Forschungsansätze im Bereich des Wissensmanagements, die sich auf die Speicherung und das Auffinden von Wissen in Artefakten oder Repositories konzentrieren. In dieser Arbeit wird ebenfalls ein auf Kodifizierung basierender Ansatz vorgestellt, da dieser Wissen explizit verfügbar macht. Dies ist von enormer Wichtigkeit, da gerade bei hoher personeller Fluktuation eine reine Personalisierungsstrategie nicht anwendbar ist und die Abhängigkeit von einzelnen Individuen reduziert wird.

Nicht behandelte Themen. In dieser Arbeit werden nur spezifische Aspekte des Wissensmanagements betrachtet. Mit den vorgestellten Methoden und Techniken soll lediglich die Ex-

plizitmachung von impliziten Entwurfsregeln in Softwaresystemen unterstützt werden. Nicht betrachtet werden Methoden zur Steigerung der Teamkommunikation, wie sie oftmals in der Literatur zum Wissensmanagement zu finden sind (vgl. [Des03]). Es wird lediglich Wissen über das zu erstellende Softwaresystem betrachtet, das sich direkt in der Beschreibung des Systems (Code oder Modelle) niederschlägt, und mit impliziten Vorgaben der Ersteller durchsetzt ist. Kreative Prozesse, wie die Findung von Ideen für Features, Anforderungserhebung oder Brainstormings sind nicht im Fokus dieser Arbeit. Auch Werkzeugunterstützung zum Austausch von Dokumenten wird nicht untersucht. Alle im Kontext dieser Arbeit vorgestellten Werkzeuge beziehen sich lediglich auf die Explizitmachung von impliziten Vorgaben in Softwaresystemen und die Überprüfung der Konformität der Implementierung des Systems bezüglich dieser Entwurfsregeln (beispielsweise der Soll-Architektur). Viele Arbeiten im Bereich des Wissensmanagements stellen auch Methoden vor, die es vereinfachen sollen, Experten mit bestimmtem Spezialwissen zu finden. Die im Rahmen dieser Arbeit vorgestellten Methoden versuchen, das Wissen einzelner Beteiligter an Softwareprojekten möglichst weitgehend in automatisiert prüfbarer Form in das System zu integrieren, um die Notwendigkeit der direkten Kommunikation mit den jeweiligen Experten möglichst gering zu halten. Damit soll eine Unabhängigkeit von bestimmten Personen erreicht und die Aktualität der systembezogenen Informationen bewahrt werden. Dies dient somit der Erhaltung der Verständlichkeit von Programmen und deren Wartbarkeit. Im Falle von Bibliotheken wird dadurch die Benutzbarkeit der APIs gesteigert.

3.2. Explizitmachung impliziter Entwurfsregeln durch Dokumentation

Um Entwurfsregeln für die Wartung eines Systems zu bewahren, ist die Dokumentation die wohl verbreitetste Lösungsstrategie. Während einerseits agile Methoden die Bedeutung von Softwaredokumentation in Frage stellen, bemüht man sich im Bereich des Re-Engineerings darum, Dokumentation aus dem System zu reproduzieren [dSAdO05]. In den folgenden Abschnitten werden Arten von Dokumentation aufgezeigt und deren Nutzen und Zweck erläutert.

3.2.1. Arten von Dokumentation

Neben mündlicher Kommunikation werden vor allem verschiedene Arten von textueller und graphischer Dokumentation verwendet, um das Wissen in einem Projekt zu konservieren und auszutauschen. In Abbildung 3.2 sind die wichtigsten Arten von Dokumentation skizziert.

Die Nutzerdokumentation soll vor allem den künftigen Anwendern des Softwaresystems die Bedienung erleichtern. Darin sind die Systemfunktionen und die Benutzerschnittstelle erklärt. Diese Dokumentation ist meist textuell gehalten und wird ggf. mit Screenshots angereichert, um einen Wiedererkennungseffekt hervorzurufen. Die Entwicklerdokumentation versucht, das Wissen über das Design und die Implementierung eines Systems zu konservieren. Sie dient den Entwicklern als Nachschlagewerk und Referenz. Für neue Entwickler ist sie eine wichtige Lektüre, um sich mit dem System vertraut zu machen. Die Entwicklerdokumentation kann

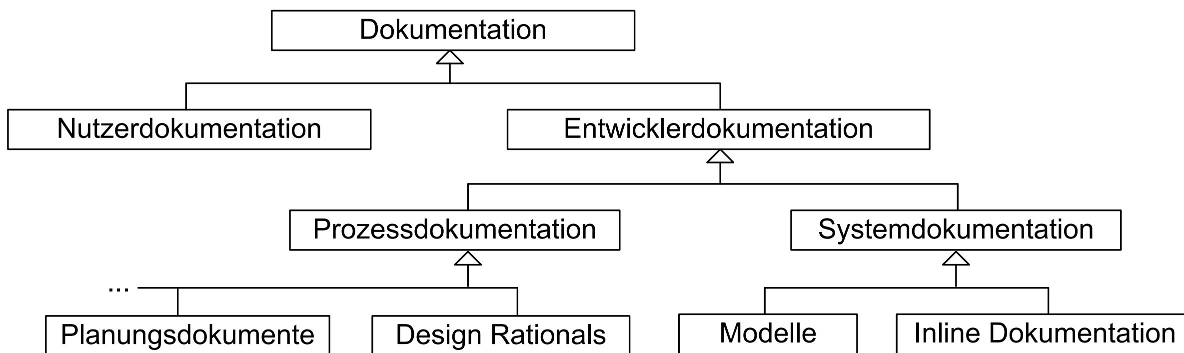


Abbildung 3.2.: Arten der Dokumentation

wiederum in Prozess- und Systemdokumentation unterteilt werden, je nachdem ob der Erstellungsprozess oder das System im Vordergrund der Beschreibung steht. Im Wesentlichen setzt sich die Systemdokumentation meist aus Modellen, die oftmals noch zusätzlich in natürlicher Sprache beschrieben werden, und aus textueller Kommentierung im Code zusammen.

Design Rationals. Design Rationals haben zum Ziel, die Gründe für die Wahl eines speziellen Entwurfs zu erläutern und den Entscheidungsprozess zu dokumentieren, der zu einem finalen Entwurf eines Softwaresystems führte. Dies sind wichtige Informationen, um später ggf. notwendige Designänderungen auf fundiertem Niveau diskutieren und durchführen zu können. Somit können Design Rationals als Dokumentation des Entwurfsprozesses (der Abfolge von Entscheidungen bis hin zu einem finalen Design) verstanden werden. [JLI92] bietet einen Überblick über die wichtigsten Arbeiten in diesem Forschungsbereich. Der Hauptfokus liegt in dieser Arbeit in der Sicherstellung, dass Änderungen an einem System die Designintention nicht verletzen, um dadurch einem Verfall des Systemdesigns vorzubeugen (anstatt des Designprozesses steht das Design selbst im Vordergrund). Damit ist insbesondere System- gegenüber Prozessdokumentation von Interesse.

Dokumentation durch Modelle. Modelle, die nicht zur Generierung von Code, sondern lediglich der Strukturierung und dem Verstehen des Problems und des Lösungsraums dienen, werden oftmals als Systemdokumentation verwendet. Derartige Modelle werden meist während der Analyse, des Grob- und Feinentwurfs erstellt und dienen als eine Art "Vorlage" für die Realisierung des Softwaresystems in der Implementierung. Da während der nachgelagerten Phasen allerdings meist ein weiterer Lerneffekt dazu beiträgt, dass sich die Vorstellungen der Entwickler über das System ändern oder dass bestimmte Details in der abstrakten Modellierung übersehen wurden, kann man sehr häufig beobachten, dass sich die Implementierung über die Zeit stark von diesen Modellen entfernt und somit sukzessive Abweichungen auftreten. Problematisch ist hierbei, dass durch die mangelnde Konsistenz zwischen Modellen und dem System diese Modelle eher einen historischen Stand der Entwicklung wiedergeben und somit einen schnellen Verstehensprozess nur unzureichend unterstützen. Ein neuer Entwickler, der schnell

in ein System eingearbeitet werden muss, sollte nicht dazu gezwungen sein, den Verlauf der Entwicklung bis zum Ist-Zustand des Systems nachvollziehen zu müssen (vgl. [PC85]).

Inline-Dokumentation. Damit Informationen nicht verloren gehen, wird meist auf informelle Dokumentation in natürlicher Sprache zurückgegriffen. Problematisch ist dabei allerdings, dass natürliche Sprache oftmals unpräzise ist. Ein Entwickler, der eine Schnittstelle (System intern oder auch extern) nutzen möchte, steht letztendlich der Frage gegenüber, wie die einzelnen Konzepte der Schnittstelle miteinander zusammenspielen und wie er diese kombinieren kann, um sein Ziel zu erreichen. Natürlich-sprachliche Dokumentation, insbesondere Inline-Dokumentation, wie beispielsweise JavaDoc [Kra99], ist ein unverzichtbarer Bestandteil, um sich initial der Semantik der implementierten Konzepte anzunähern. Da diese Art der Dokumentation jedoch meist spezifisch für bestimmte implementierte Konzepte geschrieben wird (für eine spezifische Klasse, Methode etc.), lässt sich daraus meist nur unzureichend das Zusammenspiel zwischen mehreren Konzepten ablesen [Les02]. Häufig wird nur unzureichend Aufschluss über das Zusammenspiel zwischen verschiedenen Methoden oder Klassen sowie über deren vorgesehene Verwendungsweise gegeben. Die Frage, ob ein Konzept tatsächlich im Sinne seines Erstellers verwendet wird, lässt sich somit meist nicht klar beantworten. Selbst wenn die Formulierung präzise genug ist, bedarf es immer der Überprüfung durch einen Reviewer. Angesichts der enormen Anzahl an Konzepten in heutigen Systemen ist eine vollständige manuelle Prüfung jedoch mit unrealistischen Aufwänden verbunden.

Abstraktionsgrad der Dokumentation. Die in [LSF03] beschriebenen Fallstudien zeigen, dass Inline-Dokumentation während der Evolution von Systemen meist durchaus gepflegt wird, abstraktere Modelle jedoch sukzessive veralten. Über die Antwort auf die Frage, welche Art von Dokumentation die wichtigste zur Unterstützung des Wartungsprozesses ist, herrscht in der Literatur Uneinigkeit: Während [dSAdO05] auf der Basis zweier Umfragen beschreibt, dass Wartungsingenieure den Code selbst und die Kommentierung im Code als am wichtigsten erachten und abstraktere Modelle, wie Anforderungsdefinitionen, Funktions- und Komponentenbeschreibungen als weniger wichtig einstufen, ergab eine sehr ähnliche Studie in [FL02], eine deutlich höhere Relevanz für Dokumentation von abstrakterem Wissen. Ein Erklärungsversuch könnte darin bestehen, dass sehr codenahe Dokumentation bei der konkreten Durchführung vonnöten ist. Ein Wartungsingenieur verbringt somit sehr viel Zeit mit der Studie dieser Art von Dokumentation während der Implementierung eines Change-Requests. Auch das reine Quantum an low-level Inline-Dokumentation und Kommentierung übersteigt meist die Größe abstrakterer Beschreibungen. Somit ist es durchaus erklärbar, dass ein Wartungsingenieur deutlich mehr Zeit mit dem Lesen von Code-Kommentierung verbringt als mit dem Studium der Architekturdokumentation. Jedoch wird ein einzelner Kommentar oder die konkrete Kommentierung einer bestimmten Klasse wahrscheinlich auch nicht mehr Aufmerksamkeit erhalten als abstraktere Dokumentation. Aus beiden Studien geht leider nicht hervor, inwieweit der Umfang der Dokumentationsarten mit der Wertschätzung der Entwickler korreliert. Abstraktere Dokumentation ist deutlich besser geeignet, um die Umsetzung von Change-Requests zu planen und ein initiales Verständnis von einem System zu bekommen. Auch die Korrelation des Vorwissens der einzelnen Entwickler über das System und deren Einschätzung geht aus den Studien leider

nicht hervor. In [WTMS95] wird auf Basis einer Fallstudie die These aufgestellt und begründet, dass besonders große Systeme einen höheren Bedarf an abstrakter Dokumentation aufweisen.

3.2.2. Zweck von Dokumentation

Man kann zwei Kategorien von Dokumentation unterscheiden:

- *Konstruktive Dokumentation:* Diese Art der Dokumentation wird im Rahmen der initialen Systementwicklung erstellt, meist sogar in deren frühen Phasen (Analyse und Entwurf). Sie dient dazu, den kreativen Prozess der Konzeption eines Softwaresystems zu unterstützen. Modelle, die Bestandteil konstruktiver Dokumentation sind, folgen oftmals nur unscharfer syntaktischer Vorgaben und sind meist nur wenig formal (z.B. UML Use-Case Diagramme). Entwickler setzen diese Art der Dokumentation ein, um einerseits zu verstehen, was ein Softwaresystem leisten soll, und andererseits, wie es aufgebaut werden soll. Durch die Dokumentation dieser Informationen wird es ermöglicht, die unterschiedlichen Vorstellungen der am Entwicklungsprozess beteiligten Personen zu synchronisieren und zu einem gemeinschaftlichen Bild des zu entwickelnden System zu gelangen.
- *Deskriptive Dokumentation:* Diese zweite Kategorie von Dokumentation hat einen anderen Zweck. Sie wird genutzt, um Wartungsingenieuren einen leichteren Zugang zu einem Softwaresystem zu ermöglichen. Sie soll die Einstiegshürde senken, um das System schnell zu verstehen und Änderungen vorzunehmen zu können. Deskriptive Dokumentation umfasst meist Beschreibungen und Modelle der Architektur sowie von eingesetzten Entwurfsmustern. Auch ein Glossar, das die wichtigsten Domänenbegriffe definiert, kann in die Kategorie der deskriptiven Dokumentation eingeordnet werden.

Das wichtigste Unterscheidungskriterium zwischen deskriptiver und konstruktiver Dokumentation ist die Motivation, die mit der Erstellung verbunden ist: Konstruktive Dokumentation wird von Entwicklern in gewisser Weise "für sich selbst" erstellt (zur Kommunikation im Team, Strukturierung der eigenen Vorstellungen vom System), während deskriptive Dokumentation zur Erhaltung und zum Transfer von Wissen für künftige Entwickler gedacht ist.

Die Unterscheidung zwischen konstruktiver und deskriptiver Dokumentation ist sehr wichtig, da deskriptive Dokumentation im Laufe der Evolution eines Systems mit der Implementierung konsistent gehalten werden sollte, um als wichtige Informationsquelle für Wartungsingenieure zur Verfügung zu stehen. Die Pflege von deskriptiver Dokumentation ist somit mit kontinuierlich zu erbringenden Aufwänden verbunden. Konstruktive Dokumentation hingegen sollte in den Wartungsphasen eine eher untergeordnete Rolle spielen. Sie kann jedoch in speziellen Fällen nötig sein, in denen der historische Erkenntnisstand während der Entwicklung von Bedeutung ist (Reverse Engineering aufgrund mangelnder deskriptiver Dokumentation). In [PC85] beschreibt David Parnas, dass es eine gute Dokumentation dem Wartungsingenieur nicht abverlangen sollte, in archäologischer Art und Weise alle Irrwege der Entwicklung des Systems nachzuvollziehen. Eine "Protokollierung" der Aktivitäten ist somit kein geeignetes Mittel zur Vermittlung von Wissen über das System. Wenn jeder Fehler und jede Umstrukturierung im System einzeln verstanden werden muss, um ein Verständnis über ein System zu gewinnen, lähmt dies den Wartungsprozess und senkt die Produktivität. Aus diesem Grund

ist es von enormer Bedeutung, dass auch angemessene deskriptive Dokumentation geschaffen und vor allem aktuell gehalten wird. Deskriptive Dokumentation kann durchaus aus konstruktiver Dokumentation hervorgehen, sobald diese einen gewissen Reifegrad und damit Stabilität erreicht hat.

System- und Prozessdokumentation. Sollte ein Entwickler eine Verletzung des Designs (z.B. der Architektur) hervorrufen, ist zu klären, ob diese Verletzung aufgrund eines Mangels an Wissen über die adäquate Nutzung entstanden ist oder ob (etwa aufgrund neuer Anforderungen) eine Anpassung des bestehenden Designs notwendig ist. Im zweiten Fall sind ggf. Design Rationals (als Vertreter der Prozessdokumentation) von Bedeutung, um die Beweggründe, die hinter einem bestehenden Design stehen, nachvollziehen zu können. Zunächst Bedarf es somit einer aktuellen und ausreichend präzisen Definition des Systemdesigns (Systemdokumentation), um Designabweichungen überhaupt diagnostizieren zu können. In vielen Projekten ist bereits diese nicht vorhanden, so dass keine Klarheit über die gewünschten Strukturen im System herrschen. Aus welchen Gründen eine Architektur auf eine bestimmte Art gestaltet wurde, ist erst für den Fall einer Designmodifikation interessant (architectural drift [PW92]). Hierbei ist zu beachten, dass möglichst nicht der gesamte Designprozess in archäologischer Art nachvollzogen werden muss, um die Beweggründe des Designs verstehen zu können [PC85].

Es ist zu beachten, dass Rationals zur deskriptiven Dokumentation zu zählen sind, da sie meist nach der Schaffung eines Designs dessen Entstehungsprozess beschreiben, um die Informationen über die getroffenen Entwurfsentscheidungen für die künftige Weiterentwicklung zu bewahren. Als konstruktive Prozessdokumentation wären beispielsweise Zeit- und Budgetplanungsdokumente zu betrachten.

Diese Arbeit fokussiert sich auf die Erhaltung des Wissens über das Solldesign und die Detektion von Verstößen durch kontinuierliche statische Analyse. Somit handelt es sich dabei um eine Integration von deskriptiver Systemdokumentation mit dem System selbst.

Veraltung von Dokumentation. In vielen Entwicklungsprojekten ist allerdings zu beobachten, dass deskriptive Dokumentation aus den Augen verloren wird, da im Rahmen der Weiterentwicklung eines Softwaresystems meist sehr codefokussiert gearbeitet wird. So fließen Änderungen in das System ein, ohne dass sich die Verantwortlichen darüber bewusst sind, dass damit Teile der Dokumentation ihre Gültigkeit verlieren. Durch die Separierung der Artefakte Code und Dokumentation ist es oftmals schwer, einen Überblick über die Zusammenhänge zwischen beiden zu behalten und diese entsprechend konsistent zu halten. Die Veraltung von Dokumentationsartefakten entsteht meist nicht aus "Faulheit" der Entwickler. Meist ist es den Entwicklern nicht bewusst, dass sie gerade einen Teil des Systems verändern, der auch in der Dokumentation angepasst werden muss. Der Grund hierfür liegt darin, dass Dokumentation und Code, abgesehen von Inline-Dokumentation, normalerweise nicht miteinander integriert sind. Die Aktualisierung von Dokumentation ist dadurch sehr aufwändig und teuer, da relevante Stellen nur schwer aufzufinden sind. Aus diesem Grund werden im weiteren Verlauf dieser Arbeit Konzepte vorgestellt, wie diese Integration besser erreicht werden kann und wie

bereits im Forward Engineering darauf geachtet werden kann, dass Informationen im Entwicklungsprozess nicht verloren gehen oder veralten.

Redundanz zwischen Code und Dokumentation. Ein häufiger Fehler im Bezug auf die Dokumentation von Systemen ist, dass Informationen dokumentiert werden, die ohnehin leicht im Code nachvollzogen werden können. Beispielsweise ist es meist einfach zu erkennen, welche Komponenten in einem System vorkommen. Deren Semantik und deren Interaktion ist deutlich wichtiger und schwerer aus dem Code zu ersehen. Im weiteren Verlauf dieser Arbeit werden einige Arten von Informationen aufgezeigt, die in der Implementierung, wenn überhaupt, nur noch implizit vorhanden sind. Diese sollten im Rahmen der deskriptiven Dokumentation eines Systems von vorrangigem Interesse sein. Dokumentation sollte die Unzulänglichkeiten der Code-Repräsentation überbrücken. So ist es wichtig, dass vor allem Konzepte dokumentiert werden, die nur schwer eindeutig aus dem Code reproduziert werden können. Auch Informationen, die nicht mehr direkt im Code identifizierbar sind, wie Anforderungen oder die Nutzung von Mustern und Architekturprinzipien, sollten dokumentiert werden [SLL+88]. Am besten in einer Form, in der sie auch im Code sichtbar und überprüfbar sind.

3.3. Statische Programmanalyse

Unterschiedliche Formen der statischen Programmanalyse verfolgen verschiedene Ziele. Neben der Entdeckung von Fehlern im Programmverhalten wird oftmals auch das Ziel der Identifikation von Verstößen gegen die Wartbarkeit und Verständlichkeit avisiert.

Vollständigkeit und Präzision. Techniken der statischen Analyse zum Nachweis der Korrektheit können anhand der Eigenschaften *Vollständigkeit* und *Präzision* (ähnlich zu den Begriffen 'completeness' und 'soundness' in [LBD+04]) klassifiziert werden. Eine Analysetechnik ist genau dann präzise, wenn alle durch ein Werkzeug angezeigten Fehler auch tatsächlich Fehler im Programm sind. Unterdetektion ist somit nicht ausgeschlossen. Eine präzise Analyse kann somit Fehler (der gesuchten Fehlerklasse) in einem Programm übersehen. Der praktische Einsatz präziser Analysen hat damit den großen Vorteil, dass sich die Behebung jedes gemeldeten Fehlers auszahlt. Jedoch erhält man keine Anhaltspunkte dafür, ob das Programm anschließend (bzgl. einer gewissen Eigenschaft) vollständig korrekt ist. Die Eigenschaft der Vollständigkeit eines Analyseverfahrens ist dadurch definiert, dass jede Fehlermeldung, die durch ein statisches Analysewerkzeug gefunden wird, auch tatsächlich einen Fehler im Programm darstellt. Ein vollständiges Analyseverfahren lässt somit Überdetektion zu. Im Analyseergebnis können sich also Fehlermeldungen befinden, die nicht zu einem echten Fehler im Programm korrespondieren. Die Anwendung derartiger Analysen in der Praxis ruft somit oftmals einen hohen Aufwand hervor, um eine aufgrund von Überdetektion zu große Menge an Fehlermeldungen zu untersuchen. In manchen Fällen sind die Fehlerraten derart hoch, dass dieser Aufwand den Rahmen der Anwendbarkeit sprengt. Wird dieser Aufwand jedoch aufgebracht, kann man anschließend die Behauptung aufstellen, dass das System frei von einer bestimmten Klasse an Defiziten ist. Ein ideales Analyseverfahren müsste somit sowohl vollständig als auch präzise

sein. Dies führt allerdings direkt auf das Halteproblem. Nach dem Satz von Rice [Ric53] sind nicht-triviale Eigenschaften von Programmen, die in Turing-vollständigen Programmiersprachen verfasst sind, nicht entscheidbar.

Heuristische Analysen. Zur Überprüfung bestimmter Wartbarkeitseigenschaften eines Programms werden meist auf Heuristiken basierte Analysen verwendet. Statische Analysewerkzeuge zur Überprüfung der Code-Qualität sind beispielsweise Lint/Splint (C/C++) [Joh77], FindBugs (Java), PMD (Java) oder FxCop (C#). Diese Werkzeuge führen meist einfache Analysen durch, die sogenannte Code-Smells [FBB+99] entdecken, also Stellen im Quellcode, die auf schlampige Programmierung hinweisen. Derartige Werkzeuge erzeugen Warnungen, wenn bestimmte Indizien auf schlecht lesbaren Code, die Verwendung von ungewünschten Sprachkonstrukten (z.B. 'goto') oder potentielle Fehler hindeuten. Ein Beispiel für eine durch viele Werkzeuge überprüfte Regel ist, ob jeder 'case' einer 'switch'-Anweisung mit einer 'break'-Anweisung abgeschlossen wurde. Da das Vergessen einer 'break'-Anweisung ein häufig zu beobachtendes Phänomen ist, kann ein Verstoß durchaus auf einen Laufzeitfehler hindeuten. Es gibt jedoch auch Situationen, in denen ein 'case' absichtlich nicht durch ein 'break' abgeschlossen wurde, da in verschiedenen Fällen die gleiche Reaktion erreicht und Redundanzen vermieden werden sollen. Dies wäre ein typischer Fall eines 'False Positives'.

Ein weiteres Beispiel wäre eine Regel, welche die Verwendung von Zuweisungen innerhalb von 'if'-Anweisungen verbietet. Auch hier gilt, dass jede Zuweisung innerhalb einer Fallunterscheidung angemahnt wird, unabhängig davon, ob sie einen Fehler darstellt oder nicht. Man könnte sich auf den Standpunkt stellen, dass jede Warnung, die nicht mit einem Fehler im Programm korreliert, einen 'False Positive' darstellt. Jedoch gibt es neben der Korrektheit eines Programms hinaus gute Gründe, die Verwendung von Zuweisungen in 'if'-Anweisungen zu verbieten. Beispielsweise sind derartige Programmabläufe deutlich schwerer zu verstehen und damit auch zu ändern. Auch die Lesbarkeit des Codes leidet darunter, da man das Zuweisungszeichen fälschlicherweise übersehen und als Gleichheitszeichen interpretieren könnte. Die Beschreibung des implementierten Algorithmus ist durch die Verwendung derartiger Konstrukte somit impliziter Natur. Die zwingende Verwendung einer 'break'-Anweisung in jedem 'case'-Statement könnte man auf ähnliche Art und Weise begründen. Jedoch gibt es Situationen, in denen 'case'-Anweisungen ohne ein 'break' durchaus auch sinnvoll sein können. Eine Restriktion, die derartiges verbietet, könnte den Widerspruch von Programmierern hervorrufen. Eine Entscheidung, ob Regeln immer zwingend eingehalten werden müssen, obwohl sie keine direkte Auswirkung auf die Korrektheit haben, ist somit nicht immer einfach zu treffen.

Es gängige Praxis im Rahmen der Definition von Qualitätsanforderungen an ein Projekt derartige Regeln zu nutzen, um einheitliche Coding-Guidelines zu etablieren. Werden bestimmte Regeln etabliert und keinerlei Verstöße toleriert, handelt es sich dabei faktisch um zusätzliche Constraints auf der Syntax der verwendeten Programmiersprache.

Abstraktionsniveau statischer Analysen. Die meisten statischen Analysen arbeiten somit auf dem Abstraktionsniveau der Programmiersprache. Es werden immer Einschränkungen bezüglich der Verwendung bestimmter Konstrukte der Programmiersprache definiert. Diese

Coding-Standards gelten im Allgemeinen immer global für alle Teile des Systems. Konzepte, die durch die Implementierung von Bibliotheken in ein System eingebracht werden, sind nur im Falle weniger Standard-Bibliotheken definiert [GS06]. Systemspezifische Deklarationen, wie beispielsweise eine Restriktion, dass ein bestimmter Datentyp oder eine Funktionen nur in einem bestimmten Kontext verwendet werden darf, können meist nicht adressiert werden. Derartige Restriktionen spiegeln die Vorstellungen des API-Erstellers über die adäquate Nutzung der von ihm erstellten Schnittstellen wider und würden Fehlinterpretationen seitens der Nutzer vorbeugen. In gewisser Weise können sie als Coding-Guidelines für APIs verstanden werden. Im Rahmen dieser Arbeit werden Techniken vorgestellt, um das Abstraktionsniveau statischer Analysen zu heben. Derzeit befinden sich diese Analysen auf dem Niveau der Konstrukte, die Programmiersprachen zur Verfügung stellen. Wünschenswert wäre jedoch ein Abstraktionsniveau, das auch die in Frameworks und Bibliotheken implementierten Deklarationen umfasst.

3.4. Entwicklung von Sprachen

Sprachen sind die wichtigsten Werkzeuge in der Softwareentwicklung. Heute treten Sprachen in verschiedensten Formen auf. Neben den großen Allzweckprogrammiersprachen wie C, Java, C# etc. gibt es viele kleine Helfersprachen, die in unterschiedlichen Einsatzbereichen einen speziellen Zweck erfüllen. Beispiele hierfür wären etwa SQL, dot [KN93], Make und viele andere mehr [Ben86]. Neben diesen rein textuellen Sprachen existieren noch zahlreiche verschiedene graphische Sprachen, wie etwa AutoFocus oder UML, die über CASE-Tools (**C**omputer **A**ided **S**oftware **E**ngineering) nutzbar sind.

Die meisten Sprachen werden verwendet, um aus ihnen weitere Artefakte zu erzeugen. So bieten viele CASE-Tools die Generierung von Code in Programmiersprachen an. Allzwecksprachen werden genutzt, um schließlich ausführbaren Binärcode zu erzeugen. Einige Sprachen, wie SQL oder HTML werden auch interpretiert.

Um die Erstellung eigener meist graphischer (domänenspezifischer) Sprachen zu erleichtern, wurden generische Modellierungsumgebungen (engl. Generic Modelling Environment) entwickelt. Diese Werkzeuge sind in gewisser Weise generische CASE-Tools, die anstatt ein bestimmtes Metamodell einer Sprache zu unterstützen den Benutzer ein eigenes Metamodell entwickeln lassen, auf dessen Basis schließlich Editoren erzeugt werden können. Somit unterstützen diese Werkzeuge, die nach Martin Fowler [Fow05] auch "Language Workbenches" genannt werden, die effiziente Entwicklung von eigenen meist graphischen Sprachen. Neben der Erstellung von Editoren, bieten diese Werkzeuge auch Unterstützung bei der Implementierung von Generatoren. Beispiele für diese Art von Werkzeugen sind MetaEdit+ (MetaCase) [Tol04], DSL-Tools (Microsoft), GME (Vanderbilt) [Dav03] oder das Eclipse Modeling Framework (EMF) und Graphical Modeling Framework (GMF) [Fou].

Im Folgenden soll kurz vorgestellt werden, wie eine Sprache grundsätzlich, unabhängig von speziellen Ausprägungen der Sprachentwicklung, aufgebaut ist. Jede Sprache besteht aus drei Teilen:

- *Syntax*
  Traditionell wird die Syntax einer Sprache durch eine Grammatik beschrieben. Da nur kontextfreie Sprachen effizient geparst werden können, nutzen Parsergeneratoren meist eine an die Backus-Naur Form angelehnte Notation zur Beschreibung der Produktionen einer Grammatik. Language Workbenches nutzen gewöhnlich eine Art von Datenmodellierungstechnik, die meist an Klassendiagramme oder E/R-Diagramme erinnern. Als Standard für die Metamodellierung wurde von der OMG die Meta Object Facility (MOF) [OMG06a] definiert. In vielen Werkzeugen werden jedoch proprietäre Metamodellierungstechniken genutzt (z.B. von Microsoft's DSL-Tools oder MetaEdit+). Es ist ein bekanntes Problem, dass sowohl Datenmodelle als auch kontextfreie Grammatiken nicht immer in der Lage sind, bestimmte Restriktionen einer Sprache auszudrücken. Dadurch wird durch das Metamodell bzw. die kontextfreie Grammatik meist eine Obermenge der erlaubten Worte beschrieben [Fei06]. Um die ungewünschten Worte zu entfernen, sind zusätzliche Constraints notwendig. Diese Constraints müssen meist in den Parser bzw. in der Language Workbench durch Nutzung einer Programmiersprache ausprogrammiert werden. In machen Fällen ist auch eine deklarative Beschreibung der Constraints, beispielsweise in Form einer Constraint-Sprache wie OCL [OMG06b] möglich.

  Die Syntax einer Sprache setzt sich aus der konkreten und der abstrakten Syntax zusammen. Dabei definiert die konkrete Syntax die Repräsentation der Sprachworte dem Nutzer gegenüber. Die abstrakte Syntax ist die Repräsentationsform, die benötigt wird, um ein Sprachwort zu transformieren, zu interpretieren oder etwas auf dem Wort zu evaluieren. Durch die konkrete Syntax wird festgelegt, wie Sprachworte erstellt werden können. Im Fall von textuellen Sprachen wird lediglich ein einfacher Texteditor benötigt. Ein fertiges Programm kann schließlich durch einen Parser eingelesen und ein abstrakter Syntaxgraph erzeugt werden. Bei graphischen Sprachen ist dies etwas komplizierter, da normalerweise ein spezieller Editor notwendig ist, der Operationen anbietet, Sprachworte in Form von graphischen Modellen zu erstellen und zu manipulieren. Eine Grammatik definiert grundsätzlich sowohl die abstrakte als auch die konkrete Syntax einer textuellen Sprache. Für graphische Sprachen wird lediglich die abstrakte Syntax durch ein Metamodell beschrieben. Zur Festlegung der konkreten Syntax sind somit weitere Spezifikationen in speziellen Beschreibungssprachen (vgl. Microsoft DSL-Tools) oder die manuelle Programmierung eines Editors notwendig.

  Ein abstrakter Syntaxgraph kann unabhängig davon, ob die Syntax einer Sprache durch eine Grammatik oder ein Metamodell definiert ist, als Graph verstanden werden, dessen Knoten Instanzen der Metamodellelemente bzw. der Nonterminale der Grammatik sind. Die Gleichwertigkeit beider Syntaxbeschreibungstechniken wird auch in [AP04] gezeigt, indem eine bidirektionale Abbildung von MOF-basierten Metamodellen zu kontextfreien Grammatiken definiert wird.

- *Semantik*
  Neben einer Definition der Syntax benötigt eine Sprache ein semantisches Fundament. Während die Syntax definiert, welche Worte (graphische Modelle, textuelle Programme oder Kombinationen) eine Sprache enthält, definiert die Semantik einer Sprache die Bedeutung ihrer Worte. Die Grundlage eine Semantikdefinition bildet eine semantische Domäne

[HR04]. Eine semantische Domäne sollte im besten Fall eine mathematisch fundierte Modellierungstheorie sein [Bro95]. Bildet man die Elemente der Syntax einer Sprache auf eine derartige semantische Domäne ab, erhält man eine präzise Definition der Bedeutung der Sprachelemente (vgl. [CSN05, CSAJ05]). Diese semantische Abbildung bildet eine Grundlage für vielerlei Operationen, die auf einer Sprache ausgeführt werden können, wie etwa Refactorings (Umformung einer Worts in ein anderes Wort der gleichen semantischen Bedeutung) oder die Nutzung von Beweisverfahren, die auf der semantischen Domäne beruhen. Auch Code-Generatoren können auf Basis der semantischen Definition auf Korrektheit überprüft werden. Gerade für professionelle Sprachen zur Entwicklung von (ggf. sicherheitskritischen) Systemen sollte eine fundierte semantische Basis existieren. Die der Theorie zugrunde liegende Mathematik dient dabei als allgemeingültige, eindeutige und verständliche Basis zur Kommunikation der Bedeutung der Sprache.

Im Bereich des Compilerbaus findet man häufig den Begriff "Statische Semantik" vor. Hiermit werden meist kontextsensitive Eigenschaften bezeichnet, die nicht in der kontextfreien Grammatik eines Parsergenerators definiert werden, sondern darüber hinausgehend auf Gültigkeit geprüft werden müssen (z.B. es dürfen nur Variablen genutzt werden, die zuvor deklariert wurden). Diese Begriffsgebung ist jedoch irreführend, da es sich hierbei nicht um eine Semantikdefinition handelt, sondern lediglich um syntaktische Einschränkungen, die durch kontextfreie Grammatiken nicht ausgedrückt werden können. Es ist jedoch zu beachten, dass derartige syntaktische Einschränkungen oftmals durch semantische Probleme hervorgerufen werden. Beispielsweise ist ein arithmetischer Ausdruck, der nicht deklarierte Variablen enthält, semantisch nicht interpretierbar. Aus diesem Grund sollten derartige Ausdrücke aus der Menge der Worte einer Sprache ausgeschlossen werden, so dass die Syntax der Semantik gerecht und einem Missbrauch der Sprache vorgebeugt wird. Andernfalls sind die semantischen Konzepte der Sprache nur unzulänglich in der Syntax repräsentiert.

- *Pragmatik*
  Neben der technischen Definition einer Sprache durch Syntax und Semantik ist es notwendig, einer Sprache auch ein methodisches Fundament zu geben, indem definiert wird, wie und zu welchem Zweck sie eingesetzt werden soll. Eine Sprache, die in der Softwareentwicklung genutzt wird, sollte somit definieren, in welchen Phasen diese genutzt werden soll und wie sie methodisch in einen Entwicklungsprozess integriert wird. Auch Beziehungen zu anderen Sprachen (z.B. Anforderungsbeschreibungen zu Entwurfssprachen) können als Teil der Pragmatik einer Sprache definiert werden.

4. Entstehung impliziter Entwurfsregeln

"Die Grenzen meiner Sprache bedeuten die Grenzen meiner Welt."
- Ludwig Wittgenstein

Im Laufe des Entwicklungsprozesses erarbeiten sich die Entwickler eine Vorstellung von der Anwendungsdomäne und dem zu entwickelnden System. Dieses Wissen wird schließlich über Modellierungstechniken und Programmiersprachen ausgedrückt. Obwohl die Beschreibung scheinbar immer detaillierter und damit reichhaltiger wird, geht bei der Transition zwischen den verschiedenen Darstellungsarten sukzessive Information verloren.

Zur Veranschaulichung und Präzisierung der Problematik des schleichenden Explizitheitsverlusts in Softwareentwicklungs- und Wartungsprojekten wird in diesem Kapitel eine Formalisierung des Übergangs von einem mentalen Konzeptmodell zu einer ausführbaren Implementierung eingeführt. Daran wird aufgezeigt, inwiefern eine Synchronität zwischen dem Konzeptmodell und den späteren Entwicklungsartefakten (dem Code) wünschenswert ist und inwiefern die Artefakte implizite Entwurfsregeln enthalten. Das Wissen über die in einem System zu realisierenden Konzepte kann als Konzeptmodell formalisiert werden. Dies wird im folgenden Abschnitt eingeführt. Anschließend wird illustriert, wie, ausgehend von einem Konzeptmodell Entwicklungsartefakte geschaffen werden (Implementierungsmodelle). Es wird dargestellt, welche Informationen des Konzeptmodells diese in sich tragen und welche Informationen bei den Übergängen bis hin zur Implementierung verloren gehen. Die Methoden und Techniken, die in den nachfolgenden Kapiteln eingeführt werden, ermöglichen es, diesen Explizitheitsverlust auszugleichen.

Es wird im Rahmen dieser Arbeit von einer objektorientierten Implementierung als Ziel der Entwicklung ausgegangen, da die objektorientierten Sprachen sehr gebräuchlich sind und umfangreiche Abstraktionsmechanismen bieten. Die aufgezeigten Erkenntnisse sind jedoch auf andere Programmierparadigmen übertragbar.

4.1. Modellbildung: Von der Konzeption zum System

4.1.1. Konzeptmodelle

Bevor ein Entwickler mit der Erstellung eines Modells oder eines Programms beginnen kann, muss er eine Vorstellung in Form eines mentalen Konzeptmodells (conceptual model, analog [Myl92]) haben, das die wesentlichen Konzepte und Eigenschaften definiert. Dabei kann der Modellbildungsprozess selbst wiederum Einflüsse auf das gedankliche Modell haben. Dieses

Konzeptmodell kann als eine Art Ontologie verstanden werden, das die in ein Softwaresystem zu implementierenden Konzepte (vgl. Abschnitt 2.1) und deren Relationen zueinander darstellt. Ontologien sind ein in vielen Forschungsgebieten verbreitetes Mittel, um Wissen zu sammeln, zu systematisieren, zu repräsentieren und zu verarbeiten [GN87]. Sie werden etwa im Bereich der Wissensermittlung (knowledge acquisition), der Wissensrepräsentation (knowledge representation), der Modellierung, der Informationsintegration, der Informationsgewinung/-extraktion (information retrieval) sowie des Wissensmanagements eingesetzt. In [Gua98] ist ein sehr tiefgehender Diskurs über die verschiedenen Arten und Einsatzformen von Ontologien zu finden. Ontologien sind eine explizite Spezifikation einer Konzeptionalisierung [Gru95]. Im Rahmen dieser Arbeit werden Ontologien als Formalisierung des menschlichen Wissens verwendet, das ein Entwickler als mentales Modell entwickelt, bevor die enthaltenen Konzepte in einer Implementierung kodiert werden.

Die in Ontologien formalisierten Konzepte müssen nicht zwingend der realen Welt entstammen. Oftmals werden Begrifflichkeiten genutzt, die lediglich eine bestimmte Aufgabe oder Rolle in einem System repräsentieren (z.B. Konzepte wie 'Observer', 'Controller' oder 'Widget'). Selbst wenn ein Bezug zur realen Welt vorhanden ist, stellen die Konzepte meist nur eine bestimmte Abstraktion des realen Objekts dar, da meist nur spezifische Eigenschaften für das System relevant sind. Beispielsweise wird ein Konzept 'User' im Rahmen einer Systementwicklung nicht die Gesamtheit der Eigenschaften eines Menschen repräsentieren. Vielmehr findet je nach Systemkontext eine Abstraktion auf bestimmte Eigenschaften, wie etwa dem Namen, der Größe oder dem Beruf statt. Viele darüber hinausgehende Eigenschaften werden nicht berücksichtigt.

Während des Entwicklungsprozesses ist es die Aufgabe der Entwickler, ein gemeinsames Konzeptmodell zu gewinnen und dieses sukzessive in technisch nutzbare Modelle/Programme zu überführen. In der Implementierung muss schließlich das gesamte System in der Syntax der verwendeten Programmiersprache(n) ausformuliert und die Konzepte geeignet durch die Konstrukte der Sprache repräsentiert werden. Diese Repräsentationen haben oftmals einige Defizite, die anhand folgender Formalisierung beschrieben werden können.

Formalisierung. Das idealisierte Konzeptmodell kann als (Multi-) Graph formalisiert werden:

$$\mathbb{D} = (\mathbb{D}_N, \mathbb{D}_E, e_{\mathbb{D}})$$

Die Knoten dieses Graphen repräsentieren Domänenkonzepte $\mathbb{D}_N$. Diese Konzepte stehen in verschiedensten Relationen zueinander, die über die Menge der Kanten $\mathbb{D}_E$ repräsentiert sind. Die Kanten verbinden beliebige Konzepte miteinander und sind wiederum durch einen natürlichsprachlichen Bezeichner gekennzeichnet. $e_{\mathbb{D}}$ ist eine Funktion, welche die Kanten des Graphen definiert:

$$e_{\mathbb{D}} : \mathbb{D}_N \times \mathbb{D}_N \mapsto \mathbb{D}_E \cup \{\}$$

Abbildung 4.1 zeigt eine Visualisierung eines Konzeptmodells als Graph (oben) und eine entsprechende formalisierte Repräsentation (unten). Obwohl Ontologien und damit auch Kon-

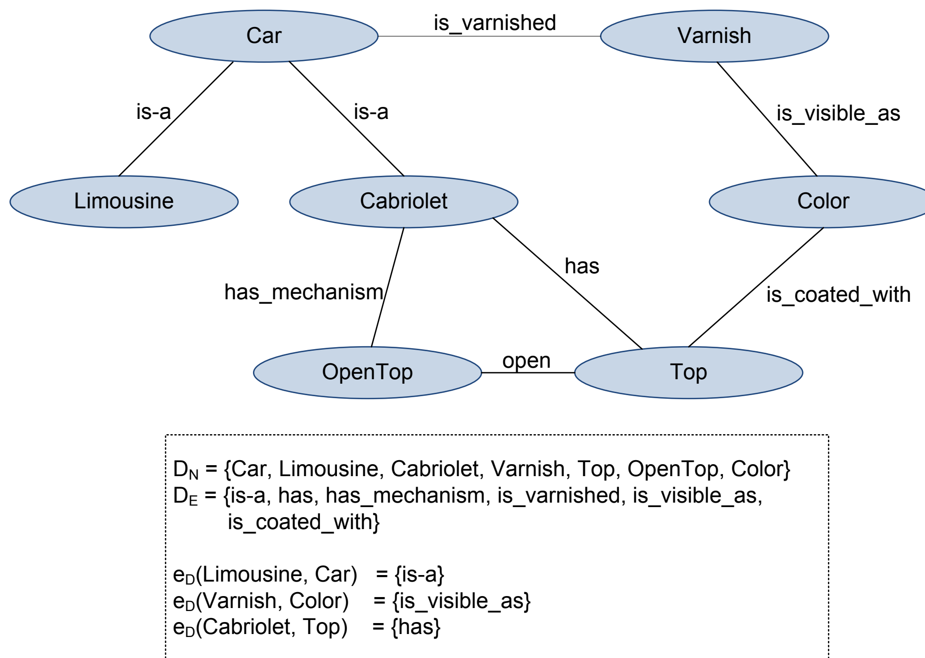


Abbildung 4.1.: Ein Beispiel eines Konzeptmodells

zeptmodelle Datenmodellen sehr ähnlich sind, sollten die Elemente eines Konzeptmodells nicht nur als Elemente verstanden werden, wie sie als Entitäten in einem Datenmodell vorkommen können. Auch Zustände und Transitionen, Dienste, Features oder sogar Hardwareelemente können Bestandteil eines derartigen Modells sein (im folgenden Abschnitt wird dies näher erläutert). Ein Konzeptmodell soll die Gesamtheit der Vorstellungen eines Systems umfassen. Eine Realisierung eines Systems entspricht schließlich einer Transformation eines Konzeptmodells in eine gegebene Modellierungs- oder Programmiersprache. Die in diesem Abschnitt definierten Konzeptmodelle sind ein Formalismus, um Information möglichst ungebunden von syntaktischen Vorgaben einer Modellierungs- oder Programmiersprache darzustellen. Selbstverständlich liegt auch den Konzeptmodellen eine Syntax zugrunde, diese wird jedoch bewusst offen gewählt. Trotz der Definition einer graphischen Repräsentation, wie in Abbildung 4.1 bespielhaft dargestellt, sollen Konzeptmodelle im Rahmen dieser Arbeit nicht als Modellierungssprache vorgestellt werden, die in der praktischen Softwareentwicklung eingesetzt werden soll. Sie dienen lediglich dazu, den schleichenden Informationsverlust im Laufe eines konventionellen Entwicklungsprozesses zu illustrieren.

Die Rolle eines Konzeptmodells. In Konzeptmodellen werden Konzepte und Relationen definiert, die teilweise bereits aus den Anforderungen hervorgehen und im Laufe der Entwicklung immer weiter angereichert werden. Dem Modell liegt lediglich eine intuitive Semantik zugrunde, die sich über die Namen der Konzepte und Relationen ergibt. In einem Projektteam muss ein eindeutiges Verständnis des Konzeptmodells ('ontological commitment') vorherrschen. Das

bedeutet, alle Mitglieder müssen sich bzgl. der Benennung und der Bedeutung der Konzepte sowie der Beziehungen zwischen diesen einig sein. Ist dies nicht der Fall, treten Widersprüche und Missverständnisse zu Tage. Ein Konzeptmodell liegt selten in expliziter Form als Entwicklungsartefakt vor. Ein Glossar ist bereits eine erste einfache Form, um eine gemeinsame Nomenklatur im Projekt zu schaffen. Eine systematische Definition der zentralen Konzepte ist der erste Schritt zu einem gemeinsamen Verständnis des zu entwickelnden Systems. Dieses Wissen sollte in Projekten bewahrt werden, um Wartbarkeit sicherzustellen.

In der Reverse Engineering Literatur [CCI90, BMW93, RW02] wird darauf verweisen, dass es eine der wichtigsten Herausforderungen in dieser Disziplin ist, die Basiskonzepte auf dem Quellcode zu extrahieren. Dies sind eben die Konzepte, die das Konzeptmodell eines Systems zentral prägen. In der Literatur zur Dokumentation von Software [Bri03, FL02] finden sich oftmals Aussagen, dass selbst veraltete Dokumentation einen Wert besitzt, obwohl die direkte Übertragung auf die Implementierung nicht mehr einfach möglich ist. Der Grund dafür ist, dass die Unterschiede in der Dokumentation meist eher in den Details als in den Kernkonzepten liegen und man damit trotz der Defizite in der Lage ist, ein Verständnis über die Funktionsweise des Systems und die Denkweise der Entwickler aufzubauen.

Selbstverständlich kann man bei der Systementwicklung nicht davon ausgehen, dass ein vollständiges Konzeptmodell bereits vor dem Start der Implementierung vorliegt. Während der Entwicklung findet immer wieder ein Erkenntnisgewinn statt, der zu Veränderungen im Konzeptmodell führt. Im Rahmen dieser Arbeit soll dies jedoch nicht weiter betrachtet werden. Da herausgearbeitet werden soll, wie ein Konzeptmodell in Entwicklungsartefakten dargestellt werden kann, wird idealisiert davon ausgegangen, dass bereits zu Beginn ein vollständiges Modell vorliegt.

4.1.2. Implementierungsmodelle

Während der frühen Phasen des Entwicklungsprozesses besteht die Aufgabe der Entwickler vor allem darin, die Domäne zu verstehen und Anforderungen an das System zu definieren. Dabei entsteht die erste Vorstellung davon, welche Domänenkonzepte existieren, wie sie zusammenhängen und welche überhaupt für das zu entwickelnde System relevant sind. Gerade die Auswahl der für ein System relevanten Domänenkonzepte ist dabei sehr wichtig (Definition der Systemgrenze). Dadurch entsteht ein Konzeptmodell, wie es in Abschnitt 4.1.1 beschrieben wurde, in den Köpfen der beteiligten Personen. Im Rahmen des Entwurfs wird dieses Wissen sukzessive mit immer mehr Details angereichert. Die Erstellung von Modellen hilft dabei bestimmte Aspekte des Systems genauer zu beschreiben. Letztendlich muss das System jedoch immer vollständig in einer Programmiersprache ausformuliert werden. Dabei vollzieht sich ein schrittweiser Übergang von einem Konzept- zu einem Implementierungsmodell.

Formalisierung. Der Wortschatz, der zur Beschreibung eines Systems zur Verfügung steht, ist dabei durch die ausgewählten Modellierungs- und Programmiersprachen bestimmt. Diese Sprachen definieren eine Menge an Konstrukten $\mathbb{K}$, die zur Modellierung instanziiert und

komponiert werden. Die Konstruktmenge einer Sprache entspricht ihrer Menge an Metamodellelementen bzw. den Variablen in ihrer Grammatik. Zusätzlich können über Abstraktionsmechanismen Konzepte definiert werden, die als Elemente zweiter Klasse den Wortschatz erweitern. Abstraktionsmechanismen bieten die Möglichkeit, eine Kombination primitiverer Konstrukte einer Programmiersprache (z.B. eine Sequenz von Anweisungen) zusammenzufassen (vgl. Lambda-Abstraktion). Im Rahmen dieser Arbeit sind lediglich Abstrationsmechanismen von Interesse, die der Abstraktion einen Bezeichner (symbolischer Name) zuweisen, so dass diese Referenzierung aus mehreren Stellen eines Programms erfolgen kann. Anonyme Abstraktionen werden somit nicht betrachtet. Formal sei ein Implementierungsmodell analog zur Definition der Konzeptmodelle wie folgt definiert:

$$\mathbb{I} = (\mathbb{I}_N, \mathbb{I}_E, e_{\mathbb{I}})$$

Die Menge $\mathbb{I}_N$ umfasst dabei alle Implementierungskonzepte, die in einem System durch Abstraktionsmechanismen beschrieben werden. Zur Implementierung von Konzepten müssen geeignete Abstraktionsmechanismen der Programmiersprache gewählt werden. Auch die Relationen zwischen den Abstraktionsmechanismen sollten den Relationen der Domänenkonzepte gerecht werden. Die Implementierung der Domänenkonzepte in Programmen kann als Implementierungsmodell in Form eines gerichteten Graphen mit beschrifteten Kanten angesehen werden. Die Knoten des Graphen entsprechen den im Programm implementierten Konzepten $\mathbb{I}_N$. Alle Deklarationen in einem Programm wären somit als Knoten im Implementierungsmodell repräsentiert. Die Kanten des Graphen bilden die Relationen $\mathbb{I}_E$, die im Programm zwischen den implementierten Konzepten bestehen können. Die genaue Menge der Kantentypen variiert je verwendeter Programmiersprache.

In Java gibt es beispielsweise die folgenden Relationstypen: $hasSupCls$, $hasType$, $hasAcc$, $hasCtr$, $hasMeth$, $hasAtt$, $hasPar$, $invMeth$ etc. Die Semantik der Beschriftungen ist dabei über den Zusammenhang der jeweiligen Abstraktionsmechanismen in der Programmiersprache definiert. $hasSupCls$ repräsentiert beispielsweise die Relation zwischen einer Klasse und ihrer Oberklasse; $hasType$ ist die Relation zwischen einem Attribut und dessen Typ; $hasAcc$ repräsentiert die Relation zwischen einer Klasse und den get/set Methoden (in Java) oder Properties (in C#); $hasCtr$ steht für die Relation zwischen einer Klasse zu ihren Konstruktoren; $hasMeth$ ist die Relation zwischen einer Klasse und ihren Methoden (die weder Konstruktoren noch Properties sind); $hasAtt$ beschreibt die Relation zwischen einer Klasse und ihren Attributen; $hasPar$ steht für die Relation zwischen einer Methode und ihren Parametern; schließlich bedeutet $invMeth$, dass eine Methode eine andere aufruft und damit ein Teil der Funktionalität durch diese Methode erbracht wird. Diese Relationen sind aus der Grammatik einer Programmiersprache ableitbar (vgl. [Rat09]).

Die in einem Implementierungsmodell instanziierten Kanten sind durch die Funktion $e_{\mathbb{I}}$ wie folgt definiert:

$$e_{\mathbb{I}} : \mathbb{I}_N \times \mathbb{I}_N \mapsto \mathbb{I}_E \cup \{\}$$

Zusätzlich sei folgende Hilfsfunktion definiert, die alle Pfade im Implementierungsmodell selektiert, die zwei bestimmte Knoten verbinden:

$$p_{\mathbb{I}} : \mathbb{I}_N \times \mathbb{I}_N \mapsto Seq_{\mathbb{I}_E, \mathbb{I}_N} \cup \{\}$$

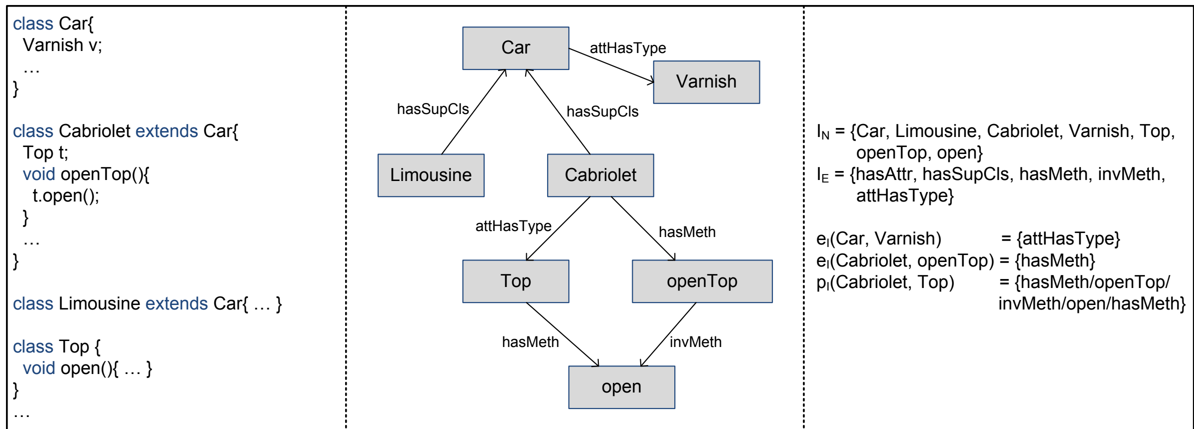


Abbildung 4.2.: Drei Repräsentationen eines Implementierungsmodells

In Abbildung 4.2 ist beispielhaft eine Implementierung des im vorigen Abschnitt eingeführten Konzeptmodells dargestellt. Auf der linken Seite ist eine fragmentarische Implementierung in Java zu sehen, in der Mitte ist eine Repräsentation als Graph des dahinter verborgenen Implementierungsmodells abgebildet und auf der rechten Seite ist analog die entsprechende Formalisierung angegeben.

Implementierungsmodelle wurden unabhängig von konkreten Beschreibungstechniken eingeführt, um im weiteren Verlauf dieser Arbeit den Verlust an Explizitheit eine bestimmten Beschreibung zu illustrieren. Jede Art von Modell oder Programm kann somit als Implementierungsmodell eines Konzeptmodells verstanden werden. Der in dieser Arbeit verwendete Begriff des Implementierungsmodells stellt somit eine Abstraktion von konkreten Beschreibungstechniken dar.

4.1.3. Von Konzept- zu Implementierungsmodellen

Im Lauf der Entwicklung eines Systems müssen Konzepte sukzessive in formalen Sprachen erfasst werden. Diese Sprachen umfassen verschiedene Modellierungstechniken und Programmiersprachen. Die intuitiven Konzepte müssen hierfür durch die Konstrukte, die diese Sprachen bieten, geeignet ausgedrückt werden.

Exkurs: Bedeutung syntaktischer Form. Abbildung 4.3 zeigt ein Bild, das in zwei verschiedene Arten von Puzzles zerschnitten wurde. Versucht man diese Puzzles zu lösen, sucht man Teile,

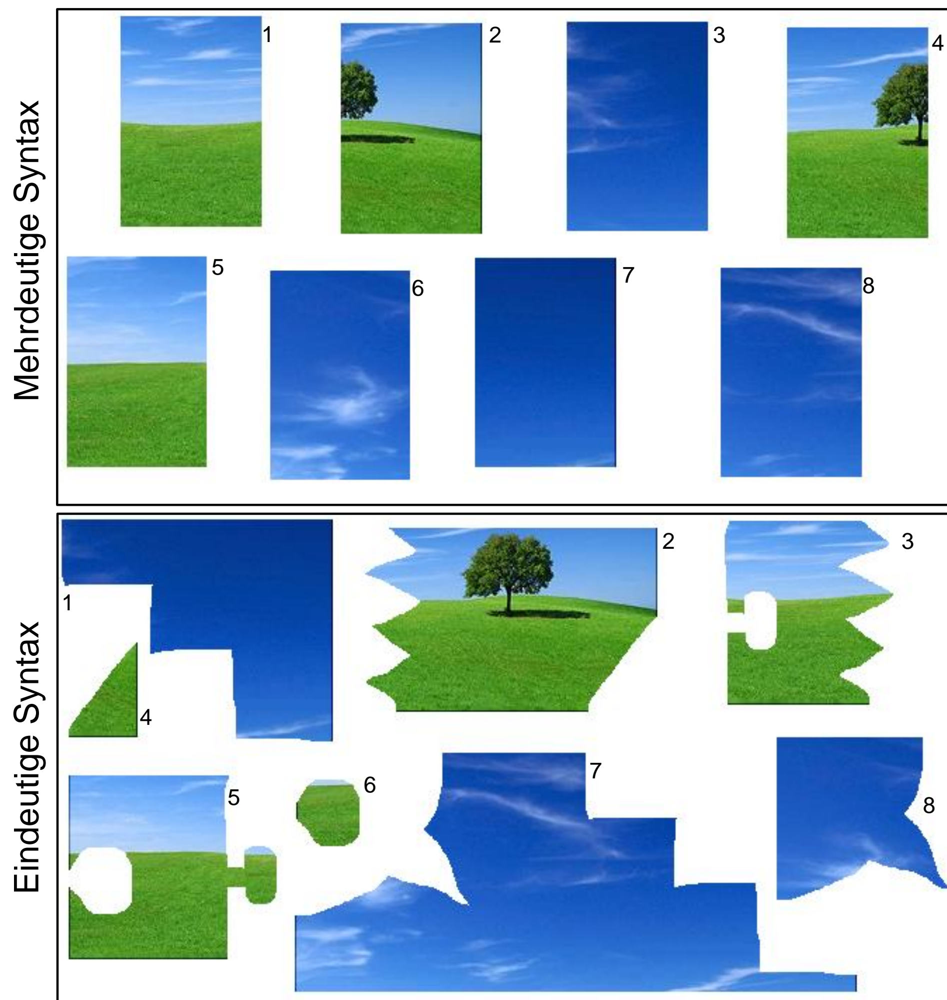


Abbildung 4.3.: Puzzle

die zueinander komponierbar sind. Hierfür hat man immer zwei Orientierungsmöglichkeiten: 1) die Übereinstimmung der Form der Teile (Syntax) und 2) die Gleichartigkeit der Motivteile (Semantik). Ein Puzzle ist auch ohne Kenntnis der Semantik eindeutig lösbar, wenn zusammenpassende Teile eine eindeutige Schnittstelle aufweisen und jedes Teil somit mit genau einem anderen Teil komponierbar ist. Aus diesem Grund ist das Puzzle, das in Abbildung 4.3-unten dargestellt ist, einfacher lösbar als das sehr homogene rechteckige Puzzle (oben), in dem viele Teile existieren, welche die gleiche Form aufweisen und die damit mehrfach komponierbar sind. Gerade wenn die Motive der Teile sehr ähnlich sind, kann ein Spieler nicht mehr entscheiden, welche Teile zusammengehören. Das rechteckige Puzzle weist somit eine höhere Komplexität auf, da die syntaktische Form nur wenige Rückschlüsse auf die adäquate Komposition der Teile zulässt und somit Lösungen des Puzzles existieren, die zwar eine (syntaktisch) mögliche Gesamtkomposition ergeben, das Bild (die Semantik) jedoch entstellt ist. Im unteren Puzz-

le sind in der rein syntaktischen Form der Teile bereits reichhaltige Informationen über die adäquate Komposition enthalten. Dadurch könnte man das unten dargestellte Puzzle in weniger Spielzügen lösen.

Auch bei der Komposition von Programmen auf Basis unterschiedlicher APIs stellt einen Entwickler vor eine ähnliche Aufgabe. Auch hierbei stehen einem zwei Hilfsmittel zur Verfügung: 1) die Bezeichner, die einen Rückschluss auf die Semantik gewähren und 2) die durch die Sprache syntaktisch erlaubten Mittel der Komposition. Die syntaktische Form von in Bibliotheken zur Verfügung gestellten Deklarationen erlauben jedoch meist Kompositionen, die nicht adäquat sind und somit der eigentlichen Semantik des Konzepts und der Intention des Erstellers nicht gerecht werden. Im übertragenen Sinne entspricht dies dem Zusammenfügen von Puzzle-Teilen, die ihrer Form nach passen, jedoch das Motiv des Bildes nicht stimmt. Die syntaktische Form einer API stellt Kommunikation zwischen dessen Ersteller und dem Nutzer dar. Ein API-Ersteller ist in seinen Kommunikationsmitteln in der Programmiersprache jedoch sehr eingeschränkt, da er die syntaktische Form nur auf Basis der Abstraktionsmechanismen der Programmiersprache wählen kann.

Kognitionspsychologischer Hintergrund. Basierend auf Erkenntnissen der kognitiven Psychologie differenziert Leveson [Lev00] drei Aspekte, die beim Entwurf von Schnittstellen beachtet werden müssen, um sicherzustellen, dass ein Nutzer alle Informationen verfügbar hat, um effizient eine Problemlösung durchführen und seine Aufgabe mit Hilfe der Schnittstelle lösen zu können:

- Inhalt (content): Die semantische Information, die in der Repräsentation enthalten sein sollte (gegeben die Ziele und Aufgaben der Benutzer).
- Struktur (structure): Das Design der Repräsentation, so dass der Nutzer die benötigte Information extrahieren kann.
- Format (form): Die Notation der Schnittstelle.

Der Inhalt wird im Wesentlichen durch das Konzeptmodell gegeben, das ein Wartungsingenieur oder ein Nutzer einer Schnittstelle reproduzieren können sollte. Die Struktur wird schließlich durch die Wahl einer Kodierung des Konzeptmodells in eine Modellierungs- oder Programmiersprache festgelegt. Das Format ist meist über die konkrete Syntax der verwendeten Sprache festgelegt und kann lediglich durch die Wahl guter Bezeichner oder durch ein angemessenes Layout und Formatierung (z.B. Zeilenumbrüche, konsistente Groß-/Kleinschreibung, o.ä.) beeinflusst werden.

Explizitheitsverlust. Der Entstehung von impliziten Entwurfsregeln in Artefakten (Modelle, Programme) liegen im Wesentlichen meist zwei Problemstellungen zugrunde: (1) Die Abstraktionsmechanismen von Programmiersprachen sind nicht in der Lage, die gewünschten Informationen/Konzepte immer derart heraus zu faktorisieren, dass einerseits alle Informationen explizit enthalten sind und zugleich Informationen nicht redundant kodiert sind. (2) Neben diesen Unzulänglichkeiten der Sprachen kommt auch noch die Fehlbarkeit der Entwickler hinzu. Oftmals wird die ‘optimale’ Implementierung der gewünschten Konzepte – so

sie denn überhaupt existiert – einfach nicht gefunden. In vielen Fällen stellt man auch fest, dass mehrere Lösungen der Codestrukturierung existieren, die jedoch jeweils unterschiedliche Vor- und Nachteile mit sich bringen. Durch diese Phänomene entstehen Festlegungen (Regeln) über Bestandteile von Modellen oder Deklarationen im Code. Diese Modellfragmente und Deklarationen könnten damit auf eine Art und Weise genutzt oder verändert werden, die den Informationen des Konzeptmodells des Erstellers widerspricht.

Der erste Schritt hin zu einer guten Repräsentation von Konzepten in Programmcode ist, dass die Konzepte in einem Implementierungsmodell wiedererkannt werden können (Concept Location) [BMW93, RW02]. Diese Information ist nur in den Bezeichnern eines Programms vorhanden. Dies führt zu einer enormen Relevanz einer intensional verständlichen Benennung der Bezeichner [AL98, GC91], die es einem Entwickler ermöglichen, das Konzeptmodell des ursprünglichen Erstellers zu reproduzieren. Die eindeutige und konsistente Benennung kann durch Techniken wie dem Einsatz eines Bezeichnerverzeichnisses verbessert werden, wie sie in [DP06] vorgestellt wurden. Ein derartiges Verzeichnis ist im Wesentlichen eine Art Glossar der Terminologie und entspricht der Menge $\mathbb{D}_N$ eines Konzeptmodells. Nach Leveson ist dies eine Frage der Identifikation des implementierten Contents.

Abbildung von Konzepten auf die Implementierung. Die Abbildung von Konzepten des Konzeptmodells auf die Elemente eines Implementierungsmodells kann durch die Funktion $m_{\mathbb{D}\to\mathbb{I}}$ ausgedrückt werden:

$$m_{\mathbb{D}\to\mathbb{I}} : \mathbb{D}_N \mapsto \mathcal{P}(\mathbb{I}_N)$$

Wenn man vernachlässigt, dass ein Konzept auf viele Knoten des Implementierungsmodellgraphen abgebildet wird, kann man idealisiert die Funktion $m^{ideal}_{\mathbb{D}\to\mathbb{I}}$ definieren, die jedem Konzept genau einen Knoten des Implementierungsmodells zuordnet:

$$m^{ideal}_{\mathbb{D}\to\mathbb{I}} : \mathbb{D}_N \mapsto \mathbb{I}_N$$

Auch die Relationen zwischen den Konzepten sollten in der Implementierung wiedererkennbar sein. Für alle Relationen in $\mathbb{D}_E$ sollten entsprechende Relationen aus $\mathbb{I}_E$ definiert sein, die der Semantik der Konzeptrelationen möglichst nahe kommen. Mit Hilfe der idealisierten Funktion $m^{ideal}_{\mathbb{D}\to\mathbb{I}}$ kann man nun den Zusammenhang zwischen den Relationen formulieren. Die Abbildung der Konzepte auf die Knoten des Implementierungsmodells sollte in einer Form erfolgen, die einen möglichst geringen Verlust an Information mit sich bringt. Somit wäre es wünschenswert, dass diese Abbildung einen Isomorphismus darstellt und damit die Knoten des Konzeptmodells auf ‘bedeutungsgleiche’ Knoten des Implementierungsmodells abgebildet würden. Es sollte somit gelten:

$$\forall d_1, d_2 \in \mathbb{D}_N : p_{\mathbb{I}}(m^{ideal}_{\mathbb{D}\to\mathbb{I}}(d_1), m^{ideal}_{\mathbb{D}\to\mathbb{I}}(d_2)) \rightleftharpoons e_{\mathbb{D}}(d_1, d_2) \tag{4.1}$$

In Formel 4.1 spielt das Zeichen ‘$\rightleftharpoons$’ die entscheidende Rolle. Es fordert eine Art intensionale Äquivalenz zwischen den beiden Seiten der Gleichung, die es einem Entwickler ermöglicht, von

den Relationen der Konzepte im Konzeptmodell auf das Implementierungsmodell zu schließen. Umgekehrt sollen auch die Relationen im Konzeptmodell (linke Seite) durch die in der Implementierung sichtbaren Relationen (rechte Seite) widergespiegelt werden. Für jede der Relationen im Konzeptmodell sollte also eine Relation in der Implementierung gegeben sein, die semantisch der Beziehung zwischen den Konzepten nahe kommt. Beispielsweise könnte man auf der linken Seite der Gleichung 4.1 die Konzeptmodellrelation 'is-a' und auf der rechten Seite die Implementierungsmodellrelation 'hasSupCls' vorfinden. In diesem Fall könnte man die Gleichung als erfüllt betrachten. Für die in den Abbildungen 4.1 und 4.2 dargestellten Konzept- und Implementierungsmodelle können beispielsweise folgende Zusammenhänge hergeleitet werden:

$\in \mathbb{D}_E$	$\in \mathbb{I}_E$
is-a	hasSupCls
has	attHasType
is_varnished	attHasType
has_mechanism	hasMeth/invMeth

Konzeptmodell eines Stacks. In Abbildung 4.4-oben ist ein Konzeptmodell der Datenstruktur 'Stack' dargestellt. Der Stack ist über die beiden Operationen 'push' und 'pop' definiert und der kann sich im Zustand 'empty' befinden. Diese Konzepte stehen in vielfältigen Relationen miteinander. So hebt eine 'pop'-Operation eine zuvor ausgeführte 'push'-Operation auf. Eine 'pop'-Operation ist nur erlaubt, wenn der Stack im Zustand 'empty' ist. Im unteren Teil von Abbildung 4.4 sind zwei Implementierungsmodelle des Stacks zu sehen. Links ist ein Implementierungsmodell in Form einer algebraischen Spezifikation zu sehen, auf der rechten Seite eine Implementierung in Java. In beiden Implementierungsmodellen kann jeweils ein Gegenpart für die im Konzeptmodell dargestellten Konzepte identifiziert werden. Die Relationen zwischen den Konzepten sind jedoch lediglich in der algebraischen Spezifikation vollständig wiederzufinden. In der Implementierung in Java ergibt sich folgende Abbildung:

$\in \mathbb{D}_E$	$\in \mathbb{I}_E$
isStateOf	hasMeth
defines	hasMeth
OnlyAllowedIfNot	
IsInverseOf	

Für die Konzeptmodellrelationen 'OnlyAllowedIfNot' und 'IsInverseOf' gibt es in der Java-Implementierung keine Entsprechung. Ein Entwickler, der diese API nutzen möchte und bisher nicht mit den 'Stack'-Konzepten vertraut ist, findet die Informationen über diese Zusammenhänge nicht in der Implementierung wieder und könnte somit den Stack auf nicht-adäquate Weise nutzen und somit Fehler in seinem Programm hervorrufen. Dieses Beispiel zeigt, wie durch eine unvollständige Repräsentation von Informationen des Konzeptmodells in einer Implementierung implizite Entwurfsregeln entstehen. Sowohl 'OnlyAllowedIfNot' als auch 'IsInverseOf' sind nicht in der Java-Implementierung ablesbar und stellen somit implizite Regeln bezüglich der Nutzung der Methoden dar.

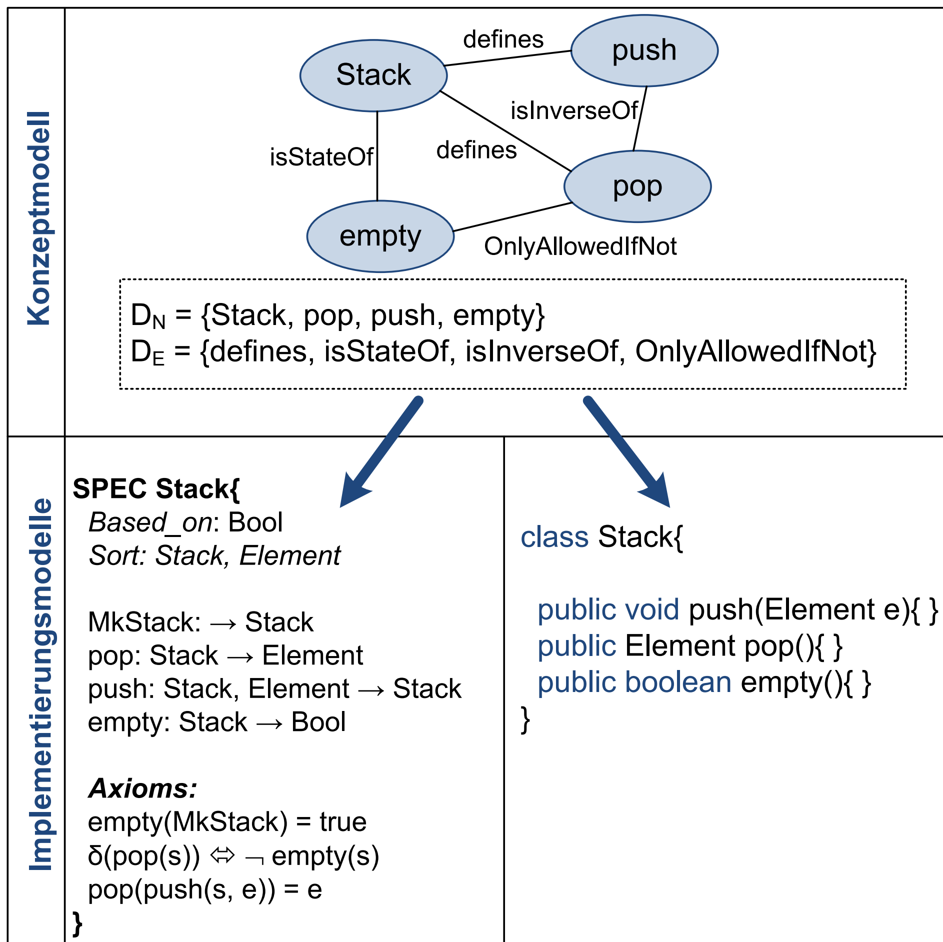


Abbildung 4.4.: Ein Konzept- und zwei Implementierungsmodelle eines Stacks

Programmiersprachen als Implementierungsmodelle. Programmiersprachen bieten meist gute Unterstützung für sehr verbreitete Relationen, wie 'is-a' oder 'has'. In vielen Fällen passen diese Standardrelationen jedoch nicht direkt zu den domänenspezifischen Relationen. Somit müssen diese spezifischen Zusammenhänge umständlich durch Kombinationen der durch die Sprache gegebenen Relationstypen ausgedrückt werden können. Dies sieht man am Beispiel der Relation 'has_mechanism', die durch Kombination zweier Implementierungsrelationen ausgedrückt wird. Je komplexer eine Einbettung einer domänenspezifischen Relation ist, desto schwieriger ist diese zu verstehen. In der Terminologie von Leveson entspricht dies der Identifikation der Struktur.

Diese Kodierung von spezifischen Konzepten und Relationen auf die durch Programmiersprachen zur Verfügung gestellten Abstraktionsmechanismen und deren Verwendungsmechanismen führt zu den im Bereich des Programmverstehens und des Reverse Engineerings bekannten Problemen des *Interleavings* und der *Delocalisation* [RSW95, LS86] von Konzepten in Pro-

grammen. Diese Phänomene erschweren das Programmverstehen, da dadurch der Rückschluss von den Implementierungskonzepten auf die Konzepte des Konzeptmodells nicht mehr eindeutig möglich ist [Rat09]. Unzulänglichkeiten von Abstraktionsmechanismen spiegeln sich auch in der "Tyrannei der dominanten Dekomposition" [TOHS99] wider, die es verhindert, dass querliegende Aspekte (Konzepte) im Programm dargestellt werden. Dies führt dazu, dass diese querliegenden Aspekte redundant im Programm auftreten, da sie nicht herausfaktorisiert und als singuläres Implementierungskonzept dargestellt werden können. Diese Redundanten stellen implizite Regeln dar, da sie im Programm konsistent weiterentwickelt werden sollten.

In den weiteren Abschnitten wird erläutert, wie bei der Nutzung heutiger Modellierungstechniken und Programmiersprachen im Lauf des Entwicklungsprozesses, in dem sukzessive ein initiales Konzeptmodell in eine Implementierung überführt wird, Informationen verloren gehen bzw. lediglich in impliziter Form vorliegen.

4.2. Entstehung impliziter Entwurfsregeln in der Modellbildung

Im folgenden werden häufig verwendete Modellierungstechniken vorgestellt und diskutiert, wie Konzeptmodelle darin formalisiert werden. Es wird aufgezeigt, inwiefern implizite Entwurfsregeln bei der Transition von einem Konzeptmodell zur Modellierung und vom Modell zu einer (objektorientierten) Implementierung auftreten. Hierfür werden verschiedene Modellierungssichten, die unterschiedliche Aspekte bzgl. der Funktionsweise eines Systems beschreiben, daraufhin untersucht, inwiefern bereits in diesen Modellen implizite Regeln auftreten und inwiefern in einer objektorientierten Implementierung zusätzliche implizite Entwurfsregeln einfließen.

Informelle Modelle. In den sehr frühen Phasen des Entwurfs eines Softwaresystems ist es wichtig, in einem Entwicklerteam eine grobe gemeinsame Vorstellung von dem zu erstellenden System oder auch Teilsystem zu schaffen (Konzeptionalisierung). Da das starre syntaktische Korsett einer Modellierungssprache oftmals als hinderlich für die Kreativität erachtet wird, werden in diesem Schritt sehr stark informelle Beschreibungen eingesetzt. In diesem frühen Schritt möchte man sich meist noch nicht einem Modellierungsparadigma unterwerfen. Auch eine formale Semantik ist an diesem Punkt in der Entwicklung noch nicht vonnöten, da noch kein exaktes Verhalten definiert wird, sondern lediglich eine Intuition vermittelt werden soll. In dieser Phase der Modellbildung ist sehr häufig die Nutzung eines Whiteboards ein sehr gutes Mittel, um die Kommunikation im Team zu unterstützen. Abbildung 4.5 zeigt vier typische Whiteboard Zeichnungen, wie sie sehr häufig in Projekträumen anzutreffen sind. Es handelt sich bei derartigen Zeichnungen meist um Mixturen verschiedenster Modellierungstechniken aus unterschiedlichen Bereichen, gemischt mit natürlicher Sprache und Symbolen. Man kann in Abbildung 4.5 deutlich erkennen, dass diese Skizzen keiner strengen syntaktischen Struktur folgen. In der Abbildung werden Teile der Topologie der Plattform, vermischt mit logischen Komponenten, Pfeilen, die eine Art von Datenfluss darstellen, und natürlich-sprachlichen Erklärungen dargestellt. Viele der dargestellten Symbole sind bereits mit Namen beschriftet, die zur Identifikation von Konzepten dienen. Diese Namen finden sehr häufig auch Einzug in die Bezeichner (Identifier) der später entwickelten Software. Die an den Whiteboards verwendete

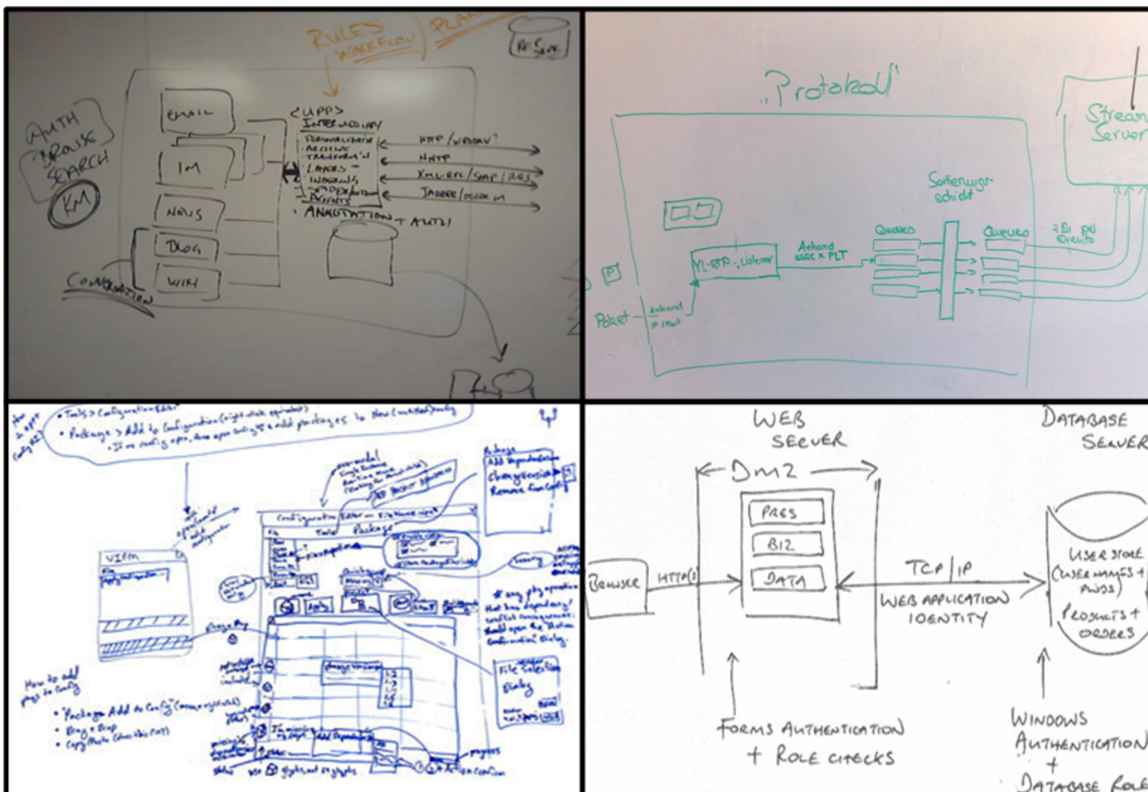


Abbildung 4.5.: Informelle Modellierung am Whiteboard

Modellierungssprache ist nicht statisch, sondern wird während der Konzeption des Systems dynamisch an die Eigenschaften des Systems angepasst, über die gerade diskutiert wird. Bereits in dieser Phase werden meist wichtige Konzepte identifiziert, Begrifflichkeiten geprägt und somit ein initiales gemeinsames Konzeptmodell erarbeitet.

Sobald diese Phase der ersten Konzeptionalisierung abgeschlossen ist und die Entwickler ein grobes Verständnis des Systems und ihrer Aufgabe haben, kann mit der sukzessiven Ausarbeitung und der genaueren Beschreibung des Systems begonnen werden. Da es ab einer bestimmten Größe und Komplexität eines Systems zunehmend schwerer fällt, ein Gesamtverständnis der Funktionalität zu entwickeln, hat es sich bewährt, ein System nicht in totaler Gänze seiner Eigenschaften zu modellieren, sondern Sichten zu bilden. Dabei werden unterschiedliche Modellierungstechniken eingesetzt, die jeweils darauf spezialisiert sind, eine Klasse an Eigenschaften möglichst einfach auszudrücken. Durch Sichten nähert man sich einem Systemverständnis an [Kru95]. Man muss die Sichten verbinden, um ein Gesamtverständnis des Softwaresystems zu bekommen. In den folgenden Abschnitten werden typische Sichten und Modellierungstechniken besprochen.

4.2.1. Datenmodellierung

Datenmodellierung findet meist in Form von Klassendiagrammen oder auch von Entity-Relationship Diagrammen (E/R-Diagrammen) statt, von denen es wiederum unterschiedliche Ausführungen gibt.

Konzeptmodell. In Abbildung 4.6 ist ein Konzeptmodell dargestellt, das Konzepte im Kontext einer Universität darstellt, wie sie in den Verwaltungssystemen einer Universität implementiert sein könnten. Die im Konzeptmodell definierten Relationen sind sehr domänenspezifisch,

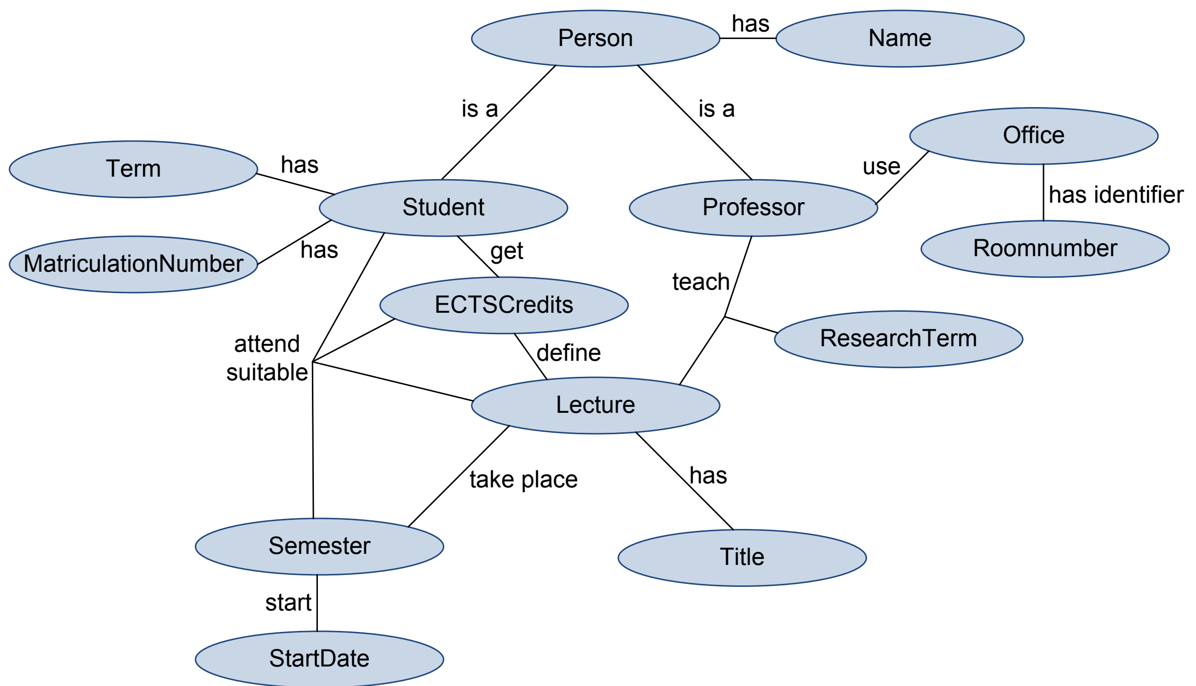


Abbildung 4.6.: Ein Ausschnitt eines Konzeptmodells einer Universität

beispielsweise verbirgt sich hinter der Relation ‘attend suitable’ zwischen den Konzepten ‘Student’ und ‘Lecture’ die Bedingung, dass Studenten nur an für sie geeigneten Vorlesungen teilnehmen. Das bedeutet, dass keine Vorlesung zweimal besucht werden darf und dass das Studiensemester des Studenten den Voraussetzungen der Vorlesung angemessen sein muss. Zusätzlich könnte die Semantik dieser Relation noch umfassen, dass die von einem Studenten in einem Semester besuchten Vorlesungen eine Mindestanzahl an ‘ECTSCredits’ übersteigen muss. Ebenso könnte die Relation “teach” beinhalten, dass Professoren in jedem Semester mindestens eine Vorlesung anbieten, außer wenn sie gerade ein Forschungssemester machen.

E/R-Modellierung. In Abbildung 4.7 ist ein E/R-Diagramm dargestellt, welches das Universitätskonzeptmodell modelliert. In diesem Datenmodell können die zentralen Konzepte direkt

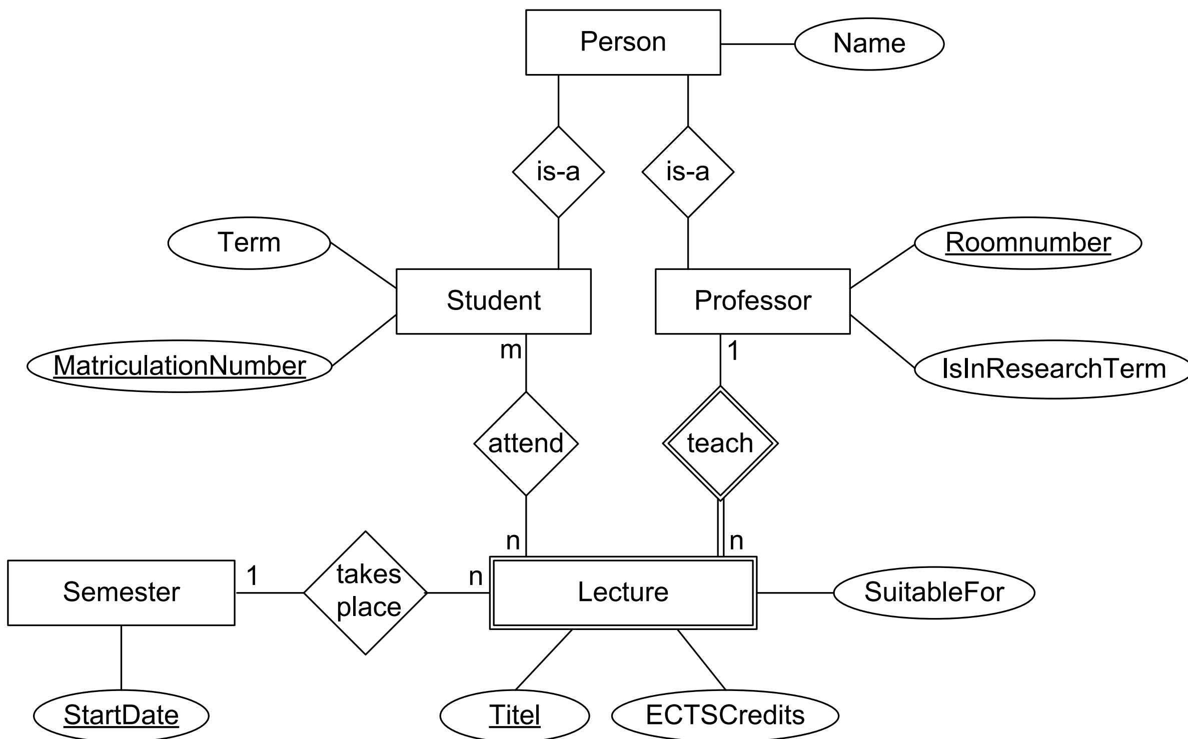


Abbildung 4.7.: Ein E/R-Modell einer Universität

als Entitäten ausgedrückt werden. Die Relationen zwischen den Konzepten werden über Relationships der Entitäten repräsentiert. Analog zur in Abschnitt 4.1.2 angegebenen Formalisierung von Implementierungsmodellen ergeben sich folgende Mengen für dieses E/R-Modell:

$$\begin{aligned}\mathbb{K} =& \{Entity, Relationship, Attribute, Multiplicity\}\\ \mathbb{I}_N =& \{Person, Name, DateOfBirth, Student, Term, Matriculationnumber, Professor,\\ & Roomnumber, IsInResearchTerm, Semester, StartDate, Lecture, Title,\\ & ECTSCredits, suitableFor, attend, teach, takes_place\}\\ \mathbb{I}_E =& \{EntityHasRelationsship, EntityHasAttribute, RelationshipHasMultiplicity\}\end{aligned}$$

Das E/R-Modell repräsentiert die Knoten des in Abbildung 4.6 dargestellten Konzeptmodells explizit als Instanzen von Entitäten und Relationships. Da die Konstrukte eines E/R-Diagramms (die Menge $\mathbb{K}$, festgelegt durch das Metamodell dieser Modellierungstechnik) jedoch auf die in $\mathbb{I}_E$ aufgeführten Beziehungen beschränkt ist, entstehen folgende implizite Regeln: Es ist ein bekanntes Problem, dass die Datenmodellierung nicht ausdrucksmächtig genug ist, um Kontextbedingungen auszudrücken, wie sie beispielsweise die 'attends suitable'- oder 'teach'-Relation definiert. Eine nicht-adäquate Nutzung dieses Schemas wäre somit die Befüllung mit Daten, die diese Kontextbedingungen verletzen. Diese Bedingungen stellen aus Sicht des Datenmodells implizite Entwurfsregeln dar, da die Information über die verpflich-

tende Einhaltung dieser Bedingungen nicht aus dem Modell hervorgeht. Andere Datenmodellierungstechniken oder auch speziellere Varianten von E/R-Diagrammen unterstützen spezielle häufig benötigte Relationships, wie Vererbung, Aggregation oder Komposition. Dadurch können bestimmte Konzeptrelationen zusätzlich ausgedrückt werden, beispielsweise die häufig benötigte Existenzabhängigkeit zwischen Konzepten. Dennoch können sehr domänenspezifische Einschränkungen wie sie im Universitätsschema auftreten nicht vollständig erfasst werden. Nicht alle Konzepte werden direkt in Entitäten übertragen. Konzepte, die am Rand der Domäne liegen, weisen im Konzeptmodell einen geringen Grad auf. Derartige Konzepte werden als Attribute der Entitäten modelliert, denen schließlich ein primitiver Datentyp zugewiesen wird. Dadurch geht oftmals Explizitheit verloren, da beispielsweise nicht jeder String auch eine gültige Raumnummer darstellt. Da Gebäudeteile, Flur- und Etagennummern nicht als Teil der Domäne angesehen werden, müssen diese implizit in einer Zeichenkette als eine Art Untersprache integriert werden. Bestimmte Konzepte werden überhaupt nicht in das E/R-Modell übernommen, im Beispiel etwa das Konzept ‘Office’, das beschreibt, dass die Raumnummer, die einem Professor zugeordnet ist, seinem Büro entspricht.

Implementierung. Eine Implementierung von Datenmodellen erfolgt durch Deklaration von entsprechenden Typen (die oftmals Entitäten in einer relationalen Datenbank entsprechen). Entsprechend dem objektorientierten Paradigma werden diese Typen als Klassen beschrieben. In einer Sprache wie Java wären somit alle Entitäten in Form einer eigenen Klasse repräsentiert und die Felder als Attribute der Klasse mit entsprechendem Datentyp. Entitäten sind somit häufig explizit in der Programmiersprache dargestellt. Felder werden häufig in Form primitiver Datentypen der Programmiersprache repräsentiert. Hier findet oftmals ein Informationsverlust statt, da in vielen Fällen nur eine Teilmenge des Wertebereichs der primitiven Typen auch eine gültige Repräsentation des Feldes im E/R-Diagramm sind. Durch die Kodierung eines Felds in einen Typ findet somit ein Informationsverlust statt. Ein ‘String’ in Java kann zwar einen Namen repräsentieren, jedoch ist nicht jeder gültige String auch ein gültiger Name. Die Relationships zwischen den Entitäten werden meist als Attribute eines entsprechenden Typs repräsentiert. Im Fall von schwachen Entitäten wird diese Kompositionsbeziehung durch Einbettung als Attribut des Typs der schwachen Entität in die starke Entität gelöst. Stehen zwei starke Entitäten in Relation, so hängt die Art der Einbettung von Kriterien ab, die nicht im Datenmodell ersichtlich sind (Kontrollfluss). Zuzüglich zur Einbettung als Attribut sind meist noch Methoden notwendig, die aus einem Objekt heraus den Zugriff auf die in Relation stehenden Objekte erlauben. Im Fall von n:m-Beziehungen behilft man sich im Bereich der relationalen Datenbanken oftmals durch Einführung von Abbildungstabellen. Auch im Bereich der Objektorientierung ist dies gängige Praxis, Relationen beispielsweise durch Nutzung des ‘Koordinator’-Musters (nach [Bal01]) als Klasse zu repräsentieren. Multiplizitäten und Constraints sind in dieser Implementierung jedoch nicht explizit modelliert.

4.2.2. Verhaltensmodellierung

Zur Modellierung von Verhalten werden oftmals Zustandsautomaten eingesetzt. Zustandsautomaten oder auch State Charts genannt, gibt es in sehr unterschiedlichen Arten, beispielsweise

Moore-, Mealy- oder auch Harel-Automaten. In manchen Ausprägungen sind auch Hierarchien in Form von Unterzuständen erlaubt.

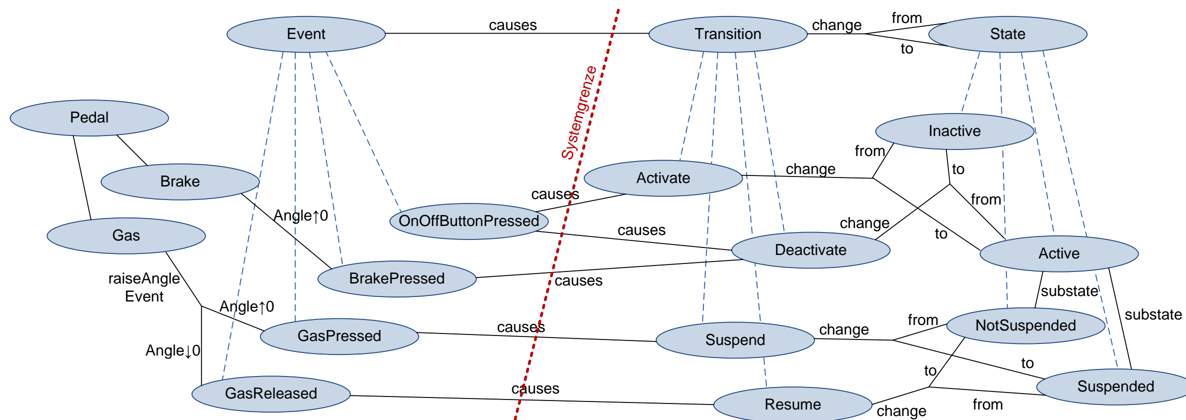


Abbildung 4.8.: Ausschnitt eines Konzeptmodells für einen Teil des Verhaltens eines ACCs

Konzeptmodell. In Abbildung 4.8 ist ein Konzeptmodell des Verhaltens eines ACCs skizziert (entsprechend dem im Projekt DENTUM implementierten System [FFH+09]). Die Konzepte ‘Event’, ‘Transition’ und ‘State’ prägen das grundsätzliche Modell der Ablaufsemantik mit der Vorstellung, dass äußere Ereignisse im System zur Durchführung von Zustandswechseln führen. In der Abbildung des Konzeptmodells sind die Relationen zwischen den konkreten domänen-/systemspezifischen ‘Events’, ‘Transitionen’ und Zuständen speziell dargestellt. Diese Relationen entsprechen einer Art Instanziierungsbeziehung und fordern, dass die Instanz alle Relationen der Typebene implementiert (beispielsweise alle systemspezifischen Events müssen eine ‘causes’-Relation zu Instanzen von ‘Transition’ eingehen).

An diesem Konzeptmodell erkennt man, dass nur ein Teil dieses Modells in ein System implementiert werden soll, aber zusätzlich die Eingaben in das System im Konzeptmodell dargestellt sind. Dadurch soll signalisiert werden, dass bei der Implementierung der Kontext der antizipierten Verwendung von Konzepten verloren geht, beispielsweise Vorgaben über die Reihenfolge der ‘Events’.

Zustandsmodellierung. Das in Abbildung 4.9 gezeigte Zustandsmodell definiert folgendes Implementierungsmodell:

$$\begin{aligned}
\mathbb{K} =& \{State, Transition, Variable, Precondition, Postcondition, Input, Output\} \\
\mathbb{I}_N =& \{acc_inactive, acc_suspended, acc_active, onOff, suspend, brake, ...\} \\
\mathbb{I}_E =& \{StateHasOutTransition, StateHasInTransition, TransitionHasPrecondition, ...\}
\end{aligned}$$

Die Konzepte ‘Transition’ und ‘State’ werden zu Konstrukten der Modellierungstechnik und sind damit als Element erster Klasse im Zustandsautomaten repräsentiert. Dementsprechend

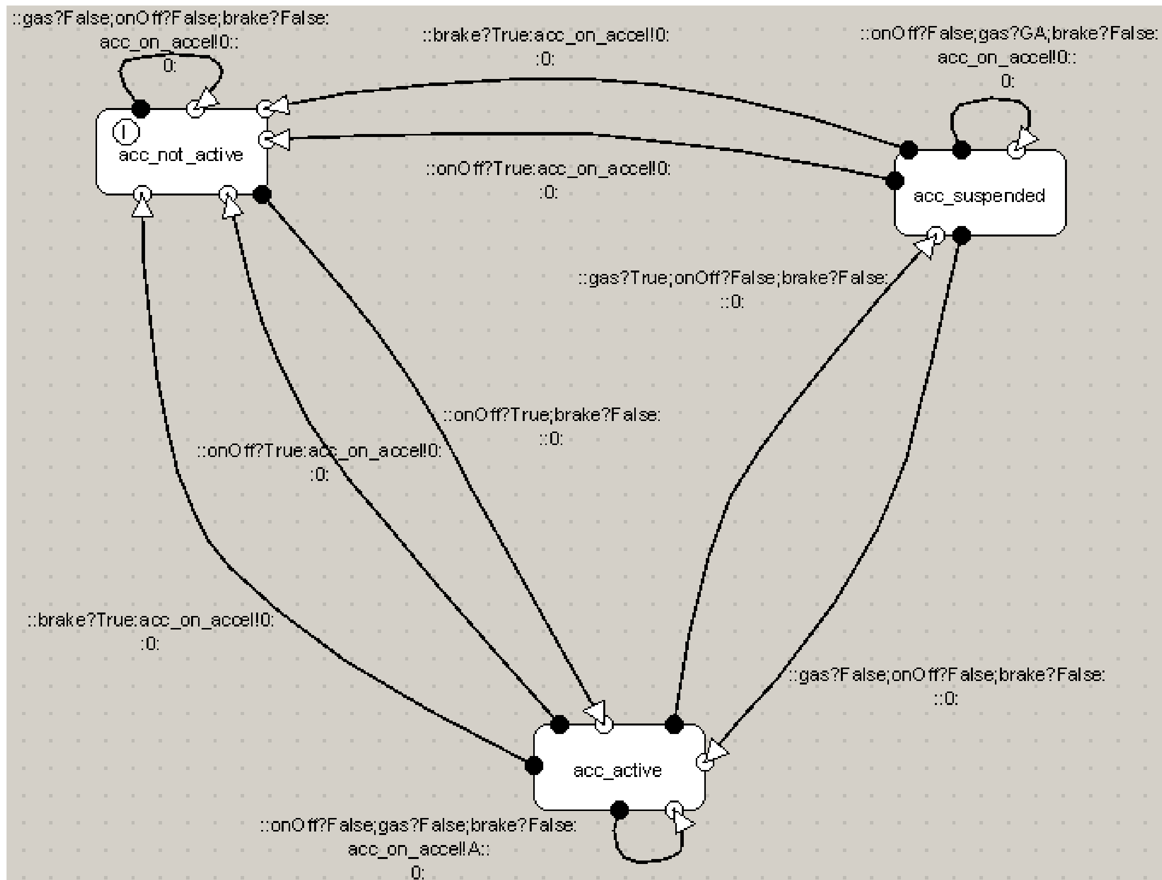


Abbildung 4.9.: Teil eines Zustandsmodells eines ACCs

können die systemspezifischen Zustände als Instanzen dieser Konstrukte modelliert werden. Da die verwendete Modellierungstechnik kein direktes Event-Konzept bietet, müssen die 'Events' in Variablen kodiert werden und deren Werte an den Transitionen überprüft werden. Die Transitionen des Konzeptmodells sind grundsätzlich im Automaten erkennbar, jedoch die Information über die nutzersichtbare Funktionalität, welche die einzelnen Transitionen kodieren, über die Interpretation der Bedingungen wieder herausgefunden werden. Die Zustände 'Suspended' und 'NotSuspended' sind im Konzeptmodell als Unterzustand von 'Active' dargestellt. In der Modellierungstechnik existieren keine Unterzustände, deshalb ist es notwendig, den Automaten flach zu modellieren. Dadurch ergeben sich zusätzliche Transitionen (z.B. von 'acc_suspended' zu 'acc_inactive'), die durch diese Modellierung grundsätzlich möglich wären, aber anhand des Konzeptmodells explizit nicht erwünscht sind. Man sieht, dass in dieser Modellierung bereits implizite Entwurfsregeln existieren, die beispielsweise die Konsistenz bestimmter Teile der an den Transitionen befindlichen Bedingungen fordern (Redundanz). Würde etwa die Teilbedingung 'onOff?False' in den Transitionen zwisch 'acc_suspend' und 'acc_active' an einer der Transitionen verändert, würde die Syntax der Modellierungstechnik dies zulassen, obwohl dies nicht adäquat im Sinne des Konzeptmodells wäre. Die Information über die geforderte Sym-

metrie dieser Transitionen ist somit lediglich implizit vorhanden. Durch eine angemessenere Modellierungstechnik (z.B. Zustandsautomaten mit Unterzuständen) könnte sogar eine noch explizitere Repräsentation geschaffen werden.

Implementierung. Eine Implementierung des ACC-Verhaltens könnte in verschiedener Weise erfolgen. Zwei häufig anzutreffende Varianten sind folgende:

- Implementierungsalternative 1: Variable definieren den Zustandsraum, der Zustand ist damit implizit über die aktuelle Belegung der Variablen definiert. Die Übergänge werden über Methoden als Schnittstelle angeboten. Der Nutzer dieser Schnittstelle muss dabei wissen, in welcher Reihenfolge die Methoden aufgerufen werden müssen. Die eigentliche Verschaltung der Transitionen und der Zustände im Automaten ist somit nur implizit in der Implementierung enthalten.

- Implementierungsalternative 2: Implementierung durch Nutzung des "State" Entwurfsmusters [GHJV95] (siehe Abbildung 4.10). Das Metamodellelement 'State' aus der Modellierungstechnik wird dabei explizit nach Java in Form einer Klasse überführt. Die Zustände 'Active', 'Inactive' und 'Suspended' können damit explizit als Konzepte in Form von Unterklassen von 'State' als konkrete Zustände eingeführt werden. Die erlaubte Reihenfolge der Schaltung der Transitionen ist jedoch nur implizit vorhanden.

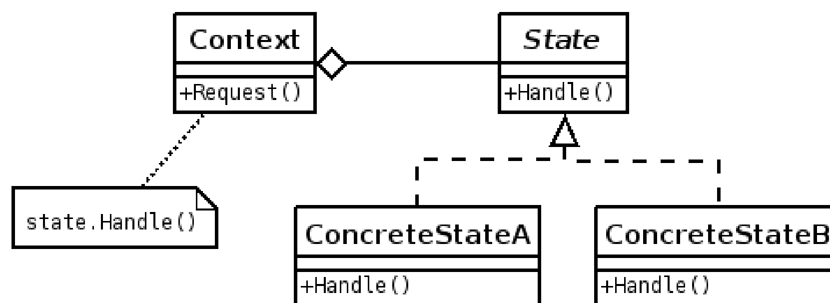


Abbildung 4.10.: Das "State" Design Pattern [GHJV95]

4.2.3. Strukturmodelle

Zur Reduktion der Komplexität wird meist eine Dekomposition eines Systems in Form von Modulen, Komponenten, Paketen oder Namensräumen vorgenommen. Diese bilden oftmals auch die Ausgangsbasis für die verteilte Entwicklung eines Systems in verschiedenen Teams.

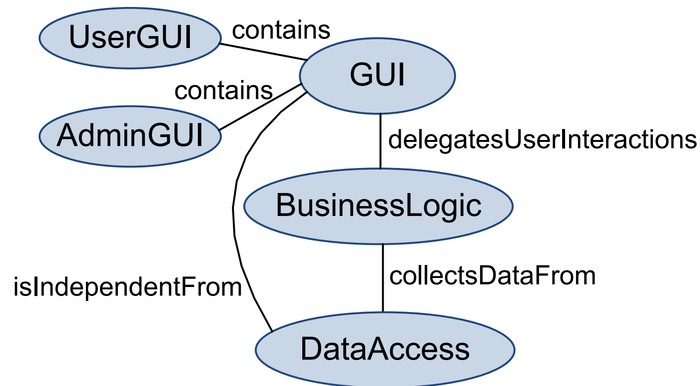


Abbildung 4.11.: Ein einfaches Konzeptmodell einer dreischichtigen Architektur

Konzeptmodell. Konzeptmodelle, welche die Struktur eines Systems betreffen, definieren Bezeichner für bestimmte Teile des Systems (Komponenten). Sie stellen die Art und Weise dar, wie die Teile des Systems zusammen die Gesamtsystemfunktionalität erbringen. Abbildung 4.11 zeigt ein einfaches Konzeptmodell einer dreischichtigen Architektur. Die Vorstellung eines Entwicklers ist hierbei, dass die 'GUI'-Teile des Systems unabhängig von den 'DataAccess'-Teilen ist. Zudem stellt dieses Konzept eine Hierarchie innerhalb der 'GUI'-Anteile dar.

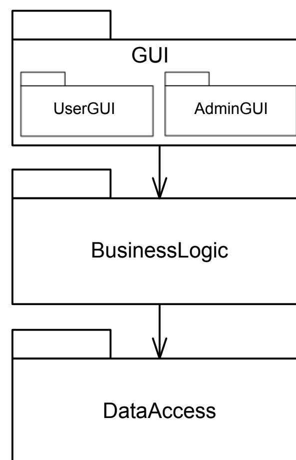


Abbildung 4.12.: Ein Komponentendiagramm der dreischichtigen Architektur

Komponentendiagramme. Zur Modellierung der statischen Struktur werden, wie in Abbildung 4.12 illustriert, meist graphische Darstellungen verwendet, in denen durch Rechtecke die Systemkomponenten dargestellt werden und durch Pfeile zwischen den Rechtecken signalisiert wird, dass eine Komponente auf eine andere zugreift. Abbildung 4.12 liegt folgendes Imple-

mentierungsmodell zugrunde:

$$\begin{aligned}\mathbb{K} =& \{Component, UsageDependency\}\\ \mathbb{I}_N =& \{GUI, UserGUI, AdminGUI, BusinessLogic, DataAccess\}\\ \mathbb{I}_E =& \{isPartOf, hasIncomingUsageDep, hasOutgoingUsageDep\}\end{aligned}$$

Die gewünschte Unabhängigkeit der 'GUI'- von der 'DataAccess'-Komponente ist in diesem Diagramm nur implizit dargestellt (da keine 'UsageDependency'-Kante zwischen diesen Komponenten eingezeichnet ist). Die einzelnen Komponenten und die Hierarchie, denen die Komponenten unterliegen, ist explizit ablesbar. Eine genauere Erläuterung der impliziten Entwurfsregeln in Komponentendiagrammen ist in Kapitel 10 zu finden.

Implementierung. In den meisten Programmiersprachen ist die Dekomposition des Systems direkt in Modul- (C), Namensraum- (C#) oder auch Package-Strukturen (Java) überführbar. Diese Arten Komponenten zu deklarieren, sind somit direkt in den Sprachen als Konstrukte verankert. In Java und C# ist es auch möglich, Hierarchien von diesen Strukturelementen (Komponenten) zu bilden und dadurch die 'isPartOf'-Relationen eines Komponentendiagramms explizit darzustellen. Die Verschaltung (UsageDependency) der einzelnen Komponenten verbleibt jedoch in allen Sprachen implizit. Es geht nicht direkt aus dem Code hervor, welche Abhängigkeiten zwischen den Komponenten erwünscht und welche durch die Architektur ausgeschlossen sind. Dies führt zum langsamen Auseinanderdriften zwischen der dokumentierten Soll- und der implementierten Ist-Architektur im Laufe der Evolution eines Systems (vgl. Fallstudie im folgenden Kapitel 10). Lediglich über Sichtbarkeiten sind einfache Einschränkungen bzgl. der Nutzung der in Komponenten verborgenen Deklarationen möglich.

4.3. Reverse Engineering von Konzeptmodellen

In [RFD$^+$08, RFJ08] wurde eine Technik zur semiautomatischen Extraktion von Domänenwissen in Form von (leichtgewichtigen) Ontologien vorgestellt. Diese Ontologien sollen das in Bibliotheken verborgene Wissen in maschinenverarbeitbarer Form enthalten und somit eine Repräsentation des Konzeptmodells des Entwicklers darstellen. Diese Extraktion ist der Versuch, den Wissensbildungsprozess zu automatisieren, den ein Entwickler vor der ersten Verwendung einer API durchzuführen hat, wenn er lediglich Informationen aus dem Code zur Verfügung hat. Dabei handelt es sich in gewisser Weise um die Rekonstruktion eines Konzeptmodells aus einem Implementierungsmodell und damit um eine Invertierung der Funktion $m_{\mathbb{D}\rightarrow\mathbb{I}}$. Selbstverständlich kann ein Entwickler auf einen Erfahrungsschatz zurückgreifen und ist im Gegensatz zu einem rein automatischen Verfahren in der Lage, die Semantik der Bezeichner zu interpretieren. Abbildung 4.13 illustriert die Intuition hinter der Ontologieextraktion. Man geht von der Annahme aus, dass Entwickler bei der Erstellung zweier APIs, welche die gleiche Domäne adressieren, das gleiche oder sehr ähnliche Konzeptmodelle besitzen, da sie sich mit der gleichen Problemstellung auseinandersetzen (links oben). Dieses Konzeptmodell bilden sie auf die Abstraktionsmechanismen von (potentiell unterschiedlichen) Programmiersprachen ab. Über die Wahl der Bezeichner wird die Semantik der zu repräsentierenden Domänenkonzepte

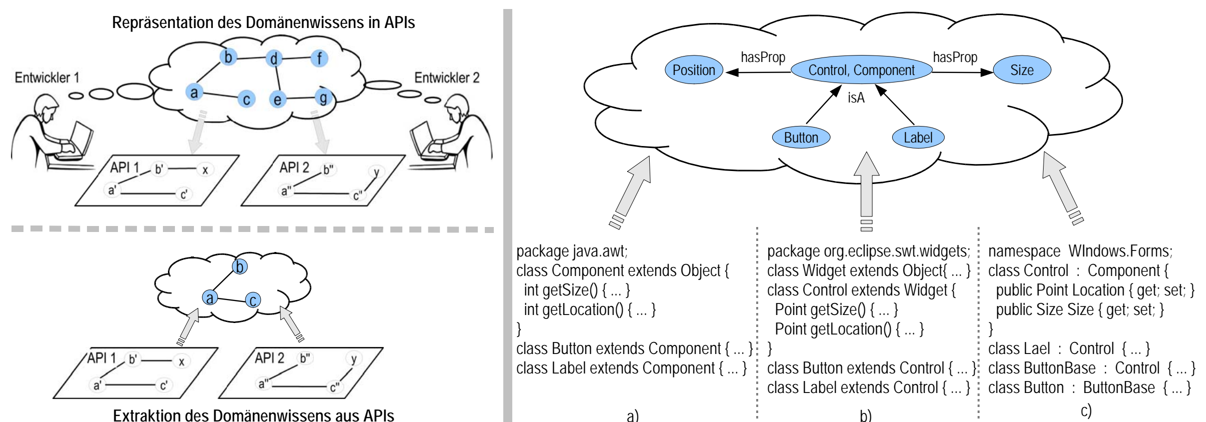


Abbildung 4.13.: Intuition der Ontologieextraktion

in die API eingebracht. Der Extraktionsprozess versucht die Gemeinsamkeiten der APIs zu nutzen, um das in den APIs verborgene Wissen zu reproduzieren und in einer Ontologie zu speichern (links unten). Stehen genügend APIs zur Verfügung, ist man somit in der Lage, eine möglichst weitgehende Ontologie der Domäne zu gewinnen (rechts).

Motivation der Ontologieextraktion. Die Motivation hinter der Extraktion und der Errichtung eines Repositories an derartigen Ontologien, die unabhängig von bestimmten Programmiersprachen Wissen über Programmiertechnologie bereitstellen, liegt in deren potentieller Nutzung für verschiedene Anwendungen im Bereich des Reverse Engineerings [RFD+08]:

- **Konzept Lokalisierung und Programmverstehen.** Ein zentraler Aspekt des Reverse Engineering Problems ist dabei die Konzept Lokalisierung (Concept Location) im Quellcode [BMW93, BMW94, RW02]. Ein Mapping von Programmteilen zu Konzepten in einer Ontologie erlaubt es, Programme gezielt nach den in ihnen implementierten Konzepten zu untersuchen.
- **Erlernen von Bibliotheken.** Nach [CCI90] ist es eines der zentralen Ziele des Reverse Engineerings, eine Repräsentation eines Systems auf einer höheren Abstraktionsebene zu erstellen. Anhand einer Ontologie kann ein Programmierer, der eine ihm bislang nicht vertraute Bibliothek verwenden möchte, einen Überblick über die in der Bibliothek implementierten Konzepte bekommen.
- **Qualitätsbewertung von APIs.** Anhand einer Domänenontologie können APIs bewertet werden, welche die gleiche Domäne adressieren wie eine gegebene Ontologie. Durch eine Analyse, wie die Konzepte in der API implementiert sind, können Situationen erkannt werden, in denen die Implementierung der Konzepte dem in der Ontologie formalisierten Domänenwissen widerspricht (vgl. [RJ07, RD07]).

Da für derartige Aufgaben kaum Ontologien verfügbar sind, die detailliert genug sind und sich auf dem richtigen Abstraktionsniveau für die Programmanalyse befinden, kann die Technik der

Ontologieextraktion aus domänenspezifischen APIs einen Beitrag zur Verbesserung derartiger Analysen leisten. Damit ist es möglich, das Depot an Wissen nutzbar zu machen, das in APIs verborgen liegt.

Technik der Ontologieextraktion. An dieser Stelle soll lediglich der grobe Ablauf der Extraktion beschrieben werden. Tiefergehendere Details können in [RFJ08] nachgelesen werden. Folgende Schritte müssen durchlaufen werden, um eine Ontologie aus einer Menge an APIs zu extrahieren, welche die gleiche Domäne adressieren:

1. Extraktion von Tripeln aus den öffentlichen Schnittstellen: Durch Nutzung eines Parsers oder eines Bytecode-Analyseframeworks müssen die in der öffentlichen Schnittstelle enthaltenen Bezeichner und deren statische Struktur extrahiert werden. In einer Programmiersprache existieren verschiedene Relationen zwischen Bezeichnern, beispielsweise ‘isA’ – durch Vererbung ausgedrückt, ‘isDoer’ – eine Klasse enthält eine Methode, ‘hasType’ – ein Attribut hat einen bestimmten Typ etc. Diese Relationen können in Form von Tripeln extrahiert werden, z.B. ‘Child isA Person’. Die Bezeichner werden dabei nicht einfach übernommen, sondern ggf. in einzelne Worte zerlegt (Pascal-Casing vorausgesetzt) und gefiltert.
2. Identifikation von Domänenkonzepten: Alle Tripel stellen potentielle Kandidaten für Relationen dar, die in die Domänenontologie aufgenommen werden. Aus den aus verschiedenen APIs gewonnenen Tripeln wird nun über einen Graph-Matching Algorithmus ein Maß für die Ähnlichkeit berechnet. Zunächst werden die in den Bezeichnern verwendeten Worte auf Ähnlichkeit überprüft. Ist diese Prüfung positiv verlaufen, werden diese als ein Konzept angesehen und die Relationen überprüft, in denen diese stehen. Sind auch diese ähnlich (nach an Isomorophie), so werden die Teilgraphen in die Domänenontologie aufgenommen.
3. Manuelle Filterung der Ergebnisse: Die erhaltene Ontologie wird abschließend manuell auf Plausibilität überprüft und ungewünschte Konzepte entfernt, die aufgrund von ‘Implementierungslärm’ induziert wurden. Der Aufwand für die manuelle Nachbearbeitung variiert je nach gewähltem Ähnlichkeitsmaß.

Ergebnisse. Ein großer Teil des zum Verstehen von Softwaresystemen notwendigen Wissens ist technischer Natur [AdOD+03, AdOdSD07], wie beispielsweise Wissen über graphische Nutzerschnittstellen, Netzwerktechnik, XML etc. Aus diesem Grund wurden in der in [RFJ08] veröffentlichten Fallstudie derartige technische Bibliotheken herangezogen, um das in ihnen verborgene Wissen zu extrahieren. Zudem bringen viele Programmiersprachen bereits in ihren Standardbibliotheken derartige Funktionalität mit, wodurch eine große Auswahl an Bibliotheken zur Verfügung steht, welche die gleiche Domäne adressieren. In der Fallstudie konnten Ontologien extrahiert werden, die das Wissen aus diesen Domänen enthalten. Diese Ontologien weisen genau das Abstraktionsniveau der analysierten Bibliotheken auf. Eine rein manuelle Erstellung dieser Ontologien wäre aufgrund deren Größe (mehrere hundert Konzepte) mit hohem Aufwand verbunden gewesen. Da alle in den Ontologien enthaltenen Konzepte bereits in mehreren APIs identifiziert werden konnten, können sie als für die jeweilige Domäne relevant betrachtet werden.

Diskussion. Die durch Anwendung dieses Verfahrens in [RFJ08] erzielten Ergebnisse und Erfahrungen zeigen, dass eine rein automatische Wissensextraktion auf Basis der syntaktischen Strukturen einer API nicht möglich ist. Obwohl viele Schritte automatisierbar sind, verbleibt die letzte Entscheidung, ob eine Relation zwischen Konzepten sinnvoll ist oder nicht, beim Menschen. Dies verdeutlicht die Problematik, dass das Verstehen von Programmstrukturen eine intellektuell stark herausfordernde Tätigkeit ist, die nicht auf rein syntaktischem Niveau realisiert werden kann. Die zur Rekonstruktion des Konzeptmodells des API-Erstellers notwendigen Informationen sind aufgrund impliziter Festlegungen und Zusammenhänge in nur sehr ungenügendem Maße in der Syntax von Bibliotheken verankert. Aus diesem Grund ist es dringend notwendig, mehr als nur eine domänenspezifische Bibliothek heranzuziehen, um überhaupt sinnvolle Ontologien extrahieren zu können. Die unterschiedlichen Bibliotheken modellieren die Konzepte der Domäne auf unterschiedliche Art und Weise, so dass in den verschiedenen Bibliotheksimplementierungen immer wieder andere Festlegungen (Relationen) explizit enthalten sind, die durch den Extraktionsprozess erkannt werden können. Erst durch die Kombination der unterschiedlichen Sichten der Bibliotheken auf die gleiche Domäne und deren unterschiedlichen syntaktischen Realisierungen, ist es möglich, brauchbare Informationen zu extrahieren. Durch Nutzung mehrerer Bibliotheken, welche die gleiche Domäne adressieren, wird es ermöglicht, unwesentliche Implementierungsdetails (Rauschen) von tatsächlichem Domänenwissen zu trennen.

Teil II.

Formalisierung impliziter Entwurfsregeln als Constraints

5. Sicherstellung der adäquaten Verwendung von Deklarationen durch Constraints

"Erklären heißt Einschränken."
- Oscar Wilde

In Kapitel 4 wurden Implementierungsmodelle als technische Umsetzung von Konzeptmodellen eingeführt, um die Entstehung von impliziten Entwurfsregeln zu beschreiben. In den folgenden Kapiteln dieses zweiten Teils soll ein leichtgewichtiger Lösungsansatz vorgestellt werden, der es ermöglicht, Entwurfsregeln durch den Einsatz geeigneter Modelle oder auch durch Annotationen im Quellcode explizit zu machen und automatisiert zu überprüfen. Die Leichtgewichtigkeit des Ansatzes besteht darin, dass dieser zum Einsatz auf Standardprogrammiersprachen geeignet ist.

Im Folgenden soll nun zunächst eine Formalisierung von Programmiersprachen sowie eine exakte Definition des Begriffs der Abstraktionsmechanismen einer Sprache gegeben werden. Anschließend wird darauf basierend ein Ansatz vorgestellt, wie durch Spezifikation von zusätzlichen Einschränkungen der Verwendung von Deklarationen (Constraints) die Adäquatheit von Nutzercode sichergestellt werden kann. Der im Weiteren vorgestellte Formalismus wurde bereits in [FR08] veröffentlicht.

5.1. Anatomie von Sprachen

In diesem Kapitel wird ein formales Framework eingeführt, das es erlaubt, Constraints über der Syntax des Nutzercodes von domänen- und systemspezifischen Deklarationen zu definieren. Zunächst wird eine Einführung in Constraints dargestellt, die Programmiersprachen enthalten, um die Menge an gültigen Worten/Programmen zu begrenzen. Anschließend wird ein Formalismus zur Spezifikation von Constraints über den in einer API oder in einem Programm implementierten Deklarationen definiert. Abbildung 5.1 zeigt ein Beispiel des formalen Frameworks: Im oberen Teil ist Java-Quellcode dargestellt, die korrespondierenden Syntaxgraphen sind im mittleren Bereich abgebildet und unten sind Beispiele des Formalismus zu finden.

Eine Programmiersprache stellt eine Menge an Sprachkonstrukten zur Verfügung. In diesem Abschnitt werden nur die folgenden Konstrukte der Sprache Java genutzt werden, die in Abbildung 5.2 dargestellt sind. Die Menge der Konstrukte sei gegeben durch:

$$\mathbb{K}_{Java} = \{\underline{ClsDecl}, \underline{MethDecl}, \underline{MethInv}, \underline{StmtBlock}, ...\}$$

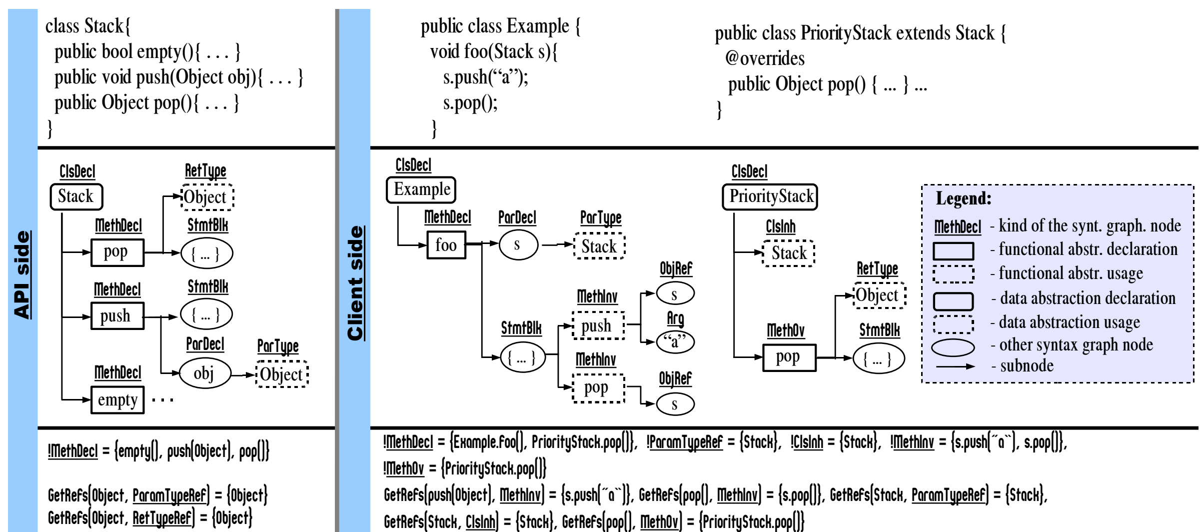


Abbildung 5.1.: Deklaration und Verwendung von Stacks

Jedes dieser Konstrukte entspricht einer Art von Knoten, die in einem abstrakten Syntaxgraphen eines Java Programms auftreten können. Die Implementierung eines Programms ist letztendlich die Instanziierung der Konstrukte der Programmiersprache. Die Grammatik der Sprache definiert die Basisstruktur, der die Kompositionen der Konstrukte folgen müssen, damit diese gültige Programme darstellen – das BNF Fragment in Abbildung 5.2 stellt beispielsweise sicher, dass die Felddeklarationen (*FieldDecl*) nur im Kontext einer Klassendeklaration (*ClsDecl*) auftreten. Die kontextfreie Grammatik einer Sprache definiert somit ein Geflecht an Relationen zwischen den Konstrukten, die einer "ist Teil von"-Semantik (Kompositionsrelation) entsprechen.

Dennoch ist die kontextfreie Grammatik einer Sprache nicht fähig, alle Regeln auszudrücken, die benötigt werden, um ausschließlich adäquate Verwendungen der Sprachkonstrukte sicherzustellen. Um einer nicht-adäquaten Verwendung entgegen zu wirken, müssen der kontextfreien Grammatik kontextsensitive Constraints hinzugefügt werden, die durch den Parser während der sogenannten 'Semantischen Analyse' geprüft werden. Beispiele für derartige kontextsensitive Constraints in der Java Grammatik sind:

- "Jede Variable muss deklariert werden, bevor sie verwendet werden darf".
- "Implementiert eine Klasse ein Interface, müssen die zu diesem Zweck implementierten Methoden die Sichtbarkeit public haben".
- "Zwei Felder einer Klasse dürfen nicht den gleichen Bezeichner haben".

Syntaktische Striktheit. Eine Sprache, deren Syntaxdefinition lediglich adäquate Kompositionen ihrer Konstrukte erlaubt, sei als *syntaktisch strikt* bezeichnet (in der Literatur auch als 'strongly checked' bezeichnet [Car96]). Eine voll-typisierte Sprache ist somit deutlich strikter

als eine untypisierte Sprache, da sie nur typkorrekte Kompositionen von Ausdrücken, Termen und Zuweisungen erlaubt.

Verfechter dynamischer, untypisierter Sprachen bezeichnen die zusätzlichen Typannotationen oftmals als überflüssigen Aufwand [Ous98], mit der Behauptung, dass damit meist nur irrelevante Fehler gefunden würden, die bereits bei der ersten Ausführung des Programms erkannt werden könnten. Die durch Typsysteme ausgeschlossenen Fehler werden damit als eine Art von 'Tippfehlern' betrachtet, die lediglich durch kleine Unachtsamkeiten von Programmierern entstehen. Der Grund hierfür ist darin zu finden, dass Typsysteme lediglich Fehler auf dem Abstraktionsniveau der Programmiersprache finden können. Professionelle Programmierer haben normalerweise kein Problem damit, die Konstrukte ihrer Programmiersprache zu beherrschen, die im Falle von Java ca. 120 Konzepte umfassen. Deutlich komplexer ist das Verstehen von Deklarationen, die in Bibliotheken und Frameworks implementiert sind und domänenspezifischen Konzepte repräsentieren. Einerseits sind diese Konzepte zweiter Klasse, die nicht über die Programmiersprache selbst, sondern durch Deklarationen repräsentiert sind, Programmierern oftmals nicht derart vertraut wie die Konstrukte der Programmiersprache. Andererseits ist die Anzahl dieser Konzepte, die durch Nutzung von Abstraktionsmechanismen in ein System einfließen, ein Vielfaches der Anzahl der Sprachkonstrukte. Aus dieser Motivation heraus, soll im folgenden Abschnitt eine Technik zur Definition von Constraints auf Deklarationen, die domänen- und systemspezifische Konzepte repräsentieren, vorgestellt werden. Diese Constraints heben somit die statische Überprüfung von dem Abstraktionsniveau der Programmiersprache auf das Niveau von in Bibliotheken und Frameworks implementierten Konzepten an. Damit werden die Vorteile der syntaktischen Striktheit von typisierten Sprachen auf Bibliotheken übertragen.

```
CompUnit ::= [PkgDecl] TypeDecl ";"
TypeDecl ::= ClsDecl | InterfaceDecl
ClsDecl ::= Visibility "class " Name " extends " ClsInh"{" (FieldDecl | MethDecl | MethOv)* "}"
MethOv ::= MethDecl
MethDecl ::= Visibility RetTypeRef Name "(" [(ParamDecl ",")* ParamDecl]") {" StmtBlock "}"
ParamDecl ::= ParaTypeRef  Name
StmtBlock ::= MethInv | IfStmt | WhileStmt
IfStmt ::= "if(" Expr ") {" StmtBlock "}" ["else{" StmtBlock "}"]
WhileStmt ::= "while(" Expr ") {" StmtBlock "}"
MethInv ::= [ObjName "." ] Name "(" [(Arg ",")* Arg] ")"
```

Abbildung 5.2.: Ein Fragment der Java Grammatik

Notation: Um den Formalismus lesbar, verständlich und übersichtlich zu halten, werden zunächst einige spezielle Notationen und abkürzende Schreibweisen definiert:

1. In diesem Abschnitt werden nur die Java Sprachkonstrukte verwendet, die in dem Fragment der (vereinfachten) Java Grammatik in Abbildung 5.2 abgebildet sind. Referenzierte Konstrukte werden immer unterstrichen dargestellt.

2. Es wird eine an Java-Programme angelehnte Notation verwendet, um auf dem Syntaxgraphen zu navigieren: Demnach steht $x.Y$ für die Menge an Knoten des Typs $Y \in \mathbb{K}_{Java}$, die zu einem spezifischen AST Knoten x korrespondiert. Wenn x beispielsweise ein Knoten vom Typ $\underline{MethDecl}$ ist, dann repräsentiert $x.\underline{Visibility}$ die Sichtbarkeit der Methode.

3. Um ein spezifisches Element in einer geordneten Menge (z.B. Statements eines Statement-Blocks) zu referenzieren, wird eine Notation ähnlich zu Java-Arrays genutzt. Zum Beispiel adressiert $m.\underline{ParamDecl}[0]$ den ersten Parameter einer $\underline{MethDecl}$ m. Das Zeichen # wird als Präfix-Operator verwendet, um die Anzahl der Einträge in einer Menge zu bekommen.

4. Es wird $!c$ geschrieben, um auf die Menge der Instanzen eines Sprachkonstrukts c ($\subseteq \mathbb{K}_{Java}$) in einem Programm zu verweisen. Beispielsweise bestimmt $!\underline{ClsDecl}$ die Menge aller Klassendeklarationen in einem Programm.

5. Um die Notation lesbarer zu gestalten, wird davon ausgegangen, dass jeder $\underline{Name}$ (Bezeichner) im abstrakten Syntaxgraphen vollqualifiziert ist.

6. Aus dem gleichen Grund wird zusätzlich die Notation $x \sqsubset y$ genutzt, welche bedeutet, dass x in einem Untergraphen von y enthalten ist. Somit bedeutet $inv \sqsubset exp$, dass eine Instanz inv eines Methodenaufrufs im Untergraphen eines Ausdrucks (Expression) exp auftritt.

5.2. Constraints über der Verwendung von implementierten Konzepten

Programmiersprachen ermöglichen es Programmierern, Konzepte abzubilden, die benötigt werden, um die Domäne zu beschreiben, für die ein System oder eine Bibliothek implementiert werden soll. Dies wird durch die Bereitstellung von Abstraktionsmechanismen bewerkstelligt. Die Abstraktionsmechanismen der Sprache Java (eine Menge AM) erlauben die Definition und die Nutzung von Konzepten (Abstraktionen, die mit durch einen symbolischen Bezeichner benannt sind, vgl. Abschnitt 4.1.2). Ein Abstraktionsmechansimus ($\in AM$) besteht aus zwei Arten von Sprachkonstrukten:

- *Deklarationsmechanismen* (AM_{decl}) ermöglichen Programmierern die Erstellung von Deklarationen. Diese Konstrukte der Programmiersprache werden verwendet, um Abstraktionen in ein Programm (oder eine Bibliothek) einzuführen und diese durch symbolische Bezeichner von anderen Stellen eines Programms nutzbar zu machen.

- *Verwendungsmechanismen* (AM_{use}) erlauben Programmierern die Nutzung von Deklarationen. Diese Art von Konstrukten verwenden Entwickler, um die Implementierung von Konzepten in ihren Programmen (wieder-)zuverwenden.

Natürlich gilt, dass sowohl Deklarations- als auch Verwendungsmechanismen durch die Programmiersprache definiert sind ($AM_{decl} \subset \mathbb{K}_{Java}, AM_{use} \subset \mathbb{K}_{Java}$). Im Folgenden werden die

Hauptabstraktionsmechanisem der objektorientierten Programmierung betrachtet (Daten- und prozedurale Abstraktion):

$$AM_{decl} = \{\underline{ClsDecl}, \underline{MethDecl}, \dots\}$$
$$AM_{use} = \{\underline{ClsInh}, \underline{MethInv}, \underline{ParaTypeRef}, \underline{MethOv}, \dots\}$$

In Abbildung 5.1 (links) sind eine Klassen- und drei Methodendeklarationen als Beispiele für Elemente aus AM_{decl} dargestellt. Auf der Nutzerseite (rechts) sind verschiedene Arten der Verwendung dieser Deklarationen zu sehen, wie ein Aufruf (MethInv) und ein Überschreiben (MethOv) der 'pop'-Methode.

Die Menge der Deklarationen, die durch die Abstraktionsmechanismen (AM) definiert sind, entspricht der Menge $\mathbb{I}_N$ (dem Implementierungsmodell, vgl. Abschnitt 4.1.2). Somit gilt entsprechend Abbildung 5.1 (links) folgendes:

$$\mathbb{I}_N = \{Stack, empty, pop, push\}$$

Der Compiler akzeptiert jede Nutzung einer Deklaration, solange es den syntaktischen Regeln der Programmiersprache gehorcht. Beispielsweise ist jeder Aufruf der 'pop'-Methode erlaubt, auch wenn der Stack leer ist, auf dem sie aufgerufen wird. Die Bedeutung des durch die Deklaration dargestellten Konzepts 'Stack' wird somit nicht berücksichtigt. Viele Implementierungen von Konzepten, die durch APIs zur Verfügung gestellt werden, sollten nicht in allen Situationen genutzt werden, welche die Programmiersprache grundsätzlich akzeptieren würde. Dies liegt daran, dass Konzepte mit spezifischen Constraints einhergehen. Genauso wie Programmiersprachen kontextsensitive Constraints definieren, um sicherzustellen, dass die eingebauten Konstrukte richtig eingesetzt werden, unterliegen auch Deklarationen ($\in \mathbb{I}_N$) bestimmten Restriktionen (aufgrund der Kanten in $\mathbb{D}_E$), die nicht explizit in der Programmiersprache ausgedrückt werden können.

Um die Referenzen auf eine bestimmte Deklaration eines Konzepts in einem Programm zu finden, wird die Funktion $GetRefs$ wie folgt definiert: Für eine Deklaration $d \in !AM_{decl}$ und einen Verwendungsmechanismus $u \in AM_{use}$, gibt $GetRefs(d, u)$ die Menge der Knoten im abstrakten Syntaxgraphen eines Programms vom Typ u zurück, welche Verwendungen von d darstellen.

$$GetRefs : !AM_{decl} \times AM_{use} \rightarrow \mathcal{P}(!AM_{use})$$

Wie in Abbildung 5.1 dargestellt, kann diese Funktion dazu genutzt werden, um alle Verwendungen der 'pop'-Methode in Form von Methodenaufrufen (MethInv) oder auch in Form von Methodenüberschreibungen (MethOv) zu finden:

$$GetRefs(pop, \underline{MethInv}) = \{s.pop()\}$$
$$GetRefs(pop, \underline{MethOv}) = \{PriorityStack.pop()\}$$

Constraints. Deklarationen werden immer anhand eines Deklarationsmechanismus erstellt. Um eine Deklaration zu verwenden, wird ein passender Verwendungsmechanismus benötigt. Zur Definition eines konzeptspezifischen Constraints werden drei Angaben benötigt: Eine Deklaration des Konzepts, ein Verwendungsmechanismus, der von dem Constraint betroffen sein soll, und eine Spezifikation des Kontextes, in welchem die Verwendung der Deklaration erlaubt sein soll. Formal sei ein Constraint $\mathcal{C}$ wie folgt definiert:

$$\mathcal{C} = \langle Decl, Usage, Ctxt \rangle$$

- $Decl$: Eine konkrete Instanz einer Deklaration $\in !AM_{decl}$ (e.g. java.util.Stack.pop).
- Use: Ein Verwendungsmechanismus $\in AM_{use}$, der auf den Typ von $Decl$ angewandt werden kann (z.B. $\underline{MethInv}$).
- $Ctxt$: Eine prädikatenlogische Formel, die den erlaubten Nutzungskontext definiert (vgl. beispielsweise Formel 5.1).

Die Semantik eines Constraints ist folgendermaßen definiert:

$$\forall GetRefs(Decl, Use) : Ctxt$$

Die Funktion $GetRefs$ gibt dabei die Menge der Nutzungen einer Deklaration $Decl$ in einem Programm zurück, bei denen der Verwendungsmechanismus Use eingesetzt wurde. Ein Constraint legt somit fest, dass für jede Verwendung einer spezifischen Deklaration ein bestimmter Kontext gelten muss. Im Weiteren wird der Bezeichner "use" als Variable im $Ctxt$ eines Constraints verwendet, um auf die konkrete Instanz des Verwendungsmechanismus (Use) zuzugreifen. Ein Beispiel für einen Constraint, der eine adäquate Verwendung (engl. well-behaved usage) von 'pop' sicherstellt, wäre:

$$\begin{gathered} PopInIf = \langle \textbf{Decl} : java.util.Stack.pop(), \quad \textbf{Use} : \underline{MethInv}, \\ \textbf{Ctxt} : \exists i \in !\underline{IfStmt} : \exists m \in \underline{MethInv} : \\ m \sqsubset i.\underline{Expr} \wedge use \sqsubset i.\underline{StmtBlock} \wedge \\ m.\underline{ObjName} = use.\underline{ObjName} \wedge m.\underline{Name} = \text{“}empty\text{”} \rangle \end{gathered} \tag{5.1}$$

Intuitiv besagt dieser Constraint, dass jeder Aufruf der 'pop'-Methode innerhalb einer 'if'-Anweisung stattfinden muss, die einen Ausdruck enthält, der durch einen Aufruf der "empty"-Methode überprüft, ob der Stack leer ist. Dieser Constraint wäre sehr restriktiv und dient an dieser Stelle lediglich zur Illustration des Formalismus.

Komposition von Constraints. In vielen Fällen ist die Beschreibung der adäquaten Verwendung eines implementierten Konzepts durch einen einzigen Constraint nicht sinnvoll möglich. Um vielfältige Einschränkungen beschreiben zu können, ohne einen einzelnen Constraint zu komplex werden zu lassen, soll nun vorgestellt werden, wie Constraints zu komplexen zusammengesetzten Constraints komponiert werden können. Auf diese Art und Weise ist es möglich, Constraints für unterschiedliche Deklarationen wiederzuverwenden. Beispielsweise wäre ein

Aufruf der 'pop'-Methode nicht nur innerhalb eines Statementblocks erlaubt, der explizit den Füllstand des Stacks überprüft, sondern auch im Anschluss an einen Aufruf der Methode 'push' auf dem gleichen Stack. Der zweite Constraint kann wie folgt definiert werden:

$$PushPopSeq = \langle \textbf{Decl} : java.util.Stack.pop(), \quad \textbf{Use} : \underline{MethInv}, \quad \textbf{Ctxt} : \exists m \in !\underline{MethInv} : m.\underline{Name} = \text{``}Stack.push\text{''} \wedge use \uparrow m \rangle \tag{5.2}$$

Der Operator $stmt_1 \uparrow stmt_2$ sei als Prädikat (Infix-Notation) definiert, das entscheidet, ob innerhalb eines StatementBlocks ($\underline{StmtBlock}$) jeder Pfad von $stmt_2$ im Kontrollflussgraphen zu einem korrespondierenden $stmt_1$ führt. Nun kann ein zusammengesetzter Constraint definiert werden, der sowohl push-pop-Sequenzen (Formel 5.2) als auch den Aufruf von 'pop' in explizit überprüften StatementBlöcken (Formel 5.1) erlaubt:

$$PopInIf \vee PushPopSeq$$

Allgemein gesprochen können zwei Constraints c_1 und c_2 zu komplexeren Constraints durch Nutzung logischer Operatoren ($\odot \in \{\wedge, \vee, \otimes, \dots\}$) komponiert werden, wenn gilt $Decl_1 = Decl_2$ und $Use_1 = Use_2$:

$$c_1 \odot c_2 := \forall GetRefs(Decl, Use) : Ctxt_1 \odot Ctxt_2$$

Auch der logische Negationsoperator $\neg$ kann auf einen Constraint c angewandt werden. Dies führt zur Negation der Kontextbedingung:

$$\neg c := \forall GetRefs(Decl, Use) : \neg Ctxt$$

Parametrierung von Constraints. Der Constraint zur Überprüfung der adäquaten Nutzung der 'pop'-Methode wurde rein für diese einzelne Methode definiert, obwohl diese Einschränkung der Nutzung potentiell auch auf andere Methoden übertragbar wäre. Um die Wiederverwendung von Constraints zu ermöglichen, ist es somit wünschenswert, Constraints parametrisieren zu können. Damit wird eine effizientere Spezifikation von Constraints ermöglicht. Da Constraints letztendlich auf der Prädikatenlogik basieren, können sie auch ähnlich zu Prädikaten parametrisiert werden. Folgender Constraint zeigt eine parametrisierte Version des $PopInIf$-Constraints (Formel 5.1), die es ermöglicht, diesen für ein beliebiges Paar von Methoden (inv und $check$) anzuwenden, bei denen $check$ überprüft, ob sich das System in einem Zustand befindet, in dem der Aufruf von inv erlaubt ist:

$$\begin{gathered} CheckBeforeInv(\underline{MethDecl}\ inv, \underline{MethDecl}\ check) = \langle \textbf{Decl} : inv, \quad \textbf{Use} : \underline{MethInv}, \\ \textbf{Ctxt} : \exists i \in !\underline{IfStmt} : \exists m \in \underline{MethInv} : \\ m \sqsubset i.\underline{Expr} \wedge use \sqsubset i.\underline{StmtBlock} \wedge \\ m.\underline{ObjName} = use.\underline{ObjName} \wedge m.\underline{Name} = check.\underline{Name} \rangle \end{gathered}$$

6. Methodische Anwendung von Constraints in Softwareprojekten

"Das Übel derer, die Fehler machen ist, dass sie etwas nicht wissen und doch denken, sie wissen es."
- Lü Bu We

In diesem Kapitel sollen die methodischen Aspekte der Verwendung von Constraints in der Softwareentwicklung vorgestellt werden. Es wird beschrieben, wie Constraints in Softwareentwicklungsprojekten praktisch eingesetzt werden können.

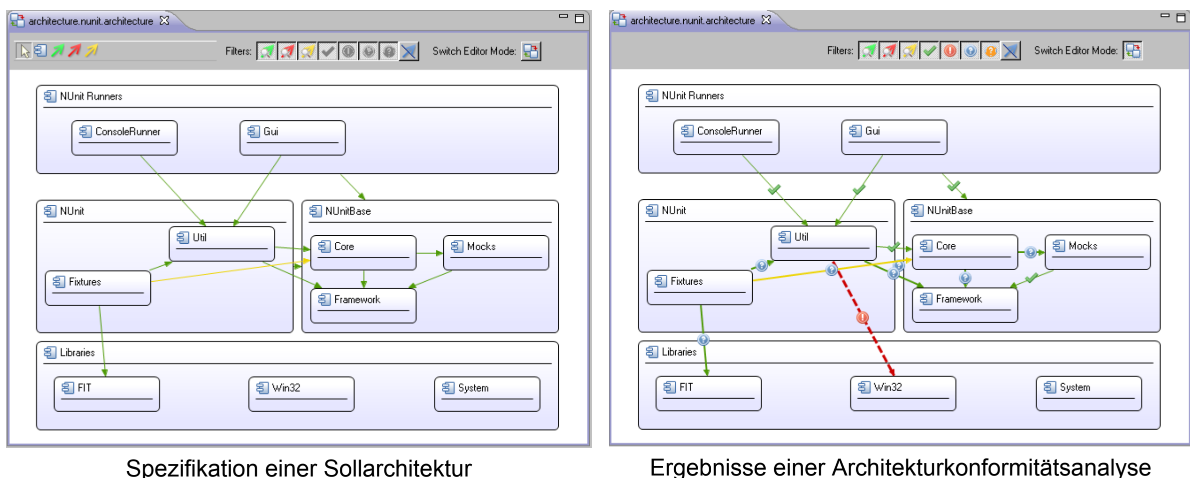
6.1. Repräsentation von Constraints in Entwicklungsartefakten

In diesem Abschnitt werden zwei Möglichkeiten der praktischen Operationalisierung von Constraints vorgestellt. Es wird dargestellt, in welcher Repräsentation Constraints in Projekten eingesetzt werden können. Die erste Möglichkeit ist die Nutzung von Modellen, in denen bestimmte Entwurfsregeln explizit ausgedrückt werden können. Constraints definieren hierbei die Abbildung von Modellen auf den Code und ermöglichen dadurch eine werkzeuggestützte Konformitätsanalyse. Die zweite Möglichkeit ist, Constraints in Form von Annotationen im Code einzusetzen, um die Information über die adäquate Verwendung von Deklarationen explizit in den Code einzubetten. Die folgenden beiden Abschnitte stellen diese beiden Möglichkeiten der Nutzung von Constraints vor.

6.1.1. Modelle als graphische Repräsentation von Constraints

Der Constraint-Formalismus wurde in Kapitel 5 eingeführt, um die impliziten Entwurfsregeln in Programmen und APIs greifbar zu machen und explizit formulieren zu können. Sie definieren dabei ein syntaktisches Muster, dem eine adäquate Nutzung einer Deklaration gerecht werden muss. Für den praktischen Einsatz in einem Softwareentwicklungsprojekt sind die Constraint-Formeln jedoch nur bedingt geeignet, da sie sehr komplex werden können und dadurch schwer zu formulieren und zu verstehen sind. Aus diesem Grund sollten in der Praxis Constraints eher im Hintergrund, integriert in geeigneten Werkzeugen, ihren Dienst tun. Ein formaler Constraint entspricht somit der formalen Definition der Überprüfungen, die ein statisches Analysewerkzeug auf einem Programm durchführt.

Wie in Abschnitt 4.2 erläutert, existieren vielerlei Modelle, die in der Lage sind, bestimmte Arten von Entwurfsregeln auszudrücken, die auf Ebene des Codes lediglich implizit vorhanden sind. Diese Modelle bieten oftmals eine bessere Art der Darstellung von Constraints als die reine Formalisierung. Sie sind leichter für Entwickler verständlich und dadurch einfacher in der Praxis einsetzbar. Damit werden sie zu einer Art "konkreten Syntax von Constraints", die eine einfachere Benutzbarkeit sicherstellt. Constraints dienen jedoch dazu, den Zusammenhang der in einem Modell beschriebenen Vorgaben auf die Programmiersprache zu spezifizieren. Bevor Modelle eingesetzt werden, muss klar definiert sein, wie sich die in den Modellen niedergelegten Informationen auf den Code auswirken sollen. Dadurch ergibt sich eine Integration zwischen den Modellen und dem Code durch die Spezifikation von Constraints. Die formalen Constraints selbst sind in diesem Zusammenhang jedoch nur für die Erstellung eines Werkzeugs zur Konformitätsanalyse von Interesse. Entwickler in einem Projekt können sich schließlich derartiger Werkzeuge bedienen und die Constraint-Spezifikation als Referenz nutzen.



Spezifikation einer Sollarchitektur — Ergebnisse einer Architekturkonformitätsanalyse

Abbildung 6.1.: Werkzeuggestützte Architekturkonformitätsanalyse

Abbildung 6.1 zeigt den Editor arch.edit, der zur Konformitätsüberprüfung einer Sollarchitekturspezifikation mit einer Implementierung genutzt werden kann. Die eigentliche Überprüfung wird durch das Werkzeug ConQAT [DJH+08] durchgeführt. Die durch arch.edit zur Verfügung gestellte Modellierungstechnik (Komponentendiagramme) ist Entwicklern meist grundsätzlich vertraut. Die Modelle können leicht anhand des Editors erstellt werden. Im Hintergrund eines derartigen Werkzeugs muss spezifiziert sein, wie die Elemente der Modelle im Code abgebildet werden und welchen Regeln der Code genügen muss. Diese Spezifikation kann durch Constraints erfolgen. Eine derartige Spezifikation dieser Constraints sowie eine formale Definition der zur Architekturkonformitätsanalyse eingesetzten Modelle werden in Kapitel 10 vorgestellt.

Ähnlich zur Architekturkonformitätsanalyse, bei der die impliziten Entwurfsregeln im Code über ein Modell explizit definiert werden, können auch andere Arten von Modellen eingesetzt werden. In [DF04a] werden beispielsweise die gewünschten Aufrufsequenzen von Methoden anhand von Automaten definiert. Durch die Automatenmodelle werden dabei die impliziten

Vorgaben bezüglich des Protokolls, dem die Nutzung der Methoden unterliegt, explizit dargestellt. Durch formale Constraints kann festgelegt werden, welche Strukturen der Nutzercode einhalten muss, um diese Vorgaben zu erfüllen. Damit kann schließlich die adäquate Verwendung der Methoden (im Sinne der Automatenspezifikation) überprüft werden.

In [MNS01] wird eine vergleichbare Methode vorgestellt. Der Zusammenhang zwischen den Modellen und dem Code kann durch Constraints jedoch klarer definiert werden, weil die Programmiersprache konkret berücksichtigt wird.

6.1.2. Deklaration von Constraints durch Programmannotationen

Für feingranularere implizite Entwurfsregeln, die weniger durch Architekturentscheidungen bedingt, sondern eher als Implementierungsentscheidungen zu betrachten sind, wäre die Anfertigung eines eigenen Modells oftmals zu schwergewichtig. Derartige Regeln werden ohnehin häufig lediglich in Form von Kommentaren direkt im Code vermerkt, anstatt diese in Form von Modellen zu dokumentieren. Diese Art von Information kann durch Nutzung von Constraints auf eine Art und Weise ausgedrückt werden, die derartige Entwurfsregeln in überprüfbarer Form explizit macht. Die Annotationen im Code würden auch in diesem Fall nicht direkt durch Constraint-Formeln erfolgen, wie sie im vorangehenden Kapitel eingeführt wurden. Die formalen Constraint-Spezifikationen werden in diesem Fall durch symbolische Bezeichner ersetzt, um die Annotationen nicht mit zu vielen Details zu überfrachten. Entwurfsregeln lassen sich häufig durch ähnliche Constraints ausdrücken. Dadurch ergibt sich ein gewisses Wiederverwendungspotential. Eine Bibliothek an Constraints, die parametrisiert und miteinander verknüpft werden können, um somit flexibel an die spezifischen Einschränkungen bestimmter Deklarationen anpassbar zu sein, würde einen effizienten Einsatz von Constraint-Annotationen ermöglichen.

Ein statisches Analysewerkzeug, wie der Java Constraint Checker, der im folgenden Kapitel näher beschrieben wird, ermöglicht Programmierern die Definition von Constraints, indem ein Constraint durch einen geeigneten Bezeichner an eine Deklaration im Code annotiert werden kann. Durch Komposition und Parametrierung von Constraints können so auch komplexere Einschränkungen formuliert werden. Ein derartiges Werkzeug muss über eine Bibliothek an gebräuchlichen Constraints verfügen, die somit einfach in Programmen eingesetzt werden können. Beispielsweise sollten gängige Entwurfsmuster direkt als Constraints in einer derartigen Bibliothek vorliegen. Auch häufige Einschränkungen der Nutzung von Deklarationen in APIs sollten darin verfügbar sein, wie etwa Einschränkungen bzgl. der Aufrufreihenfolge von Methoden im Kontrollflussgraphen (in Kapitel 9 werden einige typische Beispiele für implizite Vorgaben in einer API vorgestellt). Um auch Constraints nutzen zu können, die nicht durch Komposition und Parametrierung von vordefinierten Constraints erzeugt werden können, sollte die Architektur eines Werkzeugs auch die Erweiterung der Bibliothek zulassen.

Moderne Programmiersprachen wie Java und C# bieten Mechanismen zur typsicheren Nutzung von Annotationen. Annotationen können in diesen Sprachen explizit deklariert und mit Parametern versehen werden. Durch Typprüfungen wird schließlich sichergestellt, dass passende Argumente für die Parameter vorliegen. Aus diesem Grund bietet sich die Nutzung derartiger

Mechanismen zur Erstellung von Constraint-Bibliotheken an. Im Gegensatz zur Annotation von Constraints auf Basis von Kommentaren im Quellcode ergibt sich jedoch noch ein weiterer Vorteil: Die Annotationsmechanismen von C# und Java bieten die Möglichkeit, die Annotationen durch den Compiler auch in den Binärcode integrieren zu lassen. Damit ist es möglich, Constraints auch in Bibliotheken und Frameworks zu integrieren, die lediglich im Form des Binärcodes ausgeliefert werden.

6.2. Einsatz von Constraints im Entwicklungsprozess

In den folgenden Abschnitten wird der Einsatz von Constraints im Rahmen von Softwareentwicklungsprojekten erläutert. Dabei werden zunächst die unterschiedlichen Arten von Vorgaben kategorisiert, zu denen der Code eines Programms konform gehalten werden sollte. Anschließend wird der Lebenszyklus von Constraints erklärt und erläutert, wie eine kontinuierliche Überprüfung von Constraints bewerkstelligt werden kann.

6.2.1. Der Ursprung von (impliziten) Vorgaben in einem Projekt

Wie in Abschnitt 2.2 beschrieben, stellen implizite Vorgaben eine große Herausforderung für die Projektkommunikation dar. Constraints bieten ein Mittel, um derartige Vorgaben explizit zu erfassen und zu überprüfen, ob diese auch eingehalten werden. Grundsätzlich können zwei Arten des Ursprungs von Constraints in einem Softwareentwicklungsprojekt unterschieden werden:

- Projektexterne Vorgaben: Hierbei werden Constraints von Nicht-Projektmitarbeitern, etwa von den Erstellern von Bibliotheken, Frameworks oder extern entwickelten Programmmodulen vorgegeben, um sicherzustellen, dass die (projektinternen) Nutzer die Schnittstellen adäquat verwenden und nicht gegen Konventionen verstoßen, die mit deren Einsatz einhergehen.

- Projektinterne Vorgaben: In diesem Fall werden Constraints durch Projektmitarbeiter deklariert, um die Konformität des Programms bezüglich Vorgaben sicherzustellen, die projektintern festgelegt wurden (Architektur- oder Designvorgaben, interne Konventionen).

Somit kann man beim Einsatz von Constraints zwischen der Nutzung zur Intra- bzw. Interprojektkommunikation unterscheiden. Es ist zu beachten, dass auch Constraints, die projektintern definiert werden, projektexterne Komponenten betreffen können. Projektintern könnte in einer Art "Selbstbeschränkung" Constraints für bestimmte APIs definiert werden. Beispielsweise könnte es nur bestimmten Komponenten des Systems erlaubt sein, eine Bibliothek zu verwenden, um damit eine zu starke Kopplung an eine Bibliothek zu vermeiden und somit künftig einen Austausch der Bibliothek einfacher bewerkstelligen zu können. In diesem Fall würden projektinterne Entwickler aufgrund von internen Architekturentscheidungen Constraints für Deklarationen einführen, die sie nicht selbst erstellt haben. Da der Quellcode einer Bibliothek

nicht immer vorliegt, muss ein Werkzeug, das die Definition und Prüfung derartiger Constraints realisiert, auch deren Definition außerhalb des Quellcodes unterstützen.

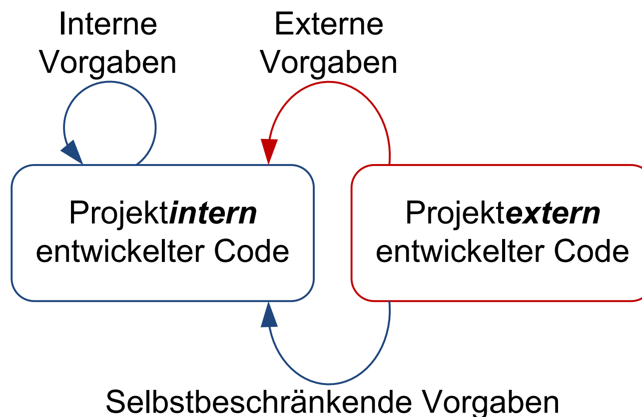


Abbildung 6.2.: Ursprung von Vorgaben in einem Projekt

Abbildung 6.2 illustriert die unterschiedlichen Arten von Vorgaben, denen der in einem Projekt entwickelte Code entsprechen muss: (1) *Interne Vorgaben* werden projektintern festgelegt und beziehen sich auf interne Deklarationen. (2) *Externe Vorgaben* werden extern festgelegt und beziehen sich auf Deklarationen, die durch projektexterne Entwickler erstellt wurden. (3) *Selbstbeschränkende Vorgaben* werden durch projektinterne Entwickler festgelegt, beziehen sich jedoch auf externe Deklarationen.

Beweggründe für den Einsatz von Constraints. Folgende Beweggründe können einem Einsatz von Constraints zugrunde liegen:

- Vermeidung von unerwünschtem Laufzeitverhalten
- Durchsetzung änderbarer Programmstrukturen
- Durchsetzung verständlicher Programmstrukturen

Alle diese Beweggründe können internen Vorgaben zugrunde liegen. Interne Vorgaben können dabei sehr restriktiv definiert werden, da die vorgesehene Nutzung der Deklarationen bereits projektintern bekannt ist. Falls notwendig, können diese Vorgaben im Lauf der weiteren Evolution des Systems einfach angepasst werden. Externe Vorgaben sollten nicht zu restriktiv definiert werden, da Bibliotheken und Frameworks oftmals ein sehr breites Spektrum an Anwendungsgebieten abdecken sollen. Den Nutzerprogrammen soll oftmals bewusst eine gewisse Freiheit im Einsatz der angebotenen Deklarationen geboten werden. Zudem ist eine Anpassung der Constraints nach der Auslieferung meist nicht kurzfristig möglich. Beweggründe für externe Vorgaben sind somit meist die Vermeidung von unerwünschtem Laufzeitverhalten. Die Änderbarkeit kann nur insofern eine Motivation für die Einführung von Constraints darstellen, dass etwa erzwungen werden soll, dass lediglich eine stabil gehaltene Schnittstelle einer Bibliothek (Fassade) verwendet und der Zugriff auf bibliotheksinternen Code vermieden werden soll.

Damit bleiben Bibliotheken änderbar, ohne dass deren Nutzerprogramme angepasst werden müssen.

Anwendungsbereiche von Constraints. In Teil III dieser Arbeit werden die unterschiedlichen Arten von Vorgaben anhand von drei wesentlichen Anwendungsbereichen von Constraints im Detail beleuchtet. Als erster Bereich werden externe Vorgaben, die in Bibliotheken zu finden sind, genauer untersucht. Der zweite Anwendungsbereich befasst sich mit internen und selbstbeschränkenden Vorgaben. Dabei wird der Fokus auf Vorgaben bezüglich der statischen strukturellen Architektur gelegt. Als Anwendungsgebiet 3 werden schließlich Entwurfsregeln untersucht, wie sie durch Entwurfsentscheidungen in Programme einfließen.

6.2.2. Lebenszyklus von Constraints

Abbildung 6.3 zeigt anhand eines Aktivitätendiagramms den Lebenszyklus von Constraints in einem Programm. Sobald eine neue Architektur- oder Designentscheidung in einem Projekt getroffen wird, sollte überprüft werden, ob diese Festlegungen durch Constraints ausdrückbar sind. Da sich Ersteller von Deklarationen nicht immer dessen bewusst sind, dass die Nutzung der Deklaration bestimmten impliziten Vorgaben unterliegt (vgl. Abschnitt 2.1), ist es unmöglich, alle impliziten Vorgaben direkt bei ihrer Entstehung explizit zu machen. Sobald jedoch ein Verstoß gegen derartige unbewusst eingeführte implizite Vorgaben erkannt wird (z.B. im Rahmen eines Reviews), sollte versucht werden, diese explizit zu machen. Falls etwa im Rahmen von Testaktivitäten unerwünschtes Laufzeitverhalten festgestellt wird, das etwa von einer nicht-adäquaten Nutzung von Schnittstellen hervorgerufen wird, sollte überprüft werden, ob dieses Fehlverhalten bereits per Design durch Nutzung von Constraints vermeidbar gewesen wäre.

Falls die Formulierung von Constraints nicht möglich oder zu komplex wäre, sollte nach anderen Methoden gesucht werden, um diese Vorgaben künftig überprüfen zu können (z.B. Einführung von automatischen Tests). Sofern sich die Einführung eines Constraints als geeignetes Mittel darstellt, sollte dieser deklariert werden (über ein Modell oder eine Annotation im Code). Nach der automatischen Überprüfung des Constraints ist zu analysieren, ob der Constraint die Bedingung korrekt wiedergibt oder ob dieser weiterer Modifikationen bedarf. Sobald der Constraint korrekt erscheint, sollte der Code entsprechend angepasst werden.

Kontinuierliche Überprüfung von Constraints. Constraints stellen einen leichtgewichtigen Ansatz dar, der versucht, bereits während der Implementierung Verletzungen von Vorgaben zu vermeiden, um dadurch Aufwände in den späteren Wartungsphasen einzusparen. Es ist bekannt, dass die Kosten für die Behebung eines Defekts umso geringer sind, je frühzeitiger dieser in einem Softwaresystem entdeckt wird. Wenn Entwickler, die aus Unwissenheit eine Deklaration inadäquat verwenden, sehr zeitnah von ihrer Entwicklungsumgebung entsprechende Warnhinweise bekommen, können sie unmittelbar darauf reagieren. Damit können Verstöße noch sehr effizient behoben werden, da die Entwickler zu diesem Zeitpunkt noch mit dem jeweiligen Code vertraut sind und sich nicht nochmals aufwändig einarbeiten müssen.

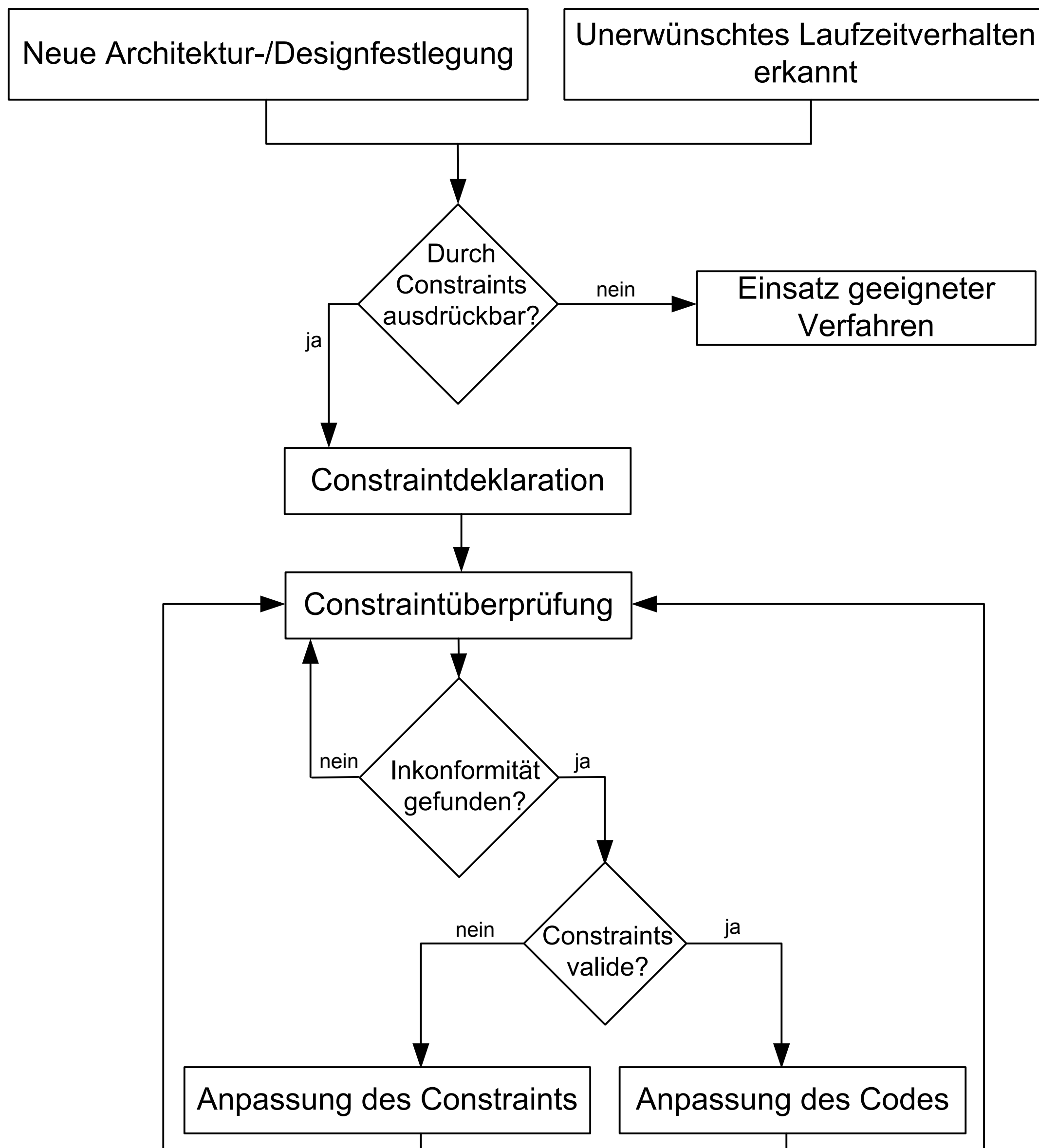


Abbildung 6.3.: Lebenszyklus von Constraints

Um diesen Effekt zu erzielen, ist es notwendig, alle Constraints, die für ein Programm definiert wurden, kontinuierlich zu überprüfen. Das in Abbildung 6.3 dargestellte Aktivitätsdiagramm definiert aus diesem Grund keinerlei Endzustand. Zur kontinuierlichen Überprüfung von Constraints können Projektleitstände (Dashboards) genutzt werden, die etwa im Rahmen eines Nightly Builds alle Constraints überprüfen und für alle Verletzungen Warnungen auflisten, so dass die Entwickler diese Liste jeden Morgen inspizieren und somit zeitnah darauf reagie-

ren können. Da die Laufzeit der Überprüfungen im Nightly Build eine untergeordnete Rolle spielt, sollten in einem Projektleitstand immer alle Constraints überprüft werden. Darüber hinaus ist es notwendig, dass Constraints, deren Überprüfung sehr effizient durchgeführt werden kann, direkt bei jedem Übersetzungsvorgang in der Entwicklungsumgebung überprüft werden. Dadurch können Verletzungen noch früher erkannt und behoben werden. Es ist ohnehin notwendig, dass Entwickler auf ihren lokalen Rechnern alle Constraints überprüfen können, um feststellen zu können, ob Änderungen am Code zur Beseitigung von Constraint-Verletzungen richtig durchgeführt wurden und ob Änderungen an Constraintdeklarationen valide sind. Es wäre nicht praktikabel, wenn die Auswirkungen derartiger Aktivitäten erst am nächsten Tag anhand des Analyseergebnisses überprüft werden könnten.

Es ist zu beachten, dass Constraints im Laufe der Evolution Anpassungen aufgrund von Architektur- und Designveränderungen unterliegen. Einige Constraints werden im Lauf der Zeit auch hinfällig und sollten entfernt werden. Auch das Löschen eines Constraints wäre somit eine Anpassung im Sinne des Aktivitätsdiagramms in Abbildung 6.3.

6.3. Die Grenze zwischen syntaktischer und semantischer Überprüfung impliziter Vorgaben

In der klassischen Verifikation, wie sie beispielsweise durch den Einsatz von Model Checkern oder Theorem Prüfern betrieben wird, ist die zentrale Fragestellung, ob eine zu verifizierende Bedingung durch ein Programm erfüllt wird oder nicht. Sowohl die Bedingungen als auch die Programme können dabei je nach gewählter Verifikationsmethode unterschiedlich ausgedrückt werden.

Constraints, wie sie im vorangehenden Kapitel definiert wurden, überprüfen die adäquate Verwendung von Deklarationen. In einigen Fällen kann man auch bis zu einem gewissen Grad absichern, dass die Nutzung von Deklarationen kein unerwünschtes Laufzeitverhalten hervorruft. Die Herangehensweise unterscheidet sich allerdings von der klassischen Verifikation: Es wird nicht versucht, zu prüfen, ob eine gegebene Bedingung von einem gegebenen Programm eingehalten wird. Vielmehr schränken Constraints die Vielfalt, in der Deklarationen verwendet werden, ein, so dass nur noch adäquate und damit auch korrekte Programme geschrieben werden. Die Fragestellung bei der Einführung eines Constraints zur Vermeidung von unerwünschtem Laufzeitverhalten lautet also "wie muss ein Programm aussehen, dass eine bestimmte Bedingung gilt". Im Gegensatz zu Verifikationstechniken sind Constraints dadurch oftmals restriktiver, da in Kauf genommen wird, dass auch korrekte Programme ausgeschlossen werden.

Die enorme Laufzeitkomplexität der meisten (automatischen) Verifikationstechniken rührt daher, dass es enorm viele Möglichkeiten gibt, ein bezüglich einer Bedingung korrektes Programm zu schreiben. Somit bedarf es in ungünstigen Fällen oftmals exponentiell vieler Unifikationsschritte, bis schließlich eine syntaktische Repräsentation erreicht wird, auf der eine Äquivalenz gezeigt werden kann. Aufgrund von Unentscheidbarkeit kann diese Repräsentation in einigen Fällen sogar niemals erreicht werden. Das Prinzip hinter der Einführung von Constraints geht

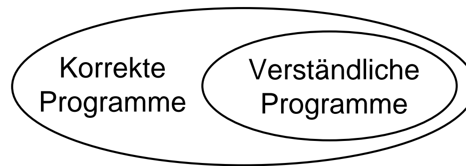


Abbildung 6.4.: Korrektheit und Verständlichkeit der Konzeptverwendung

davon aus, dass Korrektheit alleine nicht ausreichend ist. Programme sollten darüber hinaus auch verständlich und wartbar sein. Die Verwendung von Deklarationen in nicht-adäquater Weise behindert das Programmverstehen, auch wenn die Art der Verwendung keine zur Laufzeit sichtbaren negativen Effekte nach sich zieht (vgl. Abbildung 6.4). Das Verstehen ist dem Beweisen jedoch im Kern sehr nahe: Wenn ein Mensch etwas beweisen kann, dann hat er es auch verstanden. Falls nur durch sehr viele Umformungsschritte gezeigt werden kann, dass ein Programm eine Bedingung erfüllt, dann kann man im Allgemeinen auch nicht davon ausgehen, dass es für einen Menschen unmittelbar nachvollziehbar ist, warum diese Bedingung gilt. Aus Sicht der Wartung ist diese intensionale Nachvollziehbarkeit aber nötig, da Verifikationsbedingungen meist auch nach der Durchführung von Änderungen immer noch Gültigkeit haben sollen. Programme sollten vor diesem Hintergrund somit in einer Art geschrieben werden, dass die Gültigkeit bestimmter Eigenschaften leicht sichtbar und damit auch leicht nachprüfbar ist.

Betrachtet man beispielsweise folgendes Stück Code, dann sieht man, dass der notwendige Aufruf der 'close'-Funktion nur dann erfolgt, wenn $< Condition1 > \Rightarrow < Condition2 >$ gilt und zwischen den skizzierten Anweisungen keine sonstigen störenden Manipulationen der Variable f auftreten.

```
File f;
if(<Condition1>) f.open();
...
if(<Condition2>) f.close();
```

Bereits dieses einfache Beispiel bedarf somit der Lösung eines NP-vollständigen SAT-Problems. Wenn die Bedingungen komplexer Natur sind, explodiert die Laufzeit des Korrektheitsnachweises und ein Programmierer hätte Probleme damit, die Korrektheit nachzuvollziehen. Man muss sich in dieser Situation die Frage stellen, ob der im Beispiel gewählte Kontrollfluss überhaupt gut gewählt ist. Würde man fordern, dass 'open'- und 'close'-Aufrufe immer im gleichen Anweisungsblock stehen müssen, erhält man einen Aufrufkontext, der deutlich einfacher nachzuvollziehen und einfacher beweisbar ist. Zudem wird die Benutzung dieser API zwangsläufig homogener. Dies verringert das Risiko, dass durch eine Evolution der API Fehlersituationen induziert werden.

Die Wahl, inwieweit man die Nutzung einschränken sollte, ist dabei immer ein Kompromiss zwischen zu starker Restriktivität, welche die Nutzung der Deklaration erschwert, und zu großer Toleranz, unter der die Verständlichkeit leidet. Abbildung 6.5 illustriert diesen Zwiespalt anhand der Aufrufe der 'pop'-Funktion. Auf der linken Seite ist eine sehr strikte Herangehensweise

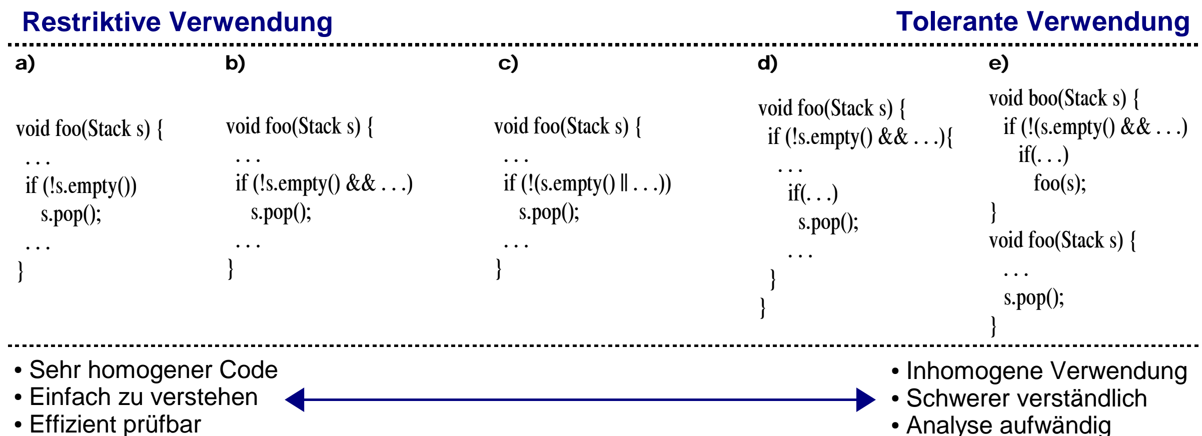


Abbildung 6.5.: Vor- und Nachteile restriktiver Constraints

dargestellt, bei der gefordert wird, dass ‘pop’ nur aufgerufen werden darf, wenn zuvor genau das dargestellte ‘if’-Statement überprüft, ob der Stack leer ist. Eine etwas tolerantere Variation wäre Alternative b, in der die ‘empty’ Überprüfung auch noch mit anderen Bedingungen verundet werden darf. Diese Entschärfung entspricht der Einführung des Axioms des neutralen Elements der Aussagenlogik ($true \Leftrightarrow true \wedge x$). In der Alternative c wären zusätzlich noch Umformungen im Sinne der DeMorganschen Regel ($(\neg F \wedge \neg G) \Leftrightarrow \neg(F \vee G)$) zugelassen. Alternative d erlaubt, dass der ‘pop’-Aufruf tiefer in verschachtelte Fallunterscheidungen eingebettet ist und schließlich toleriert Alternative e die Kapselung des Aufrufs in anderen Methoden. Somit wird dem Schnittstellennutzer sukzessive mehr Freiheit gewährt, aber die Nachvollziehbarkeit der Korrektheit von schwierigeren logischen Aussagen bis hin zur Aufhebung der Lokalität des Problems wird immer komplexer. In vielen Fällen scheint der Gewinn an Verständlichkeit und Homogenität den Verlust der syntaktischen Freiheit zu übertreffen. Oftmals lässt es sich anhand der Dokumentation nachvollziehen, dass der Ersteller einer API sogar bereits bestimmte Nutzungsmuster vorgesehen hat.

Durchsetzung von Richtlinien. Die Zielsetzung von Constraints ist breiter als die von Verifikationstechniken: Constraints sind nicht nur im Kontext von Kontroll- oder Datenfluss-Einschränkungen nützlich. Auch die Einhaltung von Architektur- und Designvorgaben oder Mustern lassen sich damit überprüfen. In den Fallstudien in Teil III dieser Arbeit werden einige derartige Beispiele gezeigt. Diese Anwendungsdimension geht über die Zielsetzung von Verifikationsverfahren hinaus. Gerade in dieser Hinsicht sollten Constraints weniger als Verifikationstechnik, sondern eher als Technik zur Sicherstellung der Konformität mit Programmierrichtlinien (Coding Standards) auf dem Abstraktionsniveau von Deklarationen verstanden werden. Programmierrichtlinien geben meist globale Einschränkungen der Syntax einer bestimmten Programmiersprache vor, wie beispielsweise ein Verbot der Nutzung von Zuweisungen in Verzweigungsbedingungen. Constraints sind in diesem Sinne auch Richtlinien, welche jedoch nicht den Einsatz bestimmter Konstrukte einer Programmiersprache beschränken, sondern die Nutzung von Deklarationen einschränken. Damit sind die Richtlinien, die durch Constraints

überprüft werden, gewissermaßen auf einem höheren Abstraktionsniveau als Richtlinien, wie sie für Programmiersprachenkonstrukte existieren.

Notwendigkeit homogener Nutzung von Deklarationen. In vielen Graphikbibliotheken (z.B. Java-Standard-Widget-Toolkit – SWT) ist es notwendig, graphische Ressourcen, z.B. Bilder für Icons, nach deren Instanziierung (durch Aufruf des Konstruktors) wieder explizit durch den Aufruf einer 'dispose'-Methode freizugeben, um dadurch die Ressourcen des Betriebsystems nicht weiter zu beanspruchen. Werden Ressourcen nicht explizit freigegeben, tritt nach einer gewissen Anzahl an Instanziierungen das Problem auf, dass keine Betriebssystemressourcen mehr verfügbar sind und das Programm mit der Meldung 'Out of Handles' abstürzt. Gerade in graphischen Editoren, die auf Basis der SWT-Bibliothek entwickelt werden, ist dies ein häufig anzutreffendes Problem, das auch bei der Entwicklung des CASE-Tools AutoFOCUS 3 aufgetreten ist. Dieser Fehler ist sehr schwer reproduzierbar und schwer lokalisierbar, da seine Auswirkungen erst stark verzögert und meist zu einem Zeitpunkt auftreten, in dem gerade nicht die den Fehler verursachenden Codestellen ausgeführt werden. Zur Lokalisierung der Fehlerursache ist die aufwändige Durchführung von Reviews des Codes notwendig, indem alle relevanten Stellen einzeln beurteilt werden. Ein Reviewer muss sich dabei in die von unterschiedlichen Programmierern erstellten heterogenen Kontrollstrukturen einarbeiten, um nachvollziehen zu können, ob ein Ressourcenleck hervorgerufen wird. Ein besseres Vorgehen wäre bereits per Konstruktion vorzuschreiben, wie mit Ressourcen im gesamten Programm umzugehen ist, etwa durch Vorgabe einer Konvention, dass im Quelltext direkt unter jeder Methode, die eine Ressource instanziiert, eine Methode zu finden sein muss, die den entsprechenden 'dispose'-Aufruf durchführt. Durch diese Designvorgabe wird der Lösungsraum für die Entwickler deutlich eingeschränkt. Ein Constraint, der diese Vorgabe überprüft, würde noch nicht die Korrektheit des Programms zur Folge haben, die Reviewaktivitäten würden jedoch deutlich vereinfacht, da die Reviewer homogenere Strukturen vorfinden, die einfacher nachvollziehbar sind. Auch die Entwickler selbst werden durch einen derartigen Constraint gewarnt, falls sie den 'dispose'-Aufruf vergessen. Der Constraint schließt somit bewusst viele grundsätzlich korrekte Programme aus, die durch komplexe hintergründige Zusammenhänge sicherstellen, dass keine Ressourcenlecks auftreten. Damit wird die Verständlichkeit der Ressourcen-Handhabung in einem Programm sichergestellt.

7. Werkzeugunterstützung zur effizienten Überprüfung von Constraints

"Es genügt nicht, zum Fluss zu kommen mit dem Wunsch, Fische zu fangen.
Man muss auch das Netz mitbringen."
- Chinesisches Sprichwort

In den vorangehenden Abschnitten wurden Constraints als prädikatenlogische Ausdrücke über der Sprachgrammatik eingeführt sowie der praktische Einsatz von Constraints in Softwareprojekten vorgestellt. In diesem Abschnitt soll die grundsätzliche Funktionsweise und die Architektur eines Werkzeugs für die Programmiersprache Java präsentiert werden, das den praktischen Einsatz von Constraints unterstützt. Dieses Werkzeug wurde im Rahmen dieser Arbeit prototypisch entwickelt und für die in Kapitel 9 vorgestellte Fallstudie erfolgreich eingesetzt.

7.1. Anforderungen an die Werkzeugunterstützung

Eine umfassende Aufzählung der Anforderungen an ein Werkzeug zur Definition und Überprüfung von Constraints würde den Rahmen dieser Arbeit sprengen. Aus diesem Grund werden an dieser Stelle nur die wesentlichen Anforderungen vorgestellt, die ein derartiges Werkzeug erfüllen sollte.

Folgende Merkmale sollte ein Werkzeug zur Überprüfung von Constraints erfüllen:

1. *Nutzung von Constraints:* Das Werkzeug sollte es ermöglichen, Constraints durch einfache Annotation von Deklarationen im Quellcode zu instanziieren. Wie in Abschnitt 6.1 ausgeführt, sind Modelle in vielen Fällen jedoch besser geeignet, um bestimmte Arten von Constraints auszudrücken. Aus diesem Grund sollte es möglich sein, das Werkzeug zur Überprüfung von Constraints auch im Hintergrund von Modellierungswerkzeugen zu verwenden. Das Modell würde somit die Informationen über die einschränkenden Kriterien zur Nutzung bestimmter Deklarationen bestimmen. Die Überprüfung auf Konformität mit dem Quellcode wird durch das im weiteren präsentierte Werkzeug erzielt.
2. *Ausgabe von Constraintverletzungen:* Das Werkzeug muss in der Lage sein, die gefundenen Verletzungen der Constraints im Code ähnlich zu Compilerfehlern in modernen Entwicklungsumgebungen in Form einer Tabelle anzuzeigen und Navigationsmöglichkeiten zur effizienten Lokalisierung und Behebung der gefundenen Verletzungen bereitzustellen. Auch eine Integration der Analyseergebnisse in Modellierungswerkzeuge, die zur Spezifikation von Constraints verwendet werden, sollte möglich sein (vgl. Abbildung 6.1). Hierfür

bietet sich ein generisches Austauschformat für identifizierte Verletzungen an, das von unterschiedlichen Modellierungswerkzeugen genutzt werden kann (vgl. [Kel09]).

3. *Implementierung von Constraints:* Das Werkzeug sollte es ermöglichen, neue Constraints zu definieren. Hierfür muss es ermöglicht werden, einen symbolischer Bezeichner für den Constraint festzulegen, der schließlich zur Referenzierung des Constraints verwendet werden kann (z.B. in Form einer Annotation im Code). Die Logik der Constraintüberprüfung soll in einer herkömmlichen Programmiersprache implementiert werden können.

4. *Wiederverwendung von Constraints:* Um eine möglichst hohe Wiederverwendbarkeit und Rekombinierbarkeit von Constraints erreichen zu können und damit den effizienten Einsatz des Werkzeugs sicherzustellen, sollten Parameter für Constraints definiert werden können. Zudem sollte, wie in Abschnitt 5.2 erläutert, die Kompositionalität von Constraints auch durch das Werkzeug unterstützt werden.

5. *Unterstützung selbstbeschränkender Constraints:* Die Annotation von Constraints sollte auch für Deklarationen möglich sein, die nicht direkt in Form von Quellcode vorliegen. Um auch die Nutzung selbstbeschränkender Constraints (vgl. Abschnitt 6.2.1) zu ermöglichen, müssen Annotationen auch unabhängig von der Verfügbarkeit des Quellcodes in einer externen Datei vorgenommen werden können.

6. *Effiziente Prüfung von Constraints:* Damit Entwicklern zeitnah die Informationen über Verletzungen von Constraints zur Verfügung stehen, ist es wichtig, dass die Constraintüberprüfungen sehr performant durchführbar sind. Die benötigte Dauer der Überprüfungen hängt zwar von den genutzten Constraints ab (vgl. Abschnitt 6.3), die Architektur des Werkzeugs sollte jedoch vorsehen, dass Constraints lediglich eine Traversierung des Syntaxbaums eines Programms benötigen. Während dieses Durchlaufs sollten alle genutzten Constraints parallel überprüft werden.

7.2. Architektur und Funktionsweise

Grundsätzliche Funktionsweise. Eine mögliche Herangehensweise zur Implementierung eines Werkzeugs zur Überprüfung von Constraints wäre die Konzeption einer Werkzeugarchitektur, die ähnlich zu Abfragen einer relationalen Datenbank ein Programm nach Verletzungen der Constraints "abfrägt". Dies ist beispielsweise das Konzept von Analysewerkzeugen, wie dem Sotograph [BKL04]. Derartige Werkzeuge haben jedoch einen hohen Initialisierungsaufwand, da, bevor Analysen ausgeführt werden können, zunächst die Daten in eine geeignete Struktur geladen werden müssen, um die Abfragen ausführen zu können. Im Fall von Änderungen im Programm muss dieser Ladevorgang wiederholt werden. Da Constraints jedoch möglichst häufig während der Programmieraktivitäten überprüft werden sollten, würde ein derartiges Laden zu inakzeptablen Wartezeiten führen (Anforderung 6). Im Gegensatz zur Berechnung von meist stark aggregierten Metriken trägt ein derartiger Ansatz nicht zur Überprüfung von Constraints bei, da hierfür nicht nur die grobgranularen Programmsturkturen (Klassen, Methoden, ...) von Interesse sind, sondern Informationen über das Programm bis auf die Ebene von Statements und Expressions benötigt werden. Die Überprüfung von Constraints sollte

sehr ähnlich zum Kompiliervorgang auch auf großen Programmen innerhalb weniger Minuten durchführbar sein. Da in Bibliotheken sehr viele Constraints vorhanden sein können, ist eine effiziente Prüfung äußerst wichtig, um den Arbeitsfluss nicht zu behindern. Zur abfragebasierten Constraintüberprüfung wäre eine enorme Anzahl an Join-Operationen, wie sie auch in Datenbankabfragen zu finden sind, über sehr große Datenmengen notwendig. Selbst ein einzelner Join aller logischen Ausdrücke eines Programms mit allen Anweisungen ist auf realen Programmgrößen mit bis zu mehreren Millionen Codezeilen auf normalen Entwicklerrechnern nicht mit den benötigten Antwortzeiten realisierbar.

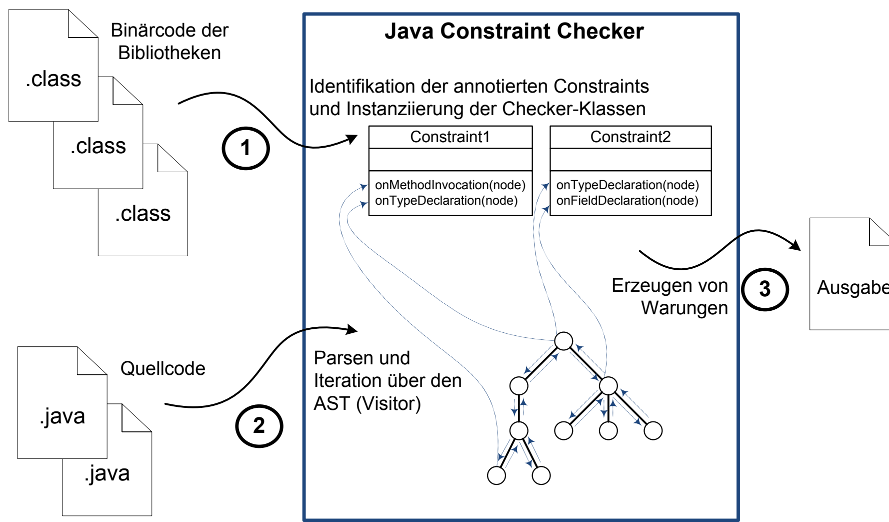


Abbildung 7.1.: Der Java Constraint Checker im Überblick

Aus diesen Gründen wird hier ein anderer Ansatz verfolgt, der die lineare Struktur des Programmtextes ausnützt. Wie Abbildung 7.1 illustriert, handelt es sich um eine gemischte Binär- und Sourcecodeanalyse. Da Bibliotheken nicht immer auch als Quellcode verfügbar sind, ist es wichtig, dass die Constraints aus den kompilierten binären Dateien extrahiert werden können (Anforderung 1). Es wird davon ausgegangen, dass der Constraint Checker nach einem Build-Vorgang ausgeführt wird und somit die Quell- und Binärdateien konsistent sind. Neben Constraints, die als Annotationen im Binärcode gefunden werden, können Constraints auch durch Angabe des vollqualifizierten Namens einer Deklaration und des entsprechenden Constraints in einer separaten Textdatei spezifiziert werden (Anforderung 5). Der Constraint Checker führt im Wesentlichen folgende drei Schritte aus:

1. *Extraktion der Constraints aus dem Binärcode und Instanziierung der passenden Checker-Klassen:* In diesem Schritt werden alle Binärdateien (sowohl Bibliotheken als auch der kompilierte Quellcode des Programms) auf annotierte Constraints untersucht. Anschließend werden separate Textdateien mit externen Constraint-Annotationen gesucht. Für jede gefundene Constraint-Annotation wird nach einer passenden Constraint-Definition in Form einer Checker-Klasse gesucht (Reflection). Sobald eine passende Checker-Klasse gefunden wurde, wird diese instanziiert und am Visitor registriert.

2. *Überprüfung der Constraints auf dem abstrakten Syntaxbaum:* Nachdem in Schritt 1 alle Constraints gefunden wurden, werden nun die zu prüfenden Quelldateien eingelesen und geparst. Anschließend wird eine Tiefensuche über dem abstrakten Syntaxbaum durch einen zentralen Visitor durchgeführt. Sobald während dieser Tiefensuche Knoten passiert werden, die für die Überprüfung eines Constraints relevant sind, wird die entsprechende Checker-Klasse aufgerufen, um die entsprechenden Überprüfungen durchzuführen.

3. *Ausgabe aller Verletzungen in Form von Warnungen:* Werden durch die Checker-Klassen Verletzungen der Constraints festgestellt, werden entsprechende Warnungen generiert. Diese werden anschließend über ein Pipes-und-Filter System (s.u.) nachbearbeitet und schließlich ausgegeben.

Diese Art der Constraintüberprüfung ist im Vergleich zu einem abfragebasierten Ansatz deutlich performanter. Obwohl die Laufzeit natürlich stark von den eingesetzten Constraints und der Komplexität der notwendigen Überprüfungen abhängt, ist für einfache Constraints lediglich ein Durchlauf über den abstrakten Syntaxbaum erforderlich.

Implementierung von Constraints. Der Constraint Checker ist als erweiterbares Werkzeug konzipiert. Neben der Anwendung von mitausgelieferten Constraints, lassen sich eigene spezifische Constraints implementieren. Hierfür muss lediglich ein neuer Annotationstyp sowie eine passende Checker-Klasse implementiert werden (Anforderung 3). Für jeden Knotentyp des abstrakten Syntaxbaums, der für den Constraint relevant ist, muss die Checker-Klasse ein spezielles Interface implementieren. Die darin definierten Methoden werden schließlich automatisch seitens des Visitors aufgerufen (Hollywood-Prinzip).

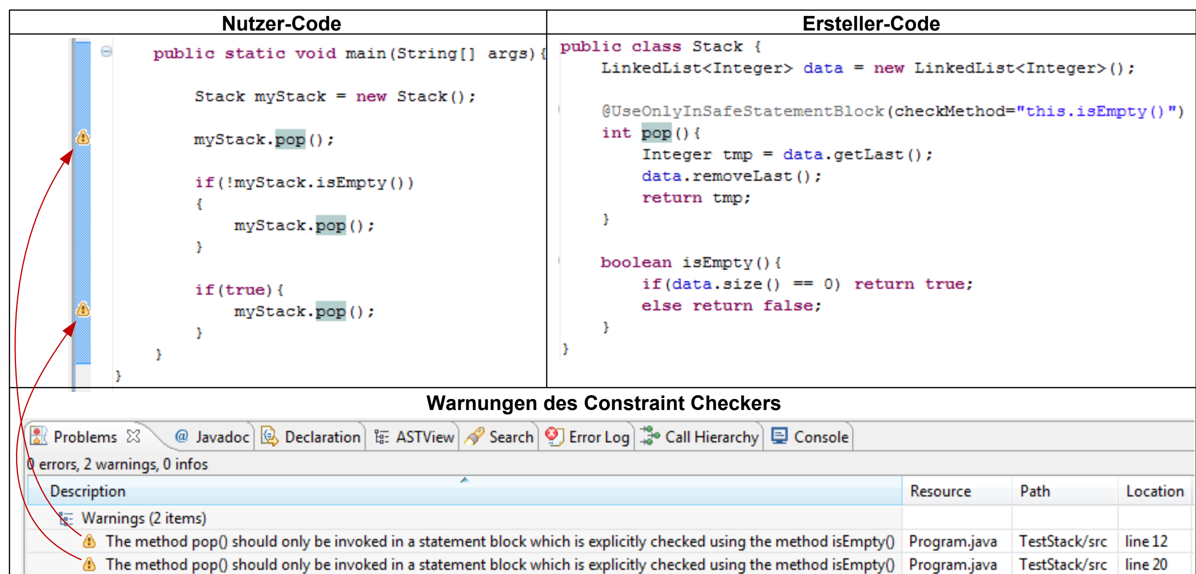


Abbildung 7.2.: Überprüfung des PopInIf-Constraints durch den Constraint Checker

Annotation von Constraints. In Abbildung 7.2 sind Screenshots des Einsatzes des Constraint Checkers in Eclipse zur Sicherstellung der adäquaten Verwendung der 'pop'-Methode (vgl. Constraint in Formel 5.1) dargestellt. Auf der rechten Seite ist eine einfache Implementierung eines Stacks zu sehen, deren 'pop'-Methode mit einem Constraint in Form einer Java Annotation versehen ist. Die Annotation spezifiziert das Argument 'this.isEmpty', welches definiert, dass die 'isEmpty'-Methode aufgerufen werden muss, um zu überprüfen, dass das System in einem Zustand ist, in dem die 'pop'-Methode aufgerufen werden darf. 'this' ist in diesem Fall ein Schlüsselwort des Constraint Checkers, das definiert, dass der Aufruf der 'isEmpty'-Methode auf dem gleichen Objekt stattfinden muss, wie der Aufruf der 'pop'-Methode, der dadurch abgesichert werden soll. Auf der linken Seite von Abbildung 7.2 ist ein Stück Code dargestellt, das die Stack-Implementierung nutzt. Am linken Rand des Editors sind Warnhinweise des Constraint Checkers erkennbar, die den Entwickler darauf hinweisen, dass die Aufrufe der 'pop'-Methode in diesen Zeilen nicht den Vorgaben des annotierten Constraints entsprechen (Anforderung 2). Im unteren Teil der Abbildung sind schließlich die Erklärungen der Verstöße dargestellt.

7.3. Wiederverwendung von Constraints

Der Constraint Checker ist so konzipiert, dass neue Constraints durch das zur Verfügung stehende Framework einfach implementiert werden können. Dennoch ist die Implementierung von Constraints mit einem gewissen Aufwand verbunden. Um den Constraint Checker möglichst einfach gewinnbringend einsetzen zu können, ist es wichtig, dass häufig benötigte Constraints direkt, ohne zusätzlichen Programmieraufwand, genutzt werden können. Hierfür sind geeignete Constraint-Bibliotheken notwendig, die generische Constraints anbieten, die durch Parametrisierung und flexible Komposition einfacher Constraints zu komplexeren Constraints auf die Problemstellungen in einem Programm angepasst werden können (Anforderung 4).

Komposition von Constraints. Abbildung 7.3 skizziert die grundsätzliche Funktionsweise der Checker-Klassen im Constraint Checker Framework. Die Klassen werden durch das Framework instanziiert, sobald eine Constraint-Annotation gefunden wird. Anschließend werden diese bei einem Visitor registriert, so dass diese bei der Durchführung der Überprüfung auf einer Quellcode-Datei immer dann angestoßen werden, wenn ein für den Constraint relevantes Sprachkonstrukt gefunden wird. Identifiziert ein Objekt einer Checker-Klasse einen Verstoß gegen einen Constraint, so wird eine Warnung erzeugt und an einen sogenannten 'OutputReporter' übergeben. Dahinter verbirgt sich ein Pipes-und-Filter Konzept, da nicht nur ein zentraler 'Reporter' existiert, der alle Warnungen sammelt und für die Ausgabe aufbereitet, sondern auch Constraints selbst die Funktion eines 'Reporters' innehaben können. In diesem Fall handelt es sich um komplexe Constraints, die sich aus anderen Constraints zusammensetzen. Eine logische Veroderung zweier Constraints wäre somit wiederum als eigene Checker-Klasse implementiert, die lediglich die Warnungen von deren Unterconstraints erhält und daraus eine Warnung erzeugt (bzw. weitergibt), wenn beide Unterconstraints an derselben Stelle einen Verstoß melden. Die Komposition von Constraints wird somit durch Einführung eines komplexen Constraints realisiert, der die zu verknüpfenden Constraints als Parameter erhält.

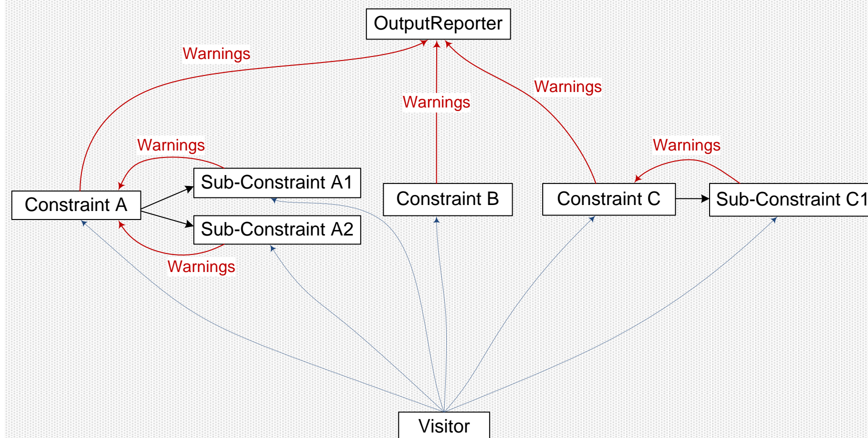


Abbildung 7.3.: Funktionsweise des Constraint Checkers

Parametrisierung von Constraints. Die Annotationen können mit Parametern versehen werden. Identifiziert der Constraint Checker eine Constraint-Annotation, wird die entsprechende Checker-Klasse instanziiert. Die Parameter werden aus der Annotation ausgelesen und dem Konstruktor der Checker-Klasse übergeben. Als Parameter sind jedoch nicht immer nur einfache Zeichenketten oder numerische Werte notwendig. Wie bereits in Abschnitt 5.2 ausgeführt, würde eine Parametrierung des $PopInIf$-Constraints für die 'pop'-Methode mit dem zu überprüfenden logischen Ausdruck eine deutlich flexibler wiederverwendbare Implementierung ergeben. Ein derartiger Parameter hätte den Typ eines Sprachkonstrukts der Sprache Java, in diesem Beispiel $\underline{Expr}$. In der Sprache Java ist es jedoch nicht möglich, komplexe Typen (Objekttypen) als Parameter in Annotationen zu nutzen. Aus diesem Grund ermöglicht der Constraint Checker die Annotation bestimmter Java Konstrukte als Zeichenketten. Diese werden beim Einlesen geparst und der Konstruktor der Checker-Klasse erhält ein Objekt des entsprechenden AST-Knotens (Typ im JDT definiert). Die Implementierung der Checker-Klasse kann somit bequem auf einen AST des annotierten Parameters zugreifen. Durch Einsatz dieses partiellen Parsens kann die Wiederverwendbarkeit von Constraint-Implementierungen gesteigert werden.

8. Verwandte Methoden und Techniken

"Die Wissenschaft fängt eigentlich erst da an interessant zu werden, wo sie aufhört."
- Justus von Liebig

In diesem Kapitel werden verwandte Arbeiten aus unterschiedlichen Bereichen präsentiert und mit dieser Arbeit in Bezug gesetzt.

8.1. Konservierung von Wissen

Es gibt vielerlei Werkzeuge und Methoden, um Programme zu analysieren. Dem aktuellen Stand der Technik mangelt es aber an Mechanismen, die es dem Ersteller von Deklarationen erlauben, präzise und in nachprüfbarer Form zu definieren, wie diese verwendet werden sollen.

Man kann eine starke Ähnlichkeit feststellen zwischen den Vorgaben, die ein Ersteller einer Deklaration über seine Klienten macht und den Zielen (Goals), die von Letovsky [LS86] in den 80er Jahren definiert wurden. Diese Goals werden genutzt, um die Intentionen eines Programmierers über einen Teil des Codes aufzuzeigen, während die Vorgaben Erwartungen des Erstellers einer Deklaration über den Code von dessen Klienten darstellen. Nach Letovsky werden die Goals durch sogenannte Programming Plans erfüllt. In dem hier vorgestellten Ansatz werden die adäquaten Strukturen der API-Verwendung (dem Plan) durch Constraints ausgedrückt, die somit die Vorgaben des API-Erstellers konservieren. Dadurch, dass die Constraints über der Syntax des Client-Codes beschrieben werden, ist es möglich, API-spezifische Kodierungsregeln zu definieren, um eine höhere Homogenität des Nutzercodes und damit eine höhere Programmverständlichkeit zu erreichen.

8.2. Dynamische Ansätze

Neben dem im Rahmen dieser Arbeit vorgestellten statischen Ansatz sind in der Literatur zahlreiche Ansätze zu finden, die versuchen, die adäquate Nutzung von Funktionen zur Laufzeit nachzuweisen. Der wohl bekannteste Ansatz stammt von David Parnas. In [BP78] präsentiert Parnas eine Methode, um das Verhalten von Softwaremodulen durch Nutzung von Trace Assertions zu spezifizieren. Diese Traces können durch ein Monitoring des laufenden Systems aufgezeichnet und auf Korrektheit bezüglich der definierten Assertions überprüft werden. Im Gegensatz dazu wurde in dieser Arbeit ein rein statischer Ansatz vorgestellt. Statische Analysen haben den Vorteil, dass man durch sie im Gegensatz zu dynamischen Verfahren tatsächlich

Aussagen über die Abwesenheit bestimmter Fehler ableiten kann. Durch Nutzung dynamischer Analysen hingegen können auch komplexe semantische Sachverhalte beispielhaft geprüft werden, die in statischen Analysen zu hohe Komplexität hervorrufen würden. Der Fokus auf die Wartbarkeit, der in dieser Arbeit verfolgt wird, geht über das Ziel der Trace Assertions hinaus.

Es gibt viele aktuelle Arbeiten, wie etwa [ABL02, RMG⁺06, KKS06], die durch dynamische Analysen in einem Reverse Engineering Ansatz versuchen, Spezifikationen aus der gegebenen Nutzung von APIs zu rekonstruieren. Die dadurch abgeleiteten Nutzungsmuster sind sehr ähnlich zu kontrollflussorientierten Constraints. Derartige Ansätze könnten sehr gut eingesetzt werden, um bestimmte Arten von Constraints in existierenden Systemen zu identifizieren.

8.3. Constraint-Sprachen

Der in diesem Kapitel vorgestellten Arbeit kommen Constraint-Sprachen sehr nahe. Am bekanntesten sind diese Sprachen im Bereich der Modellierung, beispielsweise OCL für die UML [OMG06b, EW05]. Es existieren aber auch auf Basis von Programmiersprachen Ansätze zur Beschränkung der Verwendung bestimmter Invarianten. Die Constraint Sprache SCL [HH06, Hou07] kann dazu eingesetzt werden, um die Instanziierung objektorientierter Frameworks zu überprüfen. Der vorgestellte Ansatz erweitert diese Zielsetzung in zwei Richtungen: Einerseits werden typische Arten von Vorgaben, die Ersteller von Deklarationen über ihre Klienten machen, formalisiert und über Constraints ausdrückbar gemacht, die durch Parametrisierung einfach wiederverwendet werden können. Andererseits ist das vorgestellte formale Framework, das die Spezifikation von Constraints unterstützt, programmiersprachenunabhängig. Da der Constraint-Formalismus rein auf der Syntax der Programmiersprache basiert, ist es möglich, den Ansatz leicht auf andere Sprachen zu adaptieren. Durch die explizite Berücksichtigung der Abstraktionsmechanismen einer Programmiersprache können sowohl Deklarationen (Elemente zweiter Klasse in einem Programm) als auch Konstrukte der Sprache (Elemente erster Klasse) in die Constraints einbezogen werden. In anderen Constraint-Sprachen werden die Constraints global über Modelle und Systeme definiert [HHR04]. Der in dieser Arbeit präsentierte Ansatz beschränkt sich nicht auf eine spezifische Modellierungstechnik. Constraints können die Abbildung unterschiedlichster Modelle auf den Code definieren oder sogar lokal an die zu beschränkenden Deklarationen annotiert werden.

8.4. Statische Analysen

In der Praxis kommen derzeit meist einfache regelbasierte (meist heuristische) statische Analysewerkzeuge wie FindBugs[1], PMD[2] und andere zum Einsatz (vgl. Abschnitt 3.3). Diese Werkzeuge sind meist direkt anwendbar, da keine weiteren Informationen über das implementierte System, außer der Verwendung einer bestimmten Programmiersprache, vorliegen müssen. Diese

[1] http://findbugs.sourceforge.net/
[2] http://pmd.sourceforge.net/

einfachen Werkzeuge haben jedoch genau aus diesem Grund das Problem, dass sie sehr ungenau sind und hohe Raten an False-Positives generieren. Die Analysen finden auf dem Abstraktionsniveau der Programmiersprache statt und sind damit nicht in der Lage, Aussagen über die Verwendung von implementierten Konzepten zu erzeugen. Etwas komplexere Werkzeuge schließen zusätzlich noch die Untersuchung der korrekten Verwendung von Standard Bibliotheken mit ein (z.B. [GS06] für C++). Obwohl die Untersuchungen in diesem Kapitel auf den Java Standard APIs durchgeführt wurden, ist der Ansatz nicht auf Standardbibliotheken beschränkt und kann auch auf domänenspezifische Frameworks und Bibliotheken angewandt werden.

Design-by-Contract Methoden wie JML [LLP+00] annotieren Verträge in Form von Vor-/Nachbedingungen und Invarianten an APIs. Diese Annotationen können auch als Constraints betrachtet werden. Wie bereits ausführlich in Abschnitt 6.3 erläutert, muss der Aufruf einer Methode, auch wenn er semantisch die Forderungen des Vertrags erfüllt und beispielsweise durch Methoden, wie in [Nel04, DLNS98] beschrieben, nachgewiesen werden kann, noch keine adäquate Nutzung des Konzepts darstellen, da zu komplexe Strukturen für die Wartung zu schwer nachvollziehbar sind.

In [HCXE02, CEH02] und [DF04b] werden auf Zustandsmaschinen basierende Ansätze vorgestellt, die systemspezifische statische Analysen des Aufrufverhaltens von Methoden erlauben. Diese Ansätze fokussieren sich rein auf die Absicherung des Kontrollflusses. Im Gegensatz dazu präsentieren die Autoren von [ESM05] eine Methode, um strukturelle Eigenschaften zu überprüfen. Es gibt noch viele weitere Ansätze [Hed97], um spezielle Eigenschaften der Verwendung von Typen oder Funktionen in Programmen zu analysieren. Die meisten der durch diese Ansätze analysierbaren Eigenschaften könnten grundsätzlich in einem einheitlichen Ansatz in den Constraint Checker integriert werden.

Im Projekt SLAM [BR02] wurde die adäquate Nutzung einer API zur Implementierung von Treibern analysiert. Das für eine spezielle API angewandte Verfahren ist sehr ähnlich zum Prinzip von Constraints, wie sie im Rahmen dieser Arbeit betrachtet werden. Im SLAM-Projekt wurden sehr schwergewichtige Verifikationsverfahren kombiniert, um den Nachweis der adäquaten Nutzung zu erbringen. Die Autoren berichten von der hohen Komplexität der Erstellung einer Spezifikation, die als Grundlage für die Verifikation diente. Eben diese Hürde erschwert den Einsatz derartiger Verfahren in der breiten Praxis. Die Constraints, die in dieser Arbeit vorgestellt wurden, sind ein sehr leichtgewichtiger Ansatz, ihre Spezifikation sollte somit auch normal ausgebildeten Entwicklern möglich sein, da die Spezifikation eines Constraints direkt auf der Syntax der verwendeten Programmiersprache basiert.

8.5. Erweiterbare Typsysteme

Die Vorteile von Typsystemen sind weitreichend bekannt [Bra04]:

- Typen ermöglichen eine Form von maschinell prüfbarer Dokumentation.
- Typen bieten ein konzeptionelles Rahmenwerk für Programmierer, das für den Programmentwurf, die Wartbarkeit und das Programmverstehen hilfreich ist.

- Typsysteme unterstützen frühe Fehlererkennung.

Je nach Art des eingesetzten Typsystems bieten diese unterschiedliche Analysemöglichkeiten. Im Forschungsbereich erweiterbarer oder auch austauschbarer Typsysteme (engl. pluggable type systems) existieren Arbeiten, die das Typsystem einer Programmiersprache je nach Einsatzzweck anpassbar machen (vgl. [ANMM06, PAC+08]). Durch spezielle Typsysteme, die auf zusätzlichen Annotationen im Programm basieren, können bestimmte Arten von Analysen durchgeführt werden, wie etwa Überprüfungen des Zustands von Typen (vgl. [LKR04]) oder Analysen bzgl. der Konsistenz von Datenstrukturen (vgl. [KLZR06]).

Grundsätzlich verfolgen diese Arbeiten sehr ähnliche Ziele wie die in dieser Arbeit vorgestellten Techniken. Der Hauptunterschied liegt in der Methodik des Einsatzes: Typannotationen im Code können oftmals eine gute Möglichkeit bieten, die statische Überprüfung zu verbessern und damit Informationen zu bewahren. Jedoch ist dies nicht immer eine gute Strategie Entwurfsregeln auszudrücken. Nicht alle diesbezüglichen Informationen sollten im Code annotiert werden, da beispielsweise Architekturregeln dadurch meist über das ganze System verteilt werden. Vielmehr ist es in diesem Fall wünschenswert, diese Informationen in Form eines Modells zu erstellen und zu pflegen, da diese Modelle einen besseren Überblick über das Gesamtsystem bieten und nicht in den Details des Quellcodes untergehen.

Teil III.

Praktische Anwendung von Constraints

9. Anwendungsbereich 1: Implizite Vorgaben in objektorientierten APIs

"Wissen ist gut als Unterstützung, Förderung und Aufklärung im Praktischen; wenn es aber die Praxis ersetzen soll, ist es keinen Schuß Pulver wert."
- Theodor Fontane

Bibliotheken und Frameworks sind die am weitesten verbreitete Arten der Software-Wiederverwendung. Sie bieten eine Implementierung von Konzepten einer bestimmten Domäne durch Programmabstraktionen wie Klassen und Methoden an. Meist sind dabei die Ersteller und die Nutzer der API vollständig entkoppelt. Dadurch sind die Nutzer einer Bibliothek mit dem Problem alleine gelassen, die implementierten Konzepte zu verstehen und korrekt zu verwenden. Ein Nutzer kann sich niemals sicher sein, dass er eine Methode oder eine Klasse korrekt interpretiert und schließlich im Sinne des Erstellers verwendet.

Moderne Programmiersprachen stellen eine Menge an Mechanismen, wie beispielsweise überprüfte Ausnahmen (engl. checked exceptions), statische Typisierung oder Sichtbarkeiten, zur Verfügung, um eine gutartige Verwendung von APIs sicherzustellen. Dennoch ist dadurch bei weitem noch nicht sichergestellt, dass ein Nutzer die verwendeten Bibliotheken korrekt, sicher und getreu der Architektur/des Entwurfs verwendet, nur weil sein Programm kompiliert.

In APIs existiert eine enorme Menge an impliziten Vorgaben (Regeln) über die Verwendung der darin implementierten Deklarationen. Diese Vorgaben sind meist an eine Menge an impliziten (antizipierten) Verwendungsszenarien der API angelehnt, die oftmals nur teilweise (wenn überhaupt) in der Dokumentation, in Tests oder anhand informeller Beispiele der API-Verwendung reflektiert werden. Um eine API adäquat zu verwenden, sollten deren Nutzer diese impliziten Regeln berücksichtigen. Andernfalls kann eine nicht zu den vorhergesehenen Verwendungsszenarien konforme Verwendung zu einem Missbrauch der API führen. Dies wiederum erzeugt nicht nur (latente) Fehler, sondern führt auch zu Problemen, den API-Nutzercode zu verstehen und zu warten.

In Abbildung 9.1 ist ein Fragment einer Java API dargestellt, das die Abstraktionsmechanismen von Java nutzt, um einen Stack zu implementieren. Auf der rechten Seite sind zwei mögliche Nutzungen der 'pop'-Funktionalität zu sehen: Eine adäquate Verwendung, bei der explizit geprüft wird, ob der Stack leer ist, und (b) eine andere, die ohne derartige Prüfung den Stack aufruft. Der zweite Fall (c) zeigt somit einen potentiellen (latenten) Laufzeitfehler, falls der Stack leer sein sollte.

API-Seite	Nutzer-Seite	

`class Stack { ...`
`  public Object pop() {`
`    if (elementCount == 0)`
`      throw new EmptyStackException();`
`    ...`
`  }`
	`void foo(Stack s) {`
	`  s.pop();`
	`}`
		`void boo(Stack s) {`
		`  if (!s.isEmpty())`
		`    s.pop();`
		`}`

a)	b)	c)

Abbildung 9.1.: API-Konzepte und ihre Verwendung

Meist könnten die Verwendungsszenarien als eine Menge von Verwendungsmustern (Patterns) beschrieben werden, welche die vorgesehenen adäquaten Verwendungen widerspiegeln. Im vorangehenden Beispiel wäre eine explizite Überprüfung, ob der Stack leer ist oder nicht, vor dem Aufruf von 'pop' ein derartiges Muster. Wenn die Vorgaben des API-Erstellers durch den die API nutzenden Code explizit erfüllt werden, handelt es sich um eine *adäquate* Verwendung einer API. In den nachfolgenden Abschnitten dieses Kapitels wird veranschaulicht, dass derartige Verwendungsmuster oftmals durch kontextsensitive Constraints auf der Syntax des API-Clients ausdrückbar sind. Diese Constraints beschränken die "Gestalt" des API-Nutzercodes, um nur noch valide Nutzungen zuzulassen, welche die Vorgaben des Erstellers explizit erfüllen und somit leichter zu verstehen und weiterzuentwickeln sind.

Durch die Spezifikation von statisch überprüfbaren Mustern werden API-Erstellern Möglichkeiten an die Hand gegeben, ihre Intention der Verwendung ihrer API besser an ihre Nutzer zu transportieren. Beschränkungen der Nutzung können damit längerfristig auch die Evolution einer API erleichtern, da die Heterogenität der potentiellen Nutzungsmöglichkeiten begrenzt wird.

Überblick. Im folgenden Abschnitt werden einige Beispiele gängiger Vorgaben von API Erstellern vorgestellt. Diese Vorgaben entstammen der Java-Standard-Library, auf der die im Rahmen dieses Kapitels vorgestellte Fallstudie basiert. In Abschnitt 5.2 wurde ein formales Rahmenwerk zur Definition kontextsensitiver Constraints eingeführt. Die im folgenden Abschnitt präsentierten Constraints werden anschließend in Abschnitt 9.2 klassifiziert. Dabei wird der Formalismus genutzt, um Constraints zu spezifizieren, die es erlauben, die Einhaltung der impliziten Vorgaben zu überprüfen. Das in Abschnitt 7 vorgestellte Werkzeug zur effizienten Überprüfung syntaktischer Constraints wurde verwendet, um die beschriebenen Vorgaben auf verschiedenen Programmen zu überprüfen. In Abschnitt 9.3 werden die Ergebnisse dieser Analyse dargestellt. Teile der im Folgenden präsentierten Fallstudie wurden bereits in [FR08] veröffentlicht.

9.1. Typische Vorgaben in der Java API

In diesem Abschnitt werden Beispiele für Vorgaben bezüglich der adäquaten Nutzung der Java-Standard-API präsentiert. Die Beispiele sind in Klassen ähnlicher Vorgaben kategorisiert,

```
public class Stack { //package java.util;
  /** ...
   * @throws: EmptyStackException - if this stack is empty.
   */
  public Object pop() { ... }   ...
}      a) Pop-Only-Non-Empty-Stacks
```

```
public class Object { //package java.lang;
  /** ...
   * waits should always occur in loops, like this one:
   *     synchronized (obj) {
   *         while (condition does not hold)
   *             obj.wait(timeout);
   *         ... // Perform action appropriate to condition
   *     }
   */
  public final native void wait(long timeout) { ... } ...
}             b) Wait-In-A-While
```

```
class Calendar { //package java.util;
  /**
   * @param field the time field. One of the time field constants
   */
  public void set(int field, int value) { ... } ...
} c) Parameter-Is-A-Time-Field-Constant
```

```
public class JScrollPane { //package javax.swing;
  /** ...
   * @exception ClassCastException if layout is not a ScrollPaneLayout
   */
  public void setLayout(LayoutManager layout) { ... }   ...
}   d) Parameter-Is-A-ScrollPaneLayout
```

```
public class File { //package java.io;
  /** ...
   * The return value should always be checked to make
   * sure that the rename operation was successful.
   */
  public boolean renameTo(File dest) { ... }
  ...
}e) Check-The-Return-Of-RenameTo
```

```
public class Object { //package java.lang;
  /** ...
   * it is generally necessary to override the 'hashCode' method
   * whenever this method is overridden, ...
   */
   public boolean equals(Object obj) { ... }
} f) Override-Both-Hashcode-And-Equals
```

```
public class Component { //package java.awt;
 /** ...
  * Subclasses [...] that override this method should either call
  * super.update(g), or call paint(g) directly from their update method
  */
  public void update(Graphics g) { ... } ...
}         g) Do-Not-Forget-To-Paint
```

```
public class Component { //package java.awt;
  /** ... This method is called internally by the toolkit and
   * should not be called directly by programs.
   */
   public void addNotify() { ... }   ...
}             h) Do-Not-Call-AddNotify
```

Abbildung 9.2.: Implizite Vorgaben in der Java API

jede dieser Klassen wird in einem eigenen Unterabschnitt beschrieben. Jedes der im Folgenden beschriebenen Beispiele repräsentiert einen Fall von implizitem Wissen über die adäquate Nutzung der API, in Form von Regeln über die Bibliotheksnutzung, die von den Nutzern eingehalten werden sollen, jedoch von keinem Mechanismus der Programmiersprache geprüft werden. Für einige der Vorgaben wird ihre Historie in der Java API Dokumentation angegeben, die Aufschluss über die Einführung der impliziten Vorgaben in die begleitende Dokumentation geben.

Anstatt Beispiele zu erfinden oder aus unbekannten Systemen zu präsentieren, wurden alle Beispiele der Java API entnommen, um zu verdeutlichen, dass derartige Probleme selbst in einer der am häufigsten verwendeten Bibliothek existieren. Abbildung 9.2 zeigt die Teile der Java API, die in den Beispielen verwendet werden.

9.1.1. Vorgaben über den Systemzustand

Der Zustand des Systems vor oder nach dem Aufruf einer Methode ist in vielen Fällen von entscheidender Bedeutung. Vorgaben über den Systemzustand werden in der Dokumentation meist durch Phrasen wie *"überprüfe vor dem Aufruf von X, dass ... gilt"* oder *"rufe X immer im Kontext Y auf"*. Dadurch verleiht ein API-Ersteller einer Vorgabe Ausdruck, dass seine Nutzer vor dem Aufruf einer Methode überprüfen sollten, ob das System im gewünschten Zustand ist.

Im Folgenden werden zwei typische Fälle derartiger Vorgaben vorgestellt:

"Pop-Only-Non-Empty-Stacks". Ein sehr bekannter Fall einer Vorgabe über den Systemzustand ist die Methode 'pop' in der Klasse 'java.util.Stack' (Abbildung 9.2a). Die Ersteller der

Java API setzen voraus, dass ‘pop’ nicht auf einem Objekt aufgerufen wird, das einen leeren Stack repräsentiert. Sollte ‘pop’ auf einem leeren Stack aufgerufen werden, ist der Vertrag der ‘pop’-Methode gebrochen und ein Laufzeitfehler tritt ein. Somit ergibt sich ein latenter Fehler, der sich nur zur Laufzeit manifestiert.

“Wait-In-A-While”. Jeder Aufruf der Methode ‘wait’ in der Klasse ‘java.lang.Object’ muss im Kontext einer ‘while’-Schleife erfolgen. Dies liegt an so genanntem falschen Erwachen (engl. spurious wake-ups) begründet. In bestimmten Situationen kann es passieren, dass wartende Threads durch die virtuelle Maschine von Java scheinbar grundlos aufgeweckt werden [GJSB05]. Um Überraschungen dieser Art zu vermeiden, sollte deshalb die ‘wait’ Methode immer innerhalb einer ‘while’-Schleife platziert werden, welche das ungewollte Erwachen eines Threads behandelt (siehe Abbildung 9.2b).

Dies ist eine Situation, der sich Nutzer der API ggf. nicht bewusst sind, da es sich um einen sehr speziellen und hoch technischen Fall handelt, der eintreten kann, nachdem eine Methode aufgerufen wurde. Ein Nutzer, der diese Vorgabe nicht beachtet und ‘wait’ nicht in einer ‘while’-Schleife aufruft, provoziert Fehler, die sehr schwer zu reproduzieren und zu finden sind. Die im Rahmen der Fallstudie durchgeführte Inspektion der JavaDoc-Dokumentation der vergangenen Versionen der Java API ergab, dass das “Wait-In-A-While”-Problem erst seit Java 1.5 dokumentiert ist (die Java API Dokumentation bis 1.4.2 enthält keinerlei Information über diese Vorgabe).

9.1.2. Vorgaben über die Kommunikation/Parametrierung

Eine weitere Klasse von Vorgaben über die Nutzung einer API betrifft die Kommunikation zwischen Nutzer- und Bibliothekscode durch die Argumente, die Funktionen als Parameter übergeben werden, sowie durch Rückgabewerte von Funktionen. Diese Vorgaben werden in natürlicher Sprache durch Satzgebilde wie *“der Parameter X darf nur die Werte Y und Z annehmen”* oder *“das Ergebnis der Methode X sollte immer überprüft werden”*. Die erste Vorgabe zeigt einen Fall, auf in dem eine Funktion nur partiell definiert ist und nur eine eingeschränkte Untermenge der grundsätzlich möglichen Argumentwerte akzeptiert werden. Die zweite Vorgabe repräsentiert einen Fall, in dem der Rückgabewert einer Methode eine spezielle Bedeutung trägt und immer beachtet werden soll. In den folgenden Absätzen werden drei typische Fälle derartiger Vorgaben vorgestellt:

“Parameter-Is-A-Time-Field-Constant”. Die ‘set(int field, int value)’-Methode der Klasse ‘java.util.Calendar’ benötigt als Argument für den ‘field’-Parameter eine Konstante (z.B. ‘Calendar.MONTH’, ‘Calendar.YEAR’), die das zu ändernde Kalenderfeld spezifiziert (vgl. Abbildung 9.2c). Demnach ist es nicht ausreichend, wenn ein Argument für diesen Parameter vom Typ ‘int’ ist, da die Methode ‘set’ nur zur Verwendung mit einer sehr kleinen Untermenge dieses Typs vorgesehen wurde. Wird keiner dieser Werte spezifiziert, wird ein Laufzeitfehler ausgelöst. Beachtenswert ist hierbei vor allem, dass es sich in diesem Fall nicht nur um eine

Einschränkung des Wertebereichs des Parameters handelt (wie man sie auch häufig in klassischen Vorbedingungen findet). Ein Aufruf von beispielsweise 'set(2, 11)' sollte auch vermieden werden, obwohl der Wert '2' keinen Fehler hervorrufen würde, da auch dies nicht im Sinne des Entwicklers der API ist. Diese Vorgabe ist somit keine reine Einschränkung des Wertebereichs, sondern vielmehr eine Einschränkung der Art des für 'field' einzusetzenden Arguments bezüglich der Java Grammatik. Seitens der Java Grammatik ist jeder Ausdruck (der zum Typ 'int' evaluiert) gültig, also auch jeder Term, der sich aus Additionen, Subtraktionen, Methodenaufrufen etc. gebildet werden kann. Im Kontext der 'set'-Methode sollten allerdings nur Ausdrücke, die lediglich eine Konstante darstellen, eingesetzt werden, um einerseits ein Mehr an Lesbarkeit seitens des Nutzercodes zu gewinnen und andererseits in der weiteren Evolution der API nicht auf die ganzzahligen Kodierungen der Kalenderfelder festgelegt zu sein.

"Parameter-Is-A-ScrollPaneLayout". Wie in Abbildung 9.2d) dargestellt, benötigt die Methode 'setLayout' der Klasse 'JScrollPane' einen Parameter, der ein Objekt der Unterklasse von 'ScrollPaneLayout' darstellt. Andernfalls wird ein Laufzeitfehler ausgelöst. Diese Methode überschreibt die 'setLayout'-Methode der Klasse 'Container'. Diese implizite Vorgabe liegt in der Invarianz der Parametertypen beim Überschreiben einer Methode begründet. Dies ist ein Beispiel eines noch generelleren Falles, in dem nicht jeder Untertyp eines Typs als Argument für einen Methodenparameter erlaubt ist.

"Check-The-Return-Of-RenameTo". Abbildung 9.2e präsentiert die Methode 'renameTo' der Klasse 'java.io.File'. Das Umbenennen von Dateien kann aus vielerlei Günden fehlschlagen, die in der Implementierung des Dateisystems begründet liegen. In der JavaDoc-Dokumentation der 'renameTo'-Methode gibt es aus diesem Grund einen starken Hinweis darauf, dass der Wert, den die Methode zurück gibt, immer überprüft werden soll, da er über den Erfolg oder Misserfolg der Umbenennung Auskunft gibt. Dieses Beispiel ist ein Fall, in dem der Rückgabewert einer Methode eine besondere Bedeutung hat, und deshalb nicht ignoriert werden sollte. Eine Nichterfüllung dieser Vorgabe führt zu latenten Fehlern, die in Fällen auftreten, in denen die Umbenennung fehlschlägt. Die manuelle Inspektion von früheren Versionen der Java API Dokumentation zeigt, dass dieser Kommentar erst seit Java 1.4.2 dokumentiert ist [1]. Dies lässt den Schluss zu, dass vermutlich bereits öfter Probleme dieser Art aufgetreten sind, die schließlich Anlass zur Erweiterung der Dokumentation waren.

9.1.3. Vorgaben über Unterklassen

Die dritte Kategorie von Vorgabe hängt mit der Nutzung von Klassen einer API durch Unterklassenbildung zusammen. Eine derartige Nutzung von Funktionalität ist besonders im Falle von Frameworks sehr häufig anzutreffen. In einer Dokumentation in natürlicher Sprache werden Vorgaben dieser Kategorie durch Phrasen wie *"immer wenn von X abgeleitet wird, dann sollte auch von Y abgeleitet werden"*, *"immer wenn Methode X überschrieben wird, dann sollte auch*

[1] Die Version 1.3.0 der Java API Dokumentation enthält keinen Hinweis auf dieses Problem http://java.sun.com/j2se/1.3/docs/api/java/io/File.html

Y überschrieben werden" oder *"überschreibe Methode X immer derart, dass Y gilt"*. In diesen Fällen liegt das implizite Wissen darin, wie eine Klasse in adäquater Art und Weise erweitert werden sollte. In diesen Situationen haben Erweiterungen nichtlokale Auswirkungen.

"Override-Both-Hashcode-And-Equals". Ein sehr bekanntes Beispiel einer Vorgabe über Unterklassenbildung ist das verpflichtete Überschreiben von Methoden. Um bestimmte Invarianten einer Klasse einzuhalten und um dadurch das Verhalten einer Klasse konsistent zu halten, ist es oftmals nötig, bestimmte Methoden in Unterklassen zu überschreiben. In Abbildung 9.2f wird der bekannte Fall des adäquaten Überschreibens der 'equals' Methode in der Klasse 'Object' illustriert. Immer wenn 'equals' überschrieben wird, sollte auch die Methode 'hashCode' überschrieben werden, um die Gleichheit von Objekten auch durch ihren Hashcode auszudrücken. Wird diese Vorgabe nicht eingehalten, führt dies zu einem inkonsistenten Verhalten der Gleichheitsoperatoren. Dadurch können Fehler entstehen, wenn derartige Objekte in auf Hashing basierenden Mengen (Collections), wie beispielsweise 'java.util.HashSet', eingefügt werden. Die manuelle Inspektion der früheren Java API Dokumentation ergab, dass dieser Kommentar erst ab der Version 1.4.2 aufgenommen wurde.

"Do-Not-Forget-To-Paint" In Abbildung 9.2g ist ein Fall zu sehen, in dem die Unterklassen der Klasse 'Component', welche die Methode 'update' überschreiben, darauf achten müssen, dass die Komponente gezeichnet wird. Ein Verstoß gegen diese Vorgabe führt zu graphischen Komponenten, die nicht richtig dargestellt werden. Auch in diesem Fall ergab die Inspektion der früheren Versionen der Java API Dokumentation, dass dieser Hinweis erst seit Java 1.4.2 existiert.

9.1.4. Vorgaben über den Geltungsbereich/Sichtbarkeit

Die vierte und letzte Klasse von Vorgaben betrifft die grobgranularsten Programmstrukturen, im Falle von Java sind das Pakete. Pakete definieren Namensräume und werden zur Dekomposition eines Systems eingesetzt. Paketstrukturen reflektieren Architekturentscheidungen und sind meist mit impliziten Vorgaben verbunden, welche die Verwendungsbeziehungen unter den Paketen beschränken, um monolithische Gebilde zu vermeiden. In der Dokumentation werden derartige Vorgaben durch Phrasen wie *"nutze <in einem bestimmten Kontext> keine Klassen aus Paket X"*. Auch hinter einer Formulierung wie *"die Methode X sollte nicht von extern aufgerufen werden"* verbergen sich Vorgaben über die Sichtbarkeit. Diese Fälle sind oftmals Instanzen der "public vs. published"-Problematik [Fow02].

"Do-Not-Use-SUN-Packages". Der Java API liegt eine bekannte Konvention zugrunde, die besagt, dass die Pakete, deren Name mit "sun" beginnt, nicht von Java Programmierern verwendet werden sollen, da sie den Implementierungsteil der Java API repräsentieren. Anstelle dieser Pakete sollten lediglich die "java.*"- und "javax.*"-Pakete genutzt werden, da diese die stabil gehaltene Schnittstelle als Fassade vor den 'sun'-Paketen bilden [Mic96]. Diese Konvention impliziert, dass die Java API selbst (die "java.*"- und "javax.*"-Pakete) in ihrer öffentlichen

Schnittstelle keine Typen verwenden darf, die in 'sun'-Paketen definiert sind. Andernfalls sind die Nutzer der Schnittstelle dazu gezwungen, diese Typen aus den 'sun'-Paketen in ihren Programmen zu verwenden, wenn sie gegen diese Schnittstelle programmieren.

"Do-Not-Call-AddNotify". Abbildung 9.2h illustriert ein weiteres Beispiel einer SichtbarkeitsVorgabe, nämlich, dass bestimmte Methoden, auch wenn sie als 'public' deklariert sind, nicht dafür vorgesehen sind, um direkt von Nutzern der API aufgerufen zu werden. Im dargestellten Beispiel handelt es sich um die Methode 'addNotify' in der Klasse 'java.awt.Component'. Werden entgegen der impliziten Vorgabe derartige Methoden dennoch verwendet, kann dies zu Anomalien im Verhalten des Frameworks führen. Die manuelle Inspektion der Historie der Java API Dokumentation hat gezeigt, dass dieser Kommentar bereits sehr früh in der Java API Version 1.1 eingeführt wurde.

9.2. Klassifikation von Constraints

Im Folgenden wird das formale Framework, das in Abschnitt 5.2 vorgestellt wurde, genutzt werden, um Constraints über dem API-Benutzercode zu definieren, die jede der Vorgaben aus dem vorangehenden Abschnitt explizit ausdrückbar und überprüfbar machen. Zu jeder Klasse von Vorgaben korrespondiert eine Familie von Constraints über der Syntax, die in den nachfolgenden Unterabschnitten beschrieben werden. Jeder der Abschnitte hat die folgende Struktur: Zunächst wird die Verbindung zwischen der Klasse von Vorgaben und der zugehörigen Constraint-Familie hergestellt, bevor ein Constraint für jede Vorgabe aus Abschnitt 9.1 angegeben wird. Es wird eine Formalisierung jedes Constraints zusammen mit einer intuitiven Erklärung angegeben und es werden potentielle Alternativen und Einschränkungen diskutiert. Zu jedem Constraint werden Code-Beispiele aus der Java Bibliothek präsentiert, die Verwendungen der Deklarationen darstellen. Dabei wird die Java-Standard-Bibliothek als beides, Ersteller und Nutzer der API, betrachtet. Durch diese Strategie soll klar herausgestellt werden, dass die API selbst von den Personen, die sie entwickelt haben, nicht vollständig verstanden wird. Dies soll die Notwendigkeit untermauern, das implizite Vorgaben einer Bibliothek genau zu spezifizieren und automatisch zu überprüfen. Um zu quantifizieren, wie viele API-Konzepte adäquat verwendet werden und die Constraints erfüllen, wurden die Constraints zusätzlich auf Teile der Eclipse-Implementierung angewandt. Die Ergebnisse wurden automatisch durch das Werkzeug "Java Constraint Checker" gefunden und werden in Abschnitt 9.3.2 vorgestellt.

9.2.1. Kontrollfluss-Constraints

Die Programmiersprache Java akzeptiert grundsätzlich jede Sequenz von Methodenaufrufen. In der Praxis hängt die antizipierte Verwendung von vielen API-Methoden vom Zustand ab, in dem sich das System befindet. Ist das System bei einem Aufruf nicht in dem erwarteten Zustand, treten meist Laufzeitfehler auf. Aufgrund dessen müssen API-Nutzer, bevor sie derartige Methoden aufrufen, sicherstellen, dass sich das System im notwendigen Zustand befindet.

“Pop-Only-Non-Empty-Stacks”. Die “Pop-Only-Non-Empty-Stacks”-Vorgabe kann durch den $PopInIf$-Constraint explizit gemacht werden, wie er bereits in der Formel 5.1 definiert wurde. Wenn der Nutzer gezwungen wird, diesen Constraint einzuhalten, sehen Wartungsingenieuere explizit im aufrufenden Code, dass der Nutzer sich Gedanken über den Fall gemacht hat, in dem der Stack leer ist. Der in Formel 5.1 angegebene Constraint könnte zusätzlich noch ‘while’-Schleifen, die eine entsprechende Bedingung überprüfen, ‘pop’-Aufrufe, die direkt auf ‘push’-Aufrufe folgen oder die Prüfung toleranterer Bedingungen erlauben, wie sie in Abschnitt 6.3 vorgestellt wurden.

“Wait-in-a-While”. Um die “Wait-in-a-While”-Vorgabe explizit zu machen, kann folgender Constraint definiert werden:

$$\langle \mathbf{Decl} : java.lang.Object.wait(), \quad \mathbf{Use} : \underline{MethInv}, \\ \mathbf{Ctxt} : \exists w \in !\underline{WhileStmt} : use \sqsubset w.\underline{StmtBlock} \rangle$$

Intuitiv wird dieser Constraint auf jeden Aufruf der Methode ‘java.lang.Object.wait()’ angewandt, um zu überprüfen, ob sich der Statementblock, in dem sich der Aufruf befindet, innerhalb einer ‘while’-Schleife liegt. Es wird nicht überprüft, ob die ‘wait’-Methode auch korrekt verwendet wird, da die Bedingung der ‘while’-Schleife nicht überprüft wird. Dennoch ist der Constraint geeignet, Stellen zu identifizieren, an denen Programmierer sich nicht der adäquaten Benutzung bewusst waren. Prinzipiell könnte anstelle der ‘while’-Schleife auch eine ‘for’-Schleife oder sogar eine rekursive Funktion verwendet werden, die eine Schleife simuliert. Derartige Strukturen stellen jedoch keine adäquate Verwendung dar und erfüllen somit nicht die Vorgabe. In Abbildung 9.3a sind zwei Beispiele einer adäquaten (links) und einer nicht adäquaten Verwendung (rechts) von ‘wait’ dargestellt. Durch eine manuelle Inspektion, warum die Klasse ‘EventQueue’ keine ‘while’-Schleife nutzt, wurde festgestellt, dass dies bereits ein bekannter Fehler der Java API ist (Bug Nr. 4974934)[2]. Somit konnte durch diesen Constraint tatsächlich ein Fehler gefunden werden.

9.2.2. Datenfluss-Constraints

Die Kommunikation zwischen Modulen eines Systems findet durch eine Form von Datenfluss statt. Auch diese Kommunikation kann bestimmten einschränkenden Vorgaben unterworfen sein, wie sie bereits in Abschnitt 9.1 aufgezeigt wurden. In diesem Abschnitt sollen einige datenflussorientierte Constraints vorgestellt werden, um diese Art der Vorgaben zu adressieren. In Java definieren Methoden eine Menge an Parametern. Für jeden Parameter kann bei einem Aufruf ein beliebiger Term (Expression in der Java-Grammatik) angegeben werden, solange das Typsystem diesen zu einem passenden Typen evaluieren kann. Diese Terme, die als Argumente übergeben werden, können zur Laufzeit grundsätzlich zu jedem Element im Wertebereich des Parametertypen ausgewertet werden. Somit sollten Methoden mit dem kompletten Kreuzprodukt der Wertebereiche der Parameter umgehen können. In der Praxis gibt es jedoch sehr viele

[2] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4974934

```
package javax.swing.text;
public class AbstractDocument { ...
  protected synchronized final void writeLock() { ...
    while ((numReaders > 0) || (currWriter != null)) { ...
        wait();
    } ...  } ... }
```

```
package java.awt;
public class EventQueue { ...
   public static void invokeAndWait(Runnable runnable)  ... {
      ... // no while loop
      lock.wait();
   } ... }  ... }
```

a) Clients of the 'wait' method

```
package java.awt;
import sun.dc.path.PathConsumer;
public class BasicStroke {
   public void feedConsumer(PathConsumer consumer,
           PathIterator pi) { ...
   } ... }
```

e) API Layering

```
void Foo() {
 File f1 = new File("test.txt");
 File f2 = new File("test.txt");
 HashMap h = new HashMap();
 h.put(f1, "s");
 System.out.println(h.get(f2)); // s
 System.out.println(f1.equals(f2)); // true
 Area a1 = new Area();
 Area a2 = new Area();
 HashMap m = new HashMap();
 m.put(a1, "s");
 System.out.println(m.get(a2)); // > null
  System.out.println(a1.equals(a2)); // true
}
```

d) Latent bugs example

```
package javax.swing.plaf.basic;
public class BasicDirectoryModel extends ... { ...
  public boolean renameFile(File oldFile, File newFile) {
    synchronized(fileCache) {
        if (oldFile.renameTo(newFile)) { ... } ...
    }  }  ... }
```

```
package java.util.logging;
public class FileHandler extends ... { ...
  private synchronized void rotate() {
      ...
      f1.renameTo(f2);
      } ... } ... }
```

b) Clients of the 'renameTo' method

```
package java.util;
public class File implements ... { ...
  public boolean equals(Object o) {...}
  public int hashCode() { ... }
  ...
}
```

```
package java.awt.geom;
public class Area implements ... { ...
  public boolean equals(Object o) {...}
  //no hashCode defined
  ...
}
```

c) Extensions of the equals functionality

Abbildung 9.3.: Beispiele von Verwendungen von verschiedenen Teilen der Java API

Methoden, die nicht auf der gesamten Bandbreite der Parametertypen ihrer Signatur definiert sind. Die Verwendung dieser Methoden unterliegt somit einer impliziten Einschränkung, die einem Nutzer nicht zwingend bekannt sein muss.

“Parameter-Is-A-ScrollPaneLayout”. Die “Parameter-Is-A-ScrollPaneLayout”-Vorgabe kann durch den folgenden Constraint explizit gemacht werden:

$$\langle \mathbf{Decl} : JScrollPane.setLayout(LayoutManager) \\ \mathbf{Use} : \underline{MethInv}, \quad \mathbf{Ctxt} : \exists a \in !\underline{Arg} : use.\underline{Arg}[0] = a \wedge \\ JScrollPaneLayout \in resolveType(a) \rangle$$

Intuitiv betrachtet sollten alle Aufrufe der Methode ‘setLayout’ als Argumente Objekte übergeben, deren Typ Unterklassen von ‘ScrollPaneLayout’ sind. Um diese Formel einfach zu halten, wurde eine Hilfsfunktion ‘resolveType’ verwendet, welche die Menge an Typen zurück gibt, mit denen das $\underline{Arg}$ kompatibel ist. Bei diesem Constraint muss man feststellen, dass aufgrund der Polymorphie ein Aufruf der Methode auf einem Objekt vom Typ ‘Container’ stattfinden kann. In diesem Fall ist der angegebene Constraint nicht ausreichend, um die Vorgabe vollständig auszudrücken. Die einfache syntaktische Analyse ist in diesen Fällen immer unvollständig und wird nicht jeden Aufruf finden, der auf Objekten der Super-Typen von ‘JScrollPane’ (z.B. ‘Container’) stattfindet. An dieser Stelle muss man sich bei der Definition eines Constraints entscheiden, ob man eine vollständige Analyse mit False Positives oder eine unvollständige Analyse bevorzugt.

“Parameter-Is-A-Time-Field-Constant”. Die “Parameter-Is-A-Time-Field-Constant”-Vorgabe kann wie folgt durch einen Constraint angegangen werden:

$$\langle \mathbf{Decl} : java.util.Calendar.set(int, int), \quad \mathbf{Use} : \underline{MethInv}, \\ \mathbf{Ctxt} : use.\underline{Arg}\,[0] \in \{Calendar.ERA,\ Calendar.YEAR, ...\} \rangle$$

Dieser Constraint drückt aus, dass das erste Argument von jedem Aufruf der 'set'-Methode der 'Calendar'-Klasse explizit eine der symbolischen Konstanten sein sollte, die in der Klasse 'Calendar' definiert sind (beispielsweise 'Calendar.ERA', 'Calendar.YEAR' etc.). Es gibt viele Fälle, in denen die Methode 'Calendar.set' einen allgemeinen Term als Argument übergeben bekommt. Diese Aufrufe sind dann jedoch nur schwer verständlich und auch bezüglich einer Evolution der API nicht sicher (beispielsweise bei Änderung der Werte für die symbolischen Konstanten). Derartige Nutzungen wären somit nicht adäquat, da es nicht explizit aus der Form der Aufrufe hervorgeht, dass diese die Vorgabe erfüllen. Dennoch ergab die Analyse der Nutzung der 'set'-Methode in Eclipse, dass diese nur in adäquater Art und Weise verwendet wurde (vgl. Tabelle 9.2).

"Check-The-Return-Of-RenameTo". Die "Check-The-Return-Of-RenameTo"-Vorgabe kann durch folgenden Constraint zum Ausdruck gebracht werden:

$$\langle \textbf{Decl} : java.io.File.renameTo(File), \\ \textbf{Use} : \underline{MethInv}, \textbf{Ctxt} : \exists i \in !\underline{IfStmt} : use \sqsubset i.\underline{Expr} \rangle$$

Dieser Constraint versichert, dass die 'renameTo'-Methode immer nur im Kontext der Bedingung eines 'if'-Statements aufgerufen wird. Eine Variation dieses Constraints wäre es, wenn zusätzlich der Rückgabewert in einer lokalen Variable zwischengespeichert werden darf, bevor diese Variable in der 'if'-Bedingung geprüft wird (Refactoring: Extraktion einer lokalen Variable). In Abbildung 9.3b werden zwei Beispiele dargestellt, in denen 'renameTo' in der Java Bibliothek selbst adäquat (links) und nicht adäquat (rechts) verwendet wird.

9.2.3. Vererbungs-Constraints

Vererbung ist eines der wichtigsten Konzepte, die objektorientierte Programmiersprachen bieten. Von einer Oberklasse zu erben impliziert jedoch oftmals, dass die Unterklasse bestimmten invarianten Vorgaben der Oberklasse gerecht werden muss. Gerade in objektorientierten Frameworks trifft man sehr häufig auf derartige Schnittstellen, die über die Implementierung von Unterklassen durch den Nutzer verwendet werden (z.B. Java Applets). Die in diesem Abschnitt vorgestellten Vererbungs-Constraints können dazu verwendet werden, die Vorgaben aus Abschnitt 9.1 auszudrücken und zu prüfen.

"Override-Both-Equals-And-Hashcode". Folgender Constraint überprüft die Einhaltung der "Override-Both-Equals-And-Hashcode"-Vorgabe:

$$\langle \textbf{Decl} : java.lang.Object.equals(Object), \\ \textbf{Use} : \underline{MethOv}, \quad \textbf{Ctxt} : \exists m \in !\underline{MethDecl} : \\ m.\underline{ClsDecl} = use.\underline{ClsDecl} \wedge \\ m.\underline{Name} = \text{``}hashCode\text{''} \wedge \#m.\underline{ParamDecl} = 0 \rangle$$

Dieser Constraint überprüft, ob die Klassen, die 'equals' überschreiben, auch die Methode 'hashCode' überschreiben. Abbildung 9.3c zeigt Beispiele, in denen die 'equals'-Funktionalität in adäquater (die Klasse 'File') und nicht adäquater Weise (die Klasse 'Area') überschrieben wird. In Abbildung 9.3d sind die unerwarteten Folgen illustriert, die im Falle einer nicht adäquaten Erweiterung auftreten können – beispielsweise können die beiden Objekte 'a1' und 'a2', obwohl sie gleich ('equal') sind, nicht als Schlüssel in einer auf Hashing basierenden Datenstruktur (z.B. 'HashMap' Objekten) verwendet werden.

"Do-Not-Forget-To-Paint". Die "Do-Not-Forget-To-Paint"-Vorgabe kann durch folgenden Constraint explizit gemacht werden:

$$\begin{gathered}\langle \textbf{Decl} : java.awt.Component.update(Graphics), \\ \textbf{Use} : \underline{MethOv}, \quad \textbf{Ctxt} : \exists m \in !\underline{MethInv} : \\ m \sqsubset use.\underline{StmtBlock} \wedge m.\underline{ObjName} = \text{“}super\text{”} \wedge \\ m.\underline{Name} = \text{“}update\text{”} \wedge resolveType(m.\underline{Arg}[0]) = \text{“}Graphics\text{”} \\ \wedge \# m.\underline{Arg} = 1 \rangle\end{gathered}$$

Intuitiv gesprochen, muss laut diesem Constraint jede Methode, welche die Methode 'update' der Klasse 'Component' überschreibt, die 'update' Methode der Klasse 'Component' durch Nutzung der 'super'-Referenz aufrufen. Um diesen Constraint (relativ) einfach zu halten, wird in ihm lediglich geprüft, ob ein derartiger Methodenaufruf stattfindet. Es wird darauf verzichtet, zu prüfen, ob dieser auch wiederum adäquat ausgeführt wird (mit den richtigen Argumenten). Die Analyse der Unterklassen von 'Component' in 'java.awt' ergab (nicht überraschend), dass alle die 'update' Methode adäquat überschreiben.

9.2.4. Strukturelle Constraints

Strukturelle Constraints, wie sie in diesem Abschnitt vorgestellt werden sollen, behandeln spezielle Vorgaben, die mit Sichtbarkeiten (Scoping) nicht ausgedrückt werden können. Auch die Beziehungen zwischen Komponenten eines Systems werden oftmals in der Programmiersprache nur unzureichend ausgedrückt. Strukturelle Constraints sind somit stark mit der in Kapitel 10 vorgestellten Architekturanalyse verwandt. Die Architekturanalyse prüft, ob im Code Abhängigkeiten zwischen Komponenten gefunden werden können, die den Architekturvorgaben widersprechen. Diese Architekturvorgaben sind letzendlich auch Constraints auf Paketen, Komponenten oder Namensräumen in der Implementierung.

In der Programmierliteratur ist bekannt, dass es oftmals einen Unterschied, zwischen der 'public' deklarierten Schnittstelle eines Systems und der tatsächlichen öffentlichen ('published') Schnittstelle gibt, die von Nutzern verwendet werden sollte [Fow02]. In vielen Fällen kann das Problem, dem Nutzer diesen Unterschied klar zu machen, durch explizit prüfbare Constraints gelöst werden. Diese Constraints erweitern somit das Sichtbarkeiten-Konzept einer Programmiersprache.

"Do-Not-Use-SUN-Packages". Die "Do-Not-Use-SUN-Packages"-Vorgabe besagt, dass Java-Programme nicht auf die SUN-Pakete zugreifen sollten. Eine direkte Folge daraus ist, dass die offiziellen Java APIs ('java.*'- oder 'javax.*'-Pakete) keine Typen aus den 'sun.*'-Paketen in ihrer öffentlichen Schnittstelle nutzen sollten, da dadurch Programmierer, die diese APIs nutzen, dazu gezwungen werden, Typen aus diesen verbotenen Paketen und damit Abhängigkeiten zu einer nicht gewarteten API in ihre Programme zu integrieren. Diese Bedingung kann durch die Constraints $PubMethParam$ und $PubClsInh$ ausgedrückt werden, die wie folgt definiert sind:

$$PubMethParam = \langle \mathbf{Decl} : java.sun, \quad \mathbf{Use} : \underline{ParaTypeRef}, \\ \mathbf{Ctxt} : use.\underline{MethDecl}.\underline{Visibility} = public \Rightarrow false \rangle$$

$$PubClsInh = \langle \mathbf{Decl} : java.sun, \quad \mathbf{Use} : \underline{ClsInh}, \\ \mathbf{Ctxt} : use.\underline{Visibility} = public \Rightarrow false \rangle$$

Intuitiv bedeuten diese Constraints, dass keine öffentlichen Methoden existieren sollten, die Parameter von einem Typ aus einem der SUN-Pakete deklarieren, und dass keine öffentlichen Klassen existieren dürfen, die von einer Klasse aus einem SUN-Paket abgeleitet sind. Abbildung 9.3e stellt eine Verletzung dieses Constraints in der Java API (Version 1.5) dar. Durch die Inspektion anderer Versionen dieser Klasse wurde festgestellt, dass in der Version 1.4.2 der Java API 'BasicStroke' keine 'feedConsumer'-Methode enthält ist und dass die Version 1.6 diese Methode als 'private' deklariert. Daraus kann man schließen, dass die Referenz auf ein SUN-Paket in der öffentlichen API in der Java Version 1.5 einen Defekt darstellt.

"Do-Not-Call-AddNotify". Die "Do-Not-Call-AddNotify"-Vorgabe kann durch folgenden Constraint explizit gemacht werden:

$$\langle \mathbf{Decl} : java.awt.Component.addNotify, \\ \mathbf{Use} : \underline{MethInv}, \quad \mathbf{Ctxt} : false \rangle$$

Dieser Constraint drückt aus, dass Clients der API, die Methode 'addNotify' niemals aufrufen dürfen. Der Kontext 'false' bedeutet in diesem Constraint, dass es keinen validen Kontext für einen adäquaten Aufruf von 'addNotify' gibt. Dieser Constraint ist sehr einfach, aber dennoch ist er in vielen Situationen nützlich, in denen 'public'-deklarierte Klassen, Methoden und Attribute nicht als 'published' betrachtet werden dürfen. Insbesondere in Frameworks sind derartige Probleme häufig anzutreffen.

9.3. Quantifizierung der Ergebnisse

In diesem Abschnitt werden die Erfahrungen dargestellt, die bei der Durchführung der Überprüfung von Constraints auf der Java API im Rahmen der Fallstudie gesammelt wurden. Darüber hinaus werden einige quantitative Zahlen präsentiert, welche den Bedarf an Constraints und die Anwendbarkeit des Constraint-Frameworks untermauern. Folgende Fragen sind dabei die Leitfragen der Untersuchungen:

- *Frage 1: Wie häufig treten implizite Vorgaben auf?* Durch Inspizierung von Teilen der Java API Dokumentation wurde untersucht, wie viele Vorgaben über die Verwendung der API gefunden werden, die durch Constraints adressiert werden können (siehe Abschnitt 9.3.1).
- *Frage 2: Sind (syntaktische) Constraints zu restriktiv?* Durch die Analyse einiger Nutzer der Teile der Java API, die in Abbildung 9.2 dargestellt sind, wurde gemessen, wie viele der Nutzungen in adäquater Form sind. Wenn sich nur wenige Clients adäquat verhalten, obwohl sie keine Fehler aufweisen, dann sind die Constraints zu restriktiv und ihre praktische Anwendbarkeit wäre nur eingeschränkt sinnvoll möglich (vgl. Abschnitt 9.3.2).

9.3.1. Verbreitung von impliziten Vorgaben

Die allgemeine Programmiererfahrung lässt vermuten, dass die adäquate Verwendung einer API viele Vorgaben enthält, die über Constraints formalisiert und geprüft werden können. Um Frage 1 zu beantworten, wurde nach Vorgaben gesucht, die direkt in der JavaDoc-Dokumentation der Java-Standard-API in natürlicher Sprache formuliert sind. Die Vorgaben wurden dadurch identifiziert, dass in der JavaDoc-Dokumentation nach folgenden Phrasen gesucht wurde: “must always”, “must not”, “must be called”, “should always”, “should not”, “should be called”. Für jeden Treffer wurde die Methode oder Klasse manuell inspiziert und entschieden, ob es sich dabei um eine einschränkende Vorgabe über die Nutzung der Deklaration handelt und ob die Vorgabe in Form eines Constraints als Restriktion der Aufrufsyntax ausgedrückt werden kann. In Tabelle 9.1 werden die Ergebnisse zusammengefasst dargestellt: Die erste Spalte zeigt den Namen des Teils der Java API, der untersucht wurde; die zweite Spalte (Hits) stellt die Anzahl der Stellen dar, an denen oben genannte Phrasen zu finden waren. Die dritte Spalte zeigt die Anzahl der über Constraints ausdrückbaren Vorgaben, die nach der manuellen Inspektion darin identifiziert werden konnten. Auf der rechten Seite der Tabelle ist die Verteilung der Vorgaben über die vier Constraint-Kategorien (KF – Kontrollfluss, DF – Datenfluss, V – Vererbung, S – Struktur) dargestellt. Um nicht nur die Java API zu untersuchen, enthält die Tabelle in der letzten Zeile die Ergebnisse der gleichen Untersuchung auf der Dokumentation einer Eclipse-API JDT.

Man kann feststellen, dass ein Anteil von ca. 30% der untersuchten Stellen der Dokumentation tatsächlich durch Constaints ausdrückbare Vorgaben über die adäquate Nutzung der API enthielten. Die anderen Stellen enthielten oftmals Beschreibungen der Semantik oder der durch die Deklaration implementierten Funktionalität (das implementierte Konzept).

Durch Nutzung der rein auf Phrasen basierten Suche kann man davon ausgehen, dass die Suchergebnisse nicht vollständig alle in der jeweiligen API verborgenen Vorgaben wiedergeben.

Package	Hits	Constraints	KF	DF	V	S
Java Util	23	6	1	0	4	1
Java IO	8	6	0	4	2	0
Java AWT	45	19	5	6	1	7
Eclipse JDT	37	12	4	1	4	3

Tabelle 9.1.: Vorgaben in der Java API

Wie in Abschnitt 9.1 bereits aufgezeigt wurde, ist es oftmals der Fall, dass Vorgaben erst in späteren Versionen in die Dokumentation aufgenommen wurden.

9.3.2. Restriktivität von Constraints

In diesem Abschnitt soll die Frage beantwortet werden, in welchem Maß die Verwendung der API adäquat und damit konform zu den Constraints ist und wie oft die definierten Constraints zu strikt und restriktiv sind. Diese Frage wird durch Nutzung des Werkzeugs Java Constraint Checker beantwortet. Dieses Werkzeug erlaubt es, die Constraints über den API-Deklarationen zu definieren und den Client-Code daraufhin automatisch zu überprüfen, ob die Stellen, an denen die Konzepte verwendet werden, eine Constraint konforme Verwendung darstellen. Im Rahmen dieses Experiments wurden folgende Schritte durchgeführt:

1. Definition der Constraints und Annotation der entsprechenden API Konzepte,
2. Durchführung der Analyse,
3. Inspektion der Ergebnisse, um False Positives zu entdecken.

Die Schritte 1 und 3 wurden dabei manuell durchgeführt, Schritt 2 vollautomatisch durch das Werkzeug. Tabelle 9.2 präsentiert die Ergebnisse des Experiments, bei dem die Verwendung von Teilen der Java API durch Eclipse[3] untersucht wurde. Sie zeigt, wie viele Verwendungen einer bestimmten Deklaration gefunden wurden (Hits) und wie viele davon den Constraint erfüllt haben.

Constraint	Hits	Adäquat	Nicht-adäquat
pop	144	19	125 (109 Wrapper)
wait	15	8	7
Calendar.set	27	27	0
renameTo	20	11	9 (7 in Tests)
update	4	4	0
equals	291	217	74

Tabelle 9.2.: Ergebnisse der Überprüfung der Constraints auf Eclipse

[3] Eclipse Core (org.eclipse.core.*) und JDT (org.eclipse.jdt.*) Pakete

Da keine Aufrufe der Methode 'addNotify' festgestellt wurden, ist dieser Constraint immer erfüllt und deshalb nicht in der Tabelle aufgeführt. Durch die manuelle Inspektion der neun nicht adäquaten Verwendungen von 'renameTo' fiel auf, dass sich sieben dieser Aufrufstellen im Test-Code befanden. Eine Verletzung des Constraints innerhalb von Testcode ist als weniger problematisch zu erachten als im Systemcode, da dadurch im schlimmsten Fall lediglich ein Testdurchlauf fehlschlägt, obwohl das System korrekt funktioniert. Die anderen nicht adäquaten Verwendungen im System können tatsächlich als latente Fehler angesehen werden. Auch die den Constraint verletzenden Nutzungen von 'equals' können als latente Fehler betrachtet werden. Sollten Objekte der betroffenen Klassen in Hashing basierten Strukturen verwendet werden, könnten Laufzeitfehler entstehen. Die Verletzungen des 'wait'-Constraints scheinen nicht sicher bezüglich spontanem Thread-Erwachen zu sein.

Die Beobachtungen bezüglich des 'pop'-Constraints fallen etwas aus dem durch die anderen Constraints gezeichneten Bild. Hier ist zunächst der Großteil der Verwendungen nicht adäquat. Dennoch wurde bei der manuellen Inspektion festgestellt, dass 109 der 125 Constraint-Verletzungen verschiedene Arten von Wrapper-Methoden waren, welche die 'pop'-Funktionalität lediglich kapseln, ohne wirklich neue Funktionalität damit zu implementieren. Diese Wrapper-Methoden unterstellen ihren Nutzern wiederum die Vorgabe, dass sie entweder eine Prüfung durchführen, bevor die Methode aufgerufen wird, oder dass eine andere Methode zwingend zuvor aufgerufen werden muss (oftmals enthielten die Namen der Methoden das Wort "end..." zudem existierte eine korresponierende "begin..."-Methode). In diesen Fällen müsste die Warnung des 'pop'-Constraints unterdrückt und ein entsprechender Constraint an die Wrapper-Methode annotiert werden. Die anderen Verletzungen des 'pop'-Constraints waren in hoch algorithmischen Codestellen (wie beispielsweise Parsern) zu finden. Obwohl diese Stellen wahrscheinlich eine korrekte Nutzung des Stacks darstellen, ist festzustellen, dass sie oftmals sehr schwer verständliche Methoden waren.

Diese Resultate zeigen, dass die untersuchten Deklarationen meist in adäquater Weise genutzt wurden. In Fällen, in denen die Verwendung als nicht adäquat festgestellt wurde, ergab auch die manuelle Inspektion, dass oftmals latente Fehler vorlagen.

Aus den Ergebnissen kann abgelesen werden, dass der Einsatz stärker semantikbehafteter Constraints in der Mehrzahl der Fälle kaum Vorteile gebracht hätte, da die Anzahl an False Positives relativ klein geblieben ist. Weil syntaktische Constraints im Allgemeinen sehr effizient überprüfbar sind, lassen sie sich sehr einfach und leichtgewichtig einsetzen. Da im Gegensatz zu vielen Verifikationsverfahren auf eine Definition komplexer logischer Formeln verzichtet werden kann und man lediglich die Syntax der verwendeten Programmiersprache verstehen muss, um Constraints einsetzen zu können, ist diese Technik relativ einfach in die Praxis überführbar. Es wird allerdings durchaus Situationen geben, in denen ein gewisses Maß an semantischem Wissen erforderlich ist (z.B. 'pop'), um die korrekte Verwendung bestimmter Deklarationen sicherzustellen. Es sollte bereits ein Ziel des Systementwurfs sein, zunächst Constraints insgesamt zu vermeiden und in Fällen, in denen dies nicht möglich ist, die Constraints möglichst einfach (d.h. syntaktisch) zu halten, um eine hohe Verständlichkeit und eine möglichst homogene Verwendung zu erzielen.

Die Anzahl der durch die Constraints aufgezeigten Verletzungen wäre sicherlich deutlich geringer, wenn die Überprüfung nicht erst auf einem fertigen System, sondern bereits begleitend zur

Entwicklung in einer kontinuierlichen Art und Weise eingesetzt worden wäre. Diese frühzeitige und kontinuierliche Art der Verwendung ist dem Reverse Engineering Verfahren vorzuziehen, das im Rahmen der Fallstudie eingesetzt wurde, da es im Allgemeinen mit geringerem Aufwand verbunden ist (vgl. [DPS05]).

10. Anwendungsbereich 2: Implizites Architekturwissen

"Der schlimmste aller Fehler ist, sich keines solchen bewusst zu sein."
- Thomas Carlyle

In diesem Kapitel wird eine empirische Studie präsentiert, die den Verlust an Architekturwissen untersucht, der im Lauf der Softwareevolution in industriellen Softwareentwicklungsprojekten aufgetreten ist. Es wird quantifiziert, inwieweit die Dokumentation und der Code konsistent gehalten werden. Darüber hinaus wird anhand von Interviews mit den Entwicklern evaluiert, ob diese Inkonsistenzen (Unterschiede zwischen Dokumentation und Code) Verletzungen der vorgesehenen Architektur im Code oder Unzulänglichkeiten der Dokumentation sind. Diese Fallstudie wurde auf drei industriell eingesetzten betrieblichen Informationssystemen durchgeführt, die verschiedene Funktionalitäten implementieren und unterschiedlich alt sind. Neben dem empirischen Beitrag der Quantifizierung der impliziten Architekturvorgaben wird in dieser Fallstudie auch gezeigt, wie Modelle und Analysewerkzeuge eingesetzt werden können, um Entwurfsregeln langfristig in Projekten durchzusetzen. Diese Studie wurde bereits in [FRJ09] veröffentlicht.

Im Folgenden werden Vorgaben über die Architektur (statische Struktur des Softwaresystems) als ein wichtiger Teil aller Entwurfsregeln betrachtet, die in einem System zu finden sind. Im Gegensatz zu anderen Arten von Entwurfsregeln (z.B. bezüglich des Kontroll- oder Datenflusses) sind Vorgaben über die Sollarchitektur eines Systems, wenn auch ggf. veraltet, in vielen Projekten dokumentiert. Für andere Arten von impliziten Vorgaben kann eine derartige Studie kaum durchgeführt werden, da dies mit einem enorm hohen Reverse Engineering Aufwand einhergehen würde.

10.1. Motivation

Die Architektur definiert die Struktur eines Softwaresystems anhand von Komponenten und Regeln, wie diese Komponenten voneinander abhängen. Eine geeignete Architektur ist eine fundamentale Voraussetzung für ein evolutionierbares und verständliches System [GS89, Gar00]. Entwickler benötigen bei jeder Modifikation eines Systems Wissen über die vorgesehene Architektur (engl. intended architecture). Ohne dieses Wissen verletzen Programmierer die Integrität der Architektur eines Systems oftmals versehentlich, selbst wenn sie nur kleine Änderungen am Code vornehmen.

Die heutzutage weit verbreiteten Programmiersprachen bieten nur sehr primitive Mechanismen, um die Architektur im Code explizit zu machen (z.B. Sichtbarkeiten). Aus diesem Grund ist in der industriellen Praxis die Information über die Architektur in externer Dokumentation in Form von Diagrammen und natürlich-sprachlichen Beschreibungen enthalten, die oftmals aus frühen Phasen des Entwurfs stammen (konstruktive Dokumentation). Während der Systemevolution muss die Architektur aufgrund von Änderungen in den Anforderungen, zusätzlichen Features oder lediglich durch neue Einsichten über Unzulänglichkeiten des initialen Designs oftmals angepasst werden. Diese Änderungen sind unvermeidlich, selbst wenn eine ‘optimale Designstrategie’ genutzt würde [vGB02]. Dieser Effekt wird in einem industriellen Umfeld nochmals deutlich verstärkt. Wenn diese Änderungen der vorgesehenen Architektur eintreten, werden sie meist nicht in der Architekturdokumentation angepasst und nicht zu allen Mitgliedern des Entwicklungsteams propagiert [OR92]. Dies führt zu Abweichungen zwischen der vorgesehen Architektur des Systems, wie sie unterschiedliche Entwickler verstehen, wie sie explizit in der Dokumentation beschrieben ist und wie sie tatsächlich im Code implementiert ist.

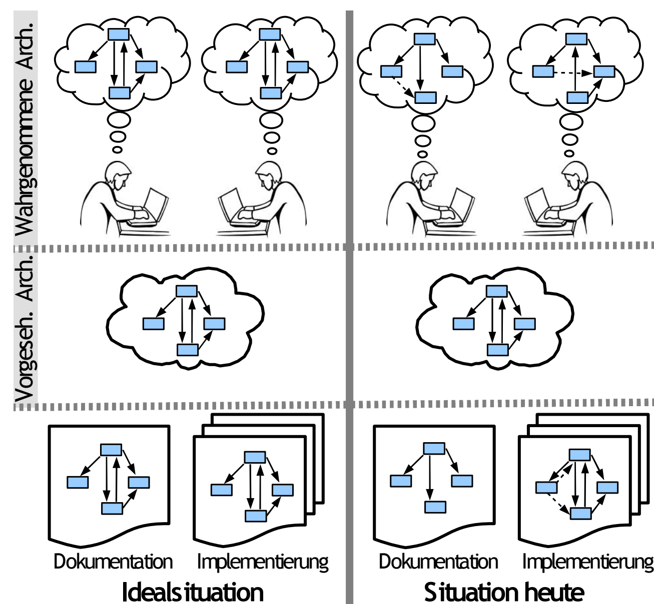


Abbildung 10.1.: Verlust an Architekturwissen

Abbildung 10.1-links illustriert intuitiv die ideale Situation, wenn alle Entwickler akkurates und einheitliches Wissen über die Architektur des Systems besitzen, die Architektur akkurat dokumentiert und akkurat im Code implementiert ist. Auf der rechten Seite der Abbildung ist die Situation veranschaulicht, die typischerweise in industriellen Projekten anzutreffen ist: Verschiedene Entwickler haben (leicht) unterschiedliche Vorstellungen von der Soll-Architektur eines Systems, niemand hat akkurate Information, wie die vorgesehene Architektur (zum aktuellen Zeitpunkt) auszusehen hat. Darüber hinaus ist nur ein Teil der vorgesehen Architektur dokumentiert und nur Teile des Codes sind dazu konform. Wie in Abbildung 10.1-rechts kann der Verlust an Architekturwissen in unterschiedlicher Gestalt beobachtet werden: Fehlende Ar-

chitekturinformation in der Dokumentation, Verletzungen der Architektur im Code, Probleme, den Code und die Dokumentation konform zu halten, sowie unterschiedliche Vorstellungen der Entwickler von der vorgesehenen Architektur. Die mentalen Konzeptmodelle der Entwickler bzgl. der Architektur des Systems sind nicht konform zueinander.

Überblick. In Abschnitt 10.3 wird kurz der Ansatz zur Analyse der Konformität zwischen der dokumentierten Architektur und der Implementierung beschrieben. In Abschnitt 10.4 werden die drei Fallstudien vorgestellt, die in Zusammenarbeit mit der Münchener Rück durchgeführt wurden. In Abschnitt 10.5 werden die Ergebnisse diskutiert und die gewonnenen Erfahrungen vorgestellt. Abschnitt 10.6 beschreibt potentielle Einschränkungen der Aussagekraft und der Übertragbarkeit der empirischen Fallstudien. Abschließend werden in Abschnitt 10.7 verwandte Arbeiten vorgestellt, bevor Abschnitt 10.8 die Ergebnisse zusammenfasst.

10.2. Implizites Architekturwissen

Eine der frühsten Entwurfsentscheidungen ist meist die Festlegung einer Architektur als grobgranulare Struktur eines Softwaresystems in Form von Komponenten und Regeln, wie diese Komponenten voneinander abhängen. Entwickler benötigen bei jeder Modifikation eines Softwaresystems Wissen über die vorgesehene Architektur (intended architecture). Ohne dieses Wissen verletzen Programmierer im Lauf der Evolution die Integrität der Architektur eines Systems (vgl. Kapitel 10). Heutige Programmiersprachen bieten zwar Mechanismen zur Kapselung von Elementen (Packages/Namespaces/Module), die vorgesehenen (und verbotenen) Abhängigkeiten können jedoch meist nicht ausgedrückt werden. Aus diesem Grund werden Mechanismen benötigt, um das in Systemen verborgene implizite Architekturwissen zu konservieren.

Constraints zur Architekturanalyse. Die Komponenten der Architektur werden in objektorientierten Programmiersprachen auf Packages (Java) oder Namensräume (C#) abgebildet. Dabei geht jedoch die Information verloren, aus welchen anderen Komponenten ein Paket oder Namensraum nicht genutzt werden darf. Die Nutzung eines Namensraums a aus einem Namensraum b kann durch folgende Art von Constraints eingeschränkt werden:

$$AllowUsingDep(b) = \langle \mathbf{Decl} : a, \quad \mathbf{Use} : \underline{UsingDecl}, \\ \mathbf{Ctxt} : use.\underline{CompUnit}.\underline{NamespDecl} = b \rangle$$

$$AllowFullQualDep(b) = \langle \mathbf{Decl} : a, \quad \mathbf{Use} : \underline{FullQualName}, \\ \mathbf{Ctxt} : use..\underline{CompUnit}.\underline{NamespDecl} = b \rangle$$

Der Constraint *AllowUsingDep* verbietet die Einbindung des annotierten Namensraums a in eine ‘CompilationUnit’ (.cs Datei) nur, wenn diese den Namensraum b deklariert. Da in

C# auch auf Namensräume zugegriffen werden kann, die nicht über eine 'using'-Anweisung eingebunden wurden, muss zusätzlich die Nutzung durch Angabe vollqualifizierter Namen eingeschränkt werden. Dies kann durch den Constraint $AllowFullQualDep$ erreicht werden, der fordert, dass, wenn der Namensraum a in einem vollqualifizierten Namen auftritt, diese Nutzung innerhalb einer 'CompilationUnit' stattfinden muss, die den Namensraum b für sich deklariert ('..' wird als abkürzende Schreibweise für die beliebig lange Navigation zum Wurzelelement des AST, der 'CompilationUnit', genutzt). Eine erlaubte Abhängigkeit des Namensraums b von a bedeutet demnach, dass für a der zusammengesetzte Constraint $AllowDep(b) = AllowUsingDep(b) \wedge AllowFullQualDep(b)$ gelten muss. Ist der Zugriff auf a von mehreren Namensräumen möglich, so können mehrere unterschiedlich parametrisierte $AllowDep$-Constraints verodert werden, um die Architektur zu beschreiben.

Diese Constraints stellen die adäquate Nutzung von Namensräumen in Systemen sicher. Zur Beantwortung der Forschungsfragen der in Kapitel 10 präsentierten Fallstudie war es notwendig, darüber hinaus Messungen durchzuführen, die den Grad der Abweichungen miteinbeziehen. Ein Constraint würde bereits die Integration eines Namensraums durch das Schlüsselwort 'using' mit einer Warnung versehen. Im Rahmen der Fallstudie wird jedoch die tatsächlich entstandene Kopplung zwischen Namensräumen erhoben, also die Nutzungen, die hinter einer 'using'-Deklaration verborgen sind.

10.3. Technik und Methodik der Architekturanalyse

In diesem Abschnitt wird der Ansatz vorgestellt, um die Architektur in maschinenlesbarer Form zu beschreiben und die Technik, die Konformität zwischen der dokumentierten Architektur und dem Code zu analysieren. Der Ansatz wird beispielhaft auf Basis der Programmiersprache C# erklärt, obwohl dieser auch auf andere Sprachen generalisierbar ist.

10.3.1. Architekturkonformitätsanalyse

Architekturbeschreibung. Um eine automatische Analyse durchführen zu können, wird die Architektur als eine Menge von hierarchischen Komponenten $comp$ beschrieben. Zusätzlich existiert eine Menge an Regeln pol (policies) zwischen diesen Komponenten. Somit kann eine Architekturbeschreibung $arch$ wie folgt formalisiert werden:

$$arch = (comp, pol)$$

Komponenten sind die strukturierenden Entitäten in diesem Beschreibungsformalismus. Die Hierarchie wird als Prädikat ausgedrückt:

$$isSubComp : comp \times comp \rightarrow bool$$

Auch Regeln können als Prädikat verstanden werden:

$$pol : comp \times comp \rightarrow bool$$

Einer Komponente ist es immer erlaubt, auf seine Unterkomponenten zuzugreifen:

$$isSubComp(c_1, c_2) \Rightarrow pol(c_1, c_2)$$

Falls eine Regel zwischen zwei Komponenten definiert ist, so spezifiziert dies, dass zwischen den Elementen in der Implementierung, die zu diesen Komponenten korrespondieren, Abhängigkeiten existieren dürfen (in der angegebenen Richtung). Jede Abhängigkeit, die nicht explizit durch eine Regel erlaubt wurde, ist verboten. In vielen Fällen beschreibt die Architektur eine Struktur des Systems, die sicherstellt, dass bestimmte Komponenten entkoppelt sind. Wenn keine Regel zwei Komponenten explizit erlaubt, aneinander gekoppelt zu sein, ist keine Abhängigkeit auf Codeebene erlaubt.

Beschreibung des Codes. In objektorientierten Systemen ist jedes Element des Codes in Typdeklarationen gekapselt (in .NET sind diese Typen definiert in Form von Klassen, Structs, Enums etc.). Zum Zweck der Architektur-Konformitätsanalyse wird ein System

$$sys = (types, dep)$$

als eine Menge von Typen *types* und Abhängigkeiten *dep* zwischen ihnen aufgefasst. Die Anzahl der *Abhängigkeiten* wird durch folgende Funktion ausgedrückt:

$$dep : types \times types \rightarrow \mathbb{N}$$

Ein Typ t_1 ist von einem anderen Typ t_2 abhängig, wenn t_2 (oder eines seiner Elemente) in t_1, wie in Tabelle 10.1, definiert genutzt wird.

Aufruf einer Methode/ eines Konstruktors
Zugriff auf ein Property oder Feld
Erweiterung einer Klasse/eines Structs, Implementierung eines Interfaces
Nutzung von Klassen/Structs/Enums als Typ (für ein Feld, Variable oder Parameter)
Annotation eines Attributs

Tabelle 10.1.: Abhängigkeiten im Code

Überprüfung der Konformität. Um den Architekturbeschreibungsmechanismus auf ein System abzubilden, muss ein *Code Mapping* als eine Funktion $map : types \to comp$ definiert werden. Wenn die Architekturbeschreibung komplett konform mit der Implementierung ist (keine nicht explizit erlaubten Abhängigkeiten), muss folgende Bedingung erfüllt sein:

$$map(t_1) = c_1 \wedge map(t_2) = c_2 \wedge \neg pol(c_1, c_2) \Rightarrow dep(t_1, t_2) = 0$$

Im Weiteren wird die Formulierung ‘es gibt x Abweichungen von Komponente c_1 zu c_2’ genutzt werden, um zu bezeichnen, dass diese Bedingung nicht erfüllt ist, obwohl die linke Seite wahr ist aber die rechte Seite mit $dep(t_1, t_2) = x$ und $x > 0$ zu falsch evaluiert.

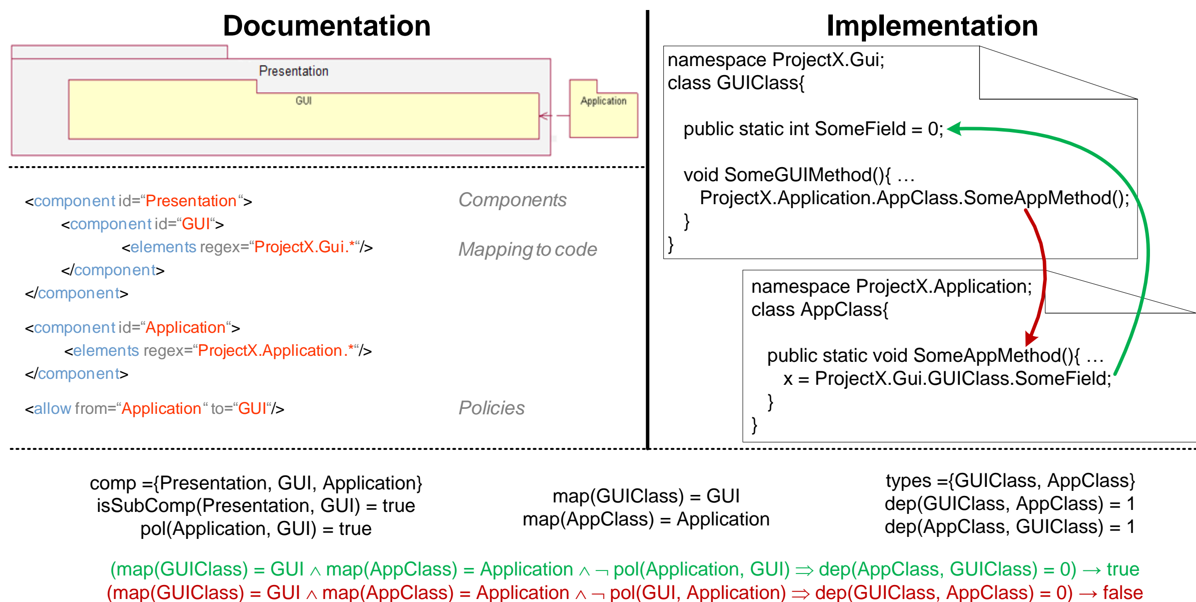


Abbildung 10.2.: Der Mechanismus zur Beschreibung der Architektur

Technische Ausführung der Analyse. Die Architektur wird anhand einer XML-Datei in einer maschinenlesbaren Form beschrieben. Abbildung 10.2-links zeigt eine Beispielarchitektur: Sie enthält drei Komponenten (Presentation, GUI und Application), GUI ist eine Subkomponente von Presentation und es ist eine Regel definiert, die der Komponente Application erlaubt, von GUI abhängig zu sein. Die Abbildung zeigt zudem, wie dieses Beispiel in XML beschrieben wird. Die XML-Datei enthält eine einfache Beschreibung hierarchischer Komponenten (die Hierarchie entspricht der Schachtelung der ‘component’-Tags) und deren Abbildung auf die Typen im Quellcode anhand von regulären Ausdrücken. Diese Ausdrücke werden verwendet, um die voll qualifizierten Namen der Typen in der Implementierung auf die Komponenten abzubilden. Zusätzlich sind ‘allow’-Tags definiert, um die Regeln zu beschrieben, wie die Komponenten voneinander abhängen dürfen. Die rechte Seite von Abbildung 10.2 illustriert die Implementierungsebene: Ein grüner Pfeil repräsentiert eine erlaubte Abhängigkeit der ‘Application’- zur

'GUI'-Komponente. Durch einen roten Pfeil wird eine Abhängigkeit dargestellt, welche die Spezifikation auf der linken Seite verletzt, da ein Zugriff auf die 'Application'-Komponente aus der 'GUI'-Komponente verboten ist. Eine Formalisierung der dargestellten Situation ist im unteren Teil der Abbildung dargestellt.

Alle Projekte, die im Rahmen der Fallstudie analysiert wurden, sind auf Basis des .NET Framework implementiert. Die Analysen wurden mit dem **Con**tinuous **Q**uality **A**ssessment **T**oolkit (ConQAT)[1] [DJH+08] durchgeführt. Die Analyse berechnet die Menge der Abhängigkeiten, die nicht explizit durch die Architekturbeschreibung erlaubt sind.

10.3.2. Methodisches Vorgehen

Analyseschritte. Abbildung 10.3 illustriert die Schritte, die notwendig sind, um eine Architekturanalyse auf einem Projekt durchzuführen:

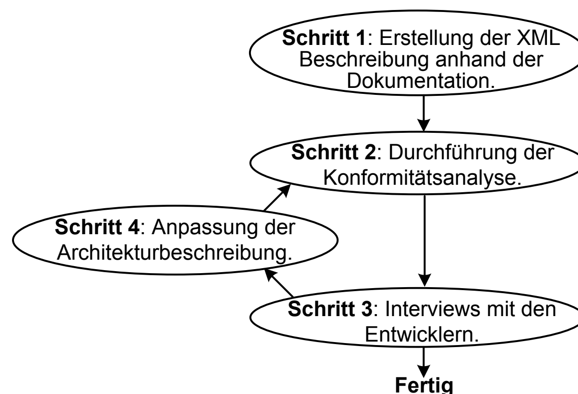


Abbildung 10.3.: Analyseschritte

Schritt 1: Transformation der Architekturdokumentation in die maschinenlesbare XML-Repräsentation. In diesem Schritt wird eine XML-Beschreibung der Architektur erstellt, die der originalen Dokumentation entspricht.

Schritt 2: Überprüfung der Konformität des Codes bezüglich der aktuellen Architekturbeschreibung in XML. In diesem Schritt wird die automatische Architekturanalyse genutzt, um eine Liste an Inkonsistenzen zu identifizieren. Diese Inkonsistenzen liegen entweder in einer unzulänglichen Beschreibung der vorgesehenen Architektur in der XML-Datei begründet oder sie sind Verletzungen der vorgesehenen Architektur im Code.

Schritt 3: Diskussion der Analyseergebnisse mit dem Entwicklerteam. Eine Diskussion der Ergebnisse aus Schritt 2 mit den Entwicklern dient der Klassifizierung der Abweichungen in Verletzungen der vorgesehenen Architektur auf der Codeebene oder Defizite der aktuellen XML-Architekturbeschreibung. Diese Klassifizierung nutzt das implizite Wissen über das

[1] www.conqat.org

System, das (wenn überhaupt) nur in den Köpfen der Entwickler zu finden ist. Falls in den Abweichungen, die durch die automatische Analyse identifiziert wurden, noch Abhängigkeiten existieren, die keine Verletzungen der vorgesehenen Architektur darstellen, dann repräsentiert die XML-Beschreibung noch nicht die vorgesehene Architektur. In diesem Fall muss die Architekturbeschreibung in Schritt 4 angepasst werden. Sobald die Entwickler alle Abweichungen als Defizite im Code ansehen, ist der Prozess beendet.

Schritt 4: Verfeinerung der Architekturbeschreibung durch Integration des impliziten Wissens, das in der Dokumentation nicht vorhanden und in Schritt 3 zu Tage getreten war. Anschließend muss Schritt 2 erneut ausgeführt werden.

Nach jeder Iteration konvergiert die in der XML-Datei beschriebene Architektur sukzessive zur vorgesehenen Architektur. Alle Modifikationen an der XML-Beschreibung, die im Lauf der Iterationen notwendig waren (Schritt 4), werden als Unzulänglichkeiten der ursprünglichen Dokumentation angesehen, die aufgrund von Änderungen der Architektur in der Evolution des Systems eingetreten sind, aber nicht dokumentiert wurden (*architectural drift* [PW92]). Nach zwei bis vier Iterationen der Schritte 2, 3 und 4 wurde die Architekturbeschreibung durch die Teammitglieder als präzise Beschreibung der vorgesehenen Architektur angesehen. Durch Nutzung dieser finalen XML-Architekturbeschreibung, welche die vorgesehene Architektur repräsentiert, konnte eine abschließende Konformitätsanalyse durchgeführt werden, um die Anzahl der Verletzungen der vorgesehenen Architektur im Code zu finden (dies entspricht dem sogenannten *architectural decay* [PW92]).

Ausgaben der Analyse. Während des gerade beschriebenen Prozesses werden folgende Mengen berechnet:

- *Missing*: Die Menge an Komponenten, die im System implementiert sind, in der Dokumentation jedoch nicht erwähnt werden.

- *Relocated*: Die Menge an Komponenten, die im Gegensatz zur ursprünglichen Dokumentation verlagert wurden. Eine Komponente x wird als 'verlagerte Komponente' bezeichnet, wenn $isSubComp(x, a)$ in der Dokumentation beschrieben ist, jedoch $isSubComp(x, b)$ die vorgesehene Architektur reflektiert ($a \neq b$).

- *Policies*: Die Menge der Regeln, die im Lauf des Prozesses hinzugefügt oder verändert wurden (in Schritt 4).

- dep_{all}: Die Menge der Abhängigkeiten zwischen den Komponenten des Systems:

$$|dep_{all}| = \sum_{t_1 \in types} \sum_{t_2 \in types} dep(t_1, t_2) \ , \qquad t_1 \neq t_2$$

- $diff_{doc}$: Die Untermenge der Abhängigkeiten ($diff_{doc} \subset dep_{all}$), die Unterschiede zwischen der originalen Dokumentation der Architektur und der Implementierung darstellen. Diese Menge wird nach der ersten Ausführung von Schritt 2 ermittelt. Diese Abweichungen reflektieren die Divergenz zwischen der dokumentierten und der implementierten Architektur.

- diff_{intend}: Die Untermenge von diff_{doc}, welche die Architekturanalyse nach Vollendung des gesamten Prozesses ermittelt. Die Architekturbeschreibung, die nach den Iterationen vorliegt, entspricht der vorgesehenen Architektur. Aus diesem Grund sind alle Abhängigkeiten in diff_{intend} Architekturverletzungen im Code.

10.4. Die Fallstudie

Dieser Abschnitt beginnt mit der Vorstellung der Forschungsfragen, die durch diese Fallstudie adressiert werden. Anschließend wird die Umgebung der Experimente beschrieben, bevor die quantitativen Ergebnisse vorgestellt werden. Abschließend werden die gemessenen Zahlen interpretiert.

10.4.1. Forschungsfragen

Frage 1: Zu welchem Grad wird die Architekturdokumentation während der Systemevolution mit der Implementierung konsistent gehalten? Der Einstiegspunkt der Analysen behandelt das Verhältnis zwischen der dokumentierten und der implementierten Architektur. Falls Abweichungen identifiziert werden können, deutet dies darauf hin, dass entweder die Implementierung die vorgesehene Architektur verletzt oder dass Änderungen der Architektur während der Systemevolution nicht in die Dokumentation übernommen wurden. Diese Frage kann anhand der Anzahl an Abweichungen an der Gesamtzahl der Abhängigkeiten im Code beantwortet werden:

$$\mathit{docdiff} = \frac{|\mathit{diff}_{doc}|}{|\mathit{dep}_{all}|}.$$

Frage 2: Wie gut reflektiert die Architekturdokumentation die vorgesehene Architektur? Eine ungenaue oder veraltete Dokumentation der Architektur ist nicht geeignet, um Architekturwissen für die Softwarewartung und -evolution zu konservieren. Die Qualität der Dokumentation entspricht dem im Projekt in expliziter Form vorhandenem Wissen, das jedem Teammitglied zugänglich ist. Falls keine präzise und aktuelle Dokumentation der Architektur existiert, werden neue Teammitglieder Schwierigkeiten haben, die Architektur zu lernen. Der Grad an implizitem Wissen in Form von Unzulänglichkeiten der Dokumentation kann über folgende Metrik quantifiziert werden:

$$\mathit{flaw}_{doc} = \frac{|\mathit{diff}_{doc}| - |\mathit{diff}_{intend}|}{|\mathit{diff}_{doc}|}.$$

Zusätzlich stellt die Anzahl an Komponenten, die nicht dokumentiert waren $|\mathit{Missing}|$ oder verlagert wurden $|\mathit{Relocated}|$, sowie die Regeln, die verändert oder hinzugefügt wurden $|\mathit{Policies}|$, Metriken für die Divergenz zwischen dokumentierter und vorgesehener Architektur dar.

Frage 3: Wie groß ist der Verfall der Architektur im System? Diese Frage soll klären, zu welchem Grad Verletzungen der vorgesehenen Architektur im Code gefunden werden können. Der Architekturverfall kann in Form von Architekturverletzungen gemessen werden:

$$flaw_{impl} = \frac{|diff_{intend}|}{|diff_{doc}|}$$

Frage 4: Was sind die Gründe für Abweichungen zwischen der vorgesehenen Architektur und dem Code? Durch diese Frage sollen die Gründe für die Einführung von Architekturverletzungen im Code sowie für die Veraltung der Dokumentation herausgefunden werden, um Möglichkeiten der Prävention identifizieren zu können. Dies ist eine qualitative Frage, die anhand der Interviews mit den Entwicklern beantwortet werden muss (Schritt 3).

10.4.2. Experimentieller Aufbau

Industrielles Umfeld. Die Fallstudie wurde im Rahmen einer Kollaboration zwischen der Münchener Rück und der Technischen Universität München durchgeführt. Münchener Rück ist ein großes Rückversicherungsunternehmen mit ca. 39.000 Mitarbeitern weltweit. Für die Versicherungsbranche ist es typisch, großen Gebrauch von individuell entwickelter Software zu machen, um Geschäftsprozesse wie den Verkauf, Risikokalkulation oder Kapitalinvestitionsrechnungen zu unterstützen. Bei der Münchener Rück wird die Softwareentwicklung hauptsächlich durch externe Entwickler durchgeführt. Dies führt zu einer relativ hohen Entwicklerfluktuation. Ein speziell zugeschnittener RUP-ähnlicher Prozess muss bei allen Softwareprojekten angewandt werden. Dieser Prozess schreibt in jedem Projekt die Erstellung einer Architekturdokumentation verpflichtend vor. Um eine möglichst reibungslose Übergabe zwischen Entwicklern zu erreichen und damit die Folgen der Entwicklerfluktuation gering zu halten, müssen die Softwareprodukte eine möglichst hohe Wartbarkeit aufweisen. Folglich müssen die Systeme in einer möglichst leicht verständlichen und homogenen Art und Weise implementiert werden. Eine Voraussetzung hierfür ist, dass das Architekturwissen im Projekt erhalten wird, indem es explizit gemacht wird.

Die Projekte. Die untersuchten Projekte sind drei typische in C# implementierte betriebliche Informationssysteme. Diese drei Projekte werden seitens der Münchener Rück als erfolgreich und professionell durchgeführt angesehen, die Systeme haben den Ruf von guter Qualität zu sein. Sie sind im produktiven Gebrauch von 10 bis 150 Nutzern in unterschiedlichen Fachabteilungen der Münchener Rück. Die drei Systeme wurden jeweils von unterschiedlichen Entwicklern entwickelt, die zudem bei unterschiedlichen Individualsoftwareherstellern angestellt sind. Während der Initialentwicklung waren bis zu 12 Entwickler pro Projekt involviert. Nachdem die Systeme in den Produktivbetrieb gingen, wurde die Anzahl an Entwicklern reduziert. Alle drei Systeme werden immer noch weiterentwickelt und erweitert.

Projekt A ist eine typische Rich-Client Anwendung, die weiterentwickelt und gewartet wird. Das System wird seit ungefähr fünf Jahren entwickelt. Es bietet Funktionalität zur Risikoanalyse. Derzeit arbeiten fünf Entwickler daran, das System weiterzuentwickeln und zu warten. Im

Projekt gab es bereits personelle Fluktuationen, keiner der derzeitigen Entwickler war in die Initialentwicklung involviert. Die Architekturdokumentation von Projekt A ist ein Textdokument, das zusätzlich Komponentendiagramme enthält. Die Komponenten werden als geschachtelte Rechtecke veranschaulicht, die durch Pfeile verbunden sind. Diese Pfeile symbolisieren erlaubte Abhängigkeiten.

Projekt B ist ein webbasiertes Informationssystem. Mit einer sechsjährigen Entwicklungszeit ist dies das älteste der untersuchten Projekte. Das System wird verwendet, um Finanzierungs- und Investierungsrechnungen durchzuführen. Derzeit entwickeln vier Entwickler das System weiter. Wie in Projekt A gab es auch bereits personelle Fluktuationen, so dass derzeit ebenfalls keiner der Initalentwickler am Projekt beteiligt ist. In Projekt B ist die Architektur vollständig durch Diagramme beschrieben, die UML Paket-Diagrammen sehr ähnlich sind.

Projekt C ist auch ein webbasiertes System. Es ist das jüngste der analysierten Systeme und seit ca. zwei Jahren in der Entwicklung. Es bietet Funktionalität, um Risikoinformationen von Kunden der Münchener Rück zu verarbeiten. Durchschnittlich entwickeln drei Entwickler das System weiter. Das System ist erst seit ca. sieben Monaten im produktiven Einsatz. Die Architekturdokumentation von Projekt C ist auch ein Textdokument, das Diagramme enthält, in denen die Architektur als Rechtecke und Pfeile illustriert wird, die Komponenten und ihre Abhängigkeiten beschreiben.

	Alter	**kLOC**	**Entwickler**
Projekt A	5	454	5
Projekt B	6	317	4
Projekt C	2	495	3

Tabelle 10.2.: Rahmendaten der Projekte

Tabelle 10.2 präsentiert die Projekte auf einen Blick: Ihr Alter, ihre Codegröße und die derzeitige Anzahl an Entwicklern, welche die Systeme warten und weiterentwickeln. Die Code-Größe der Projekte ist etwa in der gleichen Größenordnung. Dennoch ist Projekt C größer als die anderen, obwohl es die kürzeste Entwicklungszeit aufweist und derzeit mit der geringsten Anzahl an Entwicklern weiterentwickelt wird.

Tabelle 10.3 zeigt die Größe der dokumentierten Architektur hinsichtlich ihrer Anzahl an Komponenten und der Anzahl an Regeln (erlaubte Zugriffe über Komponentengrenzen hinweg). Bemerkenswert ist die hohe Anzahl an Regeln im Fall von Projekt A und C (sie enthalten nur 24 bzw. 37 Komponenten und erlauben viele Abhängigkeiten). Im Gegensatz dazu enthält Projekt B deutlich mehr Komponenten (60) aber nur etwa die gleiche Anzahl an Regeln (106) wie Projekt C. Somit ist die Beschreibung von Projekt B deutlich feingranularer und die Architektur restriktiver.

	Komponenten	**Regeln**
Projekt A	24	79
Projekt B	60	106
Projekt C	37	102

Tabelle 10.3.: Initiale Beschreibung der Architektur

10.4.3. Quantitative Ergebnisse

Tabelle 10.4 zeigt die Anzahl an Modifikationen der Architekturbeschreibung während der iterativen Verfeinerung der Architektur. Die erste Spalte präsentiert die Anzahl verlagerter Komponenten *Relocated*. Eine derartige Verlagerung ist beispielsweise das Verschieben einer Unterkomponente der Business-Komponente zur DataAccess-Komponente. Die zweite Spalte zeigt die Anzahl an Komponenten, die nicht dokumentiert waren und im Lauf des Analyseprozesses (Schritt 4) zur Architekturbeschreibung hinzugefügt wurden. Die dritte Spalte enthält die Anzahl der Regeln, die modifiziert oder zu den ursprünglich dokumentierten hinzugefügt wurden. Keine Regeln wurden entfernt, ohne sie zu ersetzen. Das zeigt, dass die dokumentierte Architektur meist weniger Regeln (die Abhängigkeiten erlauben) beinhaltet als die implementierte Architektur. In anderen Worten scheint die dokumentierte Architektur modularer als die wirklich implementierte.

	Relocated	***Missing***	***Policies***
Projekt A	0	2	8
Projekt B	6	2	24
Projekt C	2	1	9

Tabelle 10.4.: Änderungen an der Architekturbeschreibung während der Schritte 3 und 4

Tabelle 10.5 präsentiert die Hauptergebnisse der Analysen. Die dep_{all}-Spalte zeigt die Anzahl der Abhängigkeiten, welche die Architekturkonformitätsanalyse in den Systemen identifiziert hat. Diese Abhängigkeiten wurden schließlich gegen die XML-Architekturbeschreibung validiert. Die $diff_{doc}$-Spalte zeigt die Ergebnisse der ersten Konformitätsanalyse, nämlich die Anzahl der Abhängigkeiten, die nicht konform zur dokumentierten Architektur waren. Die $flaw_{doc}$- und $flaw_{impl}$-Spalten enthalten den prozentualen Anteil der nichtkonformen Abhängigkeiten, die Unzulänglichkeiten der Dokumentation bzw. Verletzungen der Architektur in der Implementierung darstellen.

System A. In System A wurden eine Untermenge von 994 (aus 8.254) Abhängigkeiten als Abweichungen zwischen der dokumentierten und der implementierten Architektur identifiziert (12%). Wie Tabelle 10.4 zeigt, mussten acht Regeln und zwei Komponenten hinzugefügt werden, um von der dokumentierten zur vorgesehenen Architektur zu gelangen. Am Ende dieses

	$\boldsymbol{dep}_{all}$	$\boldsymbol{diff}_{doc}$	$\boldsymbol{flaw}_{doc}$	$\boldsymbol{flaw}_{impl}$
Projekt A	8.254	994 (12%)	90%	10%
Projekt B	4.385	376 (9%)	72%	28%
Projekt C	5.388	(2238) 1039 (19%)	88%	12%

Tabelle 10.5.: Ergebnisse der Analysen

Analyseprozesses wurden ca. 10% der Abweichungen als Verletzungen der vorgesehenen Architektur in der Implementierung des Systems erkannt. Die anderen Abweichungen waren meist undokumentierte Änderungen der Architektur während der Evolution des Systems.

Abbildung 10.4 zeigt eine Visualisierung der Ergebnisse der Analyse von System A. Diese Art der Visualisierung wurde unter Zuhilfenahme des Graphvisualisierungswerkzeugs 'dot' [KN93] erstellt. Sie zeigt die hierachischen Komponenten als Rechtecke, Verletzungen der Architekturbeschreibung als rote Pfeile und erlaubte Abhängigkeiten durch grüne Pfeile. Diese Visualisierung der Analyseergebnisse wurde zusammen mit detaillierten Listen über die identifizierten Abweichungen in den webbasierten Projektleitstand integriert, um als Grundlage für Diskussionen zu dienen.

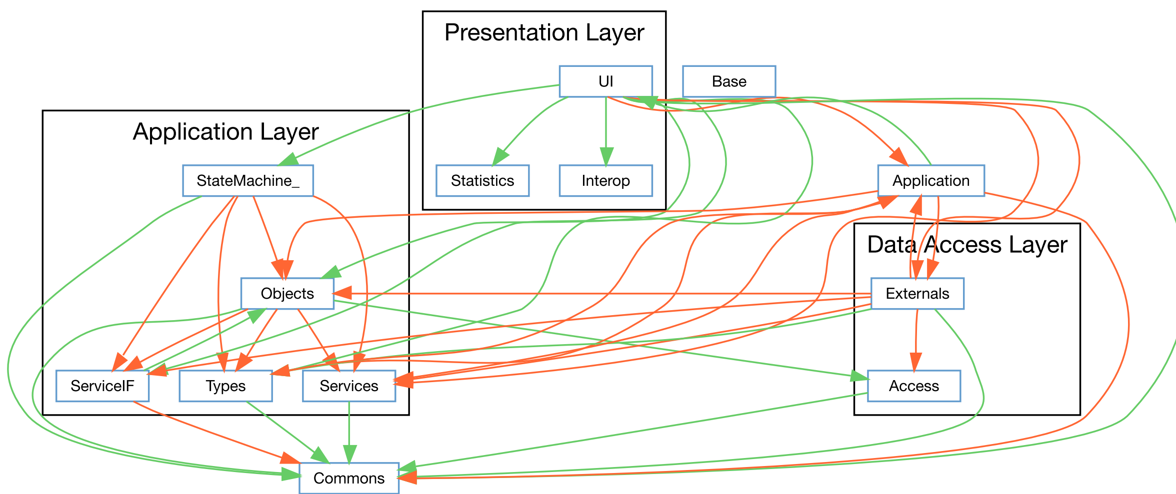


Abbildung 10.4.: Ein Beispiel einer Visualisierung der Analyseergebnisse (Projekt A)

System B. Die Ergebnisse der Analyse von Projekt B zeigten eine niedrigere Anzahl an Abweichungen zwischen der dokumentierten und der implementierten Architektur (376 von 4.385). Als Folge der Diskussionen mit den Entwicklern waren deutlich mehr Anpassungen der Architekturbeschreibung notwendig (vgl. Tabelle 10.4), um eine präzise Beschreibung der vorgesehenen Architektur zu erhalten. Die Architekturbeschreibung dieses Projekts war bereits in der Dokumentation deutlich detaillierter und feingranularer beschrieben (Tabelle 10.3). Schließlich wurde ein relativ hoher Anteil an Architekturverletzungen (28%) der 376 Abweichungen

gefunden.

System C. Nach der ersten Analyse von System C auf Basis der initialen Architekturbeschreibung wurde eine sehr große Anzahl an Abweichungen gemessen (2.238 – entspricht effektiv 42% aller Abhängigkeiten im System). Nach genauerer Betrachtung der Abweichungen stellte sich heraus, dass dieses extreme Ergebnis hauptsächlich durch einen Grund hervorgerufen wurde: Das System wurde durch Nutzung sogenannter "Datenbindung" (data bindung) erstellt, die seitens des .NET Frameworks angeboten wird. Diese Technik verursacht Abhängigkeiten, die direkt von GUI-Elementen zu Datenzugriffskomponenten reichen. Laut der Architekturspezifikation sind diese Abhängigkeiten nicht erlaubt. Obwohl diese Abhängigkeiten Abweichungen zwischen der Dokumentation und der Implementierung darstellten, wurden sie im Gegensatz zu gewöhnlichen Architekturverletzungen nicht aufgrund von mangelndem Architekturwissen eines Entwicklers in das System eingebaut. Aus diesem Grund wurde die XML-Architekturbeschreibung dementsprechend angepasst, indem eine neue Regel hinzugefügt wurde, um diese Abweichungen zu erlauben. Nach dieser Anpassung wurde ein deutlich niedrigerer Wert von 1039 Abweichungen (ca. 19%) gemessen. Dieser Wert wurde der Berechnung weiterer darauf basierender Kennzahlen zugrunde gelegt. Somit konnten schließlich 12% dieser Abweichungen als Verletzungen der Architektur entdeckt werden. Dieser Wert ist sehr nahe an dem in Projekt A gemessenen Ergebnis. Auch die Größenordnung der Anpassungen der dokumentierten Architektur war sehr ähnlich zu Projekt A (vgl. Tabelle 10.4).

10.4.4. Beantwortung der Forschungsfragen und Interpretation der Ergebnisse

Frage 1: *Zu welchem Grad wird die Architekturdokumentation während der Systemevolution mit der Implementierung konsistent gehalten?* Durch die automatische Analyse der Konformität der Architektur konnte in allen drei Projekten eine erhebliche Anzahl an Abweichungen zwischen der dokumentierten Architektur und der Implementierung gefunden werden. Wie Tabelle 10.5 zeigt, konnten zwischen 9% und 19% aller Abhängigkeiten, die im System implementiert sind, nicht in der korrespondierenden Spezifikation identifiziert werden. Diese Abweichungen repräsentieren entweder Dokumentationsdefizite oder Architekturverletzungen im Code. Die *docdiff*-Werte der Projekte zeigen, dass Projekt B die geringste Anzahl an Abweichungen vorweisen kann. Aber selbst in diesem Fall ist die Menge der Diskrepanzen erheblich (jede zehnte Abhängigkeit im System kann nicht in der Dokumentation nachvollzogen werden). Im schlechtesten Fall (Projekt C) ist fast jede fünfte Abhängigkeit nichtkonform zur Dokumentation. Diese erhebliche Desynchronisation zwischen Dokumentation und Code veranlasst Entwickler dazu, die Dokumentation als unzuverlässige Quelle für Architekturinformation anzusehen.

Frage 2: *Wie gut reflektiert die Architekturdokumentation die vorgesehene Architektur?* Die Defizite der Dokumentation wurden nach Durchführung der Interviews mit den Entwicklern (Schritt 3) erhoben. Viele dieser Interviews verursachten lebhafte Diskussionen unter den Entwicklern. Dies ist auf eine unterschiedliche Wahrnehmung und ein unterschiedliches Verständnis der vorgesehenen Architektur zurückzuführen. Das Ergebnis dieser Interviews waren Regeln,

die zur Architekturbeschreibung hinzugefügt werden mussten. Tabelle 10.5 zeigt, dass zwischen 72% und 90% der Abweichungen zwischen der dokumentierten und der implementierten Architektur auf Unzulänglichkeiten in der Dokumentation zurückzuführen sind. Tabelle 10.4 fasst die während des Analyseprozesses benötigten Änderungen der initialen Dokumentation zusammen. Während bei den Projekten A und C eine ähnliche Anzahl an Modifikationen notwendig war, rief Projekt B deutlich mehr Änderungen hervor, die in die Architekturbeschreibung eingepflegt werden mussten. Dies liegt in der feingranulareren Beschreibung der Architektur in der Dokumentation von Projekt B begründet.

Zusammenfassend wurden die meisten Abweichungen ($diff_{doc}$) als Defizite der Dokumentation angesehen. Die Anzahl der Unzulänglichkeiten der Dokumentation sind ein Maß für eine Veränderung der Architektur in der Evolution, da sie die Änderungen der initial vorgesehenen Architektur über die Zeit beschreibt.

Frage 3: *Wie groß ist der Verfall der Architektur im System?* Die letzte Spalte von Tabelle 10.5 zeigt die Architekturverletzungen, die in den einzelnen Projekten gemessen wurden. Der relative Anteil an Architekturverletzungen in den Systemen liegt zwischen 10% und 28%. Obwohl Projekt B den höchsten relativen Anteil aufweist (28%), liegt die absolute Zahl der Verletzungen, die aus den Systemen entfernt werden mussten, mit ca. 100 Abhängigkeiten auf gleichem Niveau. Die feingranularere Architekturbeschreibung in Projekt B verursacht, dass die Analyse als präziser angesehen werden kann. Dadurch konnten mehr Verletzungen gefunden werden als bei vergleichsweise grobgranularen Architekturbeschreibungen. Zusätzlich ist die Architektur von Projekt B restriktiver (vgl. Tabelle 10.3) als die der anderen Projekte. Eine Konsequenz daraus ist, dass deutlich weniger Abhängigkeiten erlaubt sind und somit die Wahrscheinlichkeit höher ist, dass Entwickler Architekturverletzungen verursachen.

Frage 4: *Was sind die Gründe für Abweichungen zwischen der vorgesehenen Architektur und dem Code?* Die meisten der Dokumentationsdefizite wurden aufgrund von während der Implementierungsphase neu gewonnenen Einsichten hervorgerufen. System A sollte beispielsweise keine direkten Abhängigkeiten der GUI- und der DataAccess-Komponente aufweisen. Die Analyse stellte jedoch derartige Abhängigkeiten im Code fest. Nach genauerer Untersuchung berichteten die Entwickler, dass diese Abhängigkeiten als unkritisch und erlaubt zu betrachten sind, da sie benötigt werden, um während des Aufstartens des Systems an spezielle Daten zu gelangen. Zur Aufstartzeit sind die Business-Komponenten, die normalerweise die Schnittstelle für die GUI-Komponente darstellen, noch nicht verfügbar. Dies ist ein Beispiel für eine Verfeinerung der Architektur, die während der Implementierungsphase durchgeführt wurde. Während der Entwurfsphase wurde es übersehen, dass eine derartige Abhängigkeit notwendig war. Dennoch wurde dieses Wissen nicht der Dokumentation hinzugefügt.

Entwickler sehen die Architekturdokumentation als Relikt aus den frühen Phasen des Projekts, als die Architektur initial kreiert wurde. Somit wird sie eher als ein Artefakt gesehen, das die konstruktive Phase der Architekturerstellung unterstützen soll und nicht so sehr als eine Beschreibung des Systems für die langfristige Wartung und Weiterentwicklung (vgl. Abschnitt 3.2.2). Die relativ hohe Anzahl an Dokumentationsdefiziten ist jedoch nicht auf die

'Faulheit' von Entwicklern zurückzuführen. Oftmals wissen Entwickler schlichtweg nicht, dass Teile in der Dokumentation existieren, die im Zuge einer Änderung am Code angepasst werden sollten, da sie während ihrer alltäglichen Arbeit die Dokumentation nur sehr wenig nutzen. Dies kann als ein Mangel an Verfügbarkeit von gewöhnlicher Dokumentation angesehen werden. Aus diesem Grund wird Dokumentation zu einem toten Artefakt, das mit zunehmender Veraltung immer noch weniger genutzt wird. Oftmals wussten die Entwickler (insbesondere in Projekt C), dass die Dokumentation veraltet war. Sie erklären, dass eine Redokumentation bereits auf ihrer Tagesordnung steht, dass derartige Aktivitäten jedoch aufgrund von Release-Druck und der höheren Priorität der Implementierung neuer Features oder Bug-Fixes oft verschoben werden.

Durch die Analysen wurden teilweise sogar kritische Architekturverletzungen gefunden. In Projekt A wurde beispielsweise folgende Verletzung identifiziert: Es wurden einige Aufrufe von Methoden implementiert, die Transaktionen auf eine nicht adäquate Art und Weise über eine falsche Schnittstelle ausführten. Dies verursachte einige unnötige Transaktionen und signifikante Performance-Einbußen. Diese Verletzung wurde von einem Entwickler hervorgerufen, der nicht die Schnittstellen verwendet hatte, die für diesen Zweck vorgesehen waren. Die Erklärung der Entwickler war, dass derjenige, der die betroffenen Code-Teile implementiert hatte, nicht wusste, welche Komponente er nutzen sollte, um diese Aufgaben zu erfüllen. Das zeigt, dass nicht alle Entwickler über die Intention und die adäquate Verwendung der Architektur Bescheid wissen.

Ein weiteres Beispiel für eine typische Verletzung ist auf Copy&Paste Programmierung zurückzuführen. Oftmals werden die Header von Dateien kopiert und als Vorlage zur Implementierung einer neuen Klasse eingefügt. Leider war auch die Deklaration des Namensraums oftmals Teil der kopierten Zeilen und es wurde schließlich vergessen, diese entsprechend anzupassen. Obwohl dies eher harmlos wirkt, ist es für einen anderen Entwickler schwierig, dies als Copy&Paste-Problem zu identifizieren und zu verstehen, dass die Klasse eigentlich in einem anderen Namensraum deklariert werden sollte.

10.5. Erfahrungen

Architekturwissen geht verloren. Entwickler kennen nicht die komplette Architektur eines Systems und insbesondere nicht die Art, wie diese auf Codeebene reflektiert wird. Der Hauptgrund hierfür sind die unzähligen Details auf der Codeebene und die große Abstraktionslücke zwischen der Architekturspezifikation und der Implementierung. Anstatt mühsame Arbeiten durchzuführen, um die vorgesehene Architektur zu verstehen und wiederherzustellen (zu erraten), ist es empfehlenswert, den Ansatz zu wählen, das Architekturwissen und dessen Abbildung auf den Code präventiv zu konservieren.

Konformitätsüberprüfungen sind ein Katalysator für Diskussionen. Die Fallstudie zeigt, dass eine strukturierte Dokumentation in einer maschinenlesbaren Form und eine automatische Analyse der Architektur eine höhere Sichtbarkeit im Entwicklerteam mit sich bringt. Viele Diskussionen über die adäquate Verwendung der Architektur flammten während der

Durchführung der Fallstudie unter den Entwicklern auf. Dadurch wurden die unterschiedlichen impliziten Sichten der einzelnen Teammitglieder synchronisiert und in die explizite Dokumentation übernommen.

Kontinuierliche Architekturanalyse. Um sicherzustellen, dass die Architekturbeschreibung auch zukünftig als Spezifikation der vorgesehenen Architektur konsistent mit dem Code gehalten wird, wurde die Architekturanalyse in den Nightly-Build der Projekte integriert. Die Architekturbeschreibung wurde in das Versionsmanagementsystem eingefügt, so dass die Entwickler leicht auf diese zugreifen können. Die Ergebnisse dieser kontinuierlichen Analyse können die Entwickler über einen Link in ihrem Projektportal einsehen. Durch diese Maßnahme bekam die Architektur eine zentralere Rolle innerhalb des Projekts. Die kontinuierlich überprüfte Architekturdokumentation hat eine bessere Verfügbarkeit und Sichtbarkeit im Projekt als die ursprünglichen Textdokumente.

Die Entwickler stimmten zu, dass durch die kontinuierliche Analyse der Konformität zwischen der Architekturbeschreibung und dem Code eine bessere Integration der Architekturvorgaben mit der Implementierung des Systems geschaffen und somit eine bessere Konservierung des Architekturwissens in den Projekten erreicht wurde. Zudem ermöglicht die kontinuierliche Analyse eine frühzeitige Detektion (und ggf. Beseitigung) von potentiellen Architekturverletzungen und Architekturmodifikationen sehr nahe an dem Zeitpunkt, an dem diese ins System einfließen. Dadurch kann sehr schnell entschieden werden, ob es sich um eine Verletzung oder um eine notwendige Veränderung der Architektur handelt. Damit werden die Kosten zur Beseitigung von Verletzungen und zur Anpassung der Dokumentation niedrig gehalten, da die verantwortlichen Entwickler immer noch wissen, woran sie vor kurzem gearbeitet haben und sich nicht erneut in den Kontext einarbeiten müssen. Auch weitere Tätigkeiten, wie etwa Tests, müssen nicht aufgrund derartiger Änderungen erneut durchlaufen werden.

Sogar einige Wochen nachdem die Fallstudie abgeschlossen war, berichteten die Entwickler, dass sie immer noch jeden Morgen die Ergebnisse der Analyse in ihrem Projektportal betrachten. Durch die Integration der Analyse in den Nightly-Build wurde eine lebendige Art der Architekturdokumentation geschaffen. Auch nachdem ein Jahr später wieder Kontakt zu den Entwicklern aufgenommen wurde, war die XML-Architekturbeschreibung noch aktuell und synchronisiert mit dem Code. Es wurden auch bereits einige Änderungen durch die Entwickler selbst an der Architekturbeschreibung vorgenommen.

Benötigte Aufwände. Die Aufwände zur Einführung der Analyse, zur Konfiguration des Projektportals sowie der Erstellung der XML-Architekturbeschreibung kosten ungefähr fünf Arbeitstage für ein Projekt. Die größten Aufwände werden durch das Reverse Engineering und die Diskussionen, wie die Architektur auszusehen hat, hervorgerufen. Nach diesen initialen Aufwänden benötigt die Architekturanalyse im Nightly-Build nur noch wenig Aufwände. Lediglich Anpassungen der Architekturbeschreibung aufgrund von Änderungen im System müssen durchgeführt werden. In Projekt A waren nur vier kleine Anpassungen in einem Jahr notwendig. Aber dies kann bei unterschiedlichen Projekten variieren.

10.6. Einschränkungen der Aussagekraft

Interne Validität. Die Auswertung der in der Fallstudie gemessenen Werte unterliegt gewissen Annahmen, welche die Ergebnisse potentiell beeinflussen.

Verstecktes Wissen. Der Prozess zur Identifikation der vorgesehenen Architektur der drei Systeme basiert auf der iterativen Untersuchung der Abweichungen zwischen der dokumentierten Architektur und dem Code. Diese Abweichungen bilden die Grundlage für die Diskussion mit den Entwicklern. Dennoch könnte es vorkommen, dass die Dokumentation und der Code in gewisser Weise gut zueinander passen, obwohl sie beide von der vorgesehenen Architektur abweichen. In derartigen Situationen würde die angewandte Methode die vorgesehene Architektur nicht herausfinden. Dieses Phänomen würde jedoch lediglich die Vollständigkeit der Ergebnisse beeinträchtigen. Die quantitativen Ergebnisse des Verlusts an Architekturwissen wären nicht beeinträchtigt, der tatsächliche Verlust wäre in diesen Situationen sogar noch größer als es die gemessenen Werte wiedergeben.

Übersetzung von textueller Dokumentation in die maschinenlesbare Form. Die Übersetzung der informellen Dokumentation in die maschinenlesbare XML-Repräsentation der Architektur könnte einige der gemessenen Ergebnisse beeinflussen. Etwa könnten Informationen, die in der Dokumentation enthalten sind, übersehen oder fehlinterpretiert werden. Dies könnte dazu führen, dass die Resultate, die nach der ersten Ausführung der automatischen Analyse (Schritt 2) gemessen wurden, verfälscht werden. Dennoch tritt derselbe Effekt auch bei Entwicklern auf, die nicht mit dem System vertraut sind und durch das Studieren der Dokumentation versuchen, die Architektur zu erlernen.

Die Entwickler selbst wissen nicht vollständig, wie die Architektur vorgesehen war. In einigen Situationen wurden die Entwickler und der Architekt über einige Details der Architektur befragt und sie waren nicht in der Lage, die Fragen sofort zu beantworten. Derartige Fragen mussten dann im Team diskutiert werden, bevor präzise Aussagen über die vorgesehene Architektur gemacht werden konnten. Dieser Effekt kann die Ergebnisse der Fallstudie beeinflussen, da eine derartige Team-Entscheidung gegebenenfalls die vorgesehene Architektur nicht widerspiegelt.

Soziale Erwünschtheit von Architekturverletzungen. In Entwicklerkreisen gibt es oftmals Wertvorstellungen, nach denen es negativ belegt ist, wenn ein Entwickler Architekturverletzungen in seinem Code erzeugt. Die Erstellung von präziser und aktueller Dokumentation steht hingegen bei einigen Entwicklern sogar in dem Ruf, eine überflüssige und bürokratische Aktivität zu sein. Dies zeigt sich auch im sogenannten "Agilen Manifest"[2], in dem als zweite Regel formuliert ist: "Funktionierende Programme gelten mehr als ausführliche Dokumentation". Dies ist ein in der Sozialforschung bekanntes Phänomen der *sozialen Erwünschtheit.* Aus diesem Grund neigen viele Entwickler eher dazu, identifizierte Abweichungen als Dokumentationsdefizite zu deklarieren und weniger dazu, sich diese als Verstöße gegen die Architektur einzugestehen. Somit könnte die reale Anzahl der Architekturverletzungen etwas höher und die reale Anzahl an Dokumentationsdefiziten entsprechend niedriger liegen, als die gemessenen Werte erkennen lassen.

[2] http://agilemanifesto.org/

Versteckte Abhängigkeiten. Auf Codeebene könnten Abhängigkeiten existieren, die durch die automatische Analyse nicht erfasst werden können, beispielsweise durch die Nutzung von Reflection hervorgerufen. In diesem Fall wären die Messungen verfälscht, da gegebenenfalls weniger Abweichungen gefunden werden als tatsächlich vorliegen. Dennoch kann diese Fehlerquelle die Ergebnisse nicht zu stark verunreinigen, da sowohl eine manuelle Inspektion des Codes als auch durch Aussagen der Entwickler bestätigt wurde, dass lediglich die in Tabelle 10.1 beschriebenen Abhängigkeitstypen verwendet werden.

Externe Validität. Einige Besonderheiten der untersuchten Projekte könnten die Generalisierung der Ergebnisse und deren Übertragbarkeit auf andere Projekte beeinflussen.

Die Umgebung bei der Münchener Rück könnte die Ergebnisse beeinflussen. Subjekt der Fallstudie waren drei Projekte, die durch unterschiedliche und nicht überlappende Entwicklerteams erstellt wurden. Zudem wurden die Projekte durch unterschiedliche Unternehmen als Auftragnehmer durchgeführt. Dennoch war der Auftraggeber immer die Münchener Rück und alle Systeme wurden unter den von der Münchener Rück vorgegebenen Rahmenbedingungen entwickelt. Alle Projekte nutzen den gleichen Entwicklungsprozess, ähnliche Infrastruktur, Werkzeuge und Technologien (.NET). Dadurch könnte die externe Aussagekraft der Ergebnisse begrenzt sein. Wäre die Fallstudie in einem anderen Umfeld durchgeführt worden, könnten die Ergebnisse von den bei der Münchener Rück gemessenen Ergebnissen abweichen.

10.7. Verwandte Arbeiten

Überprüfung der Architekturkonformität. In der Reverse Engineering Literatur gibt es einige Ansätze, um die Konformität der implementierten Architektur bezüglich höherer Modelle zu überprüfen [MNS01, FA98]. Die Autoren von [FA98] stellen einen Ansatz vor, um die Übereinstimmung von objektorientierten Entwürfen mit dem Quellcode zu überprüfen, indem in OMT spezifizierte Entwürfe auf in C++ geschriebene Programme abgebildet werden. Die hier vorgestellte Technik, um Architektur zu definieren und deren Konformität mit dem Code zu prüfen, ist ähnlich zu den von Murphy entwickelten 'Reflection Models' [MNS01]. Trotz der Ähnlichkeiten der Techniken zur Konformitätsüberprüfung ist der Fokus unterschiedlich. In dieser Fallstudie wurde der Verlust an Architekturwissen in der Systemevolution und die Nützlichkeit von Modellen (die Architektur-Constraints beschreiben) untersucht, um dieses Wissen explizit zu machen.

In [EKKM08] wird ein Ansatz zur Nutzung deklarativer Abfragen vorgestellt, um strukturelle Abhängigkeiten zu spezifizieren und damit die Konformität der Implementierung zu überprüfen. Diese Abfragen werden während der Entwicklung kontinuierlich ausgeführt. Die Ergebnisse der hier vorgestellten Fallstudie bestätigen den Bedarf einer Integration von kontinuierlichen Architekturüberprüfungen in den Entwicklungsprozess, um dem Verlust an Architekturwissen vorzubeugen.

Verwandte Fallstudien. In [vGBB05] wird untersucht, wie Unternehmen Design-Verfall (design erosion) identifizieren und wie sie der Herausforderung der Erhaltung des Designs entgegentreten. Die Fallstudie basiert auf einer qualitativen Analyse, die durch Interviews mit den Entwicklern durchgeführt wurde. Unter den wichtigsten Gründen für Design-Verfall nennen die Autoren mangelndes Wissen der Entwickler über originäre Entwurfsentscheidungen und zu wenig Aufmerksamkeit für das Design aufgrund von Release-Druck. Die Ergebnisse der hier durchgeführten Fallstudie bestätigen diese Thesen. Zudem sollte die Konservierung von Architekturwissen durch die kontinuierliche Überprüfung der Konformität zwischen dem Code und der Dokumentation vorgenommen werden. Die im Rahmen der hier vorgestellten Fallstudie angewandte Methode ist sowohl quantitativ (Messung der Diskrepanz zwischen Dokumentation und dem Code) als auch qualitativ aufgrund des Kontakts zu den Entwicklern.

Die in [TCL02] präsentierte Fallstudie zeigt die Schwierigkeit auf, Abweichungen zwischen dem Code und dem vorgesehenen Design zu detektieren, die aufgrund von personellen Fluktuationen auftreten. Die Autoren schlagen die Verwendung von Metriken als Mittel zur Erkennung von Abweichungen vor. In dieser Arbeit wurde der Bedarf aufgezeigt, die explizite Repräsentation der vorgesehenen Architektur kontinuierlich zu warten.

In [MLLF07] beschreiben die Autoren Erfahrungen, die auf Basis eines industriellen Projekts gesammelt wurden, in dem Architekturmodelle zum Einsatz kamen. Sie berichten von hohen Aufwänden, um die Modelle und die Implementierung konsistent zu halten. Die Erfahrungen aus der hier vorgestellten Fallstudie bestätigen diese Beobachtungen. Der hier vorgestellte Ansatz zeigt auf, wie der Aufwand zur Überprüfung der Konsistenz zwischen der Dokumentation (dem Modell) und der Implementierung auf Basis automatisierter Methoden reduziert werden kann.

Management von Architekturwissen. In Abschnitt 3.1 wurde bereits die Unterscheidung zwischen Kodifizierungs- und Personalisierungsstrategien nach [HNT99] eingeführt. Die im Rahmen dieser Fallstudie präsentierte Technik ist klar als Kodifizierungsstrategie einzuordnen, die besondere Vorteile in der Konservierung des Wissens bietet, um den Wissensverlust durch Entwicklerfluktuation entgegen zu wirken. Dennoch werden durch die kontinuierliche Analyse und die gefundenen Verletzungen Diskussionen zwischen den Entwicklern gefördert, die wiederum als Einstiegspunkt in eine Personalisierungsstrategie verstanden werden können.

10.8. Zusammenfassung und Fazit

In diesem Kapitel wurden Erfahrungen präsentiert, die durch eine Fallstudie zur Bestimmung des Grads an verlorenem Architekturwissen in drei industriellen Projekten gesammelt wurden. Durch diese Fallstudie wurden drei Manifestationen des Verlusts an Architekturwissen identifiziert: Verfalls des Codes in Form von Verletzungen der vorgesehenen Architektur, Verlust an Information in der Dokumentation und unterschiedliche Wahrnehmung der vorgesehenen Architektur durch unterschiedliche Entwickler.

Die wichtigsten Ergebnisse der Studie sind: Die informelle Dokumentation und der Quellcode werden nicht konsistent gehalten, keine der beiden reflektiert die vorgesehene Architektur vollständig und sogar Entwickler sind sich (einzeln befragt) der vollständigen Architektur nicht bewusst. Quantitativ werden zwischen 70% und 90% der Inkonsistenzen durch Defizite der Dokumentation hervorgerufen und 10% bis 30% repräsentieren Verletzungen der Architektur im Code.

Ein zentrales Ergebnis der Fallstudie ist, dass zwischen 9% und 19% aller Abhängigkeiten, die in den untersuchten Systemen implementiert sind, nicht mit der dokumentierten Architektur übereinstimmen. Diese Abweichungen konnten als Dokumentationsdefizite und Verletzungen der Architektur in der Implementierung nachgewiesen werden. Die echte vorgesehene Architektur war somit zwischen diesen beiden Artefakten und dem Wissen der Entwickler verborgen. Die wichtigste Erfahrung ist, dass das Wissen über die vorgesehene Architektur explizit gemacht und kontinuierlich automatisiert überprüft werden sollte, um den Verlust an Wissen zu minimieren und um die Aufmerksamkeit der Entwickler bezüglich der Integrität der Architektur zu bewahren.

11. Anwendungsbereich 3: Implizite Vorgaben im Softwareentwurf

“Abusus optimi pessima.” (Der Missbrauch des Besten ist das Schlimmste.)
- Lateinisches Sprichwort

In der in Kapitel 9 vorgestellten Fallstudie wurde gezeigt, dass in APIs eine große Menge an impliziten Entwurfsregeln zu finden ist. Zudem wurde dargestellt, wie diese Regeln durch statisch überprüfbare Constraints ausgedrückt werden können. Implizite Festlegungen sind jedoch nicht auf APIs beschränkt. Gute APIs bieten dem Nutzer meist eine Schnittstelle in Gestalt einer Fassade an. Hinter dieser Fassade ist es möglich, Implementierungsdetails zu verstecken und so dem Nutzer eine möglichst komfortable und aufgeräumte Sicht zu präsentieren, in der die Konzepte gut durch die öffentlichen Deklarationen der API reflektiert werden.

Einem Wartungsingenieur bietet sich oftmals eine deutlich weniger komfortable Sicht auf ein System, an dem er Änderungen durchzuführen hat. Er sieht meist nicht nur eine Fassade, sondern direkt in die Implementierung der Funktionalität hinein und muss sich deren Strukturen herausarbeiten. Dies ist eine Folge des Verlusts an Informationen über systeminterne Strukturen im Lauf des Entwicklungsprozesses und der Evolution des Systems. Es handelt sich somit um unzureichenden vertikalen Informationsfluss. In diesem Kapitel soll eine Einführung in die Gestalt von impliziten Entwurfsregeln in Softwaresystemen gegeben werden.

Entwurfsmuster. Muster stellen Lösungsschablonen für häufig wiederkehrende Problemstellungen dar. Wird im Lauf der Konzeption eines Systems die Anwendbarkeit von Mustern erkannt, erleichtern diese Schablonen den Übergang von einem Konzeptmodell zur Implementierung. Durch Muster wird auch die Kommunikation zwischen Entwicklern erleichtert, da sie Begrifflichkeiten definieren, die es ermöglichen, effizient über oftmals komplexe Programmstrukturen zu sprechen. Muster können selbst als Ausschnitte von Konzeptmodellen mit häufig im Systementwurf benötigten Konzepten verstanden werden.

Im Wartungsprozess kommt der Erkennung von Mustern eine wichtige Rolle zu. Nur wenn Muster erkannt werden, können Änderungen auch in adäquater Weise durchgeführt werden, ohne das ursprüngliche Design des Codes zu beeinträchtigen. Eingebettet in den Code stellen Muster jedoch implizite Regeln dar. Einem Wartungsingenieur ist nicht zwingend bewusst, dass bestimmte Muster verwendet wurden. Er erkennt diese meist nur an bestimmten Bezeichnern, beispielsweise würden Bezeichner der Art ‘...Observer’ oder ‘...Subject’ auf die Verwendung eines Observer-Patterns hindeuten. Diese Art der Mustererkennung setzt jedoch voraus, dass der Wartungsingenieur bereits mit dem Muster vertraut ist, ansonsten würde er diese Signalworte

in den Bezeichnern nicht wiedererkennen. Zudem schreiben die meisten Muster nicht die Verwendung dieser Begriffe vor. Es gibt viele Arbeiten im Bereich des Reverse Engineerings, die sich mit dem Auffinden von Mustern im Code beschäftigen (beispielsweise [HHHL03, Big89]). Die automatische Mustererkennung ist eine komplexe und aufwändige Aufgabe, die oftmals sehr unscharfe Ergebnisse liefert. Wenn bereits im Forward Engineering darauf geachtet wird, dass diese implizite Information konserviert wird, werden deutlich einfacher wartbare Softwaresysteme erstellt.

Neben den bekannten und in der Literatur häufig zitierten Mustern gibt es in verschiedenen Branchen und Domänen eine Menge an domänenspezifischen Mustern, die nur in einem deutlich eingeschränkteren Bereich eingesetzt werden können. Werden derartige Muster genutzt, ist es nochmals unwahrscheinlicher, dass diese während der Wartung als solche wiedererkannt werden.

Im Weiteren werden drei Muster vorgestellt und aufgezeigt, welche Konzeptmodelle diesen zugrunde liegen und inwiefern deren Implementierung mit impliziten Entwurfsregeln einhergeht.

11.1. Das Observer-Pattern

Konzeptmodell. In Abbildung 11.1 ist das Konzeptmodell dargestellt (oben), welches dem Observer-Pattern (unten) zugrunde liegt. Dieses Muster implementiert die Konzepte 'Subject' und 'Observer'. Damit stellt es eine Trennung zwischen den logischen Daten, die als Zustand in einem System auftreten, und deren Präsentation her (z.B. unterschiedliche Fenster einer Applikation, welche die Daten anzeigen). Die 'isa'-Beziehungen zwischen 'ConcreteSubject' und 'Subject' bzw. 'ConcreteObserver' und 'Observer' stellen dar, dass es in Systemen mehrere Ausprägungen der 'ConcreteSubjects' und 'ConcreteObserver' geben kann, die alle in gleicher Weise agieren. Die Konsistenzhaltung zwischen dem Datenzustand ('ConcreteSubject') und deren Repräsentation ('ConcreteObserver') wird nicht durch direkte Beziehungen, sondern über einen generischen Mechanismus realisiert, indem die 'ConcreteSubjects' Änderungen am Datenzustand melden ('reportUpdates') und 'Subject' diese Information an 'Observer' übermittelt ('notifiesAll'). Das Konzept 'Observer' ist schließlich für die Übermittlung an die 'ConcreteObserver' zuständig ('updates'). Durch die Einführung dieser Konzepte soll dafür gesorgt werden, dass nicht jede Präsentation (View) einen eigenen Mechanismus implementiert, sondern ein einheitliches Verfahren genutzt wird, das zentral gekapselt ist.

Implementierung. In Abbildung 11.2 ist ein leicht vereinfachtes Implementierungsmodell des Observer-Patterns zu sehen. Die beiden zentralen Konzepte 'Subject' und 'Observer' werden im Code durch Klassen repräsentiert, die Relationen über Methoden. Im Diagramm des Observer-Patterns sind die Klassen 'ConcreteSubject' und 'ConcreteObserver' dargestellt. Diese sind als Platzhalter zu interpretieren, die nicht direkt in die Implementierung übernommen werden. Sie zeigen auf, wie die konkret in einer Applikation benötigten Views und deren zugrunde liegende Daten implementiert werden sollen. Die Information, dass alle konkreten Subjects und Observer auf diese Art implementiert werden sollen, stellt somit eine Menge an Entwurfsregeln

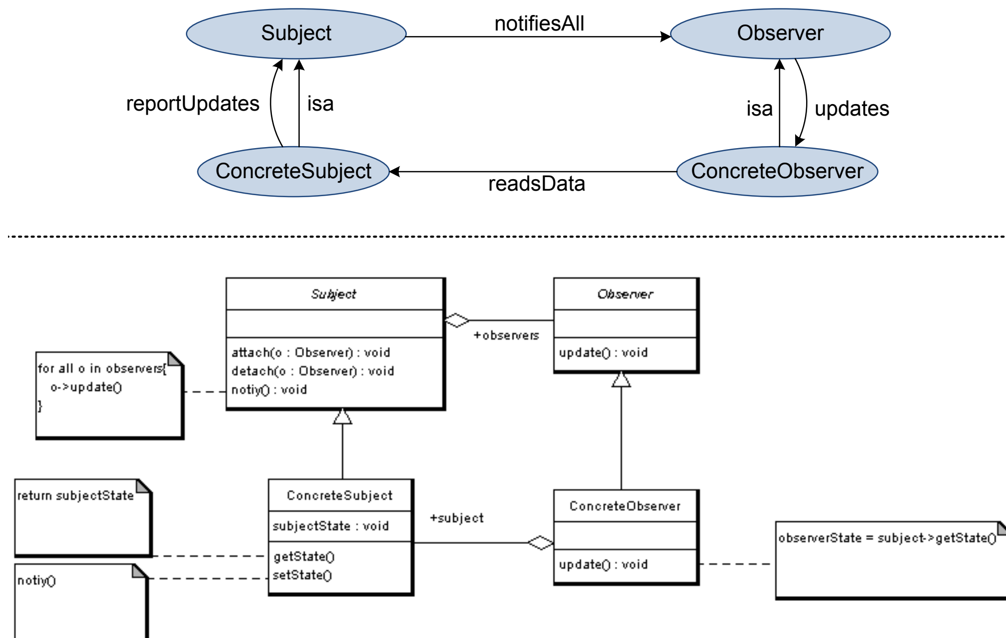


Abbildung 11.1.: Ein Konzeptmodell des Observer-Patterns [GHJV95]

dar: Oftmals sollten die Unterklassen entsprechende Namen tragen, die diese als konkrete 'Subjects' bzw. 'Observer' auszeichnen. Alle konkreten 'Subjects' sollten bestimmte Attribute aufweisen, die den darzustellenden Zustand repräsentieren. Auch diese Attribute sollten ggf. nach einem einheitlichen Schema benannt werden. Am wichtigsten ist jedoch, dass immer, wenn eine Methode schreibend auf diese Attribute zugreift, ein Aufruf der Methode 'notify' in der 'Subject'-Oberklasse erfolgen muss.

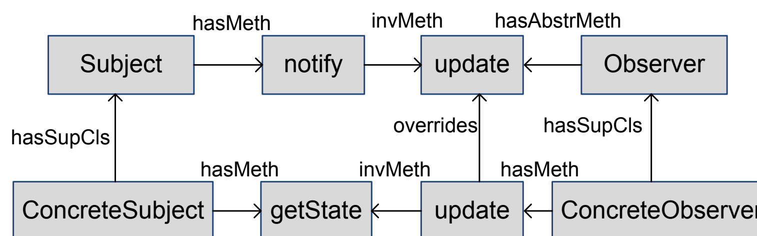


Abbildung 11.2.: Ein Implementierungsmodell des Observer-Patterns

Eine möglichst homogene Implementierung der konkreten 'Observer' und 'Subjects' stärkt die Verständlichkeit. Aber genau diese Information über diese Art von "Kodierungsrichtlinie" ist meist lediglich implizit vorhanden und geht leicht im Lauf der Evolution eines Systems verloren. Das Observer-Pattern versucht, die Logik der Benachrichtigung der graphischen Repräsentationen in den beiden Oberklassen zu kapseln. Man stellt jedoch fest, dass es durch

objektorientierte Programmiersprachen nicht möglich ist, diese Funktionalität vollkommen aus den konkreten Unterklassen zu entfernen. So ist die Information "nach einer Zustandsänderung muss 'notify' aufgerufen werden" vielfach in allen konkreten 'Subjects' implementiert. Diese stellt eine gewisse Art von Redundanz zwischen den 'ConcreteSubjects' dar, die konsistent gehalten werden muss (vgl. [SLL+88]). Die folgende Bedingung stellt diesen Zusammenhang formal dar:

$$p_{\mathbb{I}}(Subject, ConcreteSubject) \rightleftharpoons e_{\mathbb{D}}(Subject, ConcreteSubject)$$

Diese Bedingung ist in der objektorientierten Implementierung des Observer-Patterns nicht explizit, da ihre Einhaltung nicht bereits durch die Sprache überprüft wird. Im Gegensatz dazu ist die Implementierung der Relation 'updates' deutlich expliziter. Durch Nutzung der Möglichkeit zur Deklaration abstrakter Methoden in der Oberklasse können Unterklassen-Implementierungen dazu gezwungen werden, diese Methoden zu überschreiben. Damit wird vorgebeugt, dass Entwickler die Implementierung des Update-Mechanismus bei der Entwicklung neuer 'ConcreteObserver' vergessen oder dass bei einer Änderung diese Methoden entfernt werden. Die damit verbundene Entwurfsregel ist somit explizit im Quellcode realisiert. Folgende Bedingung kann somit als erfüllt betrachtet werden:

$$p_{\mathbb{I}}(Observer, ConcreteObserver) \rightleftharpoons e_{\mathbb{D}}(Observer, ConcreteObserver)$$

Überprüfung durch Constraints. Folgender Constraint überprüft, ob die oben vorgestellte implizite Entwurfsregel im Observer-Pattern ('reportUpdates') adäquat implementiert wurde:

$$\begin{gathered}\textbf{Decl: } Subject \;\; \textbf{Use: } \underline{ClsInh} \\ \textbf{Ctxt:} \;\; \forall f \in !\underline{FieldDecl} : f.use.\underline{FieldDecl} \wedge f.\underline{Name} endswith(\text{“}State\text{”}) \\ \forall a \in !\underline{Assignment} \;\; \exists inv \in !\underline{MethInv} : a.\underline{Lhs} = f \wedge inv \uparrow a.\underline{StmtBlock} \\ \wedge\, inv.\underline{ObjName} = \text{“}super\text{”} \wedge inv.\underline{Name} = \text{“}update\text{”}\end{gathered}$$

Dieser Constraint überprüft, ob nach allen schreibenden Zugriffen auf ein Attribut, das den durch die Observer überwachten Zustand hält, ein Aufruf der 'update'-Methode erfolgt, um die Observer über die Änderung im Zustand zu benachrichtigen. Diese Zustandsattribute sind Felder der 'ConcreteSubject'-Klassen, die dadurch gekennzeichnet sind, dass ihr Name auf 'State' endet. Diese Namensrichtlinie wurde eingeführt, um Zustandsattribute von ggf. vorhandenen anderen Feldern dieser Klassen unterscheiden zu können. Alternativ könnte man diese auch mit einem speziellen Kommentar kennzeichnen.

11.2. Das Visitor-Pattern

Konzeptmodell. Abbildung 11.3 zeigt das Konzeptmodell (oben) des Visitor-Patterns und beispielhaft dessen Implementierung (unten). Das Visitor-Pattern ist ein Ansatz zur Trennung

der Iterationslogik über Graphstrukturen von den auf den Graphknoten durchzuführenden Aufgaben, die Besuche der einzelnen Knoten erfordern. Ein derartiger Graph könnte beispielsweise ein abstrakter Syntaxbaum eines Programms sein. Darauf auszuführende Aufgaben wären zum Beispiel Code-Generierung oder Code-Formatierung (engl. Pretty Printing). Die Traversierungsfunktionalität wird im Konzeptmodell durch das Konzept 'Visitor' repräsentiert, der Graph selbst durch ein Geflecht an 'Nodes'. Die konkreten Aufgaben, die durch Iteration über den Graphen erfüllt werden sollen, werden durch mehrere 'VisitTasks' dargestellt. Die einzelnen Knoten im Graphen durch 'Nodes'.

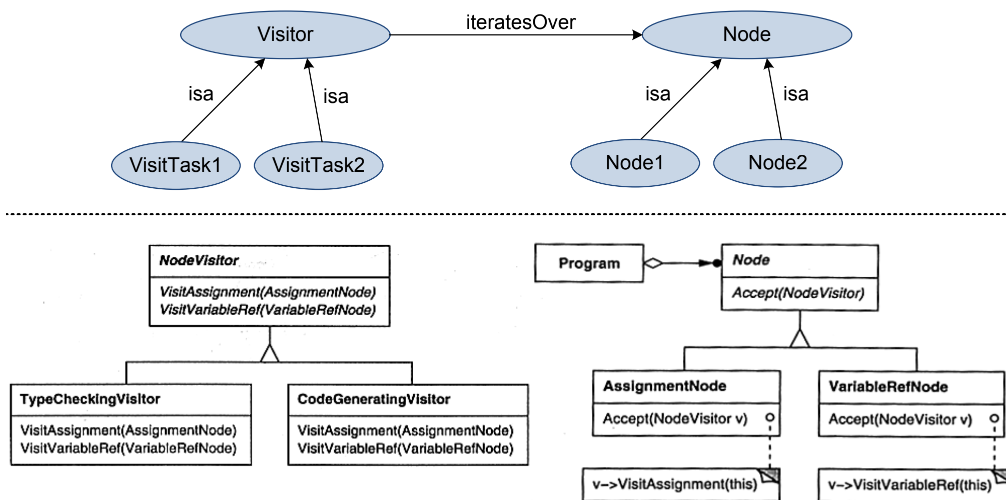


Abbildung 11.3.: Ein Konzeptmodell des Visitor-Patterns [GHJV95]

Implementierung. Das Visitor-Pattern verringert die Redundanz zwischen den unterschiedlichen Aufgaben, da die Logik der Traversierung des Graphen wiederverwendet werden kann. Zudem können Aufgaben zentral in einer Klasse anstatt verteilt über alle Knoten-Klassen implementiert werden. Abbildung 11.4 zeigt ein leicht vereinfachtes Implementierungsmodell des Visitor-Patterns. Auch hier sieht man, dass die objektorientierte Implementierung die Iterationslogik nicht vollkommen aus den Implementierungen der 'VisitTasks' und den Unterklassen von 'Node' (in Form von Klassen) heraus halten kann: Alle 'Node'-Unterklassen müssen eine entsprechende 'visit'-Methode in der Visitor-Klasse aufweisen. Diese Regel kann nicht vom Compiler überprüft werden und ist somit als implizit zu betrachten. Alle Unterklassen von 'Graph' müssen die 'accept'-Methode überschreiben. Durch die Deklaration einer abstrakten Methode in der Klasse 'Node' kann bewerkstelligt werden, dass dies durch den Compiler überprüft und diese Regel somit explizit wird. Die 'accept'-Methoden müssen alle einen Aufruf der 'visit'-Methode auf dem als Parameter übergebenen 'Visitor'-Objekt aufrufen. Da dies nicht automatisch durch die Sprache sichergestellt wird, ist dies als implizite Entwurfsregel zu betrachten. Alle diese Regeln entsprechen einer nicht vollkommen expliziten Implementierung

der ‘iteratesOver’-Kante im Konzeptmodell. Folgende Formel stellt dies formal dar:

$$p_{\mathbb{I}}(Visitor, Node) \rightleftharpoons e_{\mathbb{D}}(Visitor, Node)$$

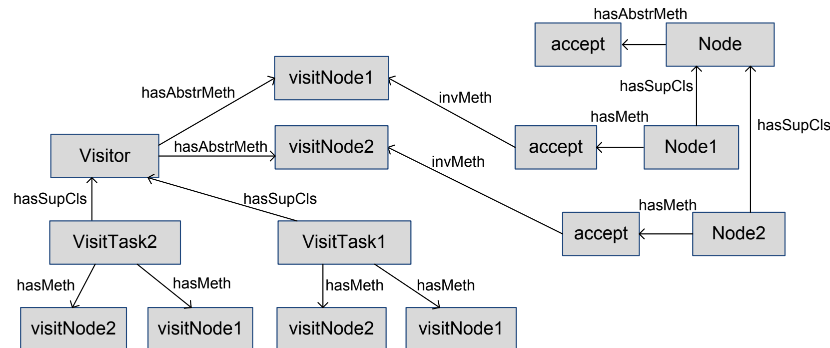


Abbildung 11.4.: Ein Implementierungsmodell des Visitor-Patterns

Auch im Visitor-Pattern findet man somit viele implizite Festlegungen, die zwingend eingehalten werden müssen, um Fehlverhalten und Design-Verstöße zu vermeiden. Obwohl das Visitor-Pattern ein sehr bewährtes Muster ist, das in vielen Anwendungen erfolgreich eingesetzt wird, sieht man daran, dass es nicht möglich ist, die Information über die Graphstrukturen nur einmal zu repräsentieren. Statt dessen müssen alle Knotentypen nicht nur in Form einer Klasse, sondern in konsistenter Weise nochmals als ‘visit’-Methoden implementiert werden. Diese Redundanz muss im Lauf der Wartung konsistent gehalten werden. Dies zeigt, dass heutige Programmiersprachen nicht in der Lage sind, diese Art von Redundanz zu vermeiden [SLL+88].

Überprüfung durch Constraints. Folgender Constraint, der an die ‘Node’-Klasse annotiert werden muss, kann die Integrität einer Implementierung des Visitor-Patterns überprüfen:

$$\textbf{Decl: } Node \quad \textbf{Use: } \underline{ClsInh}$$

$$\begin{aligned}
\textbf{Ctxt:} \quad & (\exists m \in NodeVisitor.\underline{MethDecl} : m.\underline{Name} = \text{“}Visit\text{”} + use.\underline{Name} \\
& \wedge m.\underline{ParamDecl}[0].Type.\underline{Name} = use.\underline{Name} \\
& \wedge \#m.\underline{ParamDecl} = 1 \wedge m.\underline{ParamDecl}[0].\underline{Name} = v) \\
\wedge\ (\exists a \in use.Methods : & \ a.\underline{Name} = \text{“}Accept\text{”} \wedge a.\underline{ParamDecl}[0].Type.\underline{Name} = \text{“}NodeVisitor\text{”} \\
& \wedge \exists inv \in! \underline{MethInv} : inv \sqsubset a.\underline{StmtBlock} \\
& \wedge inv.\underline{ObjName} = v \wedge inv.\underline{Name} = \text{“}Visit\text{”} + use.\underline{Name})
\end{aligned}$$

Dieser Constraint stellt sicher, dass für jeden Knotentypen eine ‘visit’-Methode (mit passendem Namen und Parameter) in der ‘NodeVisitor’-Klasse definiert ist. Darüber hinaus wird überprüft, dass die Unterklassen von ‘Node’ die ‘accept’-Methode richtig implementieren, so dass auf dem als Parameter deklarierten ‘Visitor’-Objekt die entsprechende ‘visit’-Methode aufgerufen wird.

11.3. Systemspezifische Muster

In den beiden vorangehenden Abschnitten wurden bekannte Entwurfsmuster vorgestellt, die bestimmte Strukturen in Softwaresystemen vorsehen, die im Lauf der Systemevolution konsistent gehalten werden müssen. Anhand von Signalwörtern wie 'Visitor' oder auch 'Observer' ist es durchaus wahrscheinlich, dass ein Entwickler diese bekannten Muster wiedererkennt und seinen Code adäquat in diese integriert. Dieser Wiedererkennungseffekt ist bei Mustern, die Entwickler nicht bereits während ihrer Ausbildung kennen gelernt haben, nicht gegeben. Dennoch verbergen sich in jedem größeren System spezifische Muster, die ein Entwickler kennen muss, um Änderungen in adäquater Weise durchführen zu können.

Eine nicht-eindeutige Abbildung eines Konzepts auf mehrere Implementierungskonzepte ist in folgendem Beispiel zu sehen, das aus dem Java Parser der Eclipse Entwicklungsumgebung (JDT) entnommen wurde. Die zentrale Klasse im Objektmodell des Java Parsers von Eclipse ist 'ASTNode'. Sie repräsentiert das abstrakte Konzept eines Knotens in einem Syntaxbaum. Alle Klassen, die verschiedene Arten von Java-Sprachkonstrukten repräsentieren, wie beispielsweise 'MethodDeclaration', 'Assignment' u.a., sind als Unterklassen von 'ASTNode' implementiert. Für einen Parser ist es nicht ungewöhnlich, dass während der Weiterentwicklung der von ihm akzeptieren Sprache (in diesem Fall Java), neue Sprachkonstrukte integriert werden müssen. Die Architektur eines (handgeschriebenen) Parsers sollte somit eine möglichst einfache Änderbarkeit in diese Richtung ermöglichen. Ein Kommentar in der 'ASTNode' Klasse beschreibt, was in einem derartigen Erweiterungsfall (Erweiterung um das Sprachkonstrukt 'FooBar') zu tun ist:

1. Create the FooBar AST node type class.
2. Add node type constant ASTNode.FOO_BAR.
3. Add entry to ASTNode.nodeClassForType(int).
4. Add AST.newFooBar() factory method.
5. Add ASTVisitor.visit(FooBar) and endVisit(FooBar) methods.
6. Add ASTMatcher.match(FooBar,Object) method.
7. Ensure that SimpleName.isDeclaration() covers FooBar nodes if required.
8. Add NaiveASTFlattener.visit(FooBar) method to illustrate how these nodes should be serialized.
9. Update the AST test suites.

Man sieht, dass insgesamt fünf Klassen für jede Unterklasse von 'ASTNode' angepasst werden müssen. Sollte es im Falle der Erweiterung des Parsers um ein neues Sprachkonstrukt vergessen werden, eine der Anpassungen durchzuführen, führt dies zu Fehlverhalten in den Funktionen des Parsers. Man sieht an diesem Beispiel sehr deutlich, dass das Konzept des Knotens im abstrakten Syntaxbaum nicht nur in der 'ASTNode'-Klasse implementiert ist (Delokalisierung, vgl. [SLL+88]). Durch die konsistent zu haltenden Nachbarklassen entsteht eine

gewisse Redundanz der Informationen über die Arten von Knoten im AST. Obwohl die Klassen 'AST', 'ASTVisitor', 'ASTMatcher', 'SimpleName' und 'NaiveASTFlattener' andere Konzepte repräsentieren, ist in ihnen doch die Information zu finden, welche Arten von AST-Knoten existieren können. Die Abstraktionsmechanismen von Java lassen es nicht einfach zu, diese Information redundanzfrei darzustellen.

Constraints, wie sie in Abschnitt 5.2 eingeführt wurden, machen die in einer Implementierung nur implizit repräsentierten Relationen zwischen den Konzepten explizit. Somit stellen Constraints die Festlegungen im Quellcode dar, die während dem Übergang zur Implementierung implizit geworden sind. Obwohl Redundanzen, wie sie in der 'ASTNode' Implementierung zu finden sind, prinzipiell auch durch Constraints ausgedrückt und überprüft werden können, stellen sie Dublikate von Informationen dar, die an mehreren Stellen im Code auftreten. Sie blähen den Code auf, verkomplizieren ihn und erschweren seine Wartung. Bevor Constraints für diesen Zweck eingesetzt werden, sollte deshalb immer überprüft werden, ob nicht durch Refactoring [FBB+99] eine redundanzfreie oder -ärmere Implementierungslösung gefunden werden kann.

Ein Constraint wie der Folgende könnte die Konsistenz der Implementierung der 'ASTNode'-Unterklassen automatisch überprüfen (nur die Punkte 2 bis 4 sind ausformuliert):

$$\textbf{Decl: } org.eclipse.jdt.dom.ASTNode \quad \textbf{Use: } \underline{ClsInh}$$
$$\textbf{Ctxt: } (\exists field \in ASTNode.\underline{FieldDecl}:$$
$$field.Name = reformat(usage.\underline{Name}) \ \wedge \ field.isFinal) \wedge$$
$$(\exists case \in ASTNode.nodeClassForType(int):$$
$$case.\underline{Expr} = reformat(usage.\underline{Name})) \wedge$$
$$(\exists method \in AST.\underline{MethDecl}:$$
$$method.\underline{Name} = \text{``}new'' + use.\underline{Name}) \wedge$$
$$...$$

Für die Durchführung von Wartungsarbeiten und Änderungen an einem System ist es wichtig, dass diese Strukturen identifiziert werden (Verstehen des Implementierungsmodells), um keine Fehler in das System zu induzieren. Durch die Einführung von Constraints kann diese Art der impliziten Festlegungen explizit gemacht und damit Fehlern vorgebeugt werden.

Teil IV.

Zusammenfassung und Ausblick

12. Zusammenfassung

Entstehung impliziter Entwurfsregeln. Im Rahmen dieser Arbeit wurde die Entstehung von impliziten Entwurfsregeln untersucht. Anhand der Analyse des Informationsgehalts von Modellen und von entsprechenden Implementierungen wurde aufgezeigt, wie im Lauf der Systementwicklung von der Konzeption eines Systems über den Entwurf bis hin zur Implementierung implizite Regeln entstehen. Anhand der formalen Definition von Konzept- und Implementierungsmodellen wurde ein tieferes Verständnis für die Entstehung impliziter Entwurfsregeln gewonnen.

Definition eines formalen Frameworks zur Spezifikation von Constraints. Um Verletzungen von Entwurfsregeln entgegen zu wirken, wurde im Rahmen dieser Arbeit ein formales Framework definiert, das es ermöglicht, Entwurfsregeln explizit und in nachprüfbarer Form zu erfassen. Durch dieses Framework können Constraints in Form von syntaktischen Mustern definiert werden, die festlegen, wie Deklarationen in Programmen genutzt werden sollen. Nutzer dieser Deklarationen werden somit in die Lage versetzt, zu überprüfen, ob ihr Code den Vorgaben der Ersteller der Deklarationen gerecht wird.

Methodische Anwendung von Constraints und Werkzeugunterstützung. Neben der formalen Spezifikation von Constraints wurde erläutert, wie Entwurfsregeln durch Nutzung von Modellen oder Annotationen im Quellcode ausgedrückt werden können. Es wurde die Architektur eines erweiterbaren Werkzeugs vorgestellt, das es ermöglicht, den Quellcode eines Programms auf Konformität mit Constraints zu überprüfen.

Anhand von drei Anwendungsbereichen wurde dargestellt, wie diese Konzepte in der Praxis eingesetzt werden können. Zudem wurden empirische Studien bezüglich impliziter Entwurfsregeln in Programmen durchgeführt:

Implizite Vorgaben in objektorientierten APIs. In einer Fallstudie auf Basis der Java-Standard-API wurde aufgezeigt, dass darin zahlreiche Regeln zu finden sind, die durch Nutzercode verletzt werden, obwohl diese dokumentiert waren. Diese Entwurfsregeln stellen Vorgaben an den Kontroll- und Datenfluss, Vererbungsbeziehungen und die Struktur von Nutzerprogrammen dar. Dies zeigt, dass die natürlich-sprachliche Dokumentation dieser Vorgaben nicht ausgereicht hat, um die Konformität des Nutzercodes zu diesen Vorgaben zu erzielen.

Implizites Architekturwissen. In dieser Arbeit wurde anhand einer Fallstudie auf Basis von drei industriellen Projekten gezeigt, inwiefern die Architektur eines Systems implizite Vorgaben in einem Softwaresystem darstellt. Es wurde quantifiziert, in welchem Maße diese Vorgaben über den Lauf der Systemevolution hinweg eingehalten wurden und wie viele Architekturverletzungen entstanden sind. Damit konnte der Verlust an Wissen über die vorgesehene Architektur nachgewiesen werden. Dieser Wissensverlust ist der Auslöser für aufwändige Reverse Engineering Aktivitäten, um schließlich diese Informationen wiederzugewinnen.

Implizite Festlegungen im Softwareentwurf. Es wurde gezeigt, dass sowohl verbreitete und anerkannte Entwurfsmuster als auch systemspezifische Muster mit impliziten Entwurfsregeln einhergehen. Zudem wurde präsentiert wie diese Festlegungen durch Constraints explizit und überprüfbar gemacht werden können.

13. Ausblick: Erweiterbare Sprachen

"Sprachen sind der beste Spiegel des menschlichen Geistes."
- Gottfried Wilhelm Leibniz

Modellierung von Konzepten durch Abstraktionsmechanismen. Softwareentwicklung ist immer mit der Suche nach geeigneten Ausdrucksmitteln verbunden, mit denen sich eine Problemstellung möglichst gut und effizient beschreiben lässt. Eine gute Beschreibung zeichnet sich unter anderem dadurch aus, dass sie möglichst wenige implizite Festlegungen enthält, die zu Fehlinterpretationen führen und damit eine potentielle Fehlerquelle darstellen würden. Gerade bei Bibliotheken und Frameworks, die durch eine Vielzahl an Entwicklern verwendet werden, ist es lohnenswert, diese Fehlerquellen zu begrenzen. Die meisten Bibliotheken und Frameworks werden heutzutage in Allzwecksprachen wie C, Java oder C# implementiert. In derartigen Programmiersprachen haben Entwickler die Möglichkeiten, vielfältige Abstraktionsmechanismen zu nutzen, um durch Deklarationen zu implementieren und damit gewissermaßen ein Vokabular zu definieren, das API-Nutzer verwenden können, um ihre Systemfunktionalität zu beschreiben.

Jedoch wird die syntaktische Form der verwendeten Abstraktionsmechanismen und die dafür in der Programmiersprache zur Verfügung stehenden Nutzungsmöglichkeiten der eigentlichen vorgesehenen Nutzung der Deklarationen meist nicht derart gerecht, dass implizite Entwurfsregeln vermieden werden können. Die Syntax der Programmiersprache bietet den Nutzern der Deklarationen eine festgelegte Menge an Nutzungsmöglichkeiten der Deklarationen, obwohl unter Umständen ein großer Teil dieser Möglichkeiten für konkrete Deklarationen nicht sinnvoll sind und eine nicht-adäquate Nutzung darstellen. Aus diesem Grund wurde in Abschnitt 5 dieser Arbeit das formale Framework zur Definition von Constraints eingeführt, das es Entwicklern erlaubt, den von ihnen implementierten Konzepten zusätzliche Informationen über deren adäquate Nutzung zu annotieren. Dadurch wird der Spielraum eingeschränkt, in dem andere Entwickler die Konzepte verwenden können. Sie können überprüfen, ob sie die Konzepte konform zur Intention des Erstellers nutzen. Der Mechanismus der Constraints ist somit eine zusätzliche syntaktische Restriktion der Verwendung von Deklarationen.

Wiederverwendung durch Spracherweiterung. Diese Möglichkeiten, die Überprüfungen der syntaktischen Korrektheit eines Compilers um zusätzliche statische Prüfungen der adäquaten Nutzung von Deklarationen zu erweitern, sind direkt mit heutigen Programmiersprachen einsetzbar und können sich als nützliche Hilfen in der Softwareevolution erweisen (vgl. Kapitel 10,

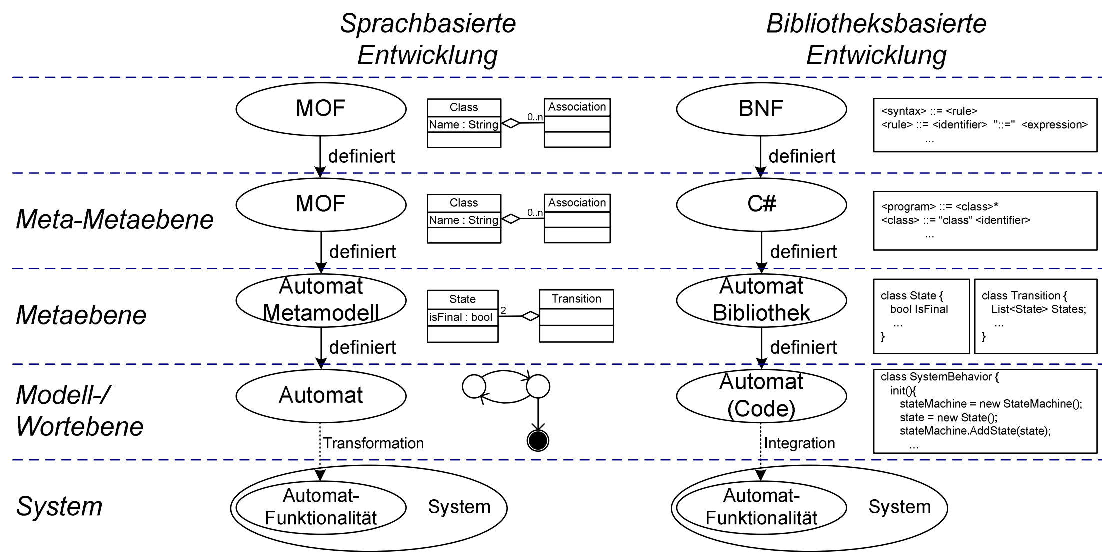


Abbildung 13.1.: Vergleich von sprach- und bibliotheksbasierter Entwicklung

9 und 11). In gewisser Weise handelt es sich dabei jedoch nur um eine Behandlung der Symptome. Der eigentliche Grund für die Entstehung impliziter Entwurfsregeln ist an der unzureichenden Ausdrucksfähigkeit der Abstraktionsmechanismen zu finden. In Kapitel 4 wurde aufgezeigt, dass bestimmte Modelle oftmals in der Lage sind, Informationen über das System (Datenstrukturen, Verhalten, statische Struktur) derart darzustellen, dass diese weniger implizite Entwurfsregeln enthalten, als eine vergleichbare Implementierung in einer objektorientierten Programmiersprache. Somit stellen diese Modellierungstechniken im Bezug auf die Explizitheit der Darstellung ein besseres Beschreibungsmittel dar.

Domänenspezifische Sprachen. Durch die Entwicklung von domänenspezifischen Sprachen (DSLs) als Ausdrucksmittel für spezifische Domänen für die heutzutage Bibliotheken und Frameworks zum Einsatz kommen, ist es möglich, diesen eine passende syntaktische Form zu geben, welche die adäquate Nutzung der darin implementierten Konstrukte sicherstellt. In der Literatur sind viele Vorteile von domänenspezifischen Sprachen beschrieben (vgl. [vDKV00]), die im Kern auf eine im Vergleich zu einer Programmiersprache explizitere Darstellung der Konzepte zurückzuführen sind. Derartige Techniken haben das Potential, durch kürzere und präzisere Beschreibungen (Programme) die Verständlichkeit zu erhöhen, da weniger Hilfskonzepte benötigt werden, wie sie in der klassischen Programmierung häufig anzutreffen sind [vDKV00]. Damit könnten Effekte, wie die Delocalisation und das Interleaving von Konzepten [SLL+88, RSW95], die das Programmverstehen erschweren, besser in den Griff bekommen werden.

Symmetrie der Sprachentwicklung zur Bibliotheksentwicklung Abbildung 13.1 zeigt beispielhaft eine Gegenüberstellung von sprach- und bibliotheksbasierter Entwicklung. Dabei wird eine Teilfunktionalität eines Systems durch einen Zustandsautomaten beschrieben. Dieser Zustandsautomat wird auf der linken Seite durch sprachbasierte Abstraktion realisiert und auf der rechten Seite durch eine Bibliothek in der Programmiersprache C#. Die beschrifteten Ebenen entsprechen den klassischen Ebenen der vierstufigen Metahierarchie [OMG06a]. Auf der Metaebene wird im Falle der sprachbasierten Entwicklung MOF genutzt, um ein Metamodell eines Automaten zu definieren. Dem steht auf Seiten der bibliotheksbasierten Entwicklung eine C# Bibliothek gegenüber, welche die Funktionalität eines Automaten in sich kapselt. Auf der Modell- oder auch Wortebene wird im Fall der sprachbasierten Entwicklung ein Automat für das konkret zu entwickelnde System spezifiziert. Auf der rechten Seite muss analog dazu die generisch gehaltene Klassenbibliothek verwendet werden, um den systemspezifischen Automaten zu instanziieren. Aus diesen beiden Spezifikationen kann schließlich das automatenspezifische (Teil-)System erstellt werden. Der unterste im Bild dargestellte Abschnitt skizziert schließlich das fertige System (welches grundsätzlich in beiden Fällen sogar den exakt gleichen Code enthalten könnte). In der Abbildung wurde auf die Bezeichnung "System*ebene*" verzichtet, da es sich streng genommen im Gegensatz zu den anderen (echten) Metaebenen nicht um eine Instanziierung, sondern um eine Transformation bzw. eine Integration handelt. Der letzte Transformations- oder auch Codegenerierungsschritt erzeugt schließlich eine Code-basierte Repräsentation (ggf. in C#) des Automaten, die der Metaebene und Modellebene auf der Seite der bibliotheksbasierten Entwicklung entspricht.

Während Bibliotheken als Technik, um Wiederverwendung zu erzielen, bekannt sind, ist dies bei Sprachen weniger selbstverständlich. Natürlich kann man auch Sprachen wiederverwenden und damit die in ihnen implementierten Konzepte. Im Gegensatz zu Bibliotheken sind die Konzepte jedoch in der Syntax der Sprache gefangen. Eine Komposition mit anderen Konzepten in anderen Sprachen ist nicht einfach möglich. Heutzutage ist oftmals eine Art 'Hybridstrategie' vorzufinden. Es werden spezifische Sprachen eingesetzt und daraus Code generiert. Die generierten Systemteile werden dann durch manuelle Programmierung integriert. Obwohl dies einen pragmatischen Ansatz darstellt, wird dabei nicht der volle Nutzen einer DSL entfaltet, da Entwickler nicht nur die Sprache selbst, sondern auch die Repräsentation der Konzepte im generierten Code verstehen müssen. Dadurch entstehen wiederum Situationen, in denen die Kodierung der Konzepte in der Zielsprache missinterpretiert werden kann, damit nicht adäquate Nutzung ermöglicht wird und somit (latente) Fehler in das System induziert werden [Fei07]. Viele DSLs sind als relativ dünne Schichten über einem darunterliegenden Framework implementiert, die dazu dienen, Implementierungsdetails des Frameworks zu verbergen [BM06]. Ein Generator muss in diesem Fall lediglich die Konzepte der Sprache auf die entsprechenden Implementierungskonzepte im Framework abbilden. Nach [RJ96] kann dadurch eine einfachere Verwendung der in einem Framework implementierten Konzepte erzielt werden.

Mangel an Kompositionalität zwischen Sprachen. Um die Vorteile der Entwicklung von domänenspezifischen Sprachen als Alternative zu Frameworks und Bibliotheken praktisch nutzbar zu machen, muss die mangelnde Kompositionalität von Sprachen überwunden werden. Heutige Softwaresysteme nutzen meist eine Vielzahl an Bibliotheken und Frameworks. Heu-

tige Ansätze aus dem Bereich der domänenspezifischen Sprachen bieten jedoch noch keine geeigneten Mechanismen, verschiedene Sprachen, die unterschiedliche Domänen adressieren, derart miteinander zu verknüpfen, dass daraus eine Beschreibungstechnik entsteht, die zur Programmentwicklung geeignet ist. Ein Mechanismus, der es ermöglicht, die Konstrukte einer Allzwecksprache um zusätzliche domänenspezifische Sprachbausteine zu erweitern, könnte eine ganzheitliche Lösung der in dieser Arbeit vorgestellten Problemstellung des Verlusts an Explizitheit bieten.

Die Entwicklung von Spracherweiterungen ist jedoch gegenüber der Definition von Constraints, wie sie in dieser Arbeit vorgestellt wurden, ein deutlich schwergewichtigerer Ansatz, der mit derzeitigen Mitteln meist deutlich aufwändiger ist als eine klassische Implementierung von Konzepten. Dieser Aufwand muss sich durch ein entsprechend hohes Wiederverwendungspotential auszahlen. Seitens der Entwickler fordert ein derartiger Ansatz neue Fähigkeiten. Durch bessere Techniken und Methoden zur Entwicklung von Sprachbausteinen könnte eine Alternative zur Erstellung von Bibliotheken und Frameworks geschaffen werden.

Beispiel: Hardwarenahe APIs. Eine Beschreibung von Systemen auf höherem Abstraktionsniveau als es heutige Programmiersprachen bieten, kann das Volumen der Beschreibung reduzieren und damit die Entwicklung vereinfachen. Die modellbasierte Entwicklung hat das Ziel, die Entwicklung auf ein höheres Abstraktionsniveau zu heben [BFH+10, Bro06a, Bro06b]. In Abbildung 13.2 wird dargestellt, wie in [Sch08] eine Integration neuer Sichten in das CASE-Tool AutoFocus vorgenommen wurde, um eine einfachere Nutzung von hardwarenahen Schnittstellen zu ermöglichen. Aus diesen Modellen kann der komplette Code (konform zu allen impliziten Regeln) generiert werden, der notwendig ist, um auf Aktuatorik, Sensorik und auf Bussysteme zuzugreifen. Die Nutzung von Peripheriekomponenten wird dadurch deutlich intuitiver, da die Modelle mit weniger impliziten Regeln durchsetzt sind, als APIs, welche deren Ansteuerung ermöglichen. Dies wird auch in der Symmetrie zwischen dem Konzept- und dem Implementierungsmodell (der neuen Sicht in AutoFocus) deutlich. Neue Arten von Hardware-Aktuatoren, Sensoren etc. können über einen PlugIn-Mechanismus in diese Sichten integriert werden und sogar eigene Sichten (etwa zur Konfiguration) zur Gesamtsprache beitragen.

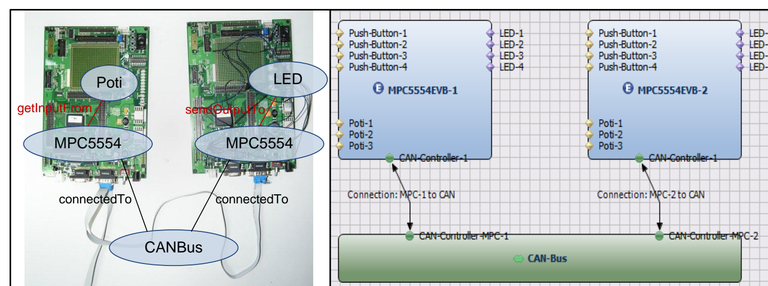


Abbildung 13.2.: Symmetrie zwischen Realität/Konzeptmodell und Modellierungssprache

Konstruktive Absicherung impliziter Entwurfsregeln. Anstatt durch Constraints die adäquate Nutzung *analytisch* zu überprüfen, kann durch die konsistente Generierung *konstruktiv* sichergestellt werden, dass die in den APIs vorhanden impliziten Vorgaben eingehalten werden. Abbildung 13.3 zeigt ein Stück Code, das aus den neuen AutoFocus-Sichten generiert wurde. An den farbigen Hervorhebungen ist deutlich zu erkennen, dass aus dem gleichen Elementen im Modell unterschiedliche Codeabschnitte generiert werden. Diese Zusammenhänge entsprechen impliziten Entwurfsregeln, die auch bei manueller Implementierung erkannt und eingehalten werden müssten. Durch die konsistente Generierung kann hier ein Fehlerpotential vermieden werden.

Generiert durch:
- Logical Architecture Generator
- Poti Generator
- Servo Generator

```
#include "os.h"
#include "poti.h"
#include "servo.h"
#include "C1.h"

/*
 * The initialize The OS of ECU "MPC5554".
 */
void StartupHook() {
    /* Initialize hardware components. */
    poti_init();
    servo_init();

    /* Initialize logical components. */
    initEnvironment_C1();
}

/*
 * The environment read-function of logical component "C1".
 */
void readEnvironment_C1() {
    /* Read sensor values and write to logical components. */
    set_INPUT_PORT_C1_Input(poti_read());
}

/*
 * The environment write-function of logical component "C1".
 */
void writeEnvironment_C1() {
    /* Read outputs from logical components and write to actuators. */
    servo_set_duty(get_OUTPUT_PORT_C1_Output());
}

/* The OSEK-Task declaration. */
TASK(MPC5554_Task) {
    /* Run the logical components. */
    readEnvironment_C1();
    doStep_C1();
    writeEnvironment_C1();

    TerminateTask();
}
```

Abbildung 13.3.: Beiträge der einzelnen PlugIns zum Code eines Steuergeräts

Arbeiten im Bereich der Sprachkomposition. In der Literatur sind bereits einige Ansätze zur Komposition bzw. zur Erweiterung von Sprachen beschrieben: MOF [OMG06a] bietet bereits einfache Operatoren zur Komposition von Modellierungssprachen, wie das Importieren

und Kombinieren von Paketen. In [BRR05] wird der Bedarf für einen neuen Operator motiviert, der es ermöglicht, Konzepte zur Kombination von Paketen wiederzuverwenden und zu generalisieren. Die Autoren von [CEK02] definieren einen neuen Kompositionsoperator, der es erlaubt, Konzepte gleichzusetzen und dadurch bei der Komposition von Paketen als ein Element zu behandeln. Karsai stellt feingranularere Operatoren vor, wie das Zusammenfassen zweier Konzepte oder spezifischere Vererbungsmechanismen [KML+03]. In [BSML07] wird gezeigt, wie diese Operatoren angewandt werden können, um existierende modellbasierte Werkzeuge zu integrieren. Emerson und Sztipanovits fordern Metamodel-Templates, die eine flexiblere Generalisierung und Anpassung von Modellierungstechniken ermöglichen [ES06]. Auf Basis textueller Sprachen beschreibt van Wyk [VWBH06, VWKSB07] einen Ansatz zur Integration neuer Konstrukte. MetaBorg ist ein Ansatz zur Komposition von Sprachmodulen, die auf einem eigenen Metamodellierungsverfahren beruhen [Rie06].

Alle diese Kompositionsansätze definieren spezielle Techniken, um Metamodelle oder Grammatiken komponieren zu können. Gerade die Operatoren zur Definition von Generalisierungen und Spezialisierungen von Metamodellelementen über Grenzen eines Sprachbausteins hinweg sind dabei vielversprechende Strategien. Die meisten Arbeiten konzentrieren sich jedoch lediglich auf die Komposition von Metamodellen und übersehen dabei, dass nicht nur die abstrakte Syntax komponiert werden sollte, sondern auch die konkrete Syntax sowie Transformationen (Generatoren) bei der Komposition berücksichtigt werden sollten. Estublier [EVI05] adressiert dieses Problem und stellt ein Prinzip von komponierbaren Modellinterpretern vor, die aus Basis aspektorientierter Programmierung realisiert werden.

Zukünftige Forschungsthemen. Die konstruktive Vermeidung impliziter Entwurfsregeln durch Erweiterung von Sprachen hat das Potential langfristig einen ganzheitlicheren Ansatz zur expliziteren Modellierung von Softwaresystemen hervorzubringen. Die Entwicklung von Sprachen ist jedoch immer noch ein Gebiet, das im Software Engineering zu wenig Beachtung findet. Künftige Arbeiten müssen für diese Disziplin einen methodischen und technischen Unterbau bereitstellen, um die Konzeption und Entwicklung von Sprachen professioneller vornehmen und Sprachbausteine besser miteinander integrieren zu können. Über die in diesem Ausblick skizzierten Vorstellungen hinaus, ergeben aus Sicht dieser Arbeit im Bereich der Entwicklung von Sprachbausteinen folgende offene Forschungsfragestellungen:

- Wie können Sprachbausteine definiert und zu einer ganzheitlichen Sprache komponiert werden?
- Welche Methodik ist geeignet, um Sprachbausteine zu entwickeln? – Wie unterscheidet sich die Entwicklung von Sprachbausteinen von der Entwicklung von APIs?
- Mit welchen Mehraufwänden ist die Entwicklung von Sprachbausteinen verbunden?
- Welche Vorteile bringen Sprachbausteine im Gegensatz zu Bibliotheken/Frameworks mit sich?

Glossar

Begriff	Beschreibung	
Abstraktionsmechanismus	Ein Abstraktionsmechanismus ist ein Mechanismus einer Programmiersprache, der es Programmierern ermöglicht, eine bestimmte Art von Deklarationen (Klassen, Methoden o.ä.) zu definieren. Im Rahmen dieser Arbeit werden lediglich Abstraktionsmechanismen betrachtet, die der Abstraktion einen Bezeichner zuweisen, um damit einen symbolischen Namen zu definieren. Diese Deklarationen können von verschiedenen Stellen im Programm auf verschiedene Weisen (z.B. bei Klassendeklarationen: Unterklassenbildung, Instanziierung etc.) genutzt werden und bilden damit eine Basis für die Wiederverwendung von Programmlogik.	48, 52, 74
Adäquate Verwendung	Die adäquate Verwendung einer Deklaration entspricht einer Nutzung bei der alle Vorgaben eingehalten werden, die seitens des Erstellers festgelegt wurden. Falls Vorgaben existieren, welche die Nutzung einer Deklaration einschränken (z.B. in Form von Constraints), dann entspricht die adäquate Verwendung einer Deklaration deren Nutzung auf eine zu den Vorgaben konforme Weise.	9, 74
Constraint	Ein Constraint ist ein syntaktisches Muster, das definiert, wie eine Deklaration in einem Programm genutzt werden darf (Definition der adäquaten Verwendung einer Deklaration). Dadurch können implizite Festlegungen, wie beispielsweise Entwurfs- oder Architekturvorgaben, explizit gemacht werden. Die Nutzung einer Deklaration kann schließich anhand des syntaktischen Musters, das durch einen Constraint festgelegt wird, auf Konformität bezüglich dieser Vorgaben überprüft werden.	74

Begriff	**Beschreibung**	
Deklaration	Eine Deklaration ist eine Einführung eines Bezeichners in ein Programm. In objektorientierten Programmiersprachen (z.B. Java) sind die wichtigsten Arten von Deklarationen beispielsweise Klassen-, Methoden- oder Attributdeklarationen.	23, 74
Entwurfsregel, implizite	Entwurfsregeln sind Bedingungen, die an die Wohlgeformtheit von Programmen gestellt werden. Sie umfassen etwa Regeln bzgl. der vorgesehenen Architektur (statische Struktur), der Einhaltung bestimmter Entwurfsmuster und andere Konventionen, die im Code eines Systems beachtet werden müssen. Diese Regeln sind im Code eines Systems implizit, wenn sie darin nicht direkt ablesbar sind. Auch Bibliotheken und Frameworks enthalten (implizite) Entwurfsregeln, die API-Nutzern Vorgaben bzgl. der adäquaten Nutzung der Schnittstellen auferlegen (vgl. Abschnitt 9.1).	19, 50
Implementierungsmodell	Jedes Artefakt, das in einem Entwicklungsprozess erstellt wird, stellt ein Implementierungsmodell eines Konzeptmodells dar. Das bedeutet, dass Entwickler durch ein Artefakt bestimmte Teile ihrer Vorstellungen (Konzeptmodell) bezüglich des zu erstellenden Systems durch Nutzung einer formalen Sprache (Modellierungs- oder Programmiersprache) Ausdruck verleihen. Im Rahmen dieser Arbeit werden lediglich Implementierungmodelle betrachtet, die eine durch eine (formale) Grammatik oder ein Metamodell festgelegte Syntax aufweisen (kein natürlich-sprachlicher Text). Anhand der Gegenüberstellung eines Konzeptmodells mit einem Implementierungsmodell kann überprüft werden, inwieweit das Implementierungsmodell die Informationen des Konzeptmodells explizit wiedergibt und inwieweit das Implementierungsmodell impliziten Vorgaben unterliegt.	48
Konstrukt	Im Rahmen dieser Arbeit wird der Begriff des Konstrukts für die in der Grammatik einer textuellen Sprache definierten Variablen (Nonterminale) oder die Entitäten (Klassen) eines Metamodells einer graphischen Sprache gebraucht.	23, 71

Begriff	Beschreibung	
Konzeptmodell	Im Rahmen dieser Arbeit ist ein Konzeptmodell als Multigraph definiert, dessen Knoten und Kanten mit natürlich-sprachlichen Bezeichnern versehen sind (analog zu leichtgewichtigen Ontologien). Konzeptmodelle werden im Rahmen dieser Arbeit verwendet, um die Vorstellungen eines Entwicklers bezüglich des zu erstellenden Systems darzustellen. Damit ist es möglich die Existenz von impliziten Festlegungen zu veranschaulichen. Ein Konzeptmodelle entspricht dem mentalen Modell eines Entwicklers und liegt nicht als Artefakt in einem Entwicklungsprozess vor. Die in Konzeptmodellen enthaltenen Informationen werden jedoch auf unterschiedliche Art und Weise in Artefakten ausgedrückt (z.B. in der Dokumentation oder im Code).	45
Konzept	Konzepte (intellektuelle Konzepte) repräsentieren im Rahmen dieser Arbeit die Vorstellungen eines Entwicklers bezüglich des zu erstellenden Systems. Konzepte sind die Basiselemente eines Konzeptmodells (Knoten des Graphen). Folgende Definitionen aus der Literatur entsprechen dem in dieser Arbeit verwendeten Konzept-Begriff: Unter einem intellektuellen Konzept wird eine Abstraktion verstanden, die normalerweise nicht greifbar ist, wie eine Idee, eine Vorstellung, ein Gedankenkonstrukt [Bjø06]. Konzepte sind Einheiten von menschlichem Wissen, die durch den menschlichen Verstand (Kurzzeitgedächtnis) in einer Instanz verarbeitet werden können [RW02].	22, 45
Vorgabe, implizite	Vorgaben sind Entwurfsregeln, welche die Möglichkeiten der Verwendung einer Deklaration in einem System einschränken. Diese Vorgaben sind im Code eines Systems implizit, wenn sie darin nicht direkt ablesbar sind. Auch APIs enthalten (implizite) Vorgaben, die API-Nutzer einhalten müssen, um die Schnittstellen adäquat zu verwenden (vgl. Abschnitt 9.1).	19, 50, 104

Literaturverzeichnis

[ABL02] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. *SIGPLAN Notices*, 37(1):4–16, 2002.

[AdOD+03] Nicolas Anquetil, Káthia Marçal de Oliveira, Márcio Greyck Batista Dias, Marcelo Ramal, and Ricardo de Moura Meneses. Knowledge for software maintenance. In *Journal of Information Technology*, pages 61–68, 2003.

[AdOdSD07] Nicolas Anquetil, Káthia Marçal de Oliveira, Kleiber D. de Sousa, and Márcio Greyck Batista Dias. Software maintenance seen as a knowledge management issue. *Journal of Information & Software Technologies*, 2007.

[AL98] Nicolas Anquetil and Timothy Lethbridge. Assessing the relevance of identifier names in a legacy software system. In *Proceedings of the 1998 conference of the Centre for Advanced Studies on Collaborative research*, page 4. IBM Press, 1998.

[ANMM06] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. *SIGPLAN Notices*, 41(10):57–74, 2006.

[AP04] Marcus Alanen and Ivan Porres. A relation between context-free grammars and meta object facility metamodels. Technical Report 606, TUCS, Mar. 2004.

[Bal01] Helmut Balzert. *Lehrbuch der Software-Technik: Software-Entwicklung.* Lehrbücher der Informatik. Spektrum Akademischer Verlag, Heidelberg, 2nd ed. edition, 2001.

[BdBDF07] Muhammad Ali Babar, Remco C. de Boer, Torgeir Dingsoyr, and Rik Farenhorst. Architectural knowlege management strategies: Approaches in research and industry. In *SHARK-ADI*, page 2, Washington, DC, USA, 2007. IEEE Computer Society.

[BDW79] T.G.R. Bower, Jane Dunkeld, and Jennifer G. Wishart. Infant perception of visually presented objects. *Science, New York, N.Y.)*, pages 1138–1139, March 1979.

[Ben86] Jon Bentley. Programming pearls: little languages. *Communications of the ACM*, 29(8):711–721, 1986.

[BFG+08] Manfred Broy, Martin Feilkas, Johannes Grünbauer, Alexander Gruler, Alexander Harhurin, Judith Hartmann, Birgit Penzenstadler, Bernhard Schätz, and Doris Wild. Umfassendes architekturmodell für das engineering eingebetteter

software-intensiver systeme. Technical Report TUM-I0816, Fakultät für Informatik, Technische Universität München, Feb. 2008.

[BFH+10] Manfred Broy, Martin Feilkas, Markus Herrmannsdoerfer, Stefano Merenda, and Daniel Ratiu. Seamless model-based development: from isolated tools to integrated model engineering environments. *Proceedings of the IEEE - Special Issue on Aerospace & Automotive*, To appear 2010.

[Big89] Ted J. Biggerstaff. Design recovery for maintenance and reuse. *Computer*, 22(7):36–49, 1989.

[Bjø06] D. Bjørner. *Software Engineering 3: Domains, Requirements and Software Design.* Teksts in Theoretical Computer Science. Springer, 2006.

[BKL04] Walter Bischofberger1, Jan Kühl, and Silvio Löffler. Sotograph - a pragmatic approach to source code architecture conformance checking. *IEEE Transactions on Software Engineering*, 3047(1):1–9, 2004.

[BL76] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[BM06] Thomas Büchner and Florian Matthes. Introspective model-driven development. *Lecture Notes in Computer Science*, pages 33–49, 2006.

[BMW93] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas Webster. The concept assignment problem in program understanding. In *Proceedings of the 15th international conference on Software Engineering.* IEEE Computer Society, 1993.

[BMW94] Ted J. Biggerstaff, Bharat G. Mitbander, and Dallas E. Webster. Program understanding and the concept assignment problem. *Communications of the ACM*, 37(5):72–82, 1994.

[Boe91] Barry W. Boehm. Software risk management: Principles and practices. *IEEE Software*, 8(1):32–41, 1991.

[BP78] Wolfram Bartussek and David Lorge Parnas. Using assertions about traces to write abstract specifications for software modules. In *2nd Conference of the European Cooperation on Informatics*. Springer, 1978.

[BR00] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 73–87, New York, NY, USA, 2000. ACM.

[BR02] Thomas Ball and Sriram K. Rajamani. The slam project: debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–3, New York, NY, USA, 2002. ACM.

[Bra04] Gilad Bracha. Pluggable type systems. In *OOPSLA Workshop on Revival of Dynamic Languages*, Oct. 2004.

[Bri03] Lionel C. Briand. Software documentation: How much is enough? In *Proceedings of the Seventh European Conference on Software Maintenance and Reengineering*, page 13, Washington, DC, USA, 2003. IEEE Computer Society.

[Bro95] Manfred Broy. Mathematical System Models as a Basis of Software Engineering. *Computer Science Today*, pages 292–306, 1995.

[Bro96] Kyle Brown. Design reverse-engineering and automated design-pattern detection in smalltalk. Technical report, North Carolina State University at Raleigh, Raleigh, NC, USA, 1996.

[Bro01] Frederick P. Brooks. *The Mythical Man-Month*, chapter No Silver Bullet - Essence and Accident in Software Engineering, pages 177–203. Addison Wesley Longman Inc., New York, 15. edition edition, 2001.

[Bro06a] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th international conference on Software Engineering*, pages 33–42, New York, NY, USA, 2006. ACM.

[Bro06b] Manfred Broy. The 'grand challenge' in informatics: Engineering software-intensive systems. In *Proceedings of the 30th Annual IEEE/NASA Software Engineering Workshop*, pages 85–88, Washington, DC, USA, 2006. IEEE Computer Society.

[BRR05] Xavier Blanc, Franklin Ramalho, and Jacques Robin. Metamodel reuse with mof. In *MoDELS*, pages 661–675, 2005.

[BSML07] Krishnakumar Balasubramanian, Douglas C. Schmidt, Zoltan Molnar, and Akos Ledeczi. Component-based system integration via (meta)model composition. In *Proceedings of the 14th International Conference on the Engineering of Computer-Based Systems*, pages 93–102, Washington, DC, USA, 2007. IEEE Computer Society.

[Car96] Luca Cardelli. Type systems. *ACM Computing Surveys*, 28(1):263–264, 1996.

[CCI90] Elliot J. Chikofsky and James H. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CEH02] Benjamin Chelf, Dawson Engler, and Seth Hallem. How to write system-specific, static checkers in metal. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*. ACM Press, 2002.

[CEK02] Tony Clark, Andy Evans, and Stuart Kent. A metamodel for package extension with renaming. In *Proceedings of the 5th International Conference on The Unified Modeling Language*, pages 305–320, London, UK, 2002. Springer Verlag.

[CKI88] Bill Curtis, Herb Krasner, and Neil Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, 1988.

[CSAJ05] Kai Chen, Janos Sztipanovits, Sherif Abdelwalhed, and Ethan Jackson. Semantic anchoring with model transformations. In *Model Driven Architecture*, volume 3748 / 2005. First European Conference, ECMDA-FA 2005, Springer Berlin / Heidelberg, Nov. 2005.

[CSN05] Kai Chen, Janos Sztipanovits, and Sandeep Neema. Toward a semantic anchoring infrastructure for domain-specific modeling languages. In *Proceedings of the 5th ACM international conference on Embedded software*, pages 35–43, New York, NY, USA, 2005. ACM Press.

[Dav03] James Davis. Gme: the generic modeling environment. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 82–83, New York, NY, USA, 2003. ACM.

[Des03] Kevin C. Desouza. Facilitating tacit knowledge exchange. *Communication of the ACM*, 46(6):85–88, 2003.

[DF04a] R. DeLine and M. Fahndrich. The fugue protocol checker: Is your software baroque. Technical Report MSR-TR-2004-07, Microsoft Research, 2004.

[DF04b] R. DeLine and M. Fahndrich. Typestates for objects. In *ECOOP*, 2004.

[DJH+08] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. Mas y Parareda, and M. Pizka. Tool support for continuous quality control. *IEEE Software*, 25(5):60–67, 2008.

[DJS09] Christoph Domann, Elmar Juergens, and Jonathan Streit. The curse of copy&paste – Cloning in requirements specifications. In *International Symposium on Empirical Software Engineering and Measurement*, 2009.

[DL99] C. R. Douce and P. J. Layzell. Evolution and errors: An empirical example. In *Proceedings of the IEEE International Conference on Software Maintenance*, page 493, Washington, DC, USA, 1999. IEEE Computer Society.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Technical Report #159, Compaq SRC, 1998.

[DP06] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Control*, 14(3):261–282, 2006.

[DPS05] Florian Deissenboeck, Markus Pizka, and Tilman Seifert. Tool support for continuous quality assessment. In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 127–136, Washington, DC, USA, 2005. IEEE Computer Society.

[dSAdO05] Sergio Cozzetti B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication*, pages 68–75, New York, NY, USA, 2005. ACM.

[dSRC+04] Cleidson R. B. de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson. Sometimes you need to see through walls: a field study of application programming interfaces. In *Proceedings of the ACM conference on Computer supported cooperative work*, pages 63–71, New York, NY, USA, 2004. ACM.

[Dvo94] Joseph Dvorak. Conceptual entropy and its effect on class hierarchies. *Computer*, 27(6):59–63, 1994.

[EGK+01] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[EKKM08] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering*, pages 391–400, New York, NY, USA, 2008. ACM.

[ES06] Matthew Emerson and Janos Sztipanovits. Techniques for metamodel composition. In *6th Workshop on Domain Specific Modeling*, pages 123–139, Oct. 2006.

[ESM05] Michael Eichberg, Thorsten Schäfer, and Mira Mezini. Using annotations to check structural properties of classes. In Maura Cerioli, editor, *FASE*, volume 3442 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2005.

[EVI05] Jacky Estublier, Germán Vega, and Anca Daniela Ionita. Composing domain-specific languages for wide-scope software engineering applications. In *MoDELS*, pages 69–83, 2005.

[EW05] Joerg Evermann and Yair Wand. Toward formalizing domain modeling semantics in language syntax. *IEEE Transactions on Software Engineering*, 31(1):21–37, 2005.

[FA98] R. Fiutem and G. Antoniol. Identifying design-code inconsistencies in object-oriented software: a case study. In *Proceedings of the International Conference on Software Maintenance*, page 94, Washington, DC, USA, 1998. IEEE Computer Society.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional, June 1999.

[Fei06] Martin Feilkas. How to represent models, languages and transformations? In *6th OOPSLA Workshop on Domain Specific Modeling*, Oct. 2006.

[Fei07] Martin Feilkas. Sprach- und bibliotheksbasierte abstraktion. In *Kolloquium Programmiersprachen und Grundlagen der Programmierung*, Okt. 2007.

[FFH+09] M. Feilkas, A. Fleischmann, F. Hölzl, C. Pfaller, K. Scheidemann, M. Spichkova, and D. Trachtenherz. A Top-Down Methodology for the Development of Automotive Software. Technical report, Technische Universität München, 2009.

[FL02] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the ACM symposium on Document engineering*, pages 26–33, New York, NY, USA, 2002. ACM.

[Fou] Eclipse Foundation. Eclipse GMF website.

[Fow02] Martin Fowler. Public versus published interfaces. *IEEE Software*, 2002.

[Fow05] Martin Fowler. Language workbenches: The killer-app for domain specific languages?, May 2005.

[FR08] Martin Feilkas and Daniel Ratiu. Ensuring well-behaved usage of apis through syntactic constraints. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 248–253, Washington, DC, USA, 2008. IEEE Computer Society.

[FRJ09] Martin Feilkas, Daniel Ratiu, and Elmar Jürgens. The loss of architectural knowledge during system evolution: An industrial case study. In *Proceedings of the 16th IEEE International Conference on Program Comprehension*, pages 188–197, Washington, DC, USA, 2009. IEEE Computer Society.

[Gar00] David Garlan. Software architecture: a roadmap. In *Proceedings of the international conference on Software Engineering*, pages 91–101, New York, NY, USA, 2000. ACM.

[GC91] Edward M. Gellenbeck and Curtis R. Cook. An investigation of procedure and variable names as beacons during program comprehension. Technical report, Oregon State University, Corvallis, OR, USA, 1991.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Third Edition: The Java Series.* Addison-Wesley, 2005.

[GN87] Michael R. Genesereth and Nils J. Nilsson. *Logical foundations of artificial intelligence.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987.

[Gru95] Thomas R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. *International Journal on Human-Computer Studies*, 43(5-6):907–928, 1995.

[GS89] Virginia R. Gibson and James A. Senn. System structure and software maintenance performance. *Communications of the ACM*, 32(3):347–358, 1989.

[GS06] Douglas Gregor and Sibylle Schupp. Stllint: lifting static checking from languages to libraries. *Software – Practice & Experience*, 36(3):225–254, 2006.

[Gua98] N. Guarino. *Formal Ontology in Information Systems: Proceedings of the 1st International Conference June 6-8, 1998, Trento, Italy.* IOS Press, Amsterdam, The Netherlands, The Netherlands, 1998.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson Engler. A system and language for building system-specific, static analyses. *SIGPLAN Notices*, 37(5), 2002.

[Hed97] Görel Hedin. Attribute extension - a technique for enforcing programming conventions. *Nordic Journal of Computing*, 4(1):93–122, 1997.

[HF10] Florian Hölzl and Martin Feilkas. Autofocus 3 - a scientific tool prototype for model-based development of component-based, reactive, distributed systems. In *Model-Based Engineering of Embedded Real-Time Systems (MBEERTS)*, Lecture Notes in Computer Science. Springer Verlag, To Appear 2010.

[HH06] Daqing Hou and H. James Hoover. Using SCL to specify and check design intent in source code. *IEEE Transactions on Software Engineering*, 32(6):404–423, 2006.

[HHHL03] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. Automatic design pattern detection. In *Proceedings of the 11th IEEE International Workshop on Program Comprehension*, page 94, Washington, DC, USA, 2003. IEEE Computer Society.

[HHR04] Daqing Hou, H. James Hoover, and Piotr Rudnicki. Specifying framework constraints with fcl. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 96–110. IBM Press, 2004.

[HNT99] Morten T. Hansen, Nitin Nohria, and Thomas Tierney. What's your strategy for managing knowledge? *Harvard Business Review 77 (2)*, pages 106–16, 1999.

[Hou07] Daqing Hou. Scl: Static enforcement and exploration of developer intent in source code. In *Companion to the proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007.

[HP00] Morten Hertzum and Annelise Mark Pejtersen. The information-seeking practices of engineers: searching for documents as well as for people. *Information Processing and Management*, 36(5):761–778, 2000.

[HP04] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Notices*, 39(12):92–106, 2004.

[HR04] David Harel and Bernhard Rumpe. Meaningful modeling: What's the semantics of "semantics"? *Computer*, 37(10):64–72, 2004.

[HRY95] David R. Harris, Howard B. Reubenstein, and Alexander S. Yeh. Reverse engineering to the architectural level. In *Proceedings of the 17th international conference on Software engineering*, pages 186–195, New York, NY, USA, 1995. ACM.

[JL83] P. N. Johnson-Laird. *Mental models: towards a cognitive science of language, inference, and consciousness.* Harvard University Press, Cambridge, MA, USA, 1983.

[JLI92] Alex P. J. Jarczyk, Peter Löffler, and Frank M. Shipman Iii. Design rationale for software engineering: A survey. In *Proceedings of the 2nd Internationale Conference on System Science, Los Alamitos, Ca*, 1992.

[JM97] Jean-Marc Jézéquel and Bertrand Meyer. Design by contract: The lessons of ariane. *Computer*, 30(1):129–130, 1997.

[Joh77] Stephen Johnson. Lint, a c program checker. Technical report, Bell Laboratories, Computer Science Technical Report 65, 1977.

[KCJ94] D. W. King, J. Casto, and H. Jones. *Communication by engineers: a literature review of engineers' information needs, seeking processes, and use.* Council on Library Resources, Washington, D.C., 1994.

[Kel09] Felix Kelm. Integration von Analysedaten in Entwicklungsumgebungen. Master's thesis, Technische Universität München, Fakultät für Informatik, 2009.

[KKS06] Johannes Koskinen, Markus Kettunen, and Tarja Systa. Profile-based approach to support comprehension of software behavior. *Proceedings of the 14th IEEE International Conference onProgram Comprehension*, 0:212–224, 2006.

[KLZR06] Viktor Kuncak, Patrick Lam, Karen Zee, and Martin C. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Trans. Softw. Eng.*, 32(12):988–1005, 2006.

[KML+03] Gabor Karsai, Miklos Maroti, Akos Ledeczi, Jeff Gray, and Janos Sztipanovits. Composition and cloning in modeling and meta-modeling. *IEEE Transactions on Control System Technology*, 2003.

[KN93] E. Koutsofios and S. North. Drawing graphs with dot, 1993.

[Kra99] Douglas Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, pages 147–153, New York, NY, USA, 1999. ACM.

[Kru95] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[Lar01] Craig Larman. Protected variation: The importance of being closed. *IEEE Software*, 18(3):89–91, 2001.

[LBD+04] James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fähndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. Righting software. *IEEE Software*, 21(3):92–100, 2004.

[Leh80] M.M. Lehman. Programs, life cycles and the laws of software evolution. *Proceedings of the IEEE*, 68(9):1060—-1076, 1980.

[Les02] Donald M. Leslie. Using javadoc and xml to produce api reference documentation. In *Proceedings of the 20th annual international conference on Computer documentation*, pages 104–109, New York, NY, USA, 2002. ACM.

[Lev00] Nancy G. Leveson. Intent specifications: An approach to building human-centered specifications. *IEEE Transactions on Software Engineering*, 26(1):15–35, 2000.

[LKR04] Patrick Lam, Viktor Kuncak, and Martin Rinard. Generalized typestate checking using set interfaces and pluggable analyses. *SIGPLAN Notices*, 39(3):46–55, 2004.

[LLP+00] Gary T. Leavens, K. Rustan M. Leino, Erik Poll, Clyde Ruby, and Bart Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA 2000 Companion*, 2000.

[LPS00] Marek Leszak, Dewayne E. Perry, and Dieter Stoll. A case study in root cause defect analysis. In *Proceedings of the 22nd international conference on Software Engineering*, pages 428–437, New York, NY, USA, 2000. ACM.

[LS86] S. Letovsky and E. Soloway. Delocalized plans and program comprehension. *IEEE Software*, 3(3):41–49, 1986.

[LSF03] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6):35–39, 2003.

[LST78] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

[Lut93] Robyn R. Lutz. Software maintenance and evolution: a roadmap. In *Proceedings of IEEE International Symposium on Requirements Engineering*, pages 126–133, 1993.

[LVD06] Thomas D. LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software Engineering*, pages 492–501, New York, NY, USA, 2006. ACM.

[Mic96] SUN Microsystems. Why developers should not write programs that call 'sun' packages, 1996.

[Min74] Marvin Minsky. A framework for representing knowledge. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1974.

[MLLF07] Anders Mattsson, Bjorn Lundell, Brian Lings, and Brian Fitzgerald. Experiences from representing software architecture in a large industrial project using model driven development. In *SHARK-ADI*, page 6, Washington, DC, USA, 2007. IEEE Computer Society.

[MNS01] Gail C. Murphy, David Notkin, and Kevin J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[MRB+05] Andrian Marcus, Vaclav Rajlich, Joseph Buchta, Maksym Petrenko, and Andrey Sergeyev. Static techniques for concept location in object-oriented code. In *Proceedings of the 13th International Workshop on Program Comprehension*, pages 33–42, Washington, DC, USA, 2005. IEEE Computer Society.

[MSRM04] Andrian Marcus, Andrey Sergeyev, Vaclav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[Myl92] John Mylopoulos. Conceptual modelling and telos. *Conceptual Modelling, Databases, and CASE: An Integrated View of Information Systems Development*, 1992.

[Nel04] Greg Nelson. Extended static checking for java. *Lecture Notes in Computer Science*, 2004.

[NT95] Ikujiro Nonaka and Hirotaka Takeuchi. *The Knowledge-Creating Company.* Oxford University Press Inc., USA, 1995.

[OMG06a] OMG. Meta Object Facility (MOF) Specification 2.0, 2006.

[OMG06b] OMG. Object Constraint Language (OCL) Specification 2.0, 2006.

[OR92] S.B. Ornburn and S. Rugaber. Reverse engineering: resolving conflicts between expected and actual software designs. *Proceedings of the IEEE Conference on Software Maintenance*, pages 32–40, Nov. 1992.

[Ous98] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[PAC+08] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa, Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for java. In *Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, New York, NY, USA, 2008. ACM.

[Par72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[Par94] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software Engineering*, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[PC85] D. L. Parnas and P. C. Clements. A rational design process: how and why to fake it. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development on Formal Methods and Software*, pages 80–100, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[PCW84] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. In *Proceedings of the 7th international conference on Software Engineering*, pages 408–417, Piscataway, NJ, USA, 1984. IEEE Press.

[Pol66] Michael Polanyi. *The Tacit Dimensions.* Doubleday, New York, 1966.

[PSV94] Dewayne E. Perry, Nancy Staudenmayer, and Lawrence G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, 1994.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the study of software architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[Rat09] Daniel Ratiu. *Intensional Meaning of Programs.* PhD thesis, Technische Universität München, 2009.

[RD07] Daniel Ratiu and Florian Deissenboeck. From reality to programs and (not quite) back again. In *ICPC*. IEEE Computer Society, 2007.

[RFD+08] Daniel Ratiu, Martin Feilkas, Florian Deissenboeck, Jan Jürjens, and Radu Marinescu. Towards a repository of common programming technologies knowledge. In *International Workshop on Semantic Technologies in System Maintenance*, Jun. 2008.

[RFJ08] Daniel Ratiu, Martin Feilkas, and Jan Jürjens. Extracting domain ontologies from domain specific apis. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pages 203–212. IEEE Computer Society, 2008.

[Ric53] H. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 83, 1953.

[Rie06] Jonathan Riehl. Assimilating metaborg:: embedding language tools in languages. In *Proceedings of the 5th international conference on Generative programming and component engineering*, pages 21–28, New York, NY, USA, 2006. ACM.

[RJ96] D. Roberts and R. Johnsonr. Evolve frameworks into domain-specific languages. In Jeff Gray, Jonathan Sprinkle, Juha-Pekka Tolvanen, and Matti Rossi, editors, *3rd International Conference on Pattern Languages*, Allerton Park, Sept. 1996.

[RJ07] Daniel Ratiu and Jan Juerjens. The reality of libraries. In *CSMR*. IEEE Computer Society, 2007.

[RL04] I. Rus and M. Lindvall. Knowledge management in software engineering. *IEEE Software*, 19(3):26 – 38, May-June 2004.

[RMG+06] Coen De Roover, Isabel Michiels, Kim Gybels, Kris Gybels, and Theo D'Hondt. An approach to high-level behavioral program documentation allowing lightweight verification. *icpc*, 0:202–211, 2006.

[Rob99] Pierre N. Robillard. The role of knowledge in software development. *Communications of the ACM*, 42(1):87–92, 1999.

[RSW95] S. Rugaber, K. Stirewalt, and L.M. Wills. The interleaving problem in program understanding. In *Proceedings of the 3rd Working Conference on Reverse Engineering*, pages 166–175, 1995.

[RW02] Vaclav Rajlich and Norman Wilde. The role of concepts in program comprehension. In *10th International Workshop on Program Comprehension*. IEEE Computer Society Press, 2002.

[Sch08] Wolfgang Schwitzer. Konzeption und Implementierung eines Deployment-Konzepts für AutoFocus 3. Master's thesis, Technische Universität München, Fakultät für Informatik, 2008.

[Sin98] Janice Singer. Practices of software maintenance. In *Proceedings of the International Conference on Software Maintenance*, page 139, Washington, DC, USA, 1998. IEEE Computer Society.

[SLL+88] Elliot Soloway, Robin Lampert, Stan Letovsky, David Littman, and Jeannine Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988.

[TCL02] R.T. Tvedt, P. Costa, and M. Lindvall. Does the code match the design? a process for architecture evaluation. In *Proceedings of the International Conference on Software Maintenance*, page 393, Washington, DC, USA, 2002. IEEE Computer Society.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st international conference on Software Engineering*, pages 107–119, New York, NY, USA, 1999. ACM.

[Tol04] Juha-Pekka Tolvanen. Metaedit+: domain-specific modeling for full code generation demonstrated [gpce]. In *Companion to the 19th ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 39–40, New York, NY, USA, 2004. ACM.

[Tve93] Barbara Tversky. Cognitive maps, cognitive collages, and spatial mental models. *Lecture Notes in Computer Science*, pages 14–24, 1993.

[vDKV00] Arie van Deursen, Paul Klint, and Joost Visser. Domain-specific languages: an annotated bibliography. *SIGPLAN Notices*, 35(6):26–36, 2000.

[vGB02] Jilles van Gurp and Jan Bosch. Design erosion: problems and causes. *The Journal of Systems & Software*, 61(2):105–119, 2002.

[vGBB05] Jilles van Gurp, Sjaak Brinkkemper, and Jan Bosch. Design preservation over subsequent releases of a software product: a case study of baan erp: Practice articles. *Journal on Software Maintainence and Evolution*, 17(4):277–306, 2005.

[VWBH06] E. Van Wyk, D. Bodin, and P. Huntington. Adding syntax and static analysis to libraries via extensible compilers and language extensions. In *Proceedings of the conference on Library-Centric Software Design*, 2006.

[VWKSB07] Eric Van Wyk, Lijesh Krishnan, August Schwerdfeger, and Derek Bodin. Attribute grammar-based language extensions for java. In *European Conference on Object Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer Verlag, Jul. 2007.

[WA04] James Ward and Aybüke Aurum. Knowledge management in software engineering - describing the process. In *Proceedings of the Australian Software Engineering Conference*, page 137, Washington, DC, USA, 2004. IEEE Computer Society.

[WTMS95] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural redocumentation: A case study. *IEEE Software*, 12(1):46–54, 1995.