Dissertation

Adaptive Erkennung von Software-Entwurfsmängeln

Schriftliche Arbeit zur Erlangung
des akademischen Grades eines
Doktors der Naturwissenschaften an der
Fakultät Elektrotechnik, Mathematik und Informatik
der Universität Paderborn

vorgelegt von

Jochen Kreimer

Paderborn, September 2005

Veröffentlichung
in den Online-Hochschulschriften
der Deutschen Nationalbibliothek,
Frankfurt am Main,
http://www.d-nb.de.

Datum der mündlichen Prüfung:
20.01.2006

Promotionskommission:
Prof. Dr. Uwe Kastens, Universität Paderborn (Gutachter)
Prof. Dr. Jürgen Ebert, Universität Koblenz-Landau (Gutachter)
PD Dr. Benno Stein, Universität Paderborn (Gutachter)
Prof. Dr. Gregor Engels, Universität Paderborn
Dr. Ulf Lorenz, Universität Paderborn

Danke!

Viele liebe Menschen haben diese Arbeit ermöglicht. Ihnen möchte ich herzlich danken.

Meinem Doktorvater Uwe Kastens danke ich für seine Geduld und seinen stetigen Zuspruch. Er gab mir häufig mehr Freiheit, als gut für mich war.

Ich danke allen Gutachtern für ihre wertvollen Hinweise und Verbesserungsvorschläge.

Den Kollegen der Fachgruppe danke ich für eine angenehme und kreative Arbeitsumgebung. Unsere Diskussionen am Kaffeetisch halfen stets, den Kopf wieder frei zu bekommen.

Viele Diplomanden und studentische Hilfskräfte haben zum Gelingen beigetragen. Ich möchte vor allem das Engagement von Carsten Lachmann und Mike Liebrecht hervorheben.

Mein besonderer Dank gilt dem Germanisten Karsten Grabenstroer. Er kämpfte sich auf der Suche nach Schreib- und Formulierungsfehlern durch die für ihn schwer verständlichen Texte.

Ich danke meinen Eltern, dass sie mich auf meinen Wegen stets begleitet und unterstützt haben.

Meine Familie und viele Freunde erinnerten mich häufig daran, dass es noch ein Leben abseits des Schreibtisches gibt.

Diese Arbeit wäre niemals entstanden ohne die Liebe und Stärke meiner Frau Dagmar und das Lachen unseres Sohnes Tillmann. Ihnen ist diese Arbeit und meine Zukunft gewidmet.

München, im Februar 2006 Jochen Kreimer

Inhaltsverzeichnis

1. Einleitung **1**

1.1. Überblick . . . 2
- 1.1.1. Objektorientierter Entwurf . . . 2
- 1.1.2. Entwurfsmängel . . . 2
- 1.1.3. Adaptive Erkennung von Entwurfsmängeln . . . 3

1.2. Kapitelübersicht . . . 4

2. Objektorientierter Entwurf **7**

2.1. Überblick . . . 8
2.2. Software-Engineering . . . 8
- 2.2.1. Programmierung bedeutet Abstraktion . . . 9
- 2.2.2. Industrielle Software-Entwicklung . . . 13

2.3. Konzepte objektorientierter Programmierung . . . 15
- 2.3.1. Objektorientierte Denkweise . . . 15
- 2.3.2. Beziehungen zwischen Objekten . . . 16
- 2.3.3. Vererbung und Polymorphie . . . 17
- 2.3.4. Schnittstellen . . . 17

2.4. Guter objektorientierter Entwurf . . . 18
- 2.4.1. Vererbungskonzepte . . . 18
- 2.4.2. Rollen . . . 19
- 2.4.3. Spezifikation . . . 20
- 2.4.4. Wiederverwendung . . . 20
- 2.4.5. Entwurfsmuster . . . 21
- 2.4.6. Komponenten . . . 24

2.5. Zusammenfassung . . . 25

3. Entwurfsmängel **27**

3.1. Übersicht . . . 28
- 3.1.1. Refactoring . . . 28
- 3.1.2. Abgrenzung . . . 29
- 3.1.3. Vielfalt der Entwurfsmängel . . . 30

3.2. Beispiele für Entwurfsmängel . . . 32
- 3.2.1. Große Klasse . . . 32
- 3.2.2. Datenklasse . . . 34

3.2.3. Lange Methode 34
3.2.4. Neid 36
3.2.5. Ausgeschlagenes Erbe 37
3.2.6. Nachrichtenketten und Vermittler 38
3.3. Klassifizierungen 38
3.4. Zusammenfassung 40

4. Maschinelles Lernen **41**
4.1. Überblick 42
4.2. Grundlagen konzeptionellen Lernens 42
4.3. Entscheidungsbaumverfahren 46
4.3.1. Problemklassen 47
4.3.2. Grundlegender Lernalgorithmus 47
4.3.3. Eigenschaften und Erweiterungen 49
4.4. Andere Lernverfahren 50
4.5. Zusammenfassung 51

5. Modellierung von Entwurfsmängeln **53**
5.1. Überblick 54
5.2. Entwurf von Modellen zu Entwurfsmängeln 54
5.2.1. Messtheorie 55
5.2.2. Metriken in der Software-Technik 56
5.2.3. Klassifizierung objektorientierter Entwurfsmetriken 56
5.2.4. Entwurf geeigneter Metriken 58
5.3. Beispielmodelle 59
5.3.1. Große Klasse 59
5.3.2. Lange Methode 64
5.3.3. Faule Klasse / Datenklasse 65
5.3.4. Neid 65
5.4. Zusammenfassung 66

6. Analyse von Programmstrukturen **69**
6.1. Überblick 70
6.2. Analyse von Java-Programmen 71
6.2.1. Methoden statischer Programmanalyse 71
6.2.2. Aspekte der Objektorientierung 72
6.3. Struktur objektorientierter Programme 73
6.3.1. Pakete, Klassen, Methoden und Attribute 73
6.3.2. Beziehungen zwischen Programmobjekten 74
6.3.3. Programmabhängigkeitsgraph 75
6.4. Berechnung von Metriken 76

6.4.1. Verwendung der relationalen Algebra 78
6.4.2. Auswertung von Relationen 80
6.4.3. Beispielberechnungen 82
6.5. Verwandte Ansätze 83
6.6. Zusammenfassung 84

7. Adaptive Erkennung von Entwurfsmängeln 85
7.1. Überblick 86
7.2. Adaptive Erkennung 86
7.2.1. Einsatz eines Lernverfahrens 90
7.2.2. Erklärungskomponente 92
7.2.3. Modellreflexion 92
7.3. Zusammenfassung 93

8. Evaluation 95
8.1. Übersicht 96
8.2. Kriterien der Evaluation 97
8.2.1. Effektivität 97
8.2.2. Effizienz 99
8.3. Untersuchungsmethoden 100
8.3.1. Voruntersuchung 100
8.3.2. Messung der Effektivität 106
8.3.3. Messung der Effizienz 127
8.4. Werkzeugunterstützung 129
8.4.1. Das Werkzeug IYC 130
8.4.2. Architektur 130
8.4.3. Benutzung 131
8.5. Fallstudie 136
8.5.1. Untersuchungsteilnehmer 136
8.5.2. Analysegegenstand 137
8.5.3. Ergebnisse der Voruntersuchung 137
8.5.4. Ergebnisse der Effektivitätsmessungen 137
8.6. Zusammenfassung 138

9. Fazit 149
9.1. Ausgangspunkt 150
9.2. Beiträge und Ergebnisse 151
9.3. Ausblick 152

A. „Design Heuristics“ nach Riel 153

B. „Bad Smells“ nach Fowler **159**

C. Konfiguration des IYC-Werkzeuges **163**

D. Verzeichnisse **167**

1. Einleitung

Inhalt

1.1. Überblick 2

1.1.1. Objektorientierter Entwurf 2

1.1.2. Entwurfsmängel 2

1.1.3. Adaptive Erkennung von Entwurfsmängeln 3

1.2. Kapitelübersicht 4

1.1. Überblick

Die Qualität von Software kann je nach Anwendungsgebiet an unterschiedlichen Kriterien gemessen werden. Für große Software-Systeme spielen u. a. Kriterien wie Wartbarkeit, Verständlichkeit und Erweiterbarkeit eine wichtige Rolle.

Mein Ziel ist es, Entwurfsmängel in Software-Systemen zu erkennen und somit „schlechte“ — unverständliche, schwer erweiter- und änderbare — Programmstrukturen zu vermeiden. Prominente Entwurfsmängel sind z. B. die von Fowler eingeführten *Bad Smells* in objektorientierten Programmen.

1.1.1. Objektorientierter Entwurf

Das objektorientierte Programmierparadigma verspricht klar strukturierte, wiederverwendbare und leicht wartbare Software. In der Praxis wird dies nur von sehr erfahrenen Programmierern und Software-Architekten erreicht.

„All data should be hidden within its class“ [74] ist nur einer der zahlreichen hilfreichen Ratschläge bekannter Vordenker und erfolgreicher Praktiker des objektorientierten Programmierparadigmas, die helfen sollen eigene Programmstrukturen kritisch zu hinterfragen.

Somit gehört die manuelle Untersuchung (auch *Software Inspection* [33] [32]) von Programmen zu den wertvollen Techniken um die Qualität von Software zu verbessern. Dabei werden Quelltexte, Entwurf und Dokumentation manuell gesichtet. In modernen agilen Software-Entwicklungsprozessen, wie dem *Extreme Programming*, spielen sie eine wichtige Rolle zur Qualitätssicherung.

Die Untersuchung erlaubt, noch vor der Testphase, und somit frühzeitig im Entwicklungsprozess, Fehler zu finden. Wegen der Arbeits- und Zeitintensität bietet sich Werkzeugunterstützung an. Mit einem Werkzeug, das automatisiert und wiederholt Software analysiert, kann z. B. die Einhaltung von Programmierrichtlinien geprüft und damit ein kontinuierlich hoher Qualitätsstandard erreicht werden.

Mein Ziel ist es, Fehler im Entwurf von Software-Systemen automatisch mit Hilfe eines Werkzeugs zu erkennen und somit unverständliche, schwer erweiter- und änderbare Programmstrukturen zu vermeiden.

1.1.2. Entwurfsmängel

Entwurfsmängel sind Programmeigenschaften, die auf potentiell fehlerhaften Entwurf eines Software-Systems hindeuten.

In der Literatur werden diese als *„Design Heuristics“* [74], *„Design Characteristics“* [92] oder *„Bad Smells“* [37] beschrieben. Die Autoren bezeichnen Entwurfsmängel i. d. R. durch Metaphern und erklären dem Software-Entwickler und Software-Architekten wie solche Entwurfsmängel erkannt und behoben werden können.

Fowler [37] beschreibt zudem *Refactoring*-Transformationen, die die innere Struktur eines Programms anpassen, ohne das von außen sichtbare Verhalten zu ändern. Die Transformationen erlauben, Programme zu bereinigen und zu vereinfachen. *„Bad Smells"* sind Entwurfsmängel, die beschreiben, welche Programmstellen durch *Refactoring*-Transformationen verbessert werden können. Beispiele für solche *„Bad Smells"* sind zu lange Methoden, Klassen mit mehreren Aufgaben, zu viele Parameter oder lokale Variable einer Methode, Verletzung von Datenkapselung, intensive Delegation oder „Erbschaftsstreitigkeiten".

Verschiedene Personen haben unterschiedliche Vorstellungen von Entwurfsmängeln [55]. Dies liegt zum Einen an der informellen und metaphorischen Beschreibungsweise von Entwurfsmängeln in der Literatur, zum Anderen am persönlichen Erfahrungsschatz und der Sichtweise des Einzelnen. Somit spielt der Faktor Mensch eine entscheidende Rolle bei der automatisierten Erkennung von Entwurfsmängeln.

1.1.3. Adaptive Erkennung von Entwurfsmängeln

Mein Ansatz beruht daher auf der Kombination verschiedener Techniken.

Zunächst ist es nötig vom konkreten Programm zu abstrahieren. Nicht die detaillierte interne Sicht auf das Programm, sondern der Überblick über Abhängigkeiten und Zusammenhänge innerhalb des Software-Systems erlauben dem menschlichen Betrachter, Problemstellen auf der Entwurfsebene zu identifizieren. Ich setze daher spezialisierte Metriken ein, die relevante Programmeigenschaften wiederspiegeln. Zur Berechnung von Metriken werden Techniken der statischen Programmanalyse eingesetzt.

Die Analyse von Metrikwerten zu einer Programmstelle erlaubt es dem Experten, verdächtige Programmstellen zu erkennen. Diese subjektive Bewertung von Programmstellen ist dann erfolgreich, wenn der Experte das Einsatzgebiet des zu beurteilenden Programms ebenso wie eigene Erfahrungen zu gutem Software-Entwurf einfließen lässt.

Ich setze daher maschinelle Lernverfahren ein — nicht um den Experten zu ersetzen — sondern um seine Erfahrung aufzunehmen und für die weitere Suche zu nutzen. Das Ergebnis ist ein adaptiver Ansatz, Entwurfsmängel zu erkennen, indem Wissen mit Hilfe von Lernverfahren generiert, bewahrt und genutzt wird.

Obwohl ich mich in dieser Arbeit besonders auf objektorientierte Programme konzentriere und das im Rahmen dieser Arbeit entstandene Werkzeug Java-Programme analysiert, ist der beschriebene Ansatz grundsätzlich auf jede Art von Programmen und auch andere Analyseziele übertragbar.

1.2. Kapitelübersicht

Ich streife in dieser Arbeite viele Wissensgebiete, sodass ich nur die im Zusammenhang dieser Arbeit wichtigen Aspekte darstelle. Auf weiterführende Literatur wird in den einzelnen Kapiteln verwiesen.

Kapitel 2: Objektorientierter Entwurf Große objektorientierte Software-Systeme sind der Analysegegenstand in dieser Arbeit. Ich erläutere daher in diesem einführenden Kapitel die Grundzüge der objektorientierten Programmierung mit besonderem Blick auf die Konstruktion großer Systeme durch eine Entwicklergruppe. Denn für den Erfolg eines Software-Projektes sind besonders die Kommunikation zwischen den Entwicklern und die einfache Wart- und Erlernbarkeit wichtig. In diesem Szenario stiftet die automatisierte Suche nach Entwurfsmängeln besonders viel Nutzen.

Kapitel 3: Entwurfsmängel In diesem Kapitel werden Entwurfsmängel charakterisiert und abgegrenzt. Einige Entwurfsmängel werden beispielhaft detailliert beschrieben. Diese werden in späteren Kapiteln jeweils wieder aufgegriffen um konkrete Szenarien der Erkennung zu beschreiben.

Kapitel 4: Maschinelles Lernen Das maschinelle Lernen erlaubt es, anhand von Beispielen konkrete Konzepte zu abstrahieren bzw. zu erlernen. In diesem Kapitel werden die Grundzüge dieser Verfahren beschrieben. Besonderes Augenmerk liegt dabei auf Verfahren, mit denen Entscheidungsbäume konstruiert werden. Diese setze ich für meinen adaptiven Ansatz ein.

Kapitel 5: Modellierung von Entwurfsmängeln Die informelle Beschreibung von Entwurfsmängel reicht nicht aus, um diese automatisiert erkennen zu können. Es wird daher jedem Entwurfsmangel eine Menge von charakteristischen Programmeigenschaften zugeordnet. Diese werden in der Form von Metriken dargestellt und bilden ein Modell für jeden zu erkennenden Entwurfsmangel. In diesem Kapitel wird für einige Mängel jeweils ein Beispielmodell entwickelt.

Kapitel 6: Programmanalyse zur Berechnung von Entwurfsmetriken Zur Berechnung von Metriken werden Verfahren der klassischen statischen Programmanalyse eingesetzt. Für den praktischen Einsatz in einem interaktiven Werkzeug ist der sparsame Gebrauch von Rechenzeit und Speicher zu betrachten. Daher wird ein Verfahren entwickelt, mit dem einzelne Programmteile mit ausreichender Genauigkeit analysiert werden können. Grundlage bildet ein Programmabhängigkeitsgraph, der mit Hilfe der relationalen Algebra ausgewertet wird, sodass sich leicht vielfältige Metriken berechnen lassen.

Kapitel 7: Adaptive Erkennung von Entwurfsmängeln Dieses Kapitel beschreibt das Gesamtkonzept zur adaptiven Erkennung von Entwurfsmängeln. Anhand des Modells eines Entwurfsmangels werden Metriken zu Programmstellen berechnet, deren Werte durch ein maschinelles Lernverfahren analysiert werden. Das Ergebnis ist ein Entscheidungsbaum für jeden Mangel, der verwendet wird um weitere Programmstellen zu untersuchen. Eine wesentliche Rolle spielt dabei der Benutzer, der den Lernprozess für seine Zwecke steuert.

Kapitel 8: Evaluation Das adaptive Verfahren passt sich dem Benutzer an. Die Erkennungsleistung lässt sich daher nur durch zeitaufwändige empirische Untersuchungen ermitteln. Ich beschreibe in diesem Kapitel ein Evaluationsverfahren, das Techniken der empirischen Sozialforschung verwendet. Konkrete Evaluationsergebnisse liefert eine Fallstudie, die im Rahmen dieser Arbeit durchgeführt werden konnte.

2. Objektorientierter Entwurf

Inhalt

2.1. Überblick **8**

2.2. Software-Engineering **8**

2.2.1. Programmierung bedeutet Abstraktion 9

2.2.2. Industrielle Software-Entwicklung 13

2.3. Konzepte objektorientierter Programmierung **15**

2.3.1. Objektorientierte Denkweise 15

2.3.2. Beziehungen zwischen Objekten 16

2.3.3. Vererbung und Polymorphie 17

2.3.4. Schnittstellen 17

2.4. Guter objektorientierter Entwurf **18**

2.4.1. Vererbungskonzepte 18

2.4.2. Rollen 19

2.4.3. Spezifikation 20

2.4.4. Wiederverwendung 20

2.4.5. Entwurfsmuster 21

2.4.6. Komponenten 24

2.5. Zusammenfassung **25**

2.1. Überblick

„Software engineering is a verb.“ schreibt Whitmire [92] und weist darauf hin, dass Software-Entwicklung häufig falsch verstanden wird. Ist Software-Entwicklung Handwerk, Ingenieurstätigkeit oder Kunst? Fest steht, Software-Entwicklung im „Großen“ ist schwer planbar, schwer kontrollierbar und noch schwerer vorhersehbar. Vergleichend mit anderen klassischen Ingenieursdisziplinen — wie z. B. dem Hoch- und Tief- oder dem Maschinenbau, in denen der Architekt plant, der Ingenieur umsetzt und der Arbeiter ausführt, vermischen diese Tätigkeiten in der Software-Entwicklung [68].

Dieses Kapitel beschreibt Eigenschaften und Wege zu qualitativ hochwertiger Software am Beispiel der objektorientierten Programmierung. Abschnitt 2.2 beleuchtet Prozesse des Software-Engineering und legt dabei den Schwerpunkt auf Konzepte von Programmiersprachen. In Abschnitt 2.3 werden die Konzepte der objektorientierten Programmierung und deren Anwendung genauer beschrieben. Zum Abschluss dieses Kapitels wird in Abschnitt 2.4 das Spektrum von gutem objektorientierten Entwurf aufgezeigt.

In dieser Arbeit untersuche ich speziell objektorientierte Software-Strukturen, die Wartung vereinfachen, die flexibel, adaptierbar und wiederverwendbar sind.

2.2. Software-Engineering

Der Begriff der *Software-Krise* existiert seit den späten 1960er Jahren. Er beschreibt das Problem, Software-Projekte nicht in der gewünschten Zeit, mit den gewünschten Kosten und bei hoher Qualität herstellen zu können.

Betrachtet man das Angebot eines gut sortierten Software-Händlers, so bemerkt man zunächst nichts von einer Krise. Eine reichhaltige Palette bunt bedruckter Schachteln, die Software-Produkte für viele Anwendungsgebiete enthalten, wird angeboten. Tatsächlich hat diese Endverbraucher-Software nur einen Marktanteil von etwa 10 %. Der überwiegende Teil der industriellen Software-Produktion wird auf speziellen Kundenwunsch hergestellt.

Die *Standish Group* untersucht seit den 90er Jahren Software-Projekte. Die letzte Untersuchung in 2003 ergab, dass 82 % aller Projekte verspätet ausgeliefert werden, 66 % aller Projekte scheitern, 52 % entsprechen nicht den vereinbarten Anforderungen, 43 % sprengen den Kostenrahmen und 15 % aller Projekte wurde vorzeitig beendet [39].

Software-Produktion scheint unkalkulierbar und damit unberechenbar zu sein. Software ist ständigen Änderungen, Fehlerkorrekturen, Erweiterungen und Anpassungen unterworfen. Dem steht die steigende Größe und damit automatisch steigende Komplexität eines Systems gegenüber. Galt in den 1980er und 1990er

Jahren ein Software-System mit mehr als 50000 Zeilen Quelltext als „großes“ Projekt, so stehen dem heute mitunter Millionen Zeilen und mehrere hundert Entwicklerjahre gegenüber.

Dijkstra schreibt hierzu bereits 1972:

> „Als es noch keine Rechner gab, war auch das Programmieren noch kein Problem, als es dann ein paar leistungsschwache Rechner gab, war das Programmieren ein kleines Problem und nun, wo wir gigantische Rechner haben, ist auch das Programmieren zu einem gigantischen Problem geworden. In diesem Sinne hat die elektronische Industrie kein einziges Problem gelöst, sondern nur neue geschaffen. Sie hat das Problem geschaffen, ihre Produkte zu benutzen.“ [Dijkstra, 1972].

Das Wissensgebiet der Software-Technik *(Software Engineering)* vermittelt Techniken, mit denen die Komplexität der Software-Entwicklung auf mehrere Personen, Aufgaben und zeitliche Phasen verteilt werden kann. Auch im Wissensgebiet der Programmiersprachen lässt sich eine stetige Entwicklung beobachten, sodass Software-Lösungen auf immer höheren Abstraktionsebenen formuliert und formalisiert werden.

2.2.1. Programmierung bedeutet Abstraktion

Programmiersprachen sind die Werkzeuge mit, denen der Software-Entwickler seine Realisierungsideen formalisiert und in eine lauffähige Form überführt. Die Ausdruckskraft der eingesetzten Sprachen hat entscheidenden Einfluss auf die Denkweise eines Entwicklers.

Zuse hat in [98] die Geschichte der Programmiersprachen zusammengetragen. Ausgehend von imperativer maschinennaher Programmierung entwickelten sich höhere Programmiersprachen (siehe Abbildung 2.1). An der Entwicklung der Paradigmen lässt sich die steigende Komplexität von Software-Systemen ablesen.

Das Lösungsprinzip bleibt dabei stets *Teile und Herrsche*: Zerlege das Problem in überschaubare Teilprobleme und löse diese isoliert. Führe die Teillösungen zu einer Gesamtlösung zusammen.

Für die industrielle Software-Entwicklung spielen imperative Programmiersprachen die größte Rolle, sodass funktionale und logische Programmierung hier nur am Rande betrachtet wird.

Strukturierung

In seinem Artikel „Goto statement considered harmful“ [26] legte Dijkstra den Grundstein der strukturierten Programmierung. Er stellte fest, dass die Anzahl von Sprüngen im Programm mit der Fehlerzahl korreliert und begründete dies mit der steigenden Komplexität, die der Programmierer nicht mehr beherrschen

1. Generation: Maschinensprache wird heute kaum noch direkt verwendet. Hier würde der Maschinencode des Zielprozessors als Folge von Zahlencodes notiert.

2. Generation: Assembler erlaubt die symbolische Notation von Maschinensprachbefehlen (Mnemonik). Makroprozessoren erlauben eingeschränkte Wiederverwendung. Hauptanwendungsgebiet der Maschinensprachen ist die Hardware-nahe Programmierung, z. B. innerhalb eines Betriebsystemkerns oder für Gerätetreiber. Heute wird auch dieses weitestgehend in höheren Programmiersprachen (wie z. B. C) programmiert.

3. Generation: höhere Programmiersprachen sind i. A. imperative strukturierte Programmiersprachen. Sie sind i. d. R. keinem bestimmten Zweck zuzuordnen und werden deshalb auch *General Purpose*-Sprachen genannt.

4. Generation: Anwendungs-spezifische Sprachen (4GL) sind Sprachen für begrenzte Anwendungsgebiete. Z. B. *SQL* zur Beschreibung von Datenbankanfragen, *XSL* zur Transformationsbeschreibung von XML-Dokumenten oder *S* zur Beschreibung von Daten und deren mathematisch statistischer Auswertung. Sie enthalten häufig Konzepte aus höheren Programmiersprachen um den Anwendungsbereich zu vergrößern und Erweiterungen zu formulieren.

5. Generation: Sprachen der 5. Generation erlauben es, statt Lösungswegen, in Form von Algorithmen nur das Problem selbst zu beschreiben. Es wird dem System überlassen eine geeignete Lösung zu finden und anzuwenden.

Logische Programmierung ist ein Beispiel. Hier werden Fakten und Regeln zu Gegenständen formuliert. Existenz- und Allquantifizierung sind Anfragen an ein solches System. Ein passender Lösungsweg (hier die Art der Suche im Lösungsraum) wird vom System gefunden.

Abbildung 2.1.: Entwicklung der Programmiersprachen — Einteilung in Generationen

könne. Es sei nicht möglich alle Programmzustände zu bedenken, die an einem Sprungziel vorliegen könnten. Dijkstra verbannte Sprünge und ersetzte sie durch die hierarchische Komposition der Konstrukte *Anweisungsfolge*, *Fallunterscheidung* und *Schleife*. Jedes Konstrukt und deren Komposition verfügt über genau einen Einstieg und einen Ausgang.

Der *Teile und Herrsche*-Ansatz der strukturierten Programmierung ist leicht zu entdecken. Teilprobleme werden durch ein Programmstück mit eindeutigem Ein- und Ausgang implementiert. Diese lassen sich beliebig mit anderen Teillösungen kombinieren, indem sie aneinander gereiht oder hierarchisch, z. B. als Fall einer Fallunterscheidung oder Rumpf einer Schleife, zusammengesetzt werden.

Prozeduren

Strukturierte Programmierung erlaubt Programmteile nur wiederzuverwenden, indem man sie dupliziert. Durch prozedurale Programmierung können Anweisungsblöcke separat definiert und an vielen Stellen durch Aufruf verwendet werden. Zusätzlich verbergen Namensräume Implementierungsdetails. Lokale Variable oder innere Prozeduren sind für den Aufrufer nicht sichtbar und brauchen vom Programmierer nicht zur Kenntnis genommen zu werden. Dies reduziert die Komplexität für den Anwender. Prozeduren erlauben Wiederverwendung „im Kleinen".

In der prozeduralen Programmierung wird Wiederverwendung „im Großen" erreicht, indem mehrere Prozeduren, die gemeinsam ein bestimmtes Konzept implementieren, zu Bibliotheken zusammengefasst werden. Der Benutzer kennt nur die Schnittstelle (die Menge der Prozeduren) einer Bibliothek und nicht die internen Abläufe.

Neben den Programmabläufen spielen vor allem die Daten eine wichtige Rolle bei der Programmierung. Um die Daten eines komplexen Software-Systems zu modellieren, reichen Mengen von Variablen primitiver Typen nicht aus. Datenstrukturen fassen Zusammengehöriges zusammen und erlauben dynamisch veränderliche Programmzustände. Bei der Verwendung von Bibliotheken werden den einzelnen Prozeduren statt einzelner Parameter häufig Referenzen auf Datenstrukturen übergeben.

Prozedurale Programmierung ist am Ablauf orientiert. Daten sind nur lose mit Prozeduren und Funktionen verbunden, z. B. indem der Typ der übergebenen Daten festgelegt wird. Prozeduren können auf Daten beliebig lesend und schreibend zugreifen. Dies führt dazu, dass in Programmteilen auf Daten zugegriffen wird, die konzeptionell für den Zugriff nicht vorgesehen sind. Da der Zugriff aber technisch möglich ist, wird dieser aus Kosten- und Zeitgründen dennoch genutzt. Degenerierte und schwer durchschaubare Architekturen sind die Folge. Die Verwendung statisch typisierter Sprachen wirkt dem entgegen; durch Typanpassungen und Zeigerarithmetik (wie in C) wird die Typsicherheit jedoch aufgeweicht.

Objekte

Die objektorientierte Programmierung entstand schon in den 1970er Jahren. *Simula* stand Pate für die Sprache *Smalltalk*, die heute als erste objektorientierte Sprache gilt. Charakteristisch für das objektorientierte Paradigma ist die Kapselung von Information und Ablauf. Die Definitionen von Daten und Operationen (auch Methoden) werden zu Klassen zusammengefasst, sodass i. d. R. nur die Operationen einer Klasse auf die eigenen Daten zugreifen dürfen. Zugriffsrechte regeln dies genauer. Vererbung erlaubt, bestehende Klassen zu spezialisieren und diese im Kontext der beerbten Klasse einzusetzen. Klassen sind Muster oder Schemata, deren Instantiierung Objekte erzeugt. Abschnitt 2.3 geht genauer auf die Konzepte der objektorientierten Programmierung ein.

Durch das Kapselungsprinzip wird das *Teile und Herrsche*-Prinzip auf einer höheren Ebene fortgeschrieben. Programmierer müssen nicht mehr wissen, welche Daten mit welchen Funktionen bearbeitet werden können. Daten, hier Objekte, sind automatisch mit allen Funktionen verknüpft, die diese betreffen.

Die objektorientierte Programmierung erfordert starkes Umdenken beim Entwurf eines Software-Systems. Während man bei der prozeduralen Programmierung Denkmuster bemüht, die an Abläufen und Datenstrukturen orientiert sind, erfordert die objektorientierte Programmierung eine an Bausteinen orientierte Denkweise. D. h. die Bausteine eines Systems sind zu definieren und mit anderen in Beziehung zu setzen.

Solche Bausteine sind Konzepte, wie Fehlerbehandlung, Protokolle, aber auch Architekturteile, z. B. Technikabstraktionen wie Datenbankschnittstellen oder Netzwerkprotokollschichten; und vor allem die Implementierung der fachlichen Konzepte des Anwendungsgebietes des Software-Systems.

Moderne objektorientierte Programmiersprachen (wie z. B. Java oder C#) verfügen z. B. über hierarchische Ausnahmen-Konzepte, die strukturierte Fehlerbehandlung erlauben. Dies erlaubt Programmteile zur Fehlerbehandlung weitgehend unabhängig vom Rest des Programms zu formulieren und so einer unübersichtlichen Verwebung vorzubeugen.

Komponenten

Programmierer sprechen häufig davon, dass ein Programmteil etwas „weiß“ oder für eine bestimmte Aufgabe „zuständig“ sei. In großen und komplexen Software-Systemen gibt es eine Vielzahl solcher Wissensgebiete, die abgebildet oder Zuständigkeiten, die wahrgenommen werden müssen. Programmiersprachen helfen dem Programmierer, indem sie Konzepte bereitstellen, die es erlauben, solche Bausteine oder auch Komponenten eines Systems isoliert zu konzipieren, zu implementieren und dann später zu konfigurieren und in ein Gesamtsystem zu integrieren.

Somit setzt sich das *Teile und Herrsche*-Prinzip bis zum Entwurf und zur Ar-

chitektur des Systems fort. Das objektorientierte Paradigma ist wohl auch deshalb weit verbreitet, weil es dazu inspiriert, Konzepte der realen Welt durch Klassendefinitionen abzubilden. Dies nimmt eine wesentliche Komplexitätshürde der Programmierung: Die Frage danach, wo ein Konzept implementiert ist, oder wo es implementiert werden soll. Klassen und deren Beziehungen enthalten eine ideale Kommunikationsstruktur, die es erlaubt, zwischen allen Projektbeteiligten zu vermitteln.

2.2.2. Industrielle Software-Entwicklung

Die vielschichtigen Probleme der industriellen Software-Entwicklung, in der größere Entwicklerteams und externe Auftraggeber beteiligt sind, lassen sich nicht allein durch geeignete Programmiersprachkonzepte lösen.

Die Beteiligung vieler Projektpartner erfordert geeignete Planungs-, Steuerungs- und Kommunikationsprozesse, die die Voraussetzungen für die Planung und Durchführung in industriellem Maßstab schaffen.

Das *Software Engineering* beschreibt eine Sammlung von Techniken, die zur Konstruktion und Wartung von Software-Systemen eingesetzt werden. Hierzu gehören Planung, Modellierung, Analyse, Spezifikation, Entwurf, Implementierung, Test und Wartung. Alle Aufgaben müssen verstanden und kontrolliert durchgeführt werden. Das Ziel ist der planbare Projektverlauf.

2.2.2.1. Arten von Software-Projekten

Die wenigsten Software-Projekte haben zum Ziel, ein völlig neues Software-System zu erstellen. Häufig sind bestehende Systeme zu ändern oder zu integrieren. Man unterscheidet folgende Arten von Software-Projekten:

Forward Engineering beschreibt die Konstruktion eines neuen Software-Systems, ausgehend von der Anforderungsanalyse bis hin zum Abnahmetest.

***Reengineering*, Wartung** Bestehende Systeme müssen gewartet werden. Die Software an sich ist zwar wartungsfrei im Sinne eines Verschleißes, muss sich aber wechselnden Anforderungen oder Umgebungen anpassen.

Die *Adaptive Wartung* bedeutet, dass die Software in einer veränderten Umgebung arbeiten soll und entsprechend angepasst wird. Es könnte z. B. das Betriebssystem, ein Übersetzer oder eine Datenbank ausgetauscht worden sein. Bei *korrigierender Wartung* werden ausschließlich Fehler behoben. *Verbesserungen* fügen neue Eigenschaften oder Funktionen hinzu. Das *Reengineering* dient der Verbesserung der Struktur (und damit des Entwurfs) eines

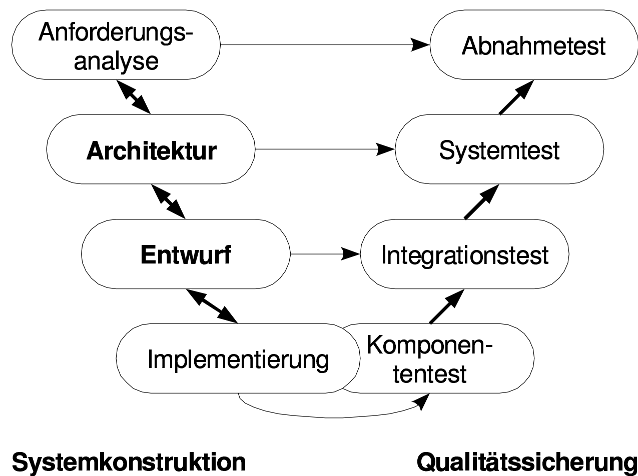

Abbildung 2.2.: V-Modell zur Konstruktion und Qualitätssicherung von Software-Systemen

Systems. I. d. R. werden keine neuen Eigenschaften oder Funktionen hinzugefügt. Es soll nur die Wartbarkeit verbessert werden. Die wichtigste Methode ist hier das *Refactoring*.

Reverse Engineering ermittelt, welche Funktionalität ein Software-System (oder ein Teil davon) hat, das nicht im Quelltext vorliegt. Ziel ist es, so viele Informationen zu erhalten, wie benötigt werden um die Software zu verändern oder zu reproduzieren.

2.2.2.2. Entwicklungsprozesse

Industrielle Software-Entwicklung benötigt Prozesse, die festlegen, in welchen Schritten Software-Systeme entstehen. Im Wesentlichen werden Wasserfall- und Spiralmodelle unterschieden.

Ein typisches Wasserfallmodell ist das V-Modell [13] aus den 1990er Jahren (siehe Abbildung 2.2). Solche Vorgehensmodelle beschreiben eine Abfolge von Aufgaben, die auch verzahnt abgearbeitet werden.

Typische Aufgaben sind die Anforderungsanalyse zu Beginn eines Projektes, aus der der Architekturentwurf hervorgeht. Daraus wird der verfeinerte Entwurf des Software-Systems entwickelt.

Für die objektorientierte Programmierung haben sich Analysetechniken etabliert, die helfen, informell formulierte Anforderungen in formalisierte objektorientierte Systemstrukturen zu übertragen. Man spricht hier von *Object Oriented*

Analysis and Design (OOAD) [9].

Die Implementierung des Software-Systems wird von einer Test- und Integrationsphase begleitet und nach Prüfung der ursprünglichen Anforderungen schließlich ausgeliefert. Verzahnung ergibt sich, wenn frühe Aufgaben der Konstruktion (Anforderungsanalyse, Entwurf) Fehler enthalten oder sich Anforderungen ändern. Im schlimmsten Fall wird dies bei der Implementierung bemerkt, sodass eine Entwurfsentscheidung überdacht werden muss.

Die Phasen der Systemkonstruktion haben ihre Entsprechung in der Qualitätssicherung. So werden Entwurfsinformationen für den Integrationstest, Architekturdaten zum Systemtest und die Ergebnisse der Anforderungsanalyse zur Bewertung des fertigen Systems verwendet.

Spiralmodelle sind heute z. B. agile Prozesse, wie das *Extreme Programming* [6] die stark von prototypischer Entwicklung geprägt sind. Sie folgen dem Prinzip der iterativen Weiterentwicklung von Prototypen mit regelmäßigen Analysephasen. Jede Iteration wird nach einem Wasserfallmodell abgearbeitet. Die Weiterentwicklung bzw. Wartung führt zur nächsten Iteration und damit der nächsten Version des Software-Systems.

2.3. Konzepte objektorientierter Programmierung

Die wesentlichen Konzepte objektorientierter Programmierung [15, 70] sind wohl verstanden. Die wichtigsten Prinzipien werden hier kurz erläutert, bevor im dann folgenden Abschnitt 2.4 der sinnvolle Einsatz, im Sinne von „gutem“ objektorientierten Entwurf beleuchtet wird.

2.3.1. Objektorientierte Denkweise

Im Vergleich zur strukturierten Programmierung, wo Funktionen lose mit Daten verbunden sind, bilden Daten und Ablauf in der objektorientierten Programmierung eine Einheit: das Objekt.

Ein zur Ausführung gebrachtes Programm besteht aus einer Menge von interagierenden Objekten. Diese haben Eigenschaften und einen Zustand, der durch Operationen verändert werden kann. Jedes Objekt legt fest, wie eine Operation ausgeführt wird. Andere Objekte schicken Nachrichten (Operation/Methode aufrufen) an ein Objekt, das selbst entscheidet, wie die Operation durchgeführt wird. In diesem Sinne handeln Objekte eigenverantwortlich.

Eigenschaften und Zustand eines Objektes werden durch zusammenwirkende Variablen (auch: Attribute oder Felder) und Operationen (auch: Methoden) implementiert.

Nach außen ist die Schnittstelle eines Objektes sichtbar. Dies sind die Methoden und Attribute, auf die zugegriffen werden darf. Das Prinzip der Kapselung *(Information Hiding)* verbirgt Implementierungsdetails vor der Außenwelt.

Objekte werden durch Klassen beschrieben, die das Verhalten eines Objektes definieren. Somit kann eine Klasse auch als abstrakter Datentyp mit zugehörigen Funktionen verstanden werden. In der Tat werden Klassen wie Typen verwendet.

Objekte existieren im Speicher und haben eine Identität: ihre Objektreferenz. Klassen definieren spezielle Methoden zur Initialisierung und Löschung von Objekten: Konstruktoren und Destruktoren. Bei Instantiierung eines Objektes wird der Konstruktor, bei Entfernen des Objektes der Destruktor ausgeführt.

2.3.2. Beziehungen zwischen Objekten

Objekte eines Programms interagieren. Hierzu sind Kommunikationsstrukturen nötig; ein Objekt muss also die Identität anderer Objekte kennen. Hierzu hält ein Objekt die Referenz eines anderen Objektes.

Objektreferenzen werden verwendet, um auf Attribute lesend und schreibend zuzugreifen oder Methoden des Objektes aufzurufen. Letzteres wird auch als Nachrichtenversand bezeichnet. Beschränkungen regeln den Zugriff auf ein Objekt von außen.

Objektreferenzen werden in Variablen gespeichert, sodass sie sowohl kurzzeitig verfügbar sein können, z. B. in lokalen Variablen oder Parametervariablen, als auch längerfristig in Attributen eines Objektes zur Verfügung stehen.

Langfristige Beziehungen zwischen Objekten werden konzeptionell unterschieden:

Assoziationen sind uni- und bidirektionale Beziehungen, bei denen die Partner eine oberflächliche Beziehung eingehen.

Aggregationen bedeuten i. d. R. unidirektionale Beziehungen, bei denen die Beziehung eines Ganzen zu seinen Teilen modelliert wird. Dabei können die Teile jedoch auch ohne das Ganze existieren.

Kompositionen sind Aggregationen, bei denen die Teile nicht ohne das Ganze existieren können.

Objektbeziehungen können uni- und bidirektional sein. Neben einfachen *1:1*-Beziehungen sind auch andere Kardinalitäten erlaubt; z. B. *1:n*- oder *n:n*-Beziehungen.

Bisher unterscheidet keine objektorientierte Programmiersprache diese Beziehungsarten sprachlich, sodass diese explizit implementiert werden müssen.

1:1-Beziehungen können einfach durch Attribute implementiert werden, die eine Referenz auf den Partner enthalten. Mehrere Partner werden durch sog. *Container*-Objekte verwaltet. Diese speichern mehrere Referenzen z. B. als Liste und stellen Zugriffsmethoden zur Verfügung. Ganzes-Teil-Beziehungen können umgesetzt werden, indem bei Objekterzeugung eines Ganzen die Teile durch den Konstruktor ergänzt werden. Analog hätte der Destruktor die Aufgabe Teile zu entfernen, wenn das Ganze nicht mehr benötigt wird.

Letzteres wird in Sprachen, die über *Garbage Collection* verfügen, automatisch erledigt. Ein *Garbage Collector* entfernt regelmäßig Objekte aus dem Speicher, zu denen keine Referenz mehr gehalten wird. Damit werden Teile eines Ganzen automatisch entfernt, nachdem das Ganze als solches entfernt wurde, sofern Referenzen der Teile nicht anderweitig weitergegeben wurden.

2.3.3. Vererbung und Polymorphie

Eine Unterklasse erbt von einer Oberklasse Attribute und Methoden und darf weitere Attribute und Methoden hinzufügen. Dabei können geerbte Methoden überschrieben werden, indem eine Methode mit gleicher Signatur in der Unterklasse definiert wird. Ein Unterklassenobjekt führt dann den Methodenaufruf mit der eigenen statt der überschriebenen Methode aus. Dieses Konzept heißt *dynamische Methodenbindung*, weil erst zur Laufzeit, anhand des Objekttyps, entschieden wird, welche Methode ausgeführt wird.

Das Untertyp-Konzept *(Subtyping)* erlaubt, dass eine Variable vom Typ einer Oberklasse Referenzen auf Objekte vom eigenen Typ und allen Unterklassen aufnehmen kann. Auch hier kommt es zu dynamischer Methodenbindung, denn zur Laufzeit wird die Methode des Objekttyps und nicht des Variablentyps aufgerufen.

Als Erweiterung des grundlegenden Typkonzeptes in statisch typisierten Programmiersprachen ist das Untertyp-Konzept sehr mächtig. Bedenkt man, dass nicht nur Attribute sondern auch lokale Variable und Methodenparameter typisiert sind.

2.3.4. Schnittstellen

Die Schnittstelle einer Klasse ist die Menge der Signaturen der von außen aufrufbaren Methoden. Innerhalb der Klasse wird die Schnittstelle durch Methodenrümpfe implementiert.

Abstrakte Methoden (z. B. in Java) enthalten keinen Rumpf. Unterklassen erben diese und müssen eine Implementierung ergänzen. Im Folgenden werden diese hier auch einfach als Verpflichtung bezeichnet.

In Java gibt es zusätzlich das *Interface*-Konzept. Ein *Interface* ist eine benannte Menge von Methodenschnittstellen (ohne Implementierung). Ähnlich wie bei Klassen, können *Interfaces* die Rolle einer Oberklasse in einer Vererbungsbeziehung spielen. Erbende Klassen werden damit verpflichtet, die Menge von Methoden zu implementieren.

Interfaces können wie Klassen als Typ einer Variablen verwendet werden. Solche Variable können dann Referenzen von allen Objekten aufnehmen, deren Klasse die Schnittstelle implementiert. Man verwendet dies um Kopplungen zu konkreten Klassen zu vermeiden.

In Java gibt es zu jeder Klasse höchstens eine Oberklasse *(single inheritance)*. Mehrfachvererbung ist nur durch das Erben von *Interfaces* erlaubt. Man spricht hier von der Implementierung eines *Interfaces*.

2.4. Guter objektorientierter Entwurf

Objektorientierte Sprachkonstrukte und Konzepte lassen sich vielfältig einsetzen. Dabei entstehen nicht zwingend „gute" Software-Strukturen. Das Ziel dieser Arbeit ist es, besonders verständliche, wartbare, erweiterbare und wiederverwendbare Strukturen zu fördern.

Dieser Abschnitt fasst grundlegende Konzepte des objektorientierten Entwurfs zusammen und diskutiert Gründe und Einsatzszenarien.

Häufig gibt es kein direkt entsprechendes Sprachkonstrukt für diese Konzepte, sodass sie in der jeweiligen Sprache abgebildet werden müssen. Als Konsequenz für die statische Analyse von Programmen ergibt sich, dass der Einsatz dieser Konzepte nicht sofort sichtbar ist.

2.4.1. Vererbungskonzepte

Vererbung kann zu unterschiedlichen Zwecken eingesetzt werden. Es sind vor allem die Konzepte *Abstraktion* und *Spezialisierung* zu unterscheiden.

2.4.1.1. Abstraktion

Bei der Abstraktion betrachtet man verschiedene Arten des selben abstrakten Konzeptes. Dieses wird durch die Oberklasse repräsentiert, Unterklassen repräsentieren unterschiedliche Ausprägungen. In einer korrekten Abstraktionsebene gibt es keine Objekte, die konzeptionell gleichzeitig verschiedenen Unterklassen zugeordnet sein könnten. Disjunkte Unterklassen führen zu Hierarchiebäumen.

Beim Entwurf fasst man im Allgemeinen alle Operationen, die allen Arten gemein sind, in einer Oberklasse zusammen. Operationen, die konzeptionell glei-

ches leisten, aber für unterschiedliche Arten jeweils anderes leisten, werden in Unterklassen implementiert. In der Oberklasse wird die Implementierungsverpflichtung definiert. Bestehende Methoden werden nicht überschrieben; es werden nur Verpflichtungen implementiert.

Es ergeben sich Hierarchieebenen, die zur Entwurfszeit vollständig bekannt sein sollen. Sonst besteht die Gefahr, dass Abstraktionsentscheidungen beim Hinzufügen neuer Klassen revidiert werden müssen.

2.4.1.2. Spezialisierung

Zu einer gegebenen Oberklasse wird eine spezialisierte Unterklasse gebildet. Es handelt sich dabei um eine Erweiterung der Oberklasse im Sinne der Ersetzbarkeit. Die Unterklasse fügt spezifische Operationen hinzu.

Die Oberklasse realisiert ein komplexes Konzept, das in anwendungsspezifischen Ausprägungen vorkommt. Man erhält eine sehr wirksame Wiederverwendung der umfangreichen Funktionalität der Oberklasse.

Wiederverwendung ist in solchen Szenarien sorgfältig geplant. Dies erfordert eine Balance zwischen mächtiger Funktionalität und vielfältiger Verwendbarkeit. Vorbereitete Erweiterungsstellen sind die vererbten „Gene“ der Oberklasse. Solche Oberklassen sind häufig Teile eines komplexen *Frameworks*. Diese implementieren umfangreiche anwendungsspezifische Konzepte und bieten diese zur Wiederverwendung an.

Ein Beispiel ist das AWT-Framework für Benutzungsoberflächen. Darin ist z. B. die Klasse *Frame*, die ein Fenster repräsentiert, enthalten. Eine Unterklasse wird verpflichtet, die *paint*-Methode zu implementieren; diese zeichnet den Fensterinhalt. Um andere Aufgaben wie Rahmen oder Benutzerinteraktion muss sich die Unterklasse nicht bemühen.

Bei der Abstraktion werden ganze vollständig bekannte Hierarchieebenen i. d. R. vom Speziellen hin zum Allgemeinen entwickelt. Anders bei der Spezialisierung: hier wird eine einzelne Klasse zum Zwecke der Wiederverwendung abgeleitet.

2.4.2. Rollen

Ein Objekt spielt eine Rolle, wenn es in der Lage ist, auf eine bestimmte Art und Weise zu reagieren. Implementiert werden Rollen durch *Interfaces*. Jede Klasse, die ein *Interface* implementiert, kann in dessen Rolle verwendet werden. *Interfaces* dienen daher als Typabstraktion. Benutzer von Objekten verwenden diese in der gewünschten Rolle, indem sie als Typ anstelle einer konkreten Klassen das *Interface* verwenden.

Ein Beispiel ist die Rolle *Serializable* aus der Java-Klassenbibliothek. Klassen, deren Objekte in einer Binärkodierung geschrieben und gelesen werden sollen, implementieren diese Rolle. Benutzer serialisierter Objekte beziehen sich nur auf die Rolle *Serializable*.

Falsche Abstraktion liegt vor, wenn eine *hat*-Beziehung statt einer *ist*-Beziehung vorliegt. D. h. ein Objekt spielt verschiedene Rollen gleichzeitig oder nacheinander. Dies lässt sich leicht ersetzen durch Delegation an solche Klassen, die diese Rolle implementieren.

2.4.3. Spezifikation

Ein abstraktes Konzept wird als Schnittstelle spezifiziert. Klassen, die dieses Konzept realisieren, implementieren die Schnittstelle. Wie bei Rollen benutzt der Anwender die Schnittstelle, und löst sich damit von einer speziellen Realisierung.

Zur Spezifikation kann statt eines *Interfaces* auch eine abstrakte Klasse mit Implementierungsverpflichtungen oder Beispielimplementierungen verwendet werden. Diese werden häufig *Adapter* genannt. Bildet man eine Unterklasse eines Adapters, so kann die gesamte vorgegebene Funktionalität übernommen werden; ggf. können einzelne Methoden überschrieben werden.

Spezifikation ist ähnlich der Abstraktion. Allerdings wird bei der Abstraktion eine vollständige Abstraktionsebene entwickelt, hier werden einzelne Ausprägungen bei Bedarf nacheinander ergänzt. Im Vergleich zu Rollen wird eine Spezifikation durch eine Klasse implementiert, wobei häufig mehrere Rollen von einer Klasse gespielt werden. Die Grenzen verschwimmen.

2.4.4. Wiederverwendung

Wiederverwendung in der prozeduralen Programmierung beschränkt sich auf die Verwendung von bestehenden Funktionen bzw. Bibliotheken und den zugehörigen Datenstrukturen. In der objektorientierten Programmierung kommt die Vererbung als mächtiges Werkzeug der Wiederverwendung hinzu.

Man unterscheidet Wiederverwendung durch Komposition und durch Vererbung. Bei der Komposition kennt eine neue Klasse bestehende Klassen und verwendet diese, indem Nachrichten an deren öffentliche Schnittstelle geschickt werden. Bei der Vererbung ist die neue Klasse eine Unterklasse der bestehenden Klasse, sodass diese alle Eigenschaften und Implementierungen der bestehenden Klasse erbt und überall in dessen Kontext eingesetzt werden kann. Die Unterklasse darf nun bestehende Implementierungen überschreiben und auf alle Attribute zugreifen. Dies erzeugt eine starke Kopplung, sodass interne Änderungen der bestehenden Klasse wahrscheinlich auch Auswirkungen auf die beerbte Klasse haben.

Wiederverwendung durch Komposition erhält die Kapselung der bestehenden Klasse und bedeutet eine lose Kopplung. Bei Vererbung öffnet sich die bestehende zur neuen Klasse und geht eine feste Bindung ein.

Die folgenden Kriterien [19, 20] sprechen für den Einsatz von Vererbung:

- Es besteht eine konzeptionelle *ist*-Beziehung zwischen Unterklasse und Oberklasse,
- Objekte gehören der Klasse für immer an, denn Objekte können ihre Klasse nicht wechseln,
- die Unterklasse erweitert die Oberklasse, sie verändert und entfernt keine Eigenschaft,
- es wird bewusst eines der beschriebenen Vererbungskonzepte eingesetzt.

Es gibt gute Gründe, dass eine neue Klasse eine bestehende Klasse durch Komposition anbindet, statt von dieser zu erben:

- Konzeptionell enthalten Objekte der neuen Klasse Objekte der bestehenden Klasse; es besteht eine *hat*-Beziehung statt einer *ist*-Beziehung,
- die neue Klasse realisiert ein anderes Konzept als die bestehende Klasse,
- Operationen der bestehenden Klasse passen schlecht für die neue Klasse und müssten ggf. „entfernt“ werden,
- es soll nicht bekannt werden, dass die neue Klasse durch eine bestehende Klasse implementiert wurde,
- die Kapselung der bestehenden Klasse soll erhalten bleiben.

2.4.5. Entwurfsmuster

Entwurfsmuster [38] beschreiben Lösungen zu häufig auftretenden Aufgaben in der Software-Entwicklung in der Form von objektorientierten Schemata. Diese repräsentieren erprobte Lösungen, die wartbare, flexible und adaptierbare Software-Strukturen erzeugen.

Die Beschreibung von Entwurfsmustern ist weitestgehend informell, folgt aber einem festen Schema:

Name Jedes Muster hat einen sprechenden Namen, der auf den Zweck hindeutet. Diese Namen sind inzwischen in die Sprechweise von Software-Entwicklern eingegangen.

Aufgabe Das Anwendungsspektrum des Musters wird durch typische Situationen und Beispiele beschrieben und durch die Beschreibung umgebender Strukturen und nötiger Voraussetzungen gestützt.

Lösung Als Lösung werden objektorientierte Strukturen vorgeschlagen, die an eigene Einsatzszenarien angepasst werden können. Erhalten bleibt jedoch stets das spezielle Zusammenspiel zwischen verschiedenen Teilaufgaben, die i. d. R. jeweils durch einzelne Klassen repräsentiert werden. Beispiele verdeutlichen die Umsetzung des Musters.

Folgerungen Jedes Muster hat seinen Nutzen, dem aber ggf. auch Kosten gegenüber stehen. Zusammen mit weiteren Beispielen werden Konsequenzen diskutiert.

Der Einsatz jedes Entwurfsmusters verbessert die Wartbarkeit von Programmen. Sofern dem Wartungsingenieur bekannt ist, dass ein Muster eingesetzt wurde, lassen sich konkrete Klassen, die für einzelne Teilaufgaben zuständig sind, schnell identifizieren. Damit entfällt der hohe Aufwand, den Quelltext detailliert zu untersuchen und zu verstehen. Die Kenntnis der spezifischen Strukturen und Abläufe des eingesetzten Entwurfsmusters erlaubt, das Programm gezielt zu ändern.

Wartung ist immer dann besonders einfach, wenn die Teilaufgaben innerhalb des Programms an ausgewiesenen Stellen implementiert sind. Einige Entwurfsmuster sind daher besonders aus Sicht der einfachen Wartung und Adaption interessant.

Entwurfsmuster leisten häufig einen wertvollen Beitrag zur Entkoppelung von Programmteilen. Im Folgenden werden zu diesem Aspekt die drei Muster *Abstract Factory*, *Bridge* und *Observer* skizziert:

Abstract Factory [38, 87ff.] beschreibt ein Schema, um verwandte Objekte zu erzeugen, ohne die konkreten Klassen zu kennen. Als Beispiel wird die Verwendung unterschiedlicher GUI-Systeme genannt. Das Programm soll sich nicht auf ein System festlegen; der Austausch ist vorgesehen.

Factories stellen Methoden bereit, mit denen verschiedene Objekte einer Familie erzeugt werden können. Dabei existieren zu einer abstrakten Oberklasse, der *AbstractFactory*, mehrere konkrete Unterklassen, die *ConcreteFactories*. Der Benutzer wählt eine und lässt sich von ihr Objekte der gewählten Familie erzeugen. Jede Methode erzeugt Objekte eines Familienmitgliedes.

Products sind die Familienmitglieder. Für jedes existiert eine Oberklasse, das *AbstractProduct*, und für jede Familie eine konkrete Unterklasse, die *ConcreteProducts*.

Der Benutzer hantiert nur mit den abstrakten Oberklassen der einzelnen Familienmitglieder und verwendet also niemals konkrete Produktklassen.

Somit wird der Benutzer nur einmal mit den verschiedenen Familien konfrontiert, genau dann, wenn er eine konkrete *Factory* erzeugt. Dies wird Konfiguration genannt. Nachdem der Benutzer die Familie festgelegt hat, werden deren konkrete Mitglieder automatisch und transparent verwendet.

Vorbedingungen und Einschränkungen dieser Programmstruktur werden hier nicht weiter diskutiert. Interessant ist, dass hier durch einen einzigen Schalter von einer Produktart auf eine andere umgestellt werden kann.

Für das Entwurfsmuster *Abstract Factory* wurde das Konzept der Spezifikation eingesetzt. Der Benutzer wird durch Einsatz der Untertypeigenschaft wirksam entkoppelt.

Bridge [38, 151ff.] beschreibt ein Schema, um Spezifikation von der Implementierung zu entkoppeln. Auch hier wird wieder die Verwendung unterschiedlicher GUI-Systeme als Beispiel genannt. Man würde zu einer abstrakten Oberklasse mehrere konkrete Implementierungen als Unterklasse binden. Eine Brücke erlaubt die Spezifikation zu verfeinern und unabhängig weitere Implementierungen hinzuzufügen.

Der Benutzer verwendet das eine Ende der Brücke, die Spezifikation. Diese unterhält eine Assoziation zu einer konkreten Implementierung — dem anderen Ende der Brücke, sodass alle Aufgaben an dieses Ende delegiert werden können. Die konkrete Implementierung kann zur Laufzeit gewählt oder ausgetauscht werden. Wird die Implementierung verändert, so muss ggf. nur die delegierende Verwendung innerhalb der Spezifikation angepasst werden; Änderungen in anderen nutzenden Programmteilen werden vermieden.

Eine Brücke erlaubt die unabhängige Verfeinerung sowohl der Spezifikation als auch der Implementierungen. Der Benutzer wird nur von Änderungen der Spezifikation betroffen; von der Implementierung ist er wirkungsvoll entkoppelt.

Observer [38, 293ff.] Objekte werden über Zustandsänderungen benachrichtigt. Ein Beispiel ist die Änderung einer Datenstruktur, deren Visualisierung angeglichen werden muss.

Beteiligt sind hier die *Observer*-Objekte, die bei einem *Subject* registriert werden. Finden Änderungen statt, wird jedes registrierte Objekt darüber informiert, sodass es auf die Änderung reagieren kann.

Ähnlich wie bei der *Abstract Factory* werden jeweils konkrete Unterklassen der Beteiligten gebildet, welche die Schnittstellen implementieren.

> Das Schema erlaubt es, Abhängigkeiten zu entkoppeln und Beziehungen zwischen den Beteiligten dynamisch zu ändern.

In Entwurfsmustern werden die bekannten objektorientierten Konzepte Abstraktion, Spezifikation, Spezialisierung und Delegation eingesetzt und jeweils in einem bestimmten anwendungsspezifischen Zusammenhang genutzt. Klar definierte Aufgaben der Beteiligten bilden Rollen aus, die einfaches Verständnis der Programmstrukturen und damit leichte Wartung und Erweiterung ermöglichen.

2.4.6. Komponenten

Das *Teile und Herrsche*-Prinzip setzt sich in der Konzeption moderner Software-Architekturen fort. Ähnlich wie Entwurfsmuster, die den Entwurf im Kleinen beschreiben, gibt es Architekturmuster die einen Entwurf im Großen beschreiben. Hier werden sie in der Sprechweise der *Quasar*-Architekturprinzipien [78] vorgestellt.

Eine Komponente exportiert eine Schnittstelle und importiert bzw. verwendet ggf. andere Schnittstellen. Der Schnittstellenbegriff beschreibt hier nicht nur die Menge der Methodensignaturen, sondern auch die Semantik der einzelnen Methoden und zusätzliche Informationen, die zur Benutzung benötigt werden. Eine Komponente versteckt seine Implementierung, nur die Schnittstelle ist sichtbar. Komponenten sind (ggf. hierarchisch) komponierbar und sind Teil des Entwurfs und der Planung eines Software-Systems.

Heutige Programmiersprachen verfügen nicht über Komponentenkonstrukte. Man bildet sie daher auf Bibliotheken (C/C++), Pakete (Java) oder *Assemblies* (C#) ab.

Im Architekturentwurf werden einzelnen Komponenten bestimmte Wissens- bzw. Themengebiete zugeordnet: sogenannte *Software-Kategorien*. Jede Komponente sollte sich möglichst nur um eine Kategorie kümmern. Damit wird das Prinzip der Trennung von Zuständigkeiten gefördert.

Ein Architekturentwurf ist ein Graph, in dem die Knoten Komponenten einer bestimmten Kategorie sind und Kanten Benutzungsrelationen darstellen. Man findet hier Komponenten, die Aufgaben implementieren, die überall verfügbar und wiederverwendbar sind, z. B. Listen, Tabellen, reguläre Ausdrücke oder Behälter. Diese Kategorie wird *0-Software* genannt. Jede andere Komponente darf *0-Software* verwenden.

Weitere wichtige Arten sind *A-* und *T-Software*. Dabei befasst sich *A-Software* nur mit den fachlichen Entitäten des Modells, sowie deren Beziehungen und Verhalten innerhalb des Software-Systems: z. B. ein Student, ein Mitarbeiter oder eine Klausur. *T-Software* implementiert bestimmte technische Aspekte: z. B. HTML-Seiten aufbauen, SQL-Anfragen zusammensetzen, auf Netzwerkdienste zugreifen.

I. d. R. ergeben sich baumartige Komponentengraphen, in denen *0-Software* die Wurzel bildet, *A-Software* diese verwendet und damit innere Knoten besetzt und *T-Software* in den Blättern zu finden ist.

Der gute Architekturentwurf unterliegt Regeln, mit denen sich der Entwurf bewerten lässt. Vermengung von *A-* und *T-Software* gilt als unfein; gewollt ist, technische Bausteine austauschen oder auf deren Änderungen an zentraler Stelle reagieren zu können. Komponenten einer verfeinerten Kategorie dürfen auf Komponenten allgemeinerer Kategorien zugreifen, aber nicht umgekehrt. Diese und weitere Regeln führen zu flexiblen, verständlichen und wartbaren Architekturentwürfen.

2.5. Zusammenfassung

Die Abstraktion ist das Kernprinzip der Software-Entwicklung. Sie erlaubt es, nach dem *Teile und Herrsche*-Muster Beteiligte, Aufgaben, Zuständigkeiten und Abläufe getrennt zu konzipieren und zu implementieren.

Abstraktionsebenen reichen vom Architekturentwurf, über den Entwurf von Komponenten, bis hin zur „Feinspezifikation" — der Implementierung. Auf allen Ebenen bieten sie Projektbeteiligten das nötige Vokabular zu effektiver Kommunikation.

Programmiersprachen bieten Sprachkonstrukte, um Lösungen auf fast allen Ebenen zu formulieren. So findet man auf der Entwurfsebene Klassen und deren Beziehungen direkt wieder. Auf der Architekturebene sind z. B. mit Java-Paketen gerade noch Komponenten erkennbar.

Dies bietet die Chance, Entwurfs- oder auch Architekturbewertung nur anhand der Implementierung vorzunehmen, sodass häufig verlorene oder veraltete Entwurfsdokumentation nicht benötigt wird.

3. Entwurfsmängel

Inhalt

3.1. Übersicht **28**
3.1.1. Refactoring 28
3.1.2. Abgrenzung 29
3.1.3. Vielfalt der Entwurfsmängel 30
3.2. Beispiele für Entwurfsmängel **32**
3.2.1. Große Klasse 32
3.2.2. Datenklasse 34
3.2.3. Lange Methode 34
3.2.4. Neid 36
3.2.5. Ausgeschlagenes Erbe 37
3.2.6. Nachrichtenketten und Vermittler 38
3.3. Klassifizierungen **38**
3.4. Zusammenfassung **40**

3.1. Übersicht

In dem vorigen Kapitel wurden Prinzipien der objektorientierten Programmierung beschrieben. Diese umzusetzen und innerhalb eines Software-Projektes mit Leben zu füllen, ist eine schwierige Aufgabe. Man setzt daher häufig auf die manuelle Inspektion der Software (*Software Inspection* oder auch *Code Reviews*) um einen gleichbleibend hohen Qualitätsstandard zu halten.

Entwurfsmängel beschreiben Indizien und Hinweise, die darauf hindeuten, dass eine Programmstelle verbessert werden kann. Hat man solche problematischen Programmstellen gefunden, kann man versuchen, diese zu überarbeiten. Hierzu bieten sich *Refactoring*-Transformationen an, die Programme restrukturieren, ohne deren Verhalten zu verändern.

In diesem Kapitel wird im folgenden Abschnitt zunächst auf die Idee der *Refactoring*-Transformationen eingegangen. Diese beheben keine Programmierfehler, aber sie lösen Strukturprobleme auf der Entwurfsebene. Die hier betrachteten Entwurfsmängel werden daher in Abschnitt 3.1.2 von anderen Fehlern außerhalb der Entwurfsebene abgegrenzt. Eine Übersicht der Entwurfsmängel zeigt dann Abschnitt 3.1.3, bevor in Abschnitt 3.2 einige beispielhaft erläutert und in Abschnitt 3.3 verschiedene Modelle der Klassifizierung skizziert werden.

3.1.1. Refactoring

Refactoring-Transformationen verbessern die innere Struktur eines Programms, ohne das von außen sichtbare Verhalten zu ändern. Diese Transformationen erlauben, Programme zu bereinigen und zu vereinfachen. Fowler greift in [37] diesen Gedanken auf und stellt einen überarbeiteten Katalog, der erstmals von Opdyke [66] [69] beschrieben Transformationen, auf.

Das Ziel dieser Semantik-erhaltenden Transformationen ist dabei stets die innere Struktur und damit die Wartbarkeit, Erweiterbarkeit und Verständlichkeit eines Systems zu verbessern. Der Benutzer des Systems bemerkt davon i. d. R. nichts.

Opdyke unterscheidet zwischen einfachen, feingranularen Transformationen, die eigentlich jedem Programmierer bekannt sind, und daraus zusammengesetzten komplexen Transformationen.

Einfache Transformationen

Zu den einfachen Transformationen gehören: Umbenennen von Methoden, Parametern, Attributen und Klassen; Erzeugen von neuen Programmobjekten; Verschieben von Methoden und Attributen in andere Klassen; Zerteilen von Methoden in mehrere Methoden; Zerteilen von Klassen und weitere. Zu jeder Transformation kann gezeigt werden, dass sie das Verhalten des Programms nicht ändert. Hierzu werden Vorbedingungen formuliert, die für eine sichere Anwendung der jeweiligen Transformation erfüllt sein müssen.

Komplexe Transformationen

Einfache Transformationen lassen sich zu komplexen Transformationen komponieren. Diese erhalten wiederum die Semantik des Programms. Eine gemeinsame Vorbedingung der komponierten Transformationen lässt sich bilden und anwenden [17].

Die Bildung einer Abstraktion ist ein Beispiel für eine solche komplexe Transformation: gleiche hierzu Methodensignaturen mehrerer Klassen aneinander an; erzeuge eine gemeinsame Oberklasse; extrahiere Methoden zu gleichartigen Methoden und verschiebe diese in die Oberklasse.

Komplexe Transformationen werden z. B. verwendet um *Design Pattern* in bestehende Programme einzupflegen [18].

3.1.2. Abgrenzung

Software-Entwicklung nach dem *Teile und Herrsche*-Prinzip lässt sich in die drei Ebenen Architektur, Entwurf und Implementierung trennen. Problemszenarien lassen sich auf allen Ebenen formulieren; Abbildung 3.1 zeigt diese Ebenen.

Programmierfehler Bekannte Untersuchungswerkzeuge (wie z. B. das Lint-Werkzeug [47]) suchen gezielt Programmierfehler. Typischerweise werden hier z. B. nicht initialisierte Variablen, *Array*-Indizierungsgrenzen, Verwendung eines Null-Zeigers oder Division durch Null und ähnliche häufig auftretende Programmierfehler durch statische Programmanalyse ermittelt.

Moderne Übersetzer zeigen dem Benutzer ähnliche Hinweise auf Fehler an.

Architektur- und Prozessfehler Der *Design Pattern*-Idee [38] stehen *AntiPattern* [14] gegenüber. Während *Design Pattern* Lösungsschemata zu Problemen beschreiben, konzentrieren sich *Anti Pattern* auf Lösungsschemata, die mehr Probleme verursachen als zur Problemlösung beizutragen.

Die Autoren unterscheiden Ebenen der Software-Entwicklung, Architektur und Projektplanung. Beispielhaft werden Probleme, Symptome und Konsequenzen beschrieben. Viele der skizzierten Probleme sind im Zwischenmenschlichen zu finden oder beruhen auf falschen Vorstellungen der Projektbeteiligten.

Ein Beispiel ist der *„Golden Hammer“*. Die Autoren zeigen auf, dass Menschen dazu neigen, alle Probleme mit dem (häufig einzigen) verfügbaren und beherrschten Werkzeug zu lösen: *„Wenn das einzige Werkzeug ein Hammer ist, ist alles andere ein Nagel.“*.

Die Grenze der *AntiPattern* zu Entwurfsmängeln verschwimmen. Vor allem im Bereich der Software-Entwicklung lassen sie sich den Entwurfsmängeln

Architektur	**„AntiPatterns“** *Software Development – Software Architecture – Software Project Management*
Entwurf	**„Entwurfsmängel“** *Bad Smells* [Fowler, 2000] – *Development AntiPatterns* [Brown, et. al., 1998] – *Design Characteristics* [Whitmire, 1997] – *Design Heuristics* [Riel, 1996]
Implemen-tierung	**„Programmierfehler“** Nicht initialisierte Variable, *Null*-Referenzen, *Array*-Grenzen, Bibliotheksbenutzung, ...

Abbildung 3.1.: Abgrenzung: Entwurfsmängel zwischen AntiPattern und Programmierfehlern

zuordnen. Hier findet man auch bekannte Metaphern wie die *„God Class“* oder den *„Spaghetti-Code“*.

Entwurfsmängel Bei der Suche nach Entwurfsmängeln sind Programmstrukturen interessant, die auf einen fehlerhaften Entwurf hindeuten. Diese Entwurfsmängel lassen sich zwischen der Architektur- und der technischen Programmebene einordnen.

Im Folgenden werden diese näher beschrieben.

3.1.3. Vielfalt der Entwurfsmängel

Ein Entwurfsmangel, bezeichnet durch eine Metapher, ist eine Menge von vagen Indizien und Hinweisen, die auf den potentiell fehlerhaften Entwurf eines Teiles eines Software-Systems hindeutet.

In der Literatur werden diese als *„Design Heuristics“* [74], *„Design Characteristics“* [92], *„Bad Smells“* [37] oder *„Software Development AntiPatterns“* [14] beschrieben. Die Autoren geben den Entwurfsmängeln i. d. R. sprechende Bezeichner und erklären dem Software-Entwickler und Software-Architekten, wie solche Entwurfsmängel erkannt und behoben werden können.

„Design Heuristics“ [74] von Riel sind die ersten Beschreibungen von Entwurfsmängeln in der Literatur. Riel fasst charakteristische Elemente objektorientierter Programmierung zusammen und beleuchtet Beziehungen zwischen

Programmobjekten. Riel formuliert mehr als 60 Merksätze, die dem Programmierer und Entwerfer helfen, Fehler im Entwurf schon im Entstehen zu erkennen.

Die Merksätze (siehe Anhang A) werden in folgende Kategorien eingeordnet:

Klassen und Objekte beschreibt Kapselung und Schnittstellen von Klassen. Riel hält es für wichtig, Implementierungsdetails zu verbergen und Abhängigkeiten einer Klasse nach außen zu vermeiden. Klassen sollten nur eine einzige Aufgabe erfüllen.

Architektur wird bei Riel im Vergleich zwischen „aktionsorientierten" (also reaktiven) und objektorientierten Programmen aufgegriffen. Probleme sieht Riel besonders in der ungleichmäßigen Verteilung der Zuständigkeiten zwischen Klassen. Er warnt hier vor zu großen und zu kleinen Klassen.

Beziehungen zwischen Klassen und Objekten werden analog zur Beschreibung in Abschnitt 2.3.2 beschrieben. Riel warnt davor, z. B. zu viele Assoziationen zwischen Objekten zu unterhalten und zu viele Methodenaufrufe zu etablieren. Desweiteren empfiehlt Riel eine Kommunikationsrichtung bei Kompositionen einzuhalten.

Später geht Riel detaillierter auf Assoziationsbeziehungen ein und gibt Entscheidungshilfen zur Verwendung von Assoziation oder Komposition.

Vererbung sieht Riel ausschließlich im Zusammenhang mit der Spezialisierung. Er betont Konzepte der Kapselung und der Zusammenfassung gleichartiger Daten und Operationen in den Oberklassen. Riel warnt anhand von Typschlüsseln Fälle zu unterscheiden und rät zum Einsatz von Polymorphie.

Auftretende Mehrfachvererbung sieht Riel generell als Anlass den Entwurf zu überdenken.

„Design Characteristics" [92] von Whitmire beschreiben Eigenschaften von gutem objektorientierten Entwurf aus einer messtheoretischen Sichtweise. Ausgehend von theoretischen Grundlagen der Definition und Berechnung von Metriken ordnet er diese in Kategorien ein (analog der Definition in Abschnitt 5.2.3).

Whitmire beschreibt eine methodische Vorgehensweise um Metriken zur Beurteilung objektorientierter Entwürfe zu erstellen. Er lässt offen, wie diese zur Beurteilung von Software-Qualität verwendet werden können.

„Bad Smells“ [37] von Fowler beschreiben informell eine Reihe von Indizien und Hinweisen, die auf fehlerhaften Entwurf hindeuten. Ausgangspunkt ist hierbei die Suche nach Programmstellen, die durch *Refactoring*-Transformationen verbessert werden können.

Beispiele für solche *„Bad Smells“* sind zu lange Methoden, Klassen mit mehreren Aufgaben, zu viele Parameter oder lokale Variable einer Methode, Verletzung von Datenkapselung, intensive Delegation oder „Erbschaftsstreitigkeiten“.

Fowlers Beschreibungen sind sehr informell und erzeugen beim interessierten Leser eine vage Vorstellung vom Wesen der beschriebenen Problemszenarien. Der Anhang B enthält eine Liste der Entwurfsmängel nach Fowler.

„AntiPatterns“ [14] beschreiben viele Probleme der industriellen Software-Entwicklung. Die Autoren konzentrieren sich dabei stark auf Fehler innerhalb von Entwicklungsprozessen und auf der Ebene des Architekturentwurfs.

Einige Antipattern betreffen auch den feiner granularen Entwurf und damit konkrete Software-Strukturen.

3.2. Beispiele für Entwurfsmängel

Im Folgenden werden einige Entwurfsmängel beschrieben. Dabei wird auf die Beschreibung aus [37] und der älteren Literatur zurück gegriffen. Entwurfsmängel werden absichtlich informell beschrieben. Fowler schreibt:

> „Etwas, das wir hier nicht versuchen werden, ist, Ihnen präzise Kriterien zu geben, wann das Refaktorisieren überfällig ist. Nach unserer Erfahrung erreicht kein System von Metriken die informierte menschliche Intuition.“ [37, Kapitel 3].

3.2.1. Große Klasse

Eine „Große Klasse“ — auch *God Class*, *Blob* oder *Winnebago* genannt — übernimmt zu viel Verantwortung innerhalb eines Software-Systems. Viele andere Klassen des Systems dienen nur als Datenbehälter oder sind mit kleineren Aufgaben betraut.

Abbildung 3.2 zeigt als Beispiel einen Ausschnitt aus einem Klassendiagramm. Man stelle sich vor, dass die Klassen *DrawingArea*, *Circle* und *Point* Teil eines Zeichenprogrammes sind. Die Klasse *DrawingArea* implementiert hier u. a. eine Methode *isInCircle*, die prüft, ob ein als Parameter übergebener Punkt innerhalb eines übergebenen Kreises liegt.

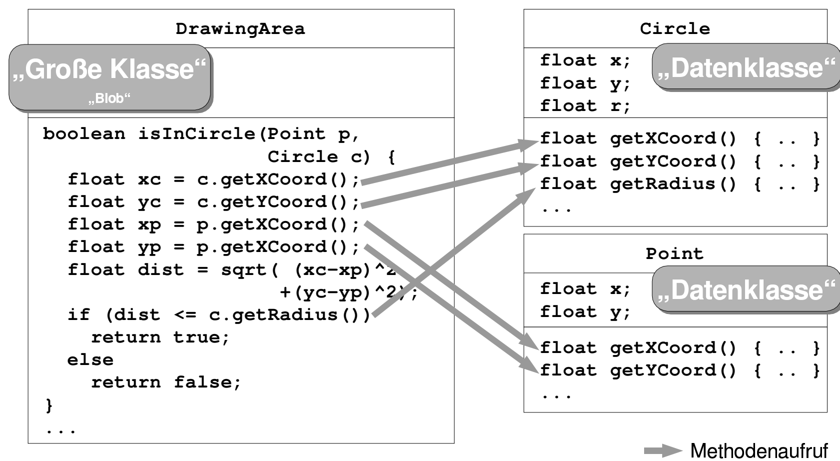

Abbildung 3.2.: Beispiele für die „Große Klasse“ und „Datenklasse“.

Auffällig ist hier, dass *isInCircle* keine Felder der eigenen Klasse verwendet. Es werden ausschließlich Werte von fremden Klassen beschafft und zu einem Ergebnis verrechnet.

Die Klassen *Circle* und *Point* besitzen einige Datenfelder und zugehörige *get*-Methoden. Diese Klassen sind typische Beispiele für „Datenklassen“, denn sie haben kein eigenes Verhalten, einen hohen inneren Zusammenhang (Kohäsion) zwischen Feldern und Methoden, sowie eine geringe Bindung zu anderen Klassen (Kopplung).

Die Klasse *DrawingArea* kann als „Große Klasse“ bezeichnet werden. Es wird das Prinzip verletzt, Daten und Verhalten zu kapseln: die Daten liegen in den Datenklassen, das Verhalten in der eigenen Klasse. Stellt man sich vor, dass diese Klasse weitere ähnliche Methoden implementiert, drängt sich der Eindruck auf, dass die Klasse für mehr als nur eine Aufgabe zuständig ist. Relativ viele Anweisungen der Methoden, eine geringe Kohäsion und starke Kopplung zu anderen Klassen sind hier die Kriterien, die auf eine „Große Klasse“ hindeuten.

Mit Hilfe von *Refactoring*-Transformationen lassen sich die beteiligten Klassen umstrukturieren. Ein Vorschlag für eine neue Struktur ist in Abbildung 3.3 zu sehen. Das Verhalten der Methode *isInCircle* bleibt unverändert, sie delegiert die Berechnung jedoch an den Kreis, der jetzt feststellen kann, ob ein übergebener Punkt innerhalb liegt. Hierzu wird wiederum die Berechnung des Abstandes zwischen zwei Punkten an die Klasse *Point* delegiert.

Kohärenzen und Kopplungen ergeben ein homogeneres Bild. Klassen operieren auf eigenen Daten, sie nehmen eigene Aufgaben war, und verbergen Details vor der Außenwelt.

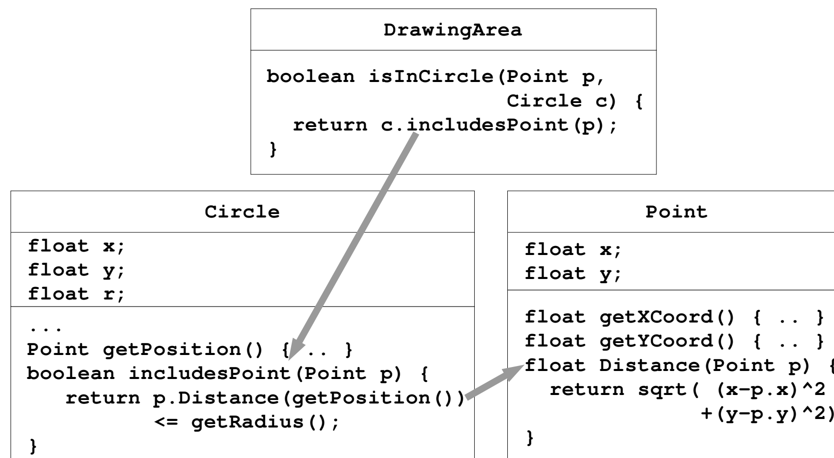

Abbildung 3.3.: Beispiel aus Abb. 3.2 nach *Refactoring*-Transformationen.

3.2.2. Datenklasse

Abbildung 3.2 zeigt auch die zwei Datenklassen *Circle* und *Point*. Wie bereits beschrieben, enthalten sie im Wesentlichen Attribute und zugehörige *set*- und *get*-Operationen, sodass eigenes Verhalten nicht zu erkennen ist.

Aufgrund der fehlenden Funktionalität sind Benutzer gezwungen, sich mit der angebotenen Datenrepräsentation auseinander zu setzen. Vermutlich manipulieren Benutzer die Objekte der Klasse viel zu detailliert. Der Benutzer benötigt genaue Kenntnisse über die Klasse: Wertebereiche der Daten müssen ebenso wie ggf. weitere Randbedingungen bekannt sein.

Das Kapselungsprinzip könnte bei solchen Klassen verletzt sein. Hier sind dann zunächst entsprechende Zugriffsoperationen zu ergänzen und der direkte Zugriff auf Attribute zu verhindern.

Eine weitere Verbesserung erhält man, indem man Nutzer der Klasse untersucht und deren Umgang mit der Klasse beobachtet. Nützliches Verhalten kann extrahiert und in die Klasse verschoben werden.

Fowler charakterisiert treffend:

> „Datenklassen sind wie Kinder. Anfangs lassen wir sie gewähren, aber um erwachsen zu werden, müssen sie Verantwortung übernehmen.“ [37, S. 81].

3.2.3. Lange Methode

Die „Lange Methode“ gehört zu den prominenten Entwurfsmängeln. Auch unter dem Namen *„Spaghetti-Code“* ist sie wohl jedem Programmierer bereits begegnet.

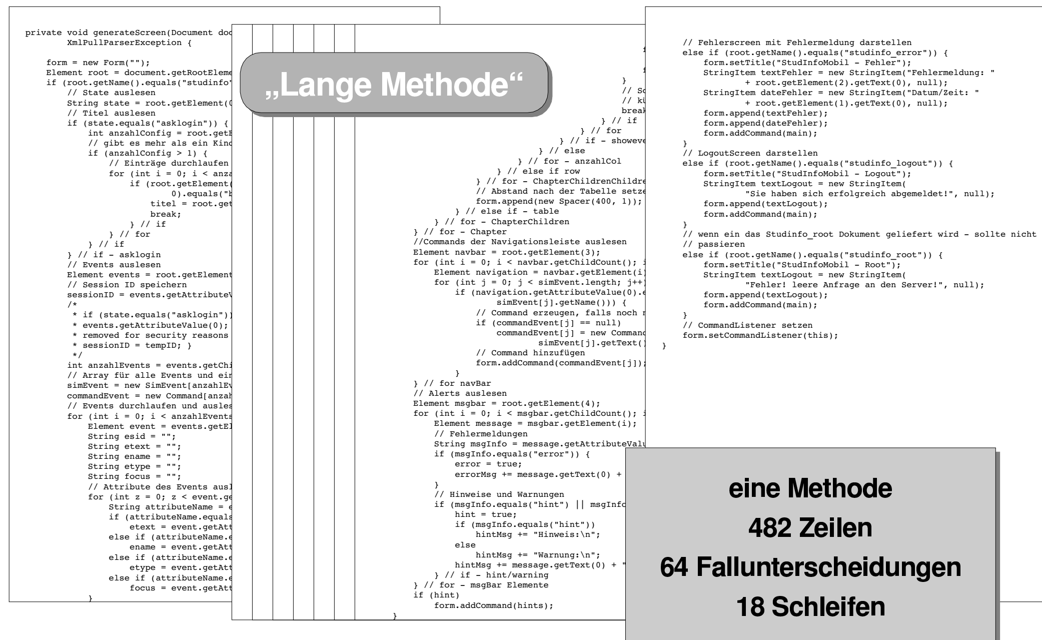

Abbildung 3.4.: Beispiel für den Entwurfsmangel „Lange Methode“.

Fowler betont, dass kurze Methoden der beste Garant für langlebige Programme seien. Dies entspricht dem in Abschnitt 2.2.1 beschriebenen *Teile und Herrsche*-Prinzip der strukturierten Programmierung. Methoden bilden getrennte Funktionseinheiten. Eine feingranulare Verteilung der Gesamtfunktionalität auf viele Funktionen erleichtert das Verständnis von jeder einzelnen Funktion und ermöglicht einzelne Konzepte wiederverwendbar — durch Aufruf — zu implementieren.

Aber auch in der objektorientierten Programmierung sind es die „kleinen Helfer“, die in der Summe komplexe Abläufe beschreiben. Dabei stehen für Fowler besonders gute Methodenbezeichner an erster Stelle. Einer Methode soll, ohne den Rumpf anzuschauen, ihre Funktionsweise bzw. Nutzen anzusehen sein. Als Faustregel soll immer dann eine neue Methode erstellt werden, wenn man das Gefühl habe, etwas kommentieren zu müssen. Demnach sind Kommentare im Programmtext Hinweise dafür, dass eine Methode zerteilt werden müsse. Dies greift Fowler auch in seinem Entwurfsmangel „Kommentare“ auf.

Abbildung 3.4 zeigt eine Java-Methode mit dem Bezeichner *generateScreen*. Sie stammt aus einem Programm für Mobiltelefone, das für das *StudInfo*-Projekt [88] implementiert wurde. Die Methode hat die Aufgabe ein als Parameter übergebenes XML-Dokument zu analysieren und eine entsprechende Darstellung für

die Anzeige des Mobiltelefons zu generieren. Der Programmierer implementierte diese Aufgabe, indem Informationen des Dokumentes in einer Reihe von lokalen Variablen gespeichert werden. Die komplexe Struktur des Dokumentes wird durch eine ebenso komplexe Struktur von geschachtelten Fallunterscheidungen und Schleifen abgebildet.

Insgesamt enthält die Methode 482 Zeilen Programmtext, 64 Fallunterscheidungen und 18 Schleifen. Das Kriterium, dass hier für eine „Lange Methode" spricht, ist nicht die Anzahl der Zeilen, sondern die Komplexität der Kontrollstruktur dieser Methode. Hätten die 482 Zeilen eine Folge von ebensovielen z. B. Ausgabeanweisungen enthalten, so könnte man diese ebensogut als eine Anweisung ansehen und man spräche nicht von einer langen Methode.

Fowler gibt sehr aggressive Empfehlungen zur Zerlegung langer Methoden. So gilt es zunächst Hemmnisse aus dem Weg zu räumen. Viele lokale Variable oder lange Parameterlisten erschweren die Extraktion von Methoden, weil häufig viele Parameter übergeben werden müssen. So könnten zunächst lokale Variable durch Abfragen ersetzt oder lange Parameterlisten in einem Parameterobjekt gekapselt werden.

Auf der Suche nach Schnittlinien innerhalb einer Methode empfiehlt Fowler nach Kommentaren zu suchen. Jeder kommentierte Abschnitt könnte eine Methode werden. Der Bezeichner der Methode ergäbe sich aus dem Kommentar. Sehr weit geht der Vorschlag, auch Kontrollstrukturen zu zerlegen, sodass aus einem Schleifenrumpf oder dem Zweig einer Fallunterscheidung eine eigene Methode wird und die Schleife bzw. Fallunterscheidung wiederum in einer eigenen Methode gekapselt wird.

3.2.4. Neid

Das Kapselungsprinzip ist eines der wesentlichen Merkmale objektorientierter Programmierung. Kapselung bedeutet für Klassen (und deren Objekte), dass keine Implementierungsdetails — seien es Methodenimplementierungen oder Attribute — nach außen bekannt werden. Die Nutzer einer Klasse verwenden ausschließlich die angebotene öffentliche Schnittstelle.

Aber auch wenn man diese einfachen Regeln beherzigt, kann es vorkommen, dass eine Klasse (bzw. deren Objekte) nur mit intimem Detailwissen verwendet werden kann. Die beiden Datenklassen in Abbildung 3.3 sind ein gutes Beispiel. Die Verwendung der öffentlichen Schnittstellen der Klassen *Circle* und *Point* erlaubt es hier, direkt die Attribute zu manipulieren. In Abschnitt 3.2.1 wurde dies bereits als Problemfall beschrieben. Das fehlende Verhalten innerhalb der Klassen führt dazu, dass das Verhalten außerhalb implementiert wird. Da nur mit den rohen Daten hantiert wird, ist entsprechendes Detailwissen beim Benutzer erforderlich. Es muss hier z. B. bekannt sein, welche Wertebereiche erlaubt sind oder

welche Rand- und Konsistenzbedingungen gelten müssen.

Der Entwurfsmangel „Neid“ charakterisiert Methoden, die sich mit den Details fremder Klassen befassen.[1] Deutlich stärker sichtbar wird der Entwurfsmangel, wenn sich die Methode intensiver mit den Details fremder Klassen als mit der eigenen Klasse auseinandersetzt. Dann wird neben der unerwünschten Kopplung zu einer fremden Klasse auch eine mangelnde Beziehung zur eigenen Klasse sichtbar.

Fowler empfiehlt hier die Methode in die fremde Klasse zu verschieben. Nicht immer gibt es jedoch genau eine beteiligte Klasse, sodass die Methode ggf. zerteilt oder das Verschiebungsziel ausgewählt werden muss.

„Neid“ beschreibt die bekannten Prinzipien Kopplung und Kohärenz, die speziell auf die Beziehung zwischen Methoden und fremden bzw. der eigenen Klasse fokussiert wird. Detailwissen, das die neidische Methode hat, ergänzt das Schema dieses Entwurfsmangels.

Innerhalb der *Refactoring*-Gemeinde diskutiert man, ob die Metapher „Neid“ den Kern des Problems trifft. Eine der vorgeschlagenen Alternativen ist „Fehlplatziertes Verhalten“.

Außerdem besteht unter den *Refactoring*-Experten nicht immer Einigkeit, ob sich „Neid“ nur auf Methoden oder auch auf Klassen bezieht. So wird der Entwurfsmangel häufig gleichbedeutend mit der „Unangebrachten Intimität“, mit der vergleichbare enge Beziehungen zwischen Klassen erklärt werden, verwendet.

3.2.5. Ausgeschlagenes Erbe

Der Entwurfsmangel „Ausgeschlagenes Erbe“ beschreibt eine besondere Form von Erbschaftsstreitigkeiten. Der Erbende zeigt kein Interesse und lehnt das Erbe ab. I. d. R. sind mit dem Erbe unangenehme Verpflichtungen verbunden, die es zu vermeiden gilt.

Auch in der objektorientierten Programmierung existiert dieses Problem: eine Klasse erbt von einer andern Klasse, ist aber nicht bereit die geerbten Verpflichtungen zu übernehmen. In Abschnitt 2.4.1 wurden bereits Kriterien beschrieben, die für eine Vererbungsbeziehung erfüllt sein müssen. Es besteht der Verdacht, dass die konzeptionelle Untertypeigenschaft verletzt wird. Die erbende Klasse ist vermutlich konzeptionell nicht in jedem Kontext einsetzbar, in dem die Oberklasse verwendet werden kann.

Ein klassisches Beispiel für ein ausgeschlagenes Erbe ist in der Java-Standardbiliothek zu finden [83]. Dort werden die Konzepte *Stapel* und *Liste* so implementiert, dass der Stapel als Erbe der Liste eingesetzt wird. Nun erlaubt

[1]Natürlich „befassen“ sich Methoden nicht mit Klassen. Gemeint ist: es existiert im Methodenrumpf ein lesender oder schreibender Zugriff auf Attribute eines Objektes bzw. ein Aufruf einer Methode der entsprechenden Klasse.

aber die Liste den wahlfreien Zugriff auf alle Listenelemente, wobei beim Stapel nur auf das jeweils oberste Element zugegriffen werden darf. Die *push*- und *pop*-Operationen werden mit Hilfe der Operationen des Listenzugriffs realisiert.

Vollständig zerstört würde die konzeptionelle Untertypeigenschaft, wenn nicht gewünschte Methoden durch Methoden mit leerem Rumpf oder einer Fehlermeldung überschrieben würden. So lässt sich aber ein Stapel noch technisch im Kontext einer Liste verwenden und gibt dazu aber den wahlfreien Elementzugriff frei.

3.2.6. Nachrichtenketten und Vermittler

„Nachrichtenketten" beschreiben Pfade durch Objektgeflechte. Ein Benutzer schickt eine Nachricht an ein Objekt und erhält ein weiteres Objekt als Ergebnis. An dieses wird wiederum eine Nachricht verschickt um ein weiteres Objekt zu erhalten. Man navigiert also durch die Objektstruktur, bis man das gewünscht Objekt erreicht hat und dieses schließlich verwendet.

Problematisch an diesem Vorgehen ist wiederum das Wissen, dass man in diesem Fall über die Struktur des vorliegende Objektgeflecht und über die Eigenschaften der beteiligten Objekte benötigt. Alle Änderungen in den beteiligten Klassen können somit an vielen Benutzungsstellen ebenfalls Anpassungen erfordern.

Das *Law of Demeter* [53] besagt, dass Nachrichten nur an Objekte verschickt werden dürfen, die entweder als Parameter einer Methode übergeben wurden oder in Attributen der eigenen Klasse gespeichert sind. Nachrichtenketten verletzen dieses.

Fowler empfiehlt hier, ausgesuchte Klassen mit speziellen Methoden auszustatten, die häufig gewünschte Navigationsziele liefern. Damit kapseln diese Klassen Teile des Objektgeflechtes.

Im Gegensatz zu den „Nachrichtenketten" steht der „Vermittler". Dieser fällt durch eine größere Anzahl von delegierenden Klassen auf, die z. B. Objekte aus einem Objektgeflecht liefern. So entstehen mit der Zeit Fassaden, die ganze Teilsysteme kapseln.

Fowler empfiehlt, eine geeignete Balance zwischen Kapselung und direkter Nutzung von Objekten zu etablieren.

3.3. Klassifizierungen

Neben der offensichtlichen Klassifizierung der Entwurfsmängel nach der Art der betroffenen Programmobjekte, findet man in der Literatur auch den Ansatz Entwurfsmängel anhand der Probleme, die sie verursachen, einzuteilen.

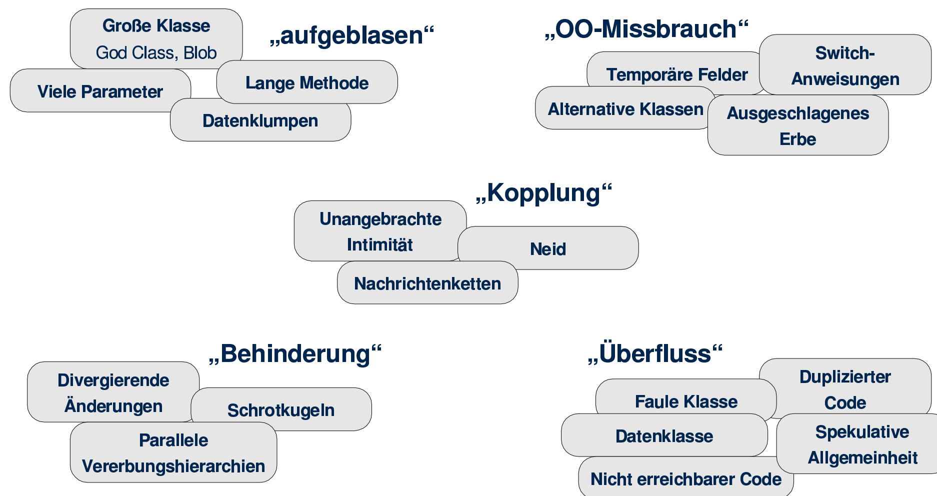

Abbildung 3.5.: Problem-orientierte Klassifizierung von Entwurfsmängeln [65].

Direkte Klassifizierung

Entwurfsmängel lasen sich direkt anhand der Art der Programmobjekte klassifizieren, die von diesen betroffen werden. So betrifft eine „Große Klasse“ nur Klassen und eine „Lange Methode“ nur Methoden.

Andere Entwurfsmängel beschreiben Beziehungen zwischen Programmobjekten. So betrifft z. B. „Neid“ die Beziehung zwischen Methoden und Klassen und „Ausgeschlagenes Erbe“ die Vererbungsbeziehung zwischen zwei Klassen.

Problem-orientierte Klassifizierung

Mäntylä beschreibt in [65] eine problem-orientierte Einteilung von Entwurfsmängeln. Abbildung 3.5 zeigt eine Übersicht, in der Mäntylä folgende Arten unterscheidet:

Kopplung Wie bereits oben beschrieben, deuten einige Entwurfsmängel auf übermäßige Kopplung zwischen Programmobjekten hin.

Behinderung Entwurfsmängel behindern Wartung und Verständnis. Muss man z. B. wegen einer Änderung an vielen anderen Stellen ebenfalls Änderungen nachpflegen, dann spricht man davon „Schrotkugeln“ heraus zu operieren.

Aufgeblasen Als Folge sukzessiver Weiterentwicklung geraten manche Programmstellen mit der Zeit zu groß. Die wichtigsten Entwurfsmängel dieser Kategorie wurden bereits beschrieben.

OO-Missbrauch Neben dem ausgeschlagenem Erbe sind z. B. auch *switch*-Anweisungen Hinweise auf missbräuchliche Anwendung oder missver-

standende objektorientierte Konzepte. Vor allem wenn ähnliche *switch*-Anweisungen häufiger auftreten, könnte eine Abstraktion, bei der dynamische Methodenbindung genutzt würde, eine Alternative sein.

Überfluss Die letzte Kategorie enthält Entwurfsmängel, die auf überflüssige Programmteile, wie duplizierten Code, unerreichbare Programmstellen oder ähnliches hindeuten.

3.4. Zusammenfassung

Entwurfsmängel beschreiben Hinweise auf den potentiell fehlerhaften Entwurf von Teilen eines objektorientierten Software-Systems. Wird ein Entwurfsmangel an einer Programmstelle entdeckt, so lohnt es sich, diese genauer zu inspizieren und die dort vorliegenden Programmstrukturen zu hinterfragen. Durch *Refactoring*-Transformationen lassen sich problematische Strukturen mit geringem Risiko umformen. Das Ziel ist es, vor allem besser wartbare, verständlichere, wiederverwendbare und leichter erweiterbare Programme zu konstruieren.

Die beschriebenen Beispiele von Entwurfsmängeln haben gezeigt, dass diese bereits bekannten Konzepte objektorientierter Programmierung aufgreifen und aus jeweils leicht unterschiedlichen Perspektiven beleuchten. Das ist auch den Autoren der Entwurfsmängel bewusst. Ihr Ziel ist es, dem Programmierer oder Architekten ein griffiges Werkzeug an die Hand zu geben, das ihnen bei der täglichen Arbeit hilft in großen Software-Systemen — sozusagen im Vorbeigehen — Problemstellen zu erkennen. Genau das leisten diese informell beschriebenen Entwurfsmängel.

Vordergründig beleuchten sie nur Hinweise und Symptome statt prüfbarer Programmeigenschaften. Ausgestattet mit griffigen Analogien können diese aber leicht erlernt, angewandt und kommuniziert werden. Somit ist die didaktische Vermittlung dieses Wissens die eigentliche Leistung der Beschreibungen von Entwurfsmängeln.

In Kapitel 2 wurde deutlich, dass bei der Konstruktion und Wartung komplexer Software-Systeme zum Einen die Abstraktion nach dem *Teile und Herrsche*-Prinzip und zum Anderen die Kommunikation zwischen den Projektbeteiligten eine wesentliche Rolle spielen.

Die Erkennung von Entwurfsmängeln durch manuelle Inspektion ist dennoch sehr zeitaufwendig, sodass mein Ziel die automatische Erkennung ist. Fowler wies bereits darauf hin, dass die menschliche Intuition benötigt würde. Ich verwende daher maschinelle Lernverfahren um dem Faktor Mensch näher zu kommen.

Im folgenden Kapitel werden die hierzu benötigten Grundlagen des maschinellen Lernens skizziert.

4. Maschinelles Lernen

Inhalt

4.1. Überblick **42**
4.2. Grundlagen konzeptionellen Lernens **42**
4.3. Entscheidungsbaumverfahren **46**
4.3.1. Problemklassen 47
4.3.2. Grundlegender Lernalgorithmus 47
4.3.3. Eigenschaften und Erweiterungen 49
4.4. Andere Lernverfahren **50**
4.5. Zusammenfassung **51**

4.1. Überblick

Im Mittelpunkt des maschinellen Lernens stehen Methoden, die es Programmen erlauben zu „lernen“. Ein lernendes Programm erfüllt eine bestimmte Aufgabe und liefert dabei durch Akkumulation von Erfahrungen eine verbesserte Leistung [59] [94] [42].

Maschinelle Lernverfahren werden erfolgreich zur Spracherkennung, zum Lenken von Fahrzeugen, zum Brettspielen und anderen Lernzielen eingesetzt. Es tragen u. a. die Wissensgebiete Statistik, Psychologie, Philosophie und künstliche Intelligenz zum maschinellen Lernen bei.

Zu den eingesetzten Techniken gehören u. a. konzeptionelles Lernen, Entscheidungsbaumkonstruktion, Neuronale Netze, Bayesches Lernen, Instanzbasiertes Lernen, Genetische Algorithmen und Verstärkungslernen.

Beim Entwurf eines Lernverfahrens wird zunächst festgelegt, welche Aufgabe das Verfahren erfüllen soll. Danach müssen die Erfahrungen, aus denen das Programm lernen soll, und die Ausgabe, die das Programm liefern soll, modelliert werden. Zuletzt wird ein Leistungsmaß festgelegt. Nur so lässt sich feststellen, ob das Verfahren durch mehr Erfahrung bessere Leistung erbringt.

Dieses Kapitel erläutert zunächst im Abschnitt 4.2 grundlegende Begriffe des konzeptionellen Lernens und geht dann in Abschnitt 4.3 detaillierter auf Verfahren Entscheidungsbäume zu erlernen und deren Eigenschaften ein. Abschließend werden in Abschnitt 4.4 weitere Lernverfahren genannt. Der Abschnitt 4.5 fasst die wesentlichen Ideen zusammen.

Ich setze maschinelle Lernverfahren ein, um die Adaption von Entwurfsmangelerkennung an Benutzer und Anwendungsgebiete zu erlauben. Dabei spiele ich die Rolle eines Anwenders solcher Lernverfahren. Das Ziel ist es also, ein möglichst geeignetes Verfahren anzuwenden und nicht spezielle Verfahren für diese Aufgabe zu entwerfen. Die Begriffsbildung orientiert sich hier an der Darstellung von Mitchell [59].

4.2. Grundlagen konzeptionellen Lernens

Konzeptionelles Lernen beschreibt Verfahren, um allgemeine Konzepte anhand von Beispielen abzuleiten (Inferenz). Das Konzept wird dabei durch eine boolsche Funktion repräsentiert. Beispiele werden durch die Ein- und Ausgabewerte der Funktion charakterisiert.

Im Alltag erlernen wir Konzepte wie z. B. einen Tisch anhand bestimmter Merkmale, die sich wiederkehrend als Muster einprägen. Sie erlauben uns Tische auch dann als solche zu erkennen, wenn wir niemals zuvor ein solches Modell gesehen hätten. Das könnten zum Beispiel Eigenschaften wie eine ebene Fläche, die

sich waagerecht in einer bestimmten Höhe befindet, sein. Das Material oder die Form der Befestigung — Tischbeine, Sockel, Aufhängung oder Wandbefestigung — würden hier keine Rolle spielen.

Verfolgt man das Beispiel weiter, so könnte man für das Konzept Tisch die Attribute *Höhe*, *Breite*, *Tiefe*, *Dicke* vom Typ *Länge* in der Einheit Meter festlegen. Zusätzlich könnte das Attribut *Material* den Wertebereich aus *Holz*, *Metall* und *unbekannt* beschreiben. Im Folgenden werden zunächst nur diskrete Wertebereiche, wie hier durch das Attribut *Material* beschrieben, verwendet.

Notation

Ein Trainingsbeispiel wird *Instanz* genannt und durch jeweils eine Folge von Attributen gebildet. Dabei wird an der jeweiligen Stelle der konkrete Wert entsprechend des vereinbarten Typs eingetragen.

Im Folgenden wird die Funktion, die das Konzept beschreibt, als $c : X \to \{0, 1\}$ bezeichnet. Dabei ist X eine Menge von Instanzen. Eine Instanz $(x, c(x))$ mit $x \in X$ ist Teil des Konzeptes — oder auch *positives Beispiel*, wenn $c(x) = 1$. Andernfalls gehört die Instanz nicht zum Konzept, bzw. sie ist ein *negatives Beispiel*.

Induktive Lernhypothese

Das einfachste Lernverfahren ist das Auswendiglernen. Damit können spielend alle „erlernten" Instanzen dem Konzept zugeordnet werden. Das Ziel ist es jedoch, das Konzept auch in unbekannten Instanzen zu erkennen. Es wird daher eine allgemeinere Konzeptbeschreibung gesucht.

Die Funktion $h : X \to \{0, 1\}$ beschreibt eine Hypothese. Das Lernverfahren bildet anhand einer Menge von Instanzen ein *allgemeineres* hypothetisches Konzept, das alle bekannten Instanzen enthält.

Man unterstellt, dass die Hypothese, welche am besten zu den bekannten Instanzen passt, auch geeignet ist, unbekannte Instanzen zu beschreiben. Voraussetzung ist allerdings, dass genügend Trainingsbeispiele vorliegen.

Einfache Hypothesensuche

Zur Bildung von Hypothesen werden zwei zusätzliche Werte verwendet. Dabei bedeute '?', dass jeder beliebige Wert angenommen werden kann und '$\emptyset$' bedeute, dass kein Wert erlaubt ist.

Damit steht in dem bisherigen Beispiel die Funktion $(?, ?, ?, ?, ?)$ für die allgemeinste Hypothese: jede Instanz ist enthalten. Die Funktion $(\emptyset, \emptyset, \emptyset, \emptyset, \emptyset)$ steht für die speziellste Hypothese und enthält keine Instanz. Eine Hypothese enthält keine Instanz, wenn mindestens an einer Stelle der Wert $\emptyset$ steht.

Auf der Suche nach der allgemeinsten Hypothese bezüglich einer gegebenen Trainingsmenge (*Find-S*-Algorithmus) beginnt man bei der speziellsten Hypothese und verallgemeinert diese, indem man schrittweise alle positiven Instanzen abarbeitet. Im ersten Schritt bildet die erste Instanz eine Hypothese. Im Vergleich der

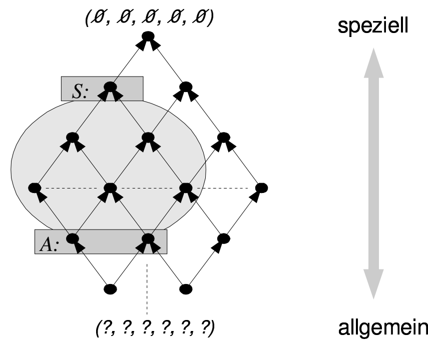

Abbildung 4.1.: Hypothesenraum des konzeptionellen Lernens

aktuellen Hypothese mit der nächsten Instanz werden jeweils die Stellen der Hypothese auf den Wert ? gesetzt, die nicht mit dem entsprechenden Wert der Instanz übereinstimmen. Man erhält schließlich eine Hypothese, die genau alle positiven Instanzen enthält.

Hypothesen ordnen

Man definiert eine Ordnung auf der Menge der Hypothesen und sagt, dass eine Hypothese h_1 allgemeiner oder gleich einer anderen Hypothese h_2 ist, wenn alle Instanzen, die h_1 enthält auch in h_2 enthalten sind. Man schreibt dann $h_1 \geq_g h_2$. Analog gibt es eine strikte Relation, bei der $h_1 >_g h_2$, genau dann wenn $(h_1 \geq_g h_2) \wedge (h_1 \ngeq_g h_2)$ ist. Die Relation bildet eine partielle Ordnung auf dem Hypothesenraum.

Abbildung 4.1 zeigt schematisch einen Hypothesenraum.

Lösungsraum

Das vorherige einfache Suchverfahren findet zwar eine Hypothese, die alle Trainingsinstanzen enthält, aber es ist nur eine von vielen gültigen Hypothesen. Außerdem ist nicht klar, warum gerade diese die am besten geeignete für unbekannte Instanzen sein sollte. Ebenso gut könnte die allgemeinste, die noch alle negativen Instanzen abweist oder eine Hypothese zwischen diesen Extremen besser geeignet sein.

Man sucht daher nach allen gültigen Hypothesen (*Candiadate-Elimination*-Algorithmus) ohne dabei alle Hypothesen aufzählen zu müssen. Die partielle Ordnung der Hypothesen erlaubt es, nur die Menge der allgemeinsten Hypothesen A und der speziellsten Hypothesen S zu beschreiben. Die Hypothesen, die entsprechend der Relation dazwischen liegen, gehören ebenfalls zum Lösungsraum (*Version Space*).

Abbildung 4.1 zeigt einen Beispielraum zwischen den Mengen S und A.

Algorithmisch nähert man sich dem Lösungsraum, indem die Grenzhypothe-

senmengen entsprechend der Trainingsinstanzen schrittweise spezialisiert bzw. verallgemeinert werden.

Unbekannte Instanzen

Der Lösungsraum, der durch die beiden Grenzmengen beschrieben ist, lässt sich verwenden um unbekannte Instanzen zu klassifizieren. Genügt eine Instanz allen Hypothesen der speziellen Grenzmenge S, so gehört die fragliche Instanz sicher zum erlernten Konzept. Genügt eine Instanz keiner der allgemeinsten Hypothesen A, so gehört sie sicher nicht zum erlernten Konzept.

Wenn jedoch beide eindeutigen Fälle nicht vorliegen, d. h. es gibt Hypothesen im Lösungsraum, für die die Instanz sowohl enthalten als auch nicht enthalten ist, dann wird durch Mehrheitsentscheid aller Hypothesen entschieden. Die Güte der Entscheidung kann am Verhältnis der positiv zu negativ entscheidenden Hypothesen abgelesen werden.

Robustheit

Konzeptionelles Lernen (in der beschriebenen Form) funktioniert nur dann, wenn die Trainingsmenge keine Fehler enthält und der initiale Hypothesenraum das gesuchte Konzept enthält.

Wenn eine Instanz fälschlich als nicht zum Konzept gehörend ausgewiesen ist, dann wird die entsprechende Hypothese entfernt. Der Algorithmus könnte solche Fehler erkennen, wenn der Lösungsungsraum schließlich keine Hypothese mehr enthält oder widersprüchliche Instanzen vorliegen.

Schwieriger ist die Frage nach der Eignung einer Konzeptrepräsentation. In diesem Beispiel kann das Konzept nur in Form einer Konjunktion mehrerer Attribute beschrieben werden. Als einzige Freiheit können einzelne Attributwerte beliebig sein. Man kann die gewählte Konzeptrepräsentation als formale Sprache auffassen, die geeignet sein muss, das Konzept zu beschreiben.

Überanpassung und Vorlieben

Man spricht daher im Bereich des konzeptionellen Lernens von einer Vorliebe *(Bias)*, die ein Lernverfahren aufgrund der gewählten Konzeptrepräsentation hat.

Wollte man tatsächlich in der Lage sein jedes Konzept zu beschreiben, so würden hier z. B. zusätzlich Disjunktion und Negation benötigt um alle Konzepte (Kardinalität der Potenzmenge der Instanzmenge) beschreiben zu können.

Dies führt leider dazu, dass das Lernverfahren die Trainingsmenge auswendig lernt, statt ein Konzept zu abstrahieren. In diesem Fall würde die speziellste Hypothesenmenge S zur Disjunktion aller positiven Instanzen und die allgemeinste Hypothesenmenge A zur Negation der Konjunktion aller negativen Instanzen entarten.

Damit kann ohne Annahmen über das zu erlernende Konzept kein Lernverfahren entwickelt werden, das unbekannte Instanzen klassifizieren könnte.

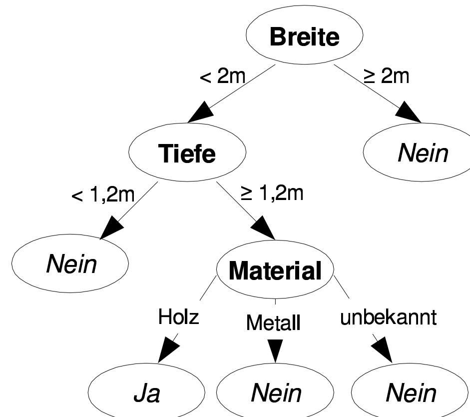

Abbildung 4.2.: Beispiel für einen Entscheidungsbaum des Konzeptes „Tisch“.

Die Wahl der Konzeptrepräsentation bzw. -sprache legt damit das Abstraktionsvermögen des Lernverfahrens über die Trainingsmenge hinaus fest. Der Hypothesenraum bzw. die Hypothesensprache ist daher stets so klein zu wählen, dass er sinnvoll abstrahiert und damit unbekannte Instanzen klassifizieren kann und groß genug um das gesuchte (aber unbekannte) Zielkonzept zu enthalten.

4.3. Entscheidungsbaumverfahren

Entscheidungsbaumverfahren gehören zu den am häufigsten verwendeten induktiven Lernverfahren. Zur Beschreibung der Hypothesen werden Konjunktion und Disjunktion von Attributen verwendet. Diese werden als Entscheidungsbaum oder alternativ als Folge von Regeln mit Fallunterscheidungen dargestellt.

Unbekannte Instanzen werden durch Interpretation des Entscheidungsbaums klassifiziert. Dabei modellieren innere Knoten des Entscheidungsbaums die Entscheidungspunkte eines einzelnen Attributes. Es wird dann ausgehend von der Wurzel an den jeweiligen Unterknoten verzweigt, für den der aktuelle Attributwert übereinstimmt. Die Blattknoten stellen dann schließlich das Ergebnis dar.

Abbildung 4.2 zeigt ein Konzept, an dem man einen Tisch erkennen könnte.

Ein Entscheidungsbaum wird rekursiv von der Wurzel zu den Blättern konstruiert. Dabei wird anhand eines speziellen Entropiemaßes der Informationsgewinn eines einzelnen Attributes beurteilt. Das Attribut mit dem höchsten Informationsgewinn bildet die Wurzel des nächsten rekursiv erzeugten Teilbaums.

Auch Entscheidungsbaumverfahren haben Vorlieben. Dem Problem der Überanpassung an die vorliegende Trainingsmenge wird begegnet, indem zu große Bäume abgeschnitten werden. Es werden daher kleinere gegenüber größeren Entscheidungsbäumen bevorzugt.

4.3.1. Problemklassen

Man setzt Entscheidungsbaumverfahren bei Lernproblemen ein, die wie folgt charakterisiert werden:

Attribute Instanzen werden durch Attributmengen beschrieben. Attribute sind jeweils Name-Wert-Paare. Dabei können Werte sowohl symbolisch (diskreter Wertebereich) als auch numerisch sein.

Zielfunktion Das Ergebnis der Zielfunktion liefert eine boolsche Klassifikation (wahr oder falsch). Es kann aber auch einfach zu mehrwertigen Klassifikationen erweitert werden.

Konzeptrepräsentation Entscheidungsbäume beschreiben Disjunktionen, die mit Konjunktionen durchmischt sein können.

Robustheit Es dürfen Fehler in der Trainingsmenge, aber auch in der Klassifikation unbekannter Instanzen auftreten. Zusätzlich dürfen Attributwerte fehlen.

4.3.2. Grundlegender Lernalgorithmus

Viele Verfahren zum Erlernen von Entscheidungsbäumen gehen auf das *ID3*-Verfahren von Quinlan [72] zurück. Dieses wurde später zum *C4.5*-Verfahren weiterentwickelt [73].

Rekursiver Algorithmus

Die grundlegende Idee beruht auf dem *Teile und Herrsche*-Prinzip. Dabei wird der Entscheidungsbaum von der Wurzel zu den Blättern rekursiv aufgebaut.

Als Eingabe für den Algorithmus dient die Trainingsmenge S und die Menge der Attributdefinitionen. Es wird nun ein Attribut für die Wurzel des Entscheidungsbaums ausgewählt. Man bestimmt das Attribut, das den größten Einfluss auf die Klassifikation der Trainingsinstanzen hat. Dies wird anhand eines Informationsmaßes bestimmt, sodass das gewählte Attribut für sich alleine die Trainingsbeispiele am besten klassifiziert. Abbildung 4.3 zeigt die Berechnung.

Für jeden diskreten Wert des Attributes wird jeweils ein Unterknoten gebildet, dem alle Trainingsinstanzen entsprechend des gerade betrachteten Attributwertes zugeordnet werden (siehe Abbildung 4.4).

Rekursiv wird nun in jedem Unterknoten mit den zugeordneten Trainingsinstanzen als Eingabe weiter verfahren. Die Rekursion endet, wenn in einem Knoten alle Instanzen gleich klassifiziert werden, also keine weitere Unterscheidung nötig ist. Der Wert der Zielfunktion bildet den jeweiligen Blattknoten.

Das Verfahren nimmt eine getroffene Entscheidung niemals zurück. Es hängt also entscheidend von der Wahl des Informationsmaßes ab, wie der Entscheidungsbaum konstruiert wird.

Die *Entropie* einer Trainingsmenge S beschreibt den Grad der Uninformiertheit bezüglich des Zielattributes. Man definiert $p_\oplus$ als Anzahl der positiven Instanzen und $p_\ominus$ als Anzahl der negativen Instanzen in S.

$$\mathsf{Entropy}(S) = -p_\oplus\, log_2\, p_\oplus - p_\ominus\, log_2\, p_\ominus$$

Man schreibt auch $S \equiv [p_\oplus, p_\ominus]$; z. B. $S \equiv [4, 5]$ für den Fall, dass S aus 4 positiven und 5 negativen Instanzen bestehe. Desweiteren ist Values(S) die Menge der Zielattributwerte von S und $S_v \subseteq S$ mit $v \in$ Values(S) ist die Menge der Instanzen deren Zielattribut gleich v ist.
Der *Nutzen* eines Attributes A bezüglich einer Trainingsmenge S berechnet sich dann wie folgt:

$$\mathsf{Gain}(S, A) = \mathsf{Entropy}(S) - \sum_{v \in \mathsf{Values}(A)} \frac{|S_v|}{|S|}\, \mathsf{Entropy}(S_v)$$

Ein anschauliches Beispiel für die Berechnung ist in [59, S. 59ff.] zu finden.

Abbildung 4.3.: Berechnung von Entropie und Nutzen.

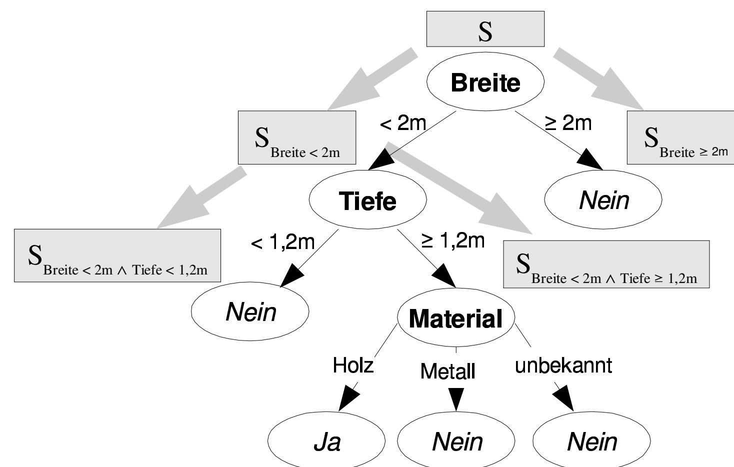

Abbildung 4.4.: Propagation der Trainingsmenge bei der Konstruktion des Entscheidungsbaums.

Informationsmaß

Das hier verwendete Informationsmaß *(information gain)* beruht auf einem allgemeinen Entropiemaß, das die Homogenität einer Trainingsmenge bezüglich ihrer Zielattribute misst. Dabei erreicht der Entropiewert 0, wenn nur negative Instanzen oder nur positive Instanzen vorliegen. Der Entropiewert ist maximal 1 bei gleich vielen positiven wie negativen Instanzen. Das Entropiemaß lässt sich leicht auf mehrwertige (statt boolsche) Zielattribute erweitern.

Das Informationsmaß beschreibt für jedes Attribut den Verlust am Wert der Entropie, wenn die Trainingsmenge nach diesem Attribut zerteilt würde. Gewählt wird dann das Attribut mit dem höchsten Gewinn bzw. niedrigstem Verlust.

4.3.3. Eigenschaften und Erweiterungen

Auch dieses Lernverfahren durchsucht einen Hypothesenraum; den Raum aller Entscheidungsbäume. Per Konstruktion wird dabei ein Entscheidungsbaum gefunden, der zur Trainingsmenge passt. Es besteht zwar nicht die Gefahr, dass die Konzeptrepräsentation nicht ausreichen könnte das Konzept zu beschreiben, aber die Gefahr der Überanpassung ist groß. Es gelingt vielleicht einen lokal optimalen Entscheidungsbaum zu finden, der aber nicht global optimal (vor allem bezüglich unbekannter Instanzen) sein muss.

Robustheit

Im Vergleich zum *Candidate-Elimination*-Verfahren wird hier statt eines Lösungsraumes nur eine Lösung gefunden. Da nicht der Anspruch besteht den vollständigen Lösungsraum zu erfassen, bietet sich die Chance auch auf falsche oder fehlende Attributwerte einzugehen. Hierzu schätzt man fehlende Werte. Man lässt also Fehler (in geringem Umfang) zu.

Vorliebe

Während sich bei Verfahren des konzeptionellen Lernens die Vorliebe *(Bias)* an der Ausdrucksstärke der Konzeptrepräsentation — also dem Suchraum — orientierte, steht hier die Suchstrategie im Vordergrund.

Zu jeder Trainingsmenge existieren mehrere passende Entscheidungsbäume. *ID3* wird charakterisiert als Verfahren, das kleine Entscheidungsbäume bevorzugt. Dies präzisiert man ehrlicherweise soweit, dass Entscheidungsbäume bevorzugt werden, die Attribute mit hohem Informationsgewinn in Wurzelnähe platzieren.

Man nennt die Vorliebe hier auch Suchvorliebe *(search bias)* im Vergleich zur Sprachvorliebe *(language bias)*. Im Allgemeinen wird die erste Form bevorzugt, weil sie sicherstellt, dass die (bzw. eine) gesuchte Hypothese im Suchraum vorhanden ist. Es sei daran erinnert, dass eine Vorliebe nötig ist um überhaupt abstrahieren und somit lernen zu können.

Überanpassung
Es gibt Fälle (z. B. bei einer kleinen Trainingsmenge oder bei fehlerhaften Attributwerten), bei denen der Entscheidungsbaum zu genau der Trainingsmenge angepasst wird. Man spricht von Überanpassung, wenn die Trainingsmenge zwar gut erkannt wird, aber unbekannte Instanzen schlecht erkannt werden. D. h. das Verfahren hat der Vorliebe zu stark nachgegeben.

Mit steigender Knotenanzahl steigt auch die Genauigkeit der Klassifizierung bezüglich der Trainingsmenge. Demgegenüber sinkt die Genauigkeit bei unbekannten Testinstanzen. Man versucht daher den Entscheidungsbaum zu kürzen, indem man ihn nicht bis zur vollständigen Trainingsüberdeckung wachsen lässt. Alternativ lässt man ihn heranwachsen und kürzt ihn später ein *(post-pruning)*.

Die Kürzung hat zur Konsequenz, dass nicht immer alle Trainingsinstanzen korrekt klassifiziert werden. Dies nimmt man zugunsten einer besseren Leistung bezüglich unbekannter Instanzen hin. Geeignete Verfahren zur Kürzung von Entscheidungsbäumen werden z. B. in [59, S. 66ff.] beschrieben.

Numerische Attribute
Bisher wurde der Wertebereich eines Attributes als eine Menge diskreter (symbolischer) Werte beschrieben. Die Kardinalität des Wertebereichs eines Attributes war gleich der Anzahl der Unterknoten eines entsprechenden Knotens.

Bei numerischen Werten, können dynamisch diskrete Wertebereiche gebildet werden, indem man geeignete Grenzwerte berechnet. Man sortiert alle Attributwerte der vorliegenden Instanzen und ordnet ihnen den Wert des Zielattributes zu. Diese Folge enthält Wechsel des Zielattributes von *falsch* nach *wahr* und umgekehrt. An diesen Stellen wird der Grenzwert als Mittelwert der beiden zugeordneten Attributwerte berechnet. Damit ergeben sich dann neue Attributwerte der Form $Attribut_{\leq x}$ und $Attribut_{>x}$. Gewählt wird das Attribut mit dem höchsten Informationsgewinn.

Erweiterungen des Verfahrens erlauben auch mehrere Wertebereiche statt nur zwei Wertebereiche, die sich auf einen einzigen Grenzwert beziehen, zu verwenden.

Das ursprüngliche *ID3*-Verfahren wurde beständig weiter entwickelt und schließlich in *C4.5* umbenannt [73]. Dieses enthält die beschriebenen Mechanismen zur Vermeidung von Überanpassung und verarbeitet sowohl diskrete als auch numerische Attribute und mehrwertige Ergebnisse der Zielfunktion.

4.4. Andere Lernverfahren

Das Fachgebiet des maschinellen Lernens bietet Raum für ein breites Spektrum unterschiedlicher Lernverfahren. Viele dieser Verfahren sind von der Natur inspiriert. Eine kleine Auswahl bilden die folgenden Verfahren.

Neuronale Netze gehören zu den effektivsten und robustesten Lernverfahren. Sie lernen aus Beispielen und bilden ein Netzwerk, das durch *Backtracking*-Verfahren an die Trainingsmenge angepasst wird. Inspiriert werden sie vom neuronalen Netz des menschlichen Gehirns.

Bayesches Lernen ist ein probabilistischer Lernansatz. Er stützt sich auf das *Bayes*-Theorem zur Berechnung bedingter Wahrscheinlichkeiten. Vergleiche haben den Ansatz mindestens so leistungsstark wie einfache Entscheidungsbaumverfahren oder Neuronale Netze eingestuft.

Genetische Algorithmen lernen, indem sie Evolution simulieren. Als Bevölkerung werden häufig Bit-Folgen eingesetzt, die Hypothesen repräsentieren. Diese entwickeln sich weiter durch Mutation oder Vermischung.

Instanzbasiertes Lernen konstruiert nicht — wie andere Lernverfahren — eine Abstraktion von Trainingsinstanzen. Stattdessen werden alle Trainingsbeispiele notiert und erst zu dem Zeitpunkt, zu dem eine neue Instanz klassifiziert werden soll, wird eine Abstraktion gebildet. Hierzu werden i. d. R. mathematische Regressionsverfahren eingesetzt.

Analytisches Lernen verwendet statt einer großen Trainingsmenge entsprechendes Vorwissen um Konzepte zu erlernen. Das Vorwissen wird genutzt um Beispielinstanzen zu erklären. Dabei werden wichtige von unwichtigen Kriterien getrennt. Diese deduktive Form des Lernens wird auch erfolgreich mit induktiven Verfahren, wie z. B. den Neuronalen Netzen, verknüpft.

4.5. Zusammenfassung

Maschinelle Lernverfahren erlauben, anhand von Beispielen Konzepte zu erlernen. Das schließt die Fähigkeit zu abstrahieren ein. Es stehen vielfältige Verfahren zur Verfügung. Ich habe hier das einfache Verfahren der Entscheidungsbaumkonstruktion ausgewählt.

Entscheidungsbäume bilden eine sehr attraktive Wissensrepräsentation. Sie sind vom menschlichen Betrachter leicht zu überblicken und nachzuvollziehen. Damit bieten sie sowohl die Chance das erlernte Konzept zu studieren als auch im Einzelinstanzen bei ihrer Klassifikation zu beobachten.

Das geplante Einsatzszenario passt hervorragend zum Einsatzspektrum der Entscheidungsbaumkonstruktion. Alternativ könnten *Bayes*-Verfahren eingesetzt werden. Da diese aber „*Black-Box*-Verfahren" sind, verlöre man den Einblick in das erlernte Konzept.

Der Einsatz von Lernverfahren in dieser Arbeit ist daher nicht primär auf die bestmögliche Lernleistung ausgerichtet, sondern auf eine transparente Form der Wissensgenerierung und -repräsentation.

Auf die Bewertung der Lernleistung wurde bisher noch nicht eingegangen. Dieses wird in Kapitel 8 im Detail beschrieben.

5. Modellierung von Entwurfsmängeln

Inhalt

5.1. Überblick **54**
5.2. Entwurf von Modellen zu Entwurfsmängeln **54**
5.2.1. Messtheorie 55
5.2.2. Metriken in der Software-Technik 56
5.2.3. Klassifizierung objektorientierter Entwurfsmetriken 56
5.2.4. Entwurf geeigneter Metriken 58
5.3. Beispielmodelle **59**
5.3.1. Große Klasse 59
5.3.2. Lange Methode 64
5.3.3. Faule Klasse / Datenklasse 65
5.3.4. Neid 65
5.4. Zusammenfassung **66**

5.1. Überblick

In Kapitel 3 wurden einige Beispiele für Entwurfsmängel informell diskutiert. Um diese automatisch erkennen zu können, müssen konkrete Programmeigenschaften identifiziert werden, die geeignet sind, den jeweiligen Entwurfsmangel zu charakterisieren. Hierzu werden Metriken eingesetzt.

Der allgemeine Qualitätsbegriff von Software-Systemen, der im Industriestandard ISO 9126 [80][81] beschrieben ist, zeigt das Prinzip der schrittweisen Verfeinerung von Qualitätszielen.

Der ISO-Standard enthält neben funktionalen Anforderungen eine Reihe von nicht funktionalen Anforderungen. Man unterscheidet *Funktionalität*, *Zuverlässigkeit*, *Benutzbarkeit*, *Effizienz*, *Wartbarkeit* und *Portierbarkeit*. Die letzten beiden Kriterien beschreiben die innere Qualität, die anderen Kriterien beschreiben die äußere Qualität, die auch der Benutzer erfährt. Dies sind die Qualitätsziele.

Der ISO-Standard operationalisiert diese allgemeinen Qualitätskriterien. Es wird z. B. die Wartbarkeit als *stabil*, *analysierbar*, *änderbar* und *testbar* konkretisiert. Allerdings lassen sich auch hieraus keine konkreten Programmeigenschaften zur Messung der Software-Qualität ableiten. Quantifiziert man einzelne Kriterien durch Mengen von Metriken, so ergibt sich daraus ein Ansatz um Qualitätskriterien zu messen [48] [30].

Entwurfsmängel können nach dem gleichen Muster erkannt werden. Einzelne Eigenschaften eines „guten" objektorientierten Entwurfs werden auf Metriken abgebildet und ein Schema zur Interpretation der auftretenden Messwerte wird ergänzt. Damit ist das Vorgehen hier zur Modellierung von Entwurfsmängeln in [56] [57] analog.

In diesem Kapitel wird im folgenden Abschnitt 5.2 zunächst ein Vorgehensmodell entwickelt, mit dem informelle Beschreibungen von Entwurfsmängeln untersucht und anhand bekannter Eigenschaften des objektorientierten Entwurfs und objektorientierter Entwurfsmetriken operationalisiert werden können. Im Abschnitt 5.3 werden dann die Beispiele von Entwurfsmängeln aus Kapitel 3 aufgegriffen und jeweils ein Modell entwickelt.

5.2. Entwurf von Modellen zu Entwurfsmängeln

Der folgende Abschnitt 5.2.1 fasst zunächst messtheoretische Grundzüge zusammen und Abschnitt 5.2.2 erläutert Einsatzzwecke von Metriken in der Software-Technik. Im Abschnitt 5.2.3 werden dann die wichtigsten Arten der speziellen objektorientierten Entwurfsmetriken erklärt, bevor in Abschnitt 5.2.4 dann das Vorgehen zum Entwurf objektorientierter Metriken speziell zur Beschreibung von Entwurfsmängeln entwickelt wird.

5.2.1. Messtheorie

Eine ausführliche Beschreibung der mathematischen Grundlagen zur Messtheorie kann z. B. in [35] oder in [92] nachgelesen werden. Hier wird nur ein kurzer Überblick gegeben.

Messung bedeutet Gegenständen Attribute zuzuordnen, um sie zu beschreiben. Dabei sind Attribute i. d. R. Name-Wert-Paare, deren Werte nach festgelegten Regeln bestimmt werden [35].

Attribute und deren Messwerte charakterisieren also Gegenstände in einer vorgegeben Weise, sodass sie auch ohne Kenntnis des Gegenstandes — nur anhand der gemessenen Attribute — einen Eindruck von den Eigenschaften des Gegenstandes vermitteln. Somit abstrahieren Messungen vom Gegenstand; oder besser: Attribute modellieren den Gegenstand.

Eine *Metrik* ist ein einzelnes Attribut, zu dem Regeln beschrieben sind, wie der entsprechende Wert anhand des Gegenstandes zu berechnen ist.

Die *Skalen* legen den Typ eines Attributes fest. Man unterscheidet i. d. R. fünf verschiedene Skalen:

- *nominal:* beschreibt einen diskreten Wertebereich, der durch einen Aufzählungstyp beschrieben wird. Es besteht keine Ordnung auf dem Wertebereich. Ein Beispiel ist der Wertebereich $\{\mathsf{rot}, \mathsf{grün}, \mathsf{blau}\}$.

- *ordinal:* beschreibt einen diskreten oder kontinuierlichen Wertebereich, auf dem zusätzlich eine Ordnung definiert ist. Ein Beispiel ist der Wertebereich $\{\mathsf{klein}, \mathsf{mittel}, \mathsf{groß}\}$ mit der Ordnung $\mathsf{klein} \leq \mathsf{mittel} \leq \mathsf{groß}$ oder die Dezibel-Skala.

- *Intervall:* beschreibt eine Ordinal-Skala, bei dem auch Differenzen zwischen Werten vergleichbar sind. Das Beispiel der ordinalen Dezibel-Skala erfüllt dies nicht, da sie logarithmisch und nicht linear steigt.

- *Rational:* beschreibt eine lineare Ordnung, auf der zusätzlich Durchschnitt und Addition definiert sind. Ein Beispiel ist die Temperatur-Skala in Celsius.

- *Absolut:* beschreibt die Anzahl von Objekten, Kardinalitäten von Mengen und Wahrscheinlichkeiten.

Eine Messung muss in dem Sinne zuverlässig sein, dass eine wiederholte Messung unter gleichen Bedingungen auch das gleiche Ergebnis liefert.

Zuletzt betrachtet man die *Validität*. Es ist zu prüfen, ob ein Messverfahren tatsächlich das gewünschte Merkmal misst. Misst z. B. ein Intelligenztest tatsächlich die Intelligenz oder vielleicht doch nur die Belastbarkeit, spezielles Fachwissen oder abstraktes Denkvermögen der Testperson?

Man spricht von einem *kausalen* Zusammenhang, wenn die Messung tatsächlich mit dem Merkmal ursächlich zusammenhängt. Gibt es nur einen regelmäßigen zufälligen Zusammenhang, spricht man von *Korrelation*. Die *Koinzidenz* beschreibt dann einen unregelmäßigen zufälligen Zusammenhang.

Die Validierung einer Metrik ist die schwierigste Aufgabe und erfordert umfangreiche empirische Untersuchungen.

5.2.2. Metriken in der Software-Technik

Eine Vielzahl von Metriken ist in der Literatur zur Software-Technik zu finden. Zuse hat allein über eintausend Metriken gesammelt und fasst Einsatzgebiete der Metriken in [2] zusammen.

So werden Metriken z. B. für die Planung und Kontrolle von Software-Projekten eingesetzt. Hier versucht man Fehlerraten vorherzusagen [60] [77] oder Kosten-, Zeit- und Aufwandsschätzungen zu verbessern [71] [23] [27] [1].

Viele Standardwerke und bekannte Metrik-Sammlungen konzentrieren sich ebenfalls auf diese Zwecke und beziehen sich i. d. R. auf strukturierte Programme. Hierzu gehören auch das Standardwerk von Fenton und Pfleeger [35] [34]. Die *Halstead*-Metriken vermessen den Aufwand der Software-Konstruktion. Die *McCabe*-Metrik [58] befasst sich mit der Komplexität von Funktionen.

Für objektorientierte Software gibt es bekannte Metrik-Sammlungen von Henderson-Sellers [43], Chidamber und Kemerer [16] [44] und Zuse [97]. Rückschlüsse auf den objektorientierten Entwurf werden in [7] und [31] gezogen.

5.2.3. Klassifizierung objektorientierter Entwurfsmetriken

Ein spezielles Gebiet der Metriken wird durch die objektorientierten Entwurfsmetriken gebildet. Diese Art der Metriken sind bestens geeignet um Entwurfsmängel zu charakterisieren. Eine pragmatische Sicht wird in [54] und [16] beschrieben. Eine formale und mathematisch messtheoretische Sicht ist in [92] zu finden. Hieraus stammt auch die folgende Klassifizierung dieser Metriken:

Größe Die am häufigsten im Alltag anzutreffenden Maße sind Größenmaße. Sie beschreiben die Kardinalität einer Menge oder die Länge eines Gegenstandes oder einer Struktur.

Kardinalitäten treten häufig im Zusammenhang mit Randbedingungen auf. So wird die Anzahl von Objekten, die von einer bestimmten Art sind oder die einer bestimmten Regel genügen, gemessen. Auch zeitliche Bedingungen, wie die größte oder kleinste Kardinalität innerhalb eines Zeitraumes, werden verwendet.

Ist die Größe eine Länge, so wird eine Einheit benötigt. Randbedingungen definieren zum Einen die genauen Start- und Zielorte einer Messung oder zum Anderen auch besondere Eigenschaften des zu messenden Weges. Bei Strukturen wie Pfaden, Listen oder Graphen kann dies die Länge von kürzesten oder längsten Wegen sein. In Bäumen spricht man von der Tiefe oder Breite.

Größenmaße haben einige wichtige Eigenschaften. Ein Größenmaß ist nicht negativ; vereinigt man disjunkte Mengen, so addieren sich die Kardinalitäten; die Kardinalität sich vereinigender Mengen ist monoton wachsend und nach oben beschränkt. Weitere Eigenschaften werden in [11][12] beschrieben.

Komplexität Der Begriff der Komplexität, insbesondere die der Komplexität von Software, ist schwer zu fassen. Man misst die Komplexität eines Gegenstandes anhand seiner umfangreichen Struktur oder anhand des Aufwandes ihn zu verstehen. Beide beschreiben ähnliches, jedoch aus gegensätzlicher Perspektive.

Die Ursprünge zur Messung von Software-Komplexität findet man bei Weyuker [91] zusammengefasst. Grundlegende Ansätze basieren auf abstrakten Strukturbäumen [87] oder Kontrollflussmodellen [58] [51].

Nüchtern betrachtet sind Komplexitätsmaße auch Größenmaße, sie messen z. B. die Anzahl der Knoten und Kanten eines Kontrollflussgraphen. Somit also eine Menge von Knoten, die durch bestimmte Randbedingungen — hier einen Graphen — festgelegt werden.

Kopplung Der Begriff der Kopplung beschreibt das Ausmaß und die Form der Abhängigkeit zwischen Teilen eines Systems.

In der strukturierten Programmierung unterscheidet man im Wesentlichen zwischen Daten- und Ablaufkopplung [64].

Besonders in der Objektorientierung spricht man von der Kopplung durch Verwendung und Vererbung [21]. Dabei ist eine Klasse gekoppelt, wenn sie die Schnittstelle einer anderen Klasse verwendet bzw. von dieser erbt.

Kohäsion Der Begriff der Kohäsion beschreibt den Zusammenhang innerhalb von Teilen eines Systems.

Diese können innerhalb von Klassen die Methoden und Attribute oder innerhalb einer Komponente die enthaltenen Klassen betreffen. Kohäsion beschreibt die Art und das Ausmaß, wie Teile, welche die gleiche Aufgabe haben, zueinander in Beziehung stehen [16]. Dies ist ein logischer Zusammenhalt.

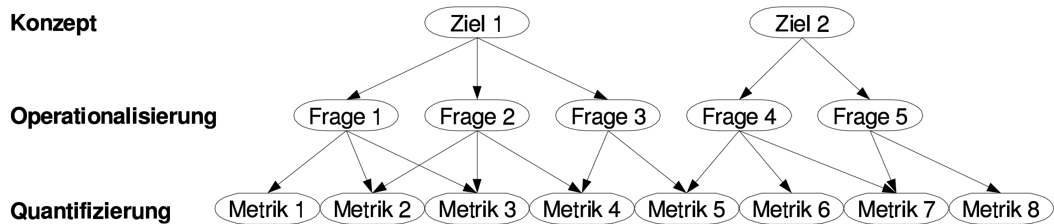

Abbildung 5.1.: Ebenen des *Ziel-Frage-Metrik*-Ansatzes [75].

> Streng genommen ist ein Kohäsionsmaß dann ein internes Kopplungsmaß. Whitmire definiert Kohäsion daher als semantischen Zusammenhalt. Danach besteht eine hohe Kohäsion zwischen den Bestandteilen eines Teil-Systems, wenn diese inhaltlich an der gleichen Aufgabe arbeiten und das gleiche Ziel verfolgen.

Whitmire [92] beschreibt noch weitere Arten von Entwurfsmetriken: *Hinlänglichkeit*, *Vollständigkeit*, *Einfachheit*, *Ähnlichkeit* und *Zerbrechlichkeit*. Diese werden hier nicht näher diskutiert.

5.2.4. Entwurf geeigneter Metriken

Es ist schwierig zu einzelnen vage formulierten Entwurfsmängeln konkrete Metriken anzugeben, die diese charakterisieren sollen. Es ist ggf. nötig eine Menge von Metriken auszuwählen und deren Nutzen zur Beschreibung eines Mangels zu prüfen. Dieser Abschnitt beschreibt ein Verfahren, mit dem systematisch geeignete Metriken gefunden werden können.

In der Übersicht zu diesem Kapitel wurde der Industriestandard ISO 9126 skizziert. Dieser verwendet den *Ziel-Frage-Metrik*-Ansatz [75] um von allgemeinen Ideen oder allgemeinen Vorstellungen zu konkreten, bis hin zu messbaren Eigenschaften zu gelangen.

Die Abbildung 5.1 zeigt die drei Ebenen dieses Ansatzes. Auf der Konzeptebene werden zunächst allgemeine Ziele formuliert. Jedes dieser Ziele wird zu einer Menge von Fragen verfeinert. Diese Ebene der Operationalisierung prüft oder bewertet das Erreichen des jeweiligen Ziels. Aus den einzelnen Fragen werden ggf. mehrere Metriken konstruiert, die diese quantitativ beantworten.

Ziele

Zielformulierungen werden aus dem Zweck, dem Problem und der Sichtweise gebildet. Der Zweck kann sein, etwas zu verbessern, vorherzusagen, zu beurteilen oder zu charakterisieren. Das Problem kann die Effizienz, die Kosten, der Fehler, die Änderungen oder die Charakterisierung sein. Die Sichtweise kann die eines Entwicklers, Kunden oder Projektleiters sein.

Das hier verfolgte Ziel ist klar: Es sollen Entwurfsmängel erkannt werden. Somit ist das Ziel, eine Programmstelle als mit einem Entwurfsmangel behaftet zu charakterisieren. Die Sichtweise ist die eines Software-Entwicklers oder -Architekten.

Jeder betrachtete Entwurfsmangel bildet in diesem Modell eine eigene Zielformulierung.

Fragen

Die Fragen dienen nun zum Einen dazu, einzelne Ziele zu konkretisieren und zum Anderen, ihr Erreichen zu beurteilen.

Wenn es sich um Ziele handelt etwas zu verbessern, dann fällt es leicht Fragen zu formulieren, die den aktuellen Ist-Stand beleuchten und Fragen zu stellen, die einen Soll-Stand bewerten oder direkt Verbesserungen aufzeigen. Hier wird das spezielle Ziel der Charakterisierung verfolgt, sodass Fragen formuliert werden müssen, die Symptome und Indizien von Entwurfsmängeln betreffen und zugleich den gewünschten Soll-Stand benennen.

5.3. Beispielmodelle

Im Folgenden werden Modelle für einige Entwurfsmängel entwickelt. Ausgehend von einer Zielformulierung dienen konkrete Fragen der Operationalisierung. Die Diskussion dieser Fragen führt zu konkreten Metriken, die in der Gesamtheit das Modell eines Entwurfsmangels bilden.

5.3.1. Große Klasse

Das Ziel ist es, eine „Große Klasse“ zu erkennen.

Frage: Ist die Art der Programmstelle eine Klasse?

Es können nur einzelne Klassen den Entwurfsmangel einer großen Klasse enthalten. Dieser Entwurfsmangel hat keinen Bezug zu anderen Programmstellen.

Metrik: *Typ der Programmstelle (Paket, Klasse oder Methode)* (Nominalskala)

Frage: Erfüllt die Klasse mehr als eine Aufgabe?

Dies ist die zentrale Frage, die sich bei der Erkennung einer großen Klasse stellt. Um diese zu beantworten werden die bei Fowler beschriebenen Indizien in konkretisierten Fragen aufgegriffen:

Frage: Enthält die Klasse eine hohe Population?

Fowler argumentiert, dass man eine große Klasse an der Anzahl der Felder erkennen kann. Die Feldanzahl kann als Maß für die Komplexität einer Klasse verstanden werden. Die Werte der Felder bilden den Zustand eines Objektes. Mit jedem zusätzlichen Feld wächst die Menge der unterscheidbaren Objektzustände um ein

Vielfaches — genauer: multiplikativ um die Anzahl der Werte, die ein Feld zur Laufzeit annimmt. Wir nehmen an, dass sich die steigende Anzahl von Feldern als steigende Komplexität der Methoden einer Klasse bemerkbar macht. Sicherlich wird in der Praxis nicht jedes Feld zu einer Fallunterscheidung herangezogen; in den meisten Fällen dienen sie lediglich der Speicherung von Werten, die sich mehr oder weniger häufig ändern.

Metrik: *Anzahl der Felder* (Absolutskala)

Metrik: *Mittlere Komplexität der Methoden einer Klasse* (Absolutskala)

Fowler argumentiert weiter, dass eine große Menge Quelltext für zu hohe Redundanz spricht und auf duplizierten Quelltext hindeutet. Eine übermäßige Anzahl an Methoden ist kein Indiz, da gerade viele kleine Methoden auf gut entworfenene Klassenschnittstellen deuten. Die Gesamtzahl der Anweisungen ist daher abhängig von der Anzahl der Methoden. Wir bestimmen daher die Anzahl von Anweisungen jeder Methode.

Metrik: *Median der Anzahl der Anweisungen aller Methoden der Klasse* (Absolutskala)

Frage: Können Teilklassen extrahiert werden?

Das Prinzip der Klassenextraktion beruht darauf, dass eine neue Klasse erstellt wird, die von der bestehenden Klasse assoziiert wird. Durch verschieben von Methoden und Feldern wird die bestehende Klasse verkleinert. Als letzter Schritt wird entschieden, ob die bestehende Klasse die neue Klasse verbergen sollte und daher die alten Methoden als Delegationen beibehält, oder ob die neue Klasse direkt verwendet werden soll.

Manche eng verwobene große Klasse, die auf diese Art zerrissen wird, erfordert u. U. eine bidirektionale Assoziation. Dies ist dann nötig, wenn eine nicht eigenständige neue Klasse entsteht, die die Funktionalität der alten Klasse benötigt. In solchen Fällen erscheint die Extraktion fragwürdig.

Zur *Analyse des internen Zusammenhangs einer Klasse* betrachten wir folgenden ungerichteten Graphen. Alle Methoden und Felder sind Knoten des Graphen. Eine Kante existiert zwischen einer Methode und einem Feld, wenn die Methode das Feld lesend oder schreibend verwendet. Eine Kante existiert zwischen zwei Methoden m_1 und m_2, wenn die Methode m_1 die Methode m_2 aufruft (siehe Beispiel in Abbildung 5.2).

Die Abbildung 5.2 zeigt ein Beispiel für einen Klassenabhängigkeitsgraphen mit den Methoden $m_1, m_2, \ldots, m_m$ und den Feldern $f_1, f_2, \ldots, f_n$. Die Abhängigkeiten bilden folgende Zusammenhangskomponenten: $\{m_1, m_2, f_1, f_2, f_3\}$, $\{m_3\}$, $\{m_4, m_m, f_4\}$ und $\{f_n\}$.

Der Graph stellt eine transitive „kennt“-Relation dar. Methoden und Felder kennen einander, wenn eine Methode ein Feld liest oder schreibt. Methoden kennen Methoden, wenn eine Methode eine andere Methode aufruft. Felder kennen ande-

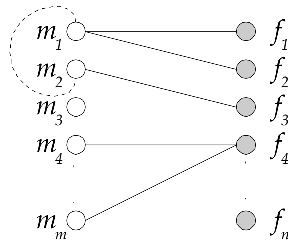

Abbildung 5.2.: Klassenabhängigkeitsgraph

re Felder, wenn deren Werte in Methoden bekannt sind, oder Werte von Feldern als Parameter an Methoden übergeben, oder als Ergebniswert zurückgeliefert werden können.

Dieser Graph liefert folgende Informationen:

- Wir betrachten die Zusammenhangskomponenten des Graphen als Mengen von Methoden mit den benötigten Feldern. Es gibt jeweils keine anderen Methoden, die enthaltene Felder benötigen. Damit lassen sich diese *vermutlich*als Klasse extrahieren.
- Es kann *einsame Knoten* geben. D. h. wir betrachten Knoten k mit Grad$(k) = 0$.

  Wenn eine Methode keine Felder benutzt, kann sie als *Hilfsmethode* bezeichnet werden. Sie ist unabhängig vom Zustand der Objekte ihrer Klasse. Ein nicht benutztes Feld kann ggf. gelöscht werden, sofern es nicht von außerhalb der Klasse zugegriffen wird.

Frage: Lassen sich Unterklassen extrahieren?

Die Voraussetzungen zur Extraktion einer Klasse oder einer Unterklasse sind sehr ähnlich. Es wird eine unabhängige, zusammenhängende Teilmenge der Methoden und Felder benötigt. Das Ziel ist jedoch sehr verschieden. Bei der Klassenextraktion wird eine neue Klasse erstellt, die per Delegation benutzt oder den Nutzern der Originalklasse zusätzlich zur Verfügung gestellt wird. Bei der Unterklassenextraktion wird ein extrahierbarer Teil in eine Unterklasse verschoben. D. h. alle Nutzer der Originalklasse müssen sich entscheiden, ob sie die — jetzt beschnittene — Klasse weiterhin verwenden können, oder die neue Klasse verwenden müssen, welche weiterhin, aufgrund der Vererbung, die vollständige Funktionalität enthält.

Fowlers primäres Indiz, das zur Extraktion einer Unterklasse führt, ist die Analyse der Benutzung der ursprünglichen Klasse. Wir betrachten hierzu die Metho-

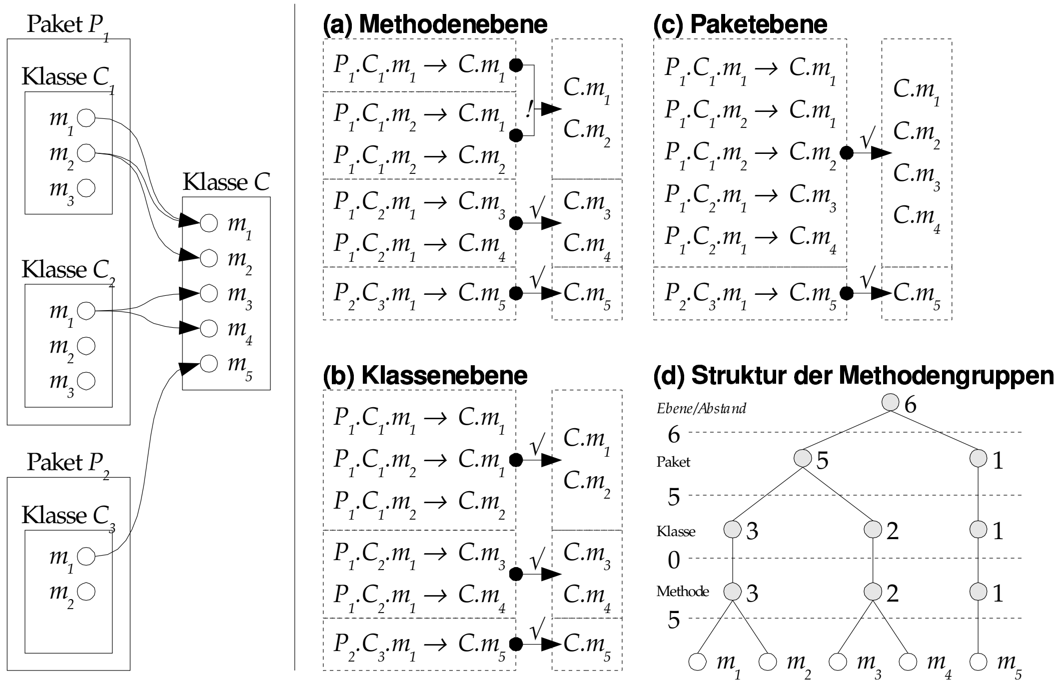

Abbildung 5.3.: Aufrufgraph zur Analyse des externen Zusammenhangs.

denaufrufstellen[1] in anderen Kontexten als der Klasse selbst. Hier kommen fremde Methoden, Klassen und Pakete in Betracht.

Zur *Analyse des externen Zusammenhangs* wird folgender ungerichteter spezieller Aufrufgraph gebildet. Knoten des Graphen sind Methoden des Systems. Es wird zwischen eigenen und fremden Methodenknoten unterschieden. Außerdem werden fremde Knoten weiterhin anhand ihres Kontextes unterschieden. Kontexte können Pakete, Klassen oder Methoden sein.

Die Abbildung 5.3 zeigt ein Beispiel für einen Aufrufgraphen zur Analyse des externen Zusammenhangs mit den Paketen P_1 und P_2, sowie den Klassen $P_1.C_1, P_1.C_2, P_2.C_3$ und C und resultierende Methodengruppen auf (a) Methoden-, (b) Klassen- und (c) Paketebene. (d) zeigt die Struktur der Methodengruppen der betrachteten Klasse mit den Methoden als Blattknoten und Methodengruppen als innere Knoten.

Eine *Methodengruppe* ist eine Menge von Methoden der betrachteten Klasse, die ausgehend vom gleichen Programmkontext aufgerufen werden. Wir bilden mehrere disjunkte Methodengruppen zu einer Klasse, sodass die Vereinigung dieser Mengen die Menge der Methoden dieser Klasse ergeben.

[1]Feldbenutzungen treten de Facto kaum auf und werden hier zunächst ausgenommen.

In Abbildung 5.3(a) ist die Gruppenbildung auf Methodenebene zu sehen. Hierzu wird die Menge der Aufrufkanten des Graphen in Teilmengen so zerlegt, dass in jeder Menge nur eine Methode als Aufrufstelle auftritt. Bilden die Aufrufziele jetzt bereits disjunkte Teilmengen, so werden sie übernommen. Nicht disjunkte Teilmengen werden vereinigt. In dem Beispiel werden die ersten beiden Mengen vereinigt, die beiden weiteren Mengen sind bereits disjunkt, sodass sich auf der Methodenebene die Methodengruppen $\{m_1, m_2\}, \{m_3, m_4\}, \{m_5\}$ für die Klasse C ergeben.

In weiteren Programmkontexten wie Klasse oder Paket wird analog vorgegangen. Die Abbildungen 5.3(b) und 5.3(c) zeigen entsprechende Methodengruppen. Es fällt auf, dass durch Vereinigung von Methodenmengen auf Methoden- und Klassenebene die gleichen Methodengruppen entstanden sind. Offensichtlich konnten auf der feinen Methodenebene nicht immer einzelne Methoden identifiziert werden, die gezielt eine Menge von Methoden aufrufen, die nicht auch von anderen Methoden aufgerufen werden. Auf Klassenebene lassen sich die Methodengruppen in diesem Beispiel gut unterscheiden: die Methoden $\{m_1, m_2\}$ werden von der Klasse C_1 aufgerufen, die Methoden $\{m_3, m_4\}$ werden von der Klasse C_2 aufgerufen und die Methode $\{m_5\}$ wird von der Klasse C_3 aufgerufen.

Frage: Können Schnittstellen extrahiert werden?

Fowlers wesentliche Idee zur Extraktion von Schnittstellen beruht darauf, dass nur ein Teil der Methoden einer Klasse aus einem Kontext heraus genutzt wird. Nutzer müssen z. B. mit mehreren Klassen interagieren, die die gleichen Methoden implementieren aber auch noch andere Methoden implementieren, die in anderen von der Klasse gespielten Rollen genutzt werden. Dies wird entweder durch Mehrfachvererbung ermöglicht oder, wie in Java, durch Schnittstellen modelliert.

Für die Extraktion von Schnittstellen können wir die externe Zusammenhangsanalyse verwenden. Zu jeder Methodenmenge wird der erzeugende Kontext notiert. Allerdings werden hier auch nicht disjunkte Mengen zugelassen. Dadurch ergeben sich Methodenmengen, die als Schnittstelle dienen können. Das Gewicht dieser Schnittstellen liefert den Hinweis, welche besonders häufig verwendet werden und extrahiert werden könnten.

Aus der Analyse des internen und externen Zusammenhangs lassen sich folgende Metriken ableiten:

Metrik: *Anzahl der internen Zusammenhangskomponenten* (Absolutskala)

Metrik: *Anzahl der externen Zusammenhangskomponenten (disjunkt)* (Absolutskala)

Metrik: *Anzahl der externen Zusammenhangskomp. (nicht disjunkt)* (Absolutskala)

5.3.2. Lange Methode

Das Ziel ist es, eine „Lange Methode“ zu erkennen.

Frage: Ist die Programmstelle eine Methode?
Nur einzelnen Methoden kann dieser Entwurfsmangel anhaften. Es gibt keinen Bezug zu anderen Methoden.

Metrik: *Typ der Programmstelle (Paket, Klasse oder Methode)* (Nominalskala)

Frage: Ist diese Methode für mehr als eine Aufgabe zuständig?
Analog zur großen Klasse werden die bei Fowler beschriebenen Indizien in konkretisierten Fragen aufgegriffen:

Frage: Wie komplex ist die Methode?
Fowler betrachtet nicht die Anzahl der Anweisungen einer Methode als das Hauptübel, sondern die Komplexität der Kontrollstruktur. Diese erschwert das Verstehen einer Methode und liefert direkte Hinweise Methoden zu zerteilen.

Als Maß wird hier das klassische Komplexitätsmaß *Cyclomatic Complexity* [58] von McCabe eingesetzt. Es basiert auf dem Kontrollflussgraphen und berechnet (vereinfacht beschrieben) die Summe der Knoten und Kanten des Graphen.

Metrik: *McCabes Cyclomatic Complexity* (Absolutskala)

Frage: Wie hoch ist die Population der Methode?
Ähnlich wie bei der großen Klasse sind die Bestandteile einer Methode interessant. Die Anzahl der Parameter und lokalen Variablen deuten ebenso in Richtung einer hohen Komplexität wie die Anzahl der Anweisungen einer Methode.

Wie bereits im Modell der großen Klasse angedeutet, ist der Zusammenhang jedoch zweifelhaft. Als Ergänzung zum Komplexitätsmaß auf der Basis des Kontrollflusses werden diese aber mit aufgenommen.

Metrik: *Anzahl der Anweisungen* (Absolutskala)

Metrik: *Anzahl der lokalen Variablen* (Absolutskala)

Metrik: *Anzahl der Parameter* (Absolutskala)

Frage: Was berechnet die Methode?
In der objektorientierten Programmierung haben Methoden bisweilen nur winzige Aufgaben. Sie dienen als *set*- oder *get*-Methode um den Zugriff auf Attribute zu kapseln. Sie sind Teil einer Fassade und delegieren ihre Aufgabe an andere Objekte oder Methoden der eigenen Klasse. Manchmal berechnen sie aber auch tatsächlich etwas und erfüllen somit eine eigene Aufgabe. Eine lange Methode ist eine Methode, die etwas berechnet, aber i. d. R. zu vieles.

Metrik: *Art der Methode (Setter, Getter, Delegator, Worker)* (Nominalskala)

5.3.3. Faule Klasse / Datenklasse

Das Ziel ist es, eine „Faule Klasse" bzw. „Datenklasse" zu erkennen.

Die Diskussion der „Großen Klasse" hat bereits die zentralen Eigenschaften von Klassen beleuchtet. Für diese beiden Entwurfsmängel werden daher die gleichen Metriken zur Charakterisierung verwendet. Allerdings erwartet man statt einer schwachen Kohäsion eine eher hohe Kohäsion und statt hoher Kopplung eine eher geringe Kopplung. Dies deutet in Richtung eines wünschenswerten Zustandes. Allerdings fehlt diesen Klassen fast vollständig das eigene Verhalten, sodass folgende Fragen ergänzt werden:

Frage: Hat die Klasse keine eigene Aufgabe?
Die Aufgaben einer Klasse müssen in der Implementierung seiner Methoden enthalten sein, sodass deren Art zu betrachten ist.

Frage: Was berechnen die Methoden?
Bei der „Langen Methode" wurde bereits die Art einer Methode verwendet. Diese Unterscheidung bietet sich hier ebenfalls an, um eine Skizze der Klasse zu erstellen. Gemessen wird die Anzahl der Methoden der verschiedenen Arten.

Metrik: *Anzahl der Setter-Methoden.* (Absolutskala)

Metrik: *Anzahl der Getter-Methoden.* (Absolutskala)

Metrik: *Anzahl der Delegator-Methoden.* (Absolutskala)

Metrik: *Anzahl der Worker-Methoden.* (Absolutskala)

Auf diesem Wege lassen sich vielleicht auch Datenklassen von faulen Klassen unterscheiden. Von einer Datenklasse wird man nur *Setter-* und *Getter*-Methoden erwarten. Eine faule Klasse wird hingegen einen Anteil von *Worker*-Methoden enthalten.

5.3.4. Neid

Das Ziel ist es, „Neid" zu erkennen.

Dieser Entwurfsmangel unterscheidet sich von den bisher modellierten dadurch, dass er nicht eine einzelne Programmstelle sondern die Beziehung zu einer anderen Programmstelle betrifft. Die Beziehung zu anderen Programmstellen wird daher durch die Metriken aufgegriffen.

Frage: Sind die beteiligten Programmstellen Methoden oder Klassen?
In Abschnitt 3.2.4 wurde bereits erläutert, dass Neid sich nicht nur auf Methoden, sondern auch allgemeiner auf Klassen beziehen kann. Gesucht werden also Methoden oder Klassen, die sich zu sehr mit den Details fremder Klassen beschäftigen.

Metrik: *Typ der Programmstelle (Paket, Klasse oder Methode)* (Nominalskala)

Frage: Beschäftigt sich die Programmstelle zu sehr mit den Details fremder Klassen?
Hierzu betrachtet man, wie eine Methode (bzw. die Methoden einer Klasse) andere Programmobjekte verwenden. Dies sind Felder, die zugegriffen oder Methoden, die aufgerufen werden. Um einen Eindruck vom Interessengebiet der Methode (bzw. der Klasse) zu erhalten unterscheidet man zwischen dem Zugriff auf Programmobjekte der eigenen Klasse im Vergleich zum Zugriff in fremde Klassen.

Metrik: *Anzahl der Felder in der eigenen Klasse* (Absolutskala)

Metrik: *Anzahl der verwendeten Felder in der eigenen Klasse* (Absolutskala)

Metrik: *Anzahl der verwendeten Felder in fremden Klassen* (Absolutskala)

Metrik: *Anzahl der Methoden in der eigenen Klasse* (Absolutskala)

Metrik: *Anzahl der aufgerufenen Methoden in der eigenen Klasse* (Absolutskala)

Metrik: *Anzahl der aufgerufenen Methoden in fremden Klassen* (Absolutskala)

Metriken dieser Art, die einfache absolute Größenmaße darstellen, sind nicht geeignet mehrere Programmobjekte untereinander zu vergleichen. Man verwendet daher relative Maße, indem man die gewünschten Maße in Relation zu einer Bezugsgröße setzt.

In diesem Fall bieten sich als Bezugsgröße die Anzahl der Felder bzw. aufgerufenen Methoden der eigenen Klasse an. Man weist dann das Verhältnis der Anzahl des eigenen Zugriffs zum fremden Zugriff aus.

5.4. Zusammenfassung

Es ist schwierig, geeignete Metriken zu finden, die einen Entwurfsmangel treffend charakterisieren. Unweigerlich fließen eigene Betrachtungsweisen des jeweiligen Entwurfsmangels in das Modell ein. Die hier vorgestellten Beispielmodelle stellen daher keinen allgemein verbindlichen oder anerkannten Katalog von Metriken dar, sondern einen initialen Vorschlag.

Bei der praktischen Umsetzung und empirischen Begleitung muss geprüft werden, ob die Kriterien geeignet sind Entwurfsmängel zu beschreiben.

Fowler beschreibt auch Entwurfsmängel, die nicht durch Programmeigenschaften charakterisiert werden können. „Divergierende Änderungen“ z. B. beschreiben den Effekt, dass eine Änderung eine andere auslöst, die wiederum eine Änderung an der ersten Stelle betrifft. „Schrotkugeln herausoperieren“ beschreibt die Auswirkung einer Änderungen auf viele andere Programmstellen. Solche Änderungsmuster lassen sich häufig nicht durch Analyse der Abhängigkeiten innerhalb des Programms aufdecken, sodass z. B. Analysen von Versionsarchiven nötig würden [96].

Zur Berechnung von Metriken wird die statische Programmanalyse eingesetzt. Da zu erwarten ist, dass andere oder veränderte Metriken benötigt werden, ist ein generischer Ansatz nötig, der es erlaubt weitere Metriken in kurzer Zeit bereitzustellen. Das folgende Kapitel beschreibt ein Verfahren hierzu.

6. Analyse von Programmstrukturen

Inhalt

6.1. Überblick **70**

6.2. Analyse von Java-Programmen **71**

6.2.1. Methoden statischer Programmanalyse 71

6.2.2. Aspekte der Objektorientierung 72

6.3. Struktur objektorientierter Programme **73**

6.3.1. Pakete, Klassen, Methoden und Attribute 73

6.3.2. Beziehungen zwischen Programmobjekten 74

6.3.3. Programmabhängigkeitsgraph 75

6.4. Berechnung von Metriken **76**

6.4.1. Verwendung der relationalen Algebra 78

6.4.2. Auswertung von Relationen 80

6.4.3. Beispielberechnungen 82

6.5. Verwandte Ansätze **83**

6.6. Zusammenfassung **84**

6.1. Überblick

In nahezu jeder Ingenieursdisziplin gehört die Bewertung eines Entwurfes zu den Pflichtaufgaben, bevor konkrete Bauwerke, Geräte oder Systeme gebaut werden. Man investiert lieber Zeit und Geld in die Architekturbewertung, als dass man später feststellen müsste, dass das Endprodukt den gewünschten Anforderungen nicht genügt.

Der Entwurf und die Entwicklung von Software sollten dabei keine Ausnahme sein. Auch hier sind Entwurfs- und Architekturbewertungen wichtige Hilfsmittel um große Software-Projekte zu steuern.

Bestehende Software-Systeme können Entwurfsfehler enthalten, welche die Wartung und Weiterentwicklung behindern. Diese Fehler zu entdecken ist das Ziel dieser Arbeit.

Im vorherigen Kapitel 5 wurden Modelle von Entwurfsmängeln entworfen. Diese sollen helfen problematische Entwürfe zu erkennen. In diesem Kapitel wird ein Analyseverfahren beschrieben, mit dem vielfältige Metriken der beschriebenen Art auf einfache Weise berechnet werden können.

Wie bereits beschrieben, existiert in der Literatur eine Vielzahl von Metrikdefinitionen für verschiedene Einsatzzwecke. Es ist daher nötig, einen leicht erweiterbaren oder generischen Ansatz zur Metrikberechnung zu finden.

Die beschriebenen Metriken gehören zur Kategorie der Entwurfsmetriken, d. h. sie charakterisieren Entwürfe und können daher auch von Systementwürfen, die als Modell (z. B. ein UML-Modell) vorliegen, berechnet werden. Analysiert man, wie in dem hier beschriebenen Ansatz, jedoch Java-Programme, so ergibt sich die Chance neben den Informationen über die Programmstruktur auch Informationen über die Implementierung und die darin verborgenden Strukturen zu erhalten.

Objektorientierte Programme sind zur Ausführungszeit ein Geflecht von Objekten, die über Nachrichten kommunizieren. Die Objekte beschreiben konkrete Abläufe, haben einen konkreten Zustand und spielen Rollen in ihren jeweiligen Verwendungskontexten.

Die Betrachtung des nicht ausgeführten Programms, die statische Programmanalyse, zeigt die Programmeigenschaften und Strukturen, die anhand der Definition der Programmbestandteile erkannt werden können. Sie beschreibt das Verhalten eines Programms in Form einer Hypothese. Die erkannten Ausführungspfade oder Kommunikationswege können genutzt werden, müssen es aber nicht.

Ziel der statischen Programmanalyse ist es, stets das dynamische Verhalten eines Programmes präzise zu modellieren. Dieses Ziel wird hier nicht verfolgt. Aus Sicht der Bewertung von Programm- oder Systementwürfen spielen Details des Programmverhaltens eine untergeordnete Rolle.

Sollte das Programm z. B. bezüglich der Laufzeit oder des Speicherbedarfs op-

timiert werden, und würden anhand von Analyseergebnissen automatische Programmtransformationen durchgeführt, so wären präzise Analyseergebnisse unerlässlich.

Hier ist das Ziel allerdings, Entwurfsmetriken zu berechnen, die später durch maschinelle Lernverfahren weiter zu abstrakten Konzepten verarbeitet werden sollen. Im Hinblick auf den Einsatz in einem interaktiven Werkzeug müssen die benötigten Informationen durch Laufzeit- und Speicherplatzsparende Verfahren ermittelt werden.

Dieses Kapitel skizziert im nun folgenden Abschnitt 6.2 Methoden der klassischen Programmanalyse und geht dabei auch auf Besonderheiten der objektorientierten Programmierung ein.

Im Abschnitt 6.3 wird die allgemeine Struktur objektorientierter Programme beschrieben und daraus die Konstruktion eines Programmabhängigkeitsgraphen abgeleitet. Dies dient als Grundlage für die Berechnung von Entwurfsmetriken in Abschnitt 6.4. In der Literatur findet man Verfahren und Werkzeuge, die Informationen über Programme für ähnliche Zwecke berechnen. Dies wird *Fact Extraction* genannt. Der Abschnitt 6.5 gibt einen Überblick.

6.2. Analyse von Java-Programmen

Statische Programmanalyseverfahren haben ihren Ursprung in der Konstruktion von Übersetzern. Sie werden dort eingesetzt um Eigenschaften eines Programmes präzise zu bestimmen, die genutzt werden um zum Einen die statische Semantik des Programms zu prüfen und zum Anderen um optimierten Code zu generieren. Analysegegenstand der klassischen Verfahren sind imperative strukturierte Programme.

6.2.1. Methoden statischer Programmanalyse

Grundsätzlich wird anhand des Analysekontextes zwischen inter- und intraprozeduralen Analysen unterschieden. Eine interprozedurale Analyse bestimmt die Eigenschaften einer einzelnen Methode oder Funktion, wohingegen intraprozedurale Analysen Eigenschaften bestimmen, die in einem größeren Kontext gültig sind. In [62] [61] findet man eine Übersicht der klassischen Analyseverfahren.

Kontrollfluss

Die Kontrollflussanalyse konstruiert einen Kontrollflussgraphen, der alle Abläufe einer Methode modelliert. Die Knoten des Graphen sind die Grundblöcke einer Methode, eine gerichtete Kante führt von einem Grundblock auf einen möglichen nachfolgenden Grundblock im Ablauf der Methode. Ein Grundblock ist eine Folge von Anweisungen, die in jedem Fall nacheinander abgearbeitet werden. Varianten

im Ablauf der Methode werden also ausschließlich durch die Kontrollflusskanten modelliert. Der Kontrollflussgraph einer Methode hat genau einen Eingang und einen Ausgang, ggf. werden künstliche Ein- und Ausgangsknoten ergänzt um den vorzeitigen Abbruch einer Methode zu modellieren.

Bei der Analyse von Java-Programmen sind *Exceptions* zu beachten. Wenn eine solche *Exception* auftritt, so wird der Ablauf einer Methode abgebrochen und mit der Behandlung der Ausnahmesituation fortgefahren. Viele Anweisungen des Java-Bytecodes können *Exceptions* auslösen (z. B. Division durch Null), sodass sehr viele fein-granulare Grundblöcke entstehen. Man beschränkt sich daher darauf nur die *Exceptions* zu modellieren, die explizit durch eine Anweisung im Programm ausgelöst werden.

Kontrollflussgraphen sind die Grundlage für die Analyse des Datenflusses.

Datenfluss

Die Datenflussanalyse ist ein schematisches Verfahren, mit dem unterschiedliche Datenflussfragen beantwortet werden können. Man unterscheidet zwischen vorwärts und rückwärts zu berechnenden Problemen.

Ein Beispiel für ein Vorwärtsproblem ist die „Erreichende Definition". Man betrachtet die Menge der Zuweisungsstellen an Variable (Definitionsstelle) und berechnet, welche Grundblöcke des Kontrollflussgraphen diese Definition unverändert erreicht. Genutzt wird diese Information z. B. um die Reihenfolge von Anweisungen zu permutieren. Im Kontext einer Schleife könnte z. B. die Definition einer unveränderlichen Variable, statt innerhalb des Schleifenrumpfes, vor dem Rumpf platziert werden.

6.2.2. Aspekte der Objektorientierung

Die Verfahren der klassischen Programmanalyse lassen sich i. d. R. auch auf objektorientierte Programme anwenden. Eine Besonderheit ist jedoch die Vererbung (siehe Abschnitt 2.3.3), aus der sich, als eine spezialisierte Form der statischen Typisierung, die dynamische Methodenbindung und die Untertypeigenschaft ableiten.

Die dynamische Methodenbindung verlangsamt Methodenaufrufe erheblich, sodass man versucht, diese durch statische Aufrufe zu ersetzen. Dies ist nur dann erlaubt, wenn der Typ des Zielobjektes bereits zur Übersetzungszeit exakte ermittelt werden kann.

Durch *Class Hierarchy Analysis* [25], *Rapid Type Analysis* [4] und intraprozedurale Klassenanalyse [40] wird dies versucht.

Meta-Programmierung

Einige Programmiersprachen (z. B. Smalltalk und Java) verfügen über eine eingeschränkte Form der Meta-Programmierung, d. h. Strukturen und Programmobjek-

te des Programms können zur Laufzeit inspiziert und verwendet werden.

In Java erlaubt der *Reflection*-Ansatz [84] z. B. Klassen und Methoden anhand ihres Namens (in Form einer Zeichenkette) zu instantiieren bzw. aufzurufen.

Dies erschwert die statische Programmanalyse bzw. lässt sie grundsätzlich scheitern. Nur der einfache und offensichtliche Einsatz lässt sich durch aufwendige abstrakte Interpretationen verfolgen.

6.3. Struktur objektorientierter Programme

Die zu berechnenden Metriken sind einerseits Eigenschaften der Programmstruktur und andererseits Eigenschaften, die durch spezialisierte Programmanalysen bestimmt werden müssen.

Bereits im Abschnitt 2.3 wurden Konzepte objektorientierter Programmierung beschrieben. Die statische Struktur objektorientierter Programme ergibt sich direkt aus der Definition der Programmobjekte.

6.3.1. Pakete, Klassen, Methoden und Attribute

Die betrachteten Programmobjekte in Java sind sind Pakete, Klassen, Methoden und Attribute.

Die Pakete bilden jeweils einen Namensraum und enthalten eine Menge von Klassen, haben aber keine weiteren Eigenschaften. Die Klassen eines Paketes werden für andere Klassen sichtbar und benutzbar, indem der Namensraum (oder Teile davon) als zu verwenden deklariert wird (*import*).

Einzelne Klassen können konkret oder abstrakt sein. Von abstrakten Klassen können keine Objekte gebildet werden. Die Sichtbarkeit einer Klasse legt fest, aus welchem Kontext sie verwendet werden kann. In Java unterscheidet man die Sichtbarkeiten: *public*, *private*, *protected* und *default*. Details werden in [85, Abs. 2.4.4] erläutert.

Die Klassen enthalten Felder sowie Methoden und innere Klassen.

Attribute sind statisch typisiert. Neben einigen Grundtypen werden in Java komplexe Typen verwendet. Diese sind Klassen und *Interfaces*. Ein Attribut von einem komplexen Typ enthält als Wert stets nur eine Referenz auf ein Objekt des jeweiligen Typs. Bei Grundtypen ist es der Wert selbst.

Einzelne Methoden haben typisierte Parameter, einen Ergebnistyp und einen Anweisungsrumpf. Überladene Methoden haben gleiche Bezeichner aber unterschiedliche Signaturen und Rückgabetypen. Der Methodenrumpf besteht aus strukturierten Anweisungen.

6.3.2. Beziehungen zwischen Programmobjekten

Zwischen einzelnen Programmobjekten bestehen statische und dynamische Beziehungen.

Die statischen Beziehungen ergeben sich direkt aus der Definition und Komposition der einzelnen Programmobjekte.

Enthaltensein Die hierarchische Komposition einzelner Programmobjekte beschreibt die Beziehungen des Enthaltensein. Pakete enthalten Klassen und *Interfaces*, Klassen und *Interfaces* enthalten Methoden und Attribute und zuletzt enthalten Klassen innere und anonyme Klassen. Die Enthaltensein-Relation ist transitiv.

Vererbung Vererbungsbeziehungen betreffen Klassen. Eine Unterklasse kann jeweils von genau einer Oberklasse erben. Nimmt man *Interfaces* hinzu, so kann eine Klasse mehrere *Interfaces* implementieren. Die Vererbungsrelation ist ebenfalls transitiv.

Dynamische Beziehungen beschreiben Eigenschaften der Anweisungsfolgen in den Methoden und lassen sich nicht alleine durch die Struktur der Programmobjekte bestimmen.

Assoziation Eine Assoziationsbeziehung zwischen zwei Klassen wird angenommen, wenn ein Attribut der einen Klasse vom Typ der weiteren Klasse existiert. Diese statische Eigenschaft betrifft nur einfache *n:1*-Beziehungen. In Abschnitt 2.3.2 wurden weitere Kardinalitäten skizziert, die durch Behälterklassen implementiert werden.

In realen Java-Programmen sind diese schwer zu erkennen. Abgesehen von bekannten Behälterklassen (z. B. aus der Java-Klassenbibliothek) werden häufig eigene Implementierungen verwendet. Das Muster des Iterators ist eine gängige Implementierungsform, sodass sich mehrstellige Assoziationen erkennen lassen.

Im Programm findet man ein Attribut, das ein Behälterobjekt referenziert. Dieses Objekt stellt ein Iterator-Objekt auf Anfrage zur Verfügung, mit dem die Elemente des Behälters durchlaufen werden. Der Rückgabetyp eines Iterators kann zunächst als Assoziationspartner verstanden werden. Behälterklassen verwenden aber als Elementtyp häufig die universelle Klasse *Object*, die Oberklasse aller Klassen in Java ist, sodass beim Beschaffen eines einzelnen Objektes eine zusätzliche Typanpassung *(down-cast)* nötig ist. Anhand dieser Typanpassungen an allen Verwendungsstellen des Iterators erkennt man den konkreten (allgemeinsten) Typ der Elemente des Behälters und damit des Assoziationspartners.

Feldzugriff Im Anweisungsrumpf einer Methode wird auf Felder zugegriffen. Dies kann ein lesender oder schreibender Zugriff sein. Die Beziehung des Feldzugriffs besteht daher zwischen den Programmobjekten Methode und Attribut. Dabei können Attribute sowohl Instanz- als auch Klassenvariable sein.

Methodenaufruf Im Anweisungsrumpf einer Methode werden Methodenaufrufe beschrieben. Es ist zwischen statisch und dynamisch gebundenen Aufrufen zu unterscheiden. Bei statisch gebundenen Aufrufen ist bereits zur Übersetzungszeit das Ziel des Aufrufs bekannt, bei dynamischer Bindung entscheidet der Typ des konkreten Zielobjektes, welche Methode ausgeführt wird. Die statische Analyse kennt nur den vermeintlichen Typ des Zielobjektes, sodass zur Laufzeit dieser Typ und dessen Untertypen auftreten können.

Hier wird die Variante gewählt, die eine Relation zur statisch bekannten Klasse beschreibt. Die vollständige Menge der möglichen Zieltypen kann anhand der Vererbungsrelation bestimmt werden.

6.3.3. Programmabhängigkeitsgraph

Ziel der statischen Programmanalyse ist es, relevante Informationen über die Struktur des Programmes zusammenzutragen, sodass die wichtigsten Aspekte des Entwurfes sichtbar werden und die benötigten Metriken einfach berechnet werden können. Dabei ist nicht die Detailsicht gefordert, die als Information für optimierende Programmtransformationen benötigt würde, sondern eine abstrahierende Übersicht des Programms.

Ein spezieller Abhängigkeitsgraph wird als Grundlage für die Berechnung vielfältiger Metriken verwendet. Knoten in dem Graphen sind vom Typ der betrachteten Programmobjekte: Paket, Klasse, Methode und Attribut. Kanten werden durch die Beziehungen *„PartOf“* (Enthaltensein), *„Inheritance“* (Vererbung), *„Implements“* (Implementierung eines *Interfaces*), *„Association“* (Objektreferenz), *„Accessedfield“* (Feldzugriff), *„Methodcall“* (Methodenaufruf) gebildet.

Die Abbildung 6.1 ist einem *Entity-Relationship*-Diagramm nachempfunden und zeigt die Beziehungen zwischen den einzelnen Entitäten des Graphen. Die Entitäten sind hier als Klassendiagramm dargestellt und enthalten die Attribute der Entität. Die Pfeile beschreiben binäre Beziehungen, wobei die Richtung der Sprechweise entspricht. Es ist z. B. eine Klasse Teil eines Paketes oder eine Methode ruft eine andere Methode auf.

Die Beziehungen Methodenaufruf, Feldzugriff, Assoziation und *Interface*-Implementierung sind von der Kardinalität *n:n*, alle anderen Beziehungen sind von der Kardinalität *n:1* (in Pfeilrichtung). Es enthält z. B. ein Paket mehrere Klassen und eine Methode kann mehrere Methoden aufrufen, aber auch von mehre-

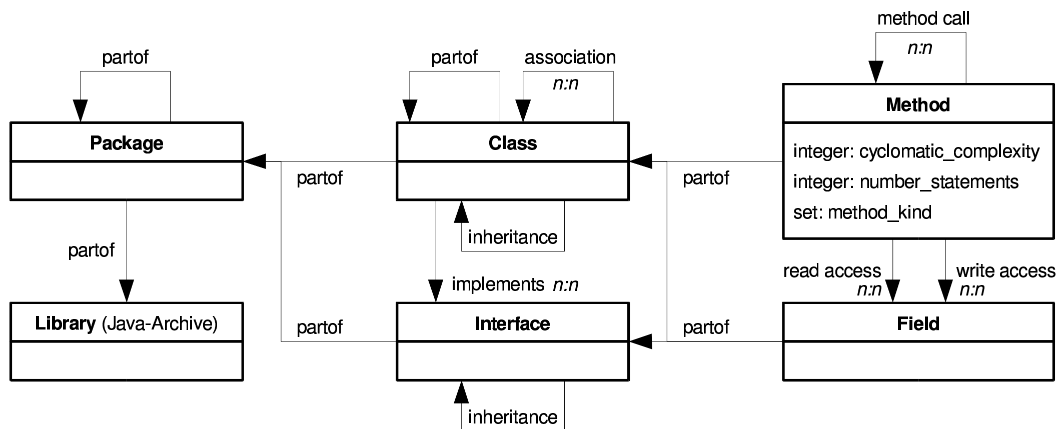

Abbildung 6.1.: Beziehungen zwischen Entitäten des Programmabhängigkeitsgraphen.

ren aufgerufen werden. Die *n:1*-Beziehungen bilden also Baumstrukturen und *n:n*-Beziehungen beschreiben bipartite Graphen.

Jede Entität wird durch ein Schlüsselattribut beschrieben. Diese sind hier nicht dargestellt. Es wird der eindeutige Name des jeweiligen Programmobjektes verwendet.

Die Abbildung 6.2 zeigt ein Beispiel für einen solchen Programmabhängigkeitsgraphen aus [50].

Analysekontext

Programmabhängigkeitsgraphen benötigen zum Aufbau viel Speicherplatz und Rechenzeit. Daher wird nicht das vollständige Programm, sondern nur der Kontext nach dem der Benutzer gefragt hat, analysiert; z. B. ein Paket oder eine Klasse.

Innerhalb des Kontextes werden alle Programmobjekte analysiert und für jedes ein Knoten angelegt. Bei der Analyse treten Beziehungen zu Programmobjekten auf, die außerhalb des Kontextes liegen. Diese werden dem Graphen ebenfalls als Knoten zugefügt und als kontextfremd markiert.

6.4. Berechnung von Metriken

Aus dem Programmabhängigkeitsgraphen lassen sich verschiedenartige Metrikwerte einfach ablesen. Im Folgenden werden einige Beispiele beschrieben.

Viele der Metriken lassen sich durch Einschränkung des Graphen auf bestimmte Beziehungsarten und durch Grapheigenschaften wie Zusammenhangskomponenten ermitteln.

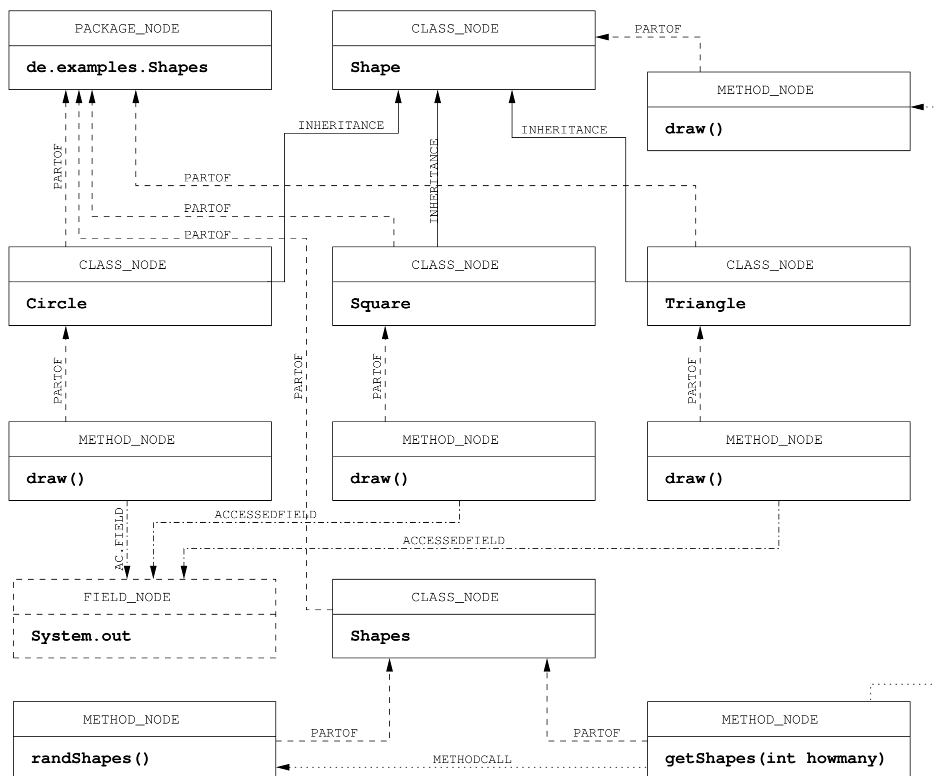

Abbildung 6.2.: Beispiel eines Programmabhängigkeitsgraphen [50].

Häufig müssen sehr ähnliche Metriken berechnet werden, die sich nur in der Wahl des Kontextes unterscheiden, z. B. die Anzahl der zugegriffenen Felder einer Methode innerhalb und außerhalb der eigenen Klasse. Durch Ausweitung des Kontextes unter Nutzung der Enthaltensein-Beziehung ist dies einfach möglich.

6.4.1. Verwendung der relationalen Algebra

Mit Hilfe einer relationalen Algebra [22] lassen sich benötigte Teile des Programmabhängigkeitsgraphen extrahieren und damit sehr einfach Metrikwerte berechnen.

Das hierzu nötige relationale Schema wurde im Abschnitt 6.3.3 nur angedeutet. Um konkrete Anfragen formulieren zu können, wird die relationale Modellierung wie folgt zu einem relationalen Schema ergänzt:

- Jede Entität ist eine Relation, die aus dem kartesischen Produkt der einzelnen Attribute besteht. Hierzu enthält jede Entität ein (in Abb. 6.1 unsichtbares) Attribut mit Namen *id*, das den eindeutigen Namen des Programmobjektes enthält. *id* ist also die Menge aller Programmobjektbezeichner. Es wird daher angenommen, dass jedes Programmobjekt — über alle Entitäten hinweg — einen eindeutigen Namen hat. In Java wird hierzu der voll qualifizierte Name verwendet. Das boolsche Attribut *visibility* zeigt an, ob die Entität innerhalb des aktuellen Analysekontextes liegt.
- Jede binäre Relation aus Abb. 6.1 wird abgebildet auf eine Relation gleichen Namens und den Attributen *from* und *to*. Beide bilden einen gemeinsamen Schlüssel und enthalten jeweils Werte der *id*-Attribute der beteiligten Entitäten.

Daraus ergibt sich das relationale Schema in Abbildung 6.3. Die Schlüsselattribute sind durch Unterstreichen gekennzeichnet.

Die Ergebnisse statischer Programmanalysen werden als konkrete Ausprägung des relationalen Schemas notiert, sodass durch die relationale Algebra konkrete Anfragen gestellt werden können um Metriken zu berechnen.

Die relationale Algebra ist *abgeschlossen*, d. h. das Ergebnis einer Anfrage ist wieder eine Relation. Damit lassen sich Anfragen hierarchisch komponieren.

Es werden folgende Operatoren der relationalen Algebra verwendet:

- *Selektion:* die Operation $\sigma_F(R)$ wählt die Elemente der Relation R aus, für welche die boolsche Formel F wahr ist.
- *Projektion:* die Operation $\Pi_A(R)$ extrahiert eine Menge von Attributen A aus der Relation R. D. h. es sind alle Elemente der Relation enthalten. Diese bestehen aber nur noch aus den Attributen der Menge A.

$$\begin{aligned}
\text{Library} &\subseteq \underline{id} \times visibility \\
\text{Package} &\subseteq \underline{id} \times visibility \\
\text{Class} &\subseteq \underline{id} \times visibility \\
\text{Interface} &\subseteq \underline{id} \times visibility \\
\text{Field} &\subseteq \underline{id} \times visibility \\
\text{Method} &\subseteq \underline{id} \times visibility \\
& \quad \times cyclomatic_complexity \\
& \quad \times number_statements \\
& \quad \times method_kind
\end{aligned}$$

$$\begin{aligned}
\text{partof} &\subseteq \underline{from} \times \underline{to} \\
\text{inheritance} &\subseteq \underline{from} \times \underline{to} \\
\text{implements} &\subseteq \underline{from} \times \underline{to} \\
\text{methodcall} &\subseteq \underline{from} \times \underline{to} \\
\text{readaccess} &\subseteq \underline{from} \times \underline{to} \\
\text{writeaccess} &\subseteq \underline{from} \times \underline{to}
\end{aligned}$$

Abbildung 6.3.: Relationales Schema zur Abfrage des Programmabhängigkeitsgraphen

- *Umbenennung:* die Operation $\rho_{R'}(R)$ benennt eine Relation R in die Relation R' um.

  Die Operation ist überladen, sodass auch Attribute einer Relation umbenannt werden können. Dann ist $\rho_{a' \leftarrow a}(R)$ die Relation R in der das Attribut a in a' umbenannt wurde.

  Zur Verkürzung der Schreibweise wird hier auch bei der Projektion die Umbenennung verwendet. Dabei gelte:

  $$\Pi_{a' \leftarrow a}(R) \equiv \rho_{a' \leftarrow a}(\Pi_a(R))$$

- *Vereinigung:* zwei Relationen R und S lassen sich vereinigen zu $R \cup S$, wenn beide das gleiche Schema (d. h. gleiche Attribute und gleiche Schlüssel) haben.

- *Differenz:* die Differenz zweier Relationen $R - S$ ist die Menge der Tupel, die in R aber nicht in S vorkommen.

- *Kartesisches Produkt:* das kartesische Produkt zweier Relationen $R \times S$ enthält für jedes Element aus R alle Elemente aus S. Die Kardinalität ist daher $|R| * |S|$. Das Schema des kartesischen Produktes ist die Vereinigung der Attribute beider Relationen.

- *Verbund:* Der Verbund $R \bowtie S$ ist ähnlich dem kartesischen Produkt. Er enthält aber nur die Elemente aus R und S, bei denen alle Attribute gleichen

Namens auch gleiche Werte enthalten.

Als Beispiel beschreibt die folgende Anfrage alle Methoden einer Klasse k:

$$M_k = \rho_{to \leftarrow id}(\sigma_{id=k}(\mathsf{Class})) \bowtie \mathsf{partof} \bowtie \rho_{from \leftarrow id}(\mathsf{Method})$$

Die Selektion wählt zunächst die Klasse aus der Class-Relation, die den Bezeichner k hat. Das Attribut id spielt für die partof-Relation die Rolle des Ziels. Daher wird das Attribut *id* in *to* umbenannt. Die Methoden spielen die Rolle der Quelle in der partof-Relation, sodass deren Attribut *id* in *from* umbenannt wird. Der Verbund der drei Relationen mit jeweils zu verknüpfenden *to*- bzw. *from*-Attributen ist dann die Relation

$$\begin{aligned} M_k \subseteq\; & to \times from \times visibility \\ & \times cyclomatic_complexity \times number_statements \times method_kind, \end{aligned}$$

wobei das Attribut to den Bezeichner der Klasse k enthält und aus der Class-Relation gefüllt wurde und alle anderen Attribute aus der Method-Relation gefüllt wurden.

Zur besseren Übersicht werden die Attribute to und $from$ umbenannt in den Namen der beteiligten Relation.

$$M'_k = \rho_{Class \leftarrow to, Method \leftarrow from}(M_k).$$

Um die folgende Darstellung zu vereinfachen, werden Verbunde dieser Art im Folgenden notiert als:

$$\begin{aligned} M'_k &= \sigma_{id=k}(\mathsf{Class}) \bowtie_{\mathsf{partof}} \mathsf{Method} \\ M'_k &\subseteq Class \times Method \times visibility \\ & \quad \times cyclomatic_complexity \times number_statements \times method_kind. \end{aligned}$$

Mit Hilfe dieser Operationen lassen sich nun gezielt Informationen aus dem Abhängigkeitsgraphen extrahieren. Als Ergebnis erhält man stets eine Relation, die ausgewertet werden kann um einen Metrikwert zu berechnen.

6.4.2. Auswertung von Relationen

Die zur Berechnung von Metriken extrahierten Relationen müssen ausgewertet werden um konkrete Metrikwerte zu erhalten. Im einfachsten Fall ist die Kardinalität einer Relation bereits der Metrikwert. In anderen Fällen werden z. B. Graphalgorithmen verwendet.

Im Abschnitt 6.3.3 wurde zwischen Entitäten und Beziehungen unterschieden. Die Entitäten sind dabei Relationen, die durch das kartesische Produkt der Attribute gebildet werden und somit Mengen von Entitäten darstellen. Die Beziehungen sind Relationen, die durch das kartesische Produkt $from \times to$ dargestellt werden und Baumstrukturen bzw. bipartite Graphen repräsentieren.

Die Relation einer Entität oder den Verbund mehrerer Entitäten kann man sich als Tabelle vorstellen. Jedes Attribut der Relation repräsentiert eine Tabellenspalte, jede Zeile der Tabelle steht für ein Element der Relation. Für die Relation R seien folgende Auswertungsfunktionen definiert:

- *Kardinalität:* die Kardinalität $|R|$ bestimmt die Anzahl der Elemente der Relation, bzw. die Länge der Entitätentabelle.
- *Summe:* beschränkt man die Relation auf ein einzelnes Attribut, bzw. die Tabelle auf eine einzelne Spalte, so lässt sich bei numerischen Werten die Gesamtsumme berechnen. Dies wird hier als $sum(R)$ notiert.
- *Mittelwerte:* analog zur Berechnung einer Summe sind verschiedene Mittelwerte wie z. B. das arithmetische Mittel, notiert als $mean(R)$ oder der Median, notiert als $median(R)$, nützlich.
- *Extremwerte:* ebenfalls hilfreich sind Extremwerte wie das Maximum $max(R)$ oder Minimum $min(R)$ einer numerischen Folge.
- *Sonstiges:* grundsätzlich steht für die Auswertung solcher Entitätsrelationen das gesamte Spektrum statistischer Auswertungsverfahren zur Verfügung. Die hier dargestellten Funktionen sind daher nur Beispiele.

  Wenn die betrachteten Werte nicht numerisch sind, sondern aus einem diskreten nominalen Wertebereich stammen, dann bietet es sich an, z. B. die Anzahl konkreter einzelner Elemente des Wertebereichs oder deren Anteil am Gesamtumfang zu berechnen.

Die Relation einer Beziehung besteht in dem hier beschriebenen Schema aus den Attributen *from* und *to*, die jeweils eindeutige Namen der in Beziehung stehenden Entitäten enthalten. Diese Struktur beschreibt entweder Bäume oder Graphen, sodass spezialisierte Funktionen deren Eigenschaften ermitteln können.

Betrachtet man Baumstrukturen, so sind z. B. folgende Eigenschaften interessant: Tiefe $depth(R)$ und Breite $width(R)$ des Baumes, Abstand eines bestimmten Knotens zur Wurzel $level(id, R)$ und weitere.

Bei bipartiten Graphen sind für diesen Zweck vor allem deren Zusammenhangskomponenten $cc(R)$ bzw. die Anzahl der Zusammenhangskomponenten $ncc(R)$ zu bestimmen. Vergleiche hierzu auch Abschnitt 5.3.

6.4.3. Beispielberechnungen

Die folgenden Beispiele zeigen, wie konkrete relationale Anfragen genutzt werden, um Teilaspekte der bekannten Analyseergebnisse zu extrahieren und zur Metrikberechnung zu nutzen.

Anzahl der Methoden einer Klasse
Um die Anzahl der Methoden einer Klasse k zu bestimmen, werden die Methoden selektiert, die mit der Klasse k in partof-Beziehung stehen:

$$n = |\sigma_{id=k}(\mathsf{Class}) \ltimes_{\mathsf{partof}} \mathsf{Method}|$$

Anzahl der Methoden eines Paketes
In diesem Fall kann der Kontext leicht anhand der partof-Beziehung auf das Paket p ausgedehnt werden. Es wird die Menge von Methoden selektiert, die zu Klassen gehören, die innerhalb des Paketes p definiert sind:

$$n = |\sigma_{id=p}(\mathsf{Package}) \ltimes_{\mathsf{partof}} \mathsf{Class} \ltimes_{\mathsf{partof}} \mathsf{Method}|$$

Mittlere McCabe-Komplexität einer Klasse
Wurden, wie im Beispiel der Methoden, zusätzliche Eigenschaften der Programmobjekte berechnet, so sind diese im relationalen Schema als Attribut des Programmobjektes zu finden.

Hier werden alle Methoden einer Klasse k ausgewählt. Die Ergebnisrelation wird auf das Attribut cyclomatic_complexity eingeschränkt, sodass das arithmetische Mittel der Komplexitäten aller betrachteten Methoden akkumuliert wird:

$$n = \mathsf{mean}(\Pi_{\mathsf{cyclomatic_complexity}}(\sigma_{id=k}(\mathsf{Class}) \ltimes_{\mathsf{partof}} \mathsf{Method}))$$

Kenndaten der Vererbungshierarchie
In den bisherigen Beispielen wurden Mengen von Programmobjekten nach bestimmten Kriterien ausgewählt. Ebenso lassen sich Teilgraphen des Programmabhängigkeitsgraphen extrahieren.

Um die im Graphen abgebildete Vererbungshierarchie z. B. innerhalb eines Paketes p zu bestimmen wird die Menge der Klassen eines Paketes C_p benötigt:

$$C_p = \sigma_{id=p}(\mathsf{Package}) \ltimes_{\mathsf{partof}} \mathsf{Class}$$

Diese Klassen werden verwendet, um die inheritance-Relation durch einen Verbund auf die Elemente einzuschränken, die sowohl auf der *from*-Seite, als auch auf der *to*-Seite mit einer Klasse innerhalb des Paketes verbunden sind.

$$\mathsf{inheritance}_p = \Pi_{from \leftarrow Class}(C_p) \bowtie \mathsf{inheritance} \bowtie \Pi_{to \leftarrow Class}(C_p)$$

Kenndaten, wie z. B. die Breite der (eingeschränkten) Vererbungshierarchie können dann leicht bestimmt werden:

$$n = width(\mathsf{inheritance}_p).$$

Feldzugriffe innerhalb einer Klasse

Im Folgenden wird der Zusammenhang zwischen den Methoden und den Feldern einer Klasse gemessen. Hierzu wird ein Teilgraph aus dem Programmabhängigkeitsgraphen extrahiert. Die Knoten des Teilgraphen sind alle Methoden und Felder der Klasse k. Die Kanten sind entweder lesender oder schreibender Feldzugriff.

Zunächst werden die Mengen der Methoden M_k und Felder F_k der Klasse k bestimmt:

$$\begin{aligned} M_k &= \Pi_{\mathsf{from} \leftarrow \mathsf{Method}}(\sigma_{id=k}(\mathsf{Class}) \bowtie_{\mathsf{partof}} \mathsf{Method}) \\ F_k &= \Pi_{\mathsf{to} \leftarrow \mathsf{Field}}(\sigma_{id=k}(\mathsf{Class}) \bowtie_{\mathsf{partof}} \mathsf{Field}) \end{aligned}$$

Für den Verbund mit den readaccess- und writeaccess-Beziehungen wurden die Attribute mit den Methoden- bzw. Feldbezeichnern bereits in *from* bzw. *to* umbenannt.

Jetzt ist nur noch der Teil der Feldzugriffs-Relationen zu extrahieren, der zwischen den gewünschten Methoden und Feldern besteht:

$$\begin{aligned} Read &= M_k \bowtie \mathsf{readaccess} \bowtie F_k \\ Write &= M_k \bowtie \mathsf{writeaccess} \bowtie F_k \\ RW &= Read \cup Write \end{aligned}$$

Die Anzahl der Zusammenhangskomponenten n ergibt sich dann aus der Auswertung der Ergebnisrelation:

$$n = ncc(RW).$$

6.5. Verwandte Ansätze

Im Bereich des *Reengineering* [63] [49] werden Modelle eingesetzt, die es erlauben, vom konkreten Programm bzw. verschiedenen Programmiersprachen zu abstrahieren. Ausprägungen des Modells spiegeln Eigenschaften des Programms wieder. Auf der Basis dieser Modellinstanzen lassen sich dann höherwertige Eigenschaften, z. B. auch OO-Metriken, ableiten. Diese Modellinstanziierung wird als *Fact Extraction* bezeichnet.

Zu den prominenten Werkzeugen gehört das *Rigi*-System [82]. Programmabhängigkeitsgraphen werden an anderer Stelle vor allem zum Zwecke des *Slicing* eingesetzt. Hier sind auch Ansätze für Java [41] [89] bekannt. Ursprünglich wurden Programmabhängigkeitsgraphen 1984 von Ottenstein und Ottenstein [67] eingeführt.

Werkzeuge wie *„jCosmo“* [29], *„CodeCrawler“* [52] und *„Crocodile/CrocoCosmos“* [79] visualisieren quantitative und strukturelle Eigenschaften eines analysierten Programms und bieten dem Suchenden unterschiedliche Sichtweisen zur Analyse großer Programme an.

In [8] wird ein Ansatz beschrieben, Anfragen an Abhängigkeitsgraphen mit Mitteln der Prädikatenlogik zu stellen.

6.6. Zusammenfassung

Mit Hilfe eines Programmabhängigkeitsgraphen, der durch einfache statische Programmanalysen konstruiert wird, lassen sich Entwurfsmetriken berechnen. Als allgemeiner Ansatz wurde hierzu die relationale Algebra verwendet um Anfragen an den Graphen zu stellen und deren Ergebnisse auszuwerten. Dies erlaubt, sehr schnell weitere Metrikberechnungen zu ergänzen und ähnliche Metriken auf der Grundlage bestehender Metriken weiter zu entwickeln.

Dieser Ansatz ist besonders für ein interaktives Werkzeug geeignet. Der Benutzer soll nicht auf die Analyse warten müssen, sondern in kurzer Zeit bedient werden. Der Analysekontext bestimmt die Größe des Abhängigkeitsgraphen und gleichzeitig die Präzision der Analyse. Denn die Analyse ist ausgehend vom Analysekontext nach außen gerichtet. D. h. Beziehungen von Programmobjekten innerhalb des Kontextes zu Programmobjekten außerhalb des Kontextes werden sichtbar, Beziehungen in der Gegenrichtung gehen verloren.

Setzt man voraus, dass das Programm nicht verändert wurde und genügend Speicherplatz zur Verfügung steht, dann kann der Abhängigkeitsgraph über mehrere Iterationen der Analyse erhalten bleiben und ergänzt werden.

7. Adaptive Erkennung von Entwurfsmängeln

Inhalt

7.1. Überblick **86**
7.2. Adaptive Erkennung **86**
7.2.1. Einsatz eines Lernverfahrens 90
7.2.2. Erklärungskomponente 92
7.2.3. Modellreflexion 92
7.3. Zusammenfassung **93**

7.1. Überblick

Große und komplexe Software-Systeme haben gemeinsam, dass sie nicht von einzelnen oder wenigen Software-Ingenieuren beherrscht werden können. Für die Arbeit an einem solchen System ist ein *Team* nötig, das in der Lage ist, geeignete Abstraktionsebenen und Kommunikationsstrukturen zu etablieren. In Kapitel 2 wurden hierzu Prinzipien und Konzepte des objektorientierten Entwurfs beschrieben. Das *Teile und Herrsche*-Prinzip findet sich auf allen Ebenen wieder und reicht von Themen der Kommunikation bis hin zu konkreten Software-Strukturen.

Die Entwurfsmängel wurden in Kapitel 3 als ein didaktisches Gerüst zur Vermittlung objektorientierter Entwurfskompetenz beschrieben. Bildhafte Sprache und plakative Beschreibungsform sind Zutaten, mit denen der Programmierer und der Architekt lernt, einen problematischen Entwurf zu erkennen.

Die maschinellen Lernverfahren, speziell Entscheidungsbaumverfahren (Kapitel 4), erlauben anhand von Beispielen Regeln abzuleiten. Ich verwende diese Verfahren hier, um zum Einen ein gewisses Maß der fehlenden menschlichen Intuition nachzubilden, zum Anderen erlauben diese Verfahren, quasi automatisch Regeln und damit Wissen zu generieren.

In Kapitel 5 habe ich Kriterien zusammengetragen, die es erlauben, typische Strukturen in objektorientierter Software zu identifizieren, die mit Entwurfsmängeln in Zusammenhang gebracht werden können. Diese Strukturen lassen sich durch Metriken ausdrücken. Metriken erlauben, Programmstellen oder -teile zu charakterisieren, ohne das konkrete Programm im Detail zu verstehen.

Die Verfahren zur Berechnung der Metriken auf der Basis von statischen Programmanalysen habe ich in Kapitel 6 vorgestellt. Das Modell der Entwurfsmängel in der Form von Metriken ist keine allgemeingültige oder allgemein anerkannte Sichtweise auf Entwurfsmängel. Die flexible Form der Metrikberechnung auf der Grundlage von Programmabhängigkeitsgraphen erlaubt es, in kurzer Zeit weitere spezielle Metriken zu implementieren und somit für die Erkennung von Entwurfsmängeln nutzbar zu machen.

In diesem Kapitel wird ein Konzept zur adaptiven Erkennung von Entwurfsmängeln beschrieben. Hierzu werden die beschriebenen Komponenten zu einem Gesamtsystem verknüpft. Der Einsatz von Lernverfahren wird gesondert beleuchtet.

7.2. Adaptive Erkennung

Um Entwurfsmängel automatisch zu erkennen, kombiniere ich objektorientierte Metriken mit Verfahren des maschinellen Lernens. Gestützt durch Entwurfsmangelmodelle bilden Metriken eine geeignete Grundlage, um die qualitativen Eigen-

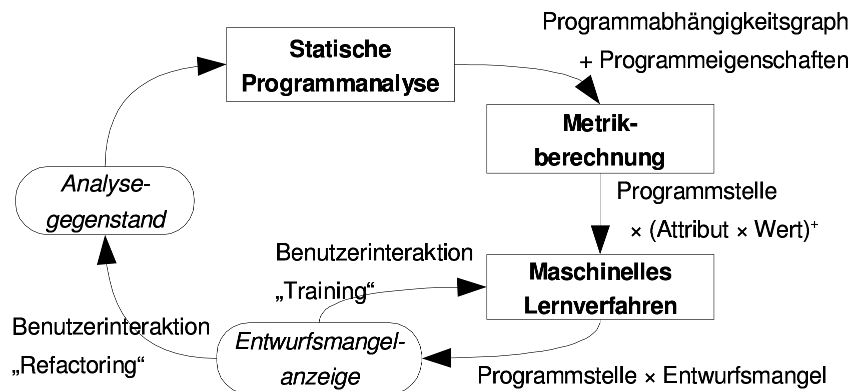

Abbildung 7.1.: Grundkonzept zur adaptiven Erkennung von Entwurfsmängeln.

schaften des Entwurfs eines Software-Systems zu beurteilen. Maschinelle Lernverfahren werden eingesetzt, um die menschliche Intuition zu unterstützen, aber nicht zu ersetzen.

Die Abbildung 7.1 zeigt das Grundkonzept zur adaptiven Erkennung von Entwurfsmängeln.

Ausgehend vom Analysegegenstand — dem Programm — werden anhand von statischen Programmanalysen objektorientierte Metriken berechnet. Die Messwerte einzelner Programmstellen werden entsprechend des Modells an ein maschinelles Lernverfahren weitergeleitet. Dieses wird mit Beispielen von vorhandenen Entwurfsmängeln trainiert und ist nach einiger Zeit in der Lage, anhand der Messwerte Entwurfsmängel in Programmstellen zu erkennen.

Der Benutzer hat dabei die Aufgabe, vorgeschlagene Programmstellen zu begutachten und das Ergebnis dem Verfahren zurück zu liefern. Wenn der Benutzer eine Programmstelle als mit einem Entwurfsmangel behaftet beurteilt, dann kann diese z. B. durch *Refactoring*-Transformationen umstrukturiert und somit ggf. verbessert werden.

Im Folgenden wird die Verknüpfung der einzelnen Komponenten im Detail beschrieben.

Analysegegenstand

Der Analysegegenstand ist das Programm oder ein Teil des Programms, dessen Struktur verbessert und in dem nach Entwurfsmängeln gesucht werden soll. Hierzu werden einzelne Programmstellen des Programms betrachtet. Zusätzlich zum Namen einer Programmstelle ist deren Art A bekannt. Dieser Wertebereich (siehe auch Abb. 6.1, S. 76) sei definiert als

$$A = \{\mathsf{Methode}, \mathsf{Feld}, \mathsf{Klasse}, \mathsf{Interface}, \mathsf{Paket}, \mathsf{Bibliothek}\}.$$

Die Notation einer Programmstelle wird hier nicht näher definiert. In der Sprache Java z. B. kann man sich den vollqualifizierten Namen einer Methode oder einer Klasse vorstellen. Die Menge aller Programmstellen P sei damit definiert als Tupel aus Art und Bezeichner einer Programmstelle:

$$P \subset A \times B,$$

wobei B die Menge aller Bezeichner eines gegebenen Programms bedeute.

Statische Programmanalyse und Metrikberechnung

Die fraglichen Programmstellen werden mit statischen Analysemethoden untersucht. Das Ergebnis der Analyse sind ein Programmabhängigkeitsgraph und weitere Ergebnisse spezieller Analysen. Diese ermöglichen, wie in Kapitel 6 beschrieben, vielfältige Entwurfsmetriken zu berechnen. Jeder Metrik wird eine Programmstellenart und ein Typ zugeordnet und durch einen Namen identifiziert. Die Menge M der Metriken sei definiert als:

$$M \subset A \times T \times \mathsf{MB},$$

wobei MB die Menge der Metrikbezeichner ist und der Typ T den Wertebereich der Metrik charakterisiert.

Hier wird zwischen nominalen und ordinalen Metriken unterschieden. Wobei der genaue Wertebereich bei nominalen Metriken implizit der Metrikdefinition zu entnehmen ist. Es werden daher nur zwei Typen T unterschieden[1]:

$$T = \{\mathsf{nominal}, \mathsf{ordinal}\}.$$

Häufig wird die Teilmenge aller Metriken verwendet, in der alle Metriken von einer bestimmten Art sind. Im Folgenden bezeichne M_a die Teilmenge aller Metriken, für welche die Art der Programmstelle gleich $a \in A$ ist:

$$M_a = \{(a', t', m') \in M \mid a' = a\}.$$

Modellierung von Entwurfsmängeln

Für die Suche muss eine Menge von Entwurfsmängeln E festgelegt werden, sodass ein Entwurfsmangel durch eine Teilmenge der zuvor festgelegten Metriken M modelliert wird. Zu jeder Metrik $m \in M$ ist wiederum die Art der Programmstelle bekannt, auf die sich diese bezieht. Damit ist:

$$E_a \subset \wp(M_a).$$

[1] Vergleicht man mit Typkonzepten aus Programmiersprachen, so ist der ordinale Typ ein Grundtyp, z. B. ein ganzzahliger Typ. Der nominale Typ ist ein komplexer Aufzählungstyp, wobei konkrete Aufzählungen implizit durch Metrikdefinitionen beschrieben werden.

Es besteht also jeder Entwurfsmangel in E_a nur aus Metriken M_a, die von der gleichen Art $a \in A$ wie der Entwurfsmangel selbst sind.

Programmvermessung

Die statische Programmanalyse berechnet zu den gewünschten Programmstellen i. d. R. alle Metriken von der Art der jeweiligen Programmstelle. Als Ergebnis der Programmanalyse liegt also eine Folge von Messwerten zu jeder Programmstelle vor. Diese wird als Programmvermessung PV bezeichnet und beschreibt eine Menge von Programmstellen, denen jeweils eine Folge von Metriken M und deren Werte W zugeordnet sind.

Ein Wert W wird als Tupel des Typs $t \in T$ und eines konkreten Wertes $\omega \in \Omega$ dargestellt. Die Menge Ω_t beschreibe den Wertebereich des Typs $t \in T$:

$$W = \{(t, \omega_t) \mid \omega_t \in \Omega_t\}.$$

Für einen einzelnen Messwert gilt, dass der Typ des Wertes $w \in W$ gleich dem Typ t der Metrik $m \in M$ ist und die Art a der Metrik $m \in M$ gleich der Art der Programmstelle $p \in P$ ist. Somit gelte zunächst:

$$\begin{aligned} P_a &= \{(a', b) \in P \mid a = a'\}, \\ M_a^t &= \{(a', t', \mathsf{mb}) \in M \mid a = a' \wedge t = t'\} \text{ und} \\ W^t &= \{(t', \omega) \in W \mid t = t'\}. \end{aligned}$$

Die Menge PV ist dann die Menge aller Programmvermessungen:

$$\begin{aligned} \mathsf{PV}_a &\subset P_a \times (M_a^t \times W^t) \\ \mathsf{PV} &= \bigcup_{a \in A} \mathsf{PV}_a. \end{aligned}$$

Instanzbildung

Aus den Entwurfsmangelmodellen und der Programmvermessung werden Instanzen für das Lernverfahren gebildet. Für jeden Entwurfsmangel wird jeweils eine eigene Trainingsmenge und damit ein eigener Entscheidungsbaum konstruiert.

Ein konkretes Entwurfsmangelmodell $e \in E$ ist eine Menge von Metriken:

$$e \subset M_a.$$

Eine aktuelle Programmvermessung besteht aus einer Menge von Programmstellen und jeweils einer zugeordneten Menge von Metriken und ihrer Messwerte. Eine Menge von Trainingsinstanzen I_e für einen Entwurfsmangel e ist eine Abbildung, deren Ergebnis bei einer gegebenen Programmvermessung nur von den

Metriken und Metrikwerten abhängt. Im Sinne des Lernverfahrens heißen die Metriken dann Attribute. Eine Instanzmenge I_e eines Entwurfsmangels e ist dann:

$$I_e = \mathsf{PV} \rightarrow W_z.$$

Ein zusätzliches Attribut, das Zielattribut $Z = \{\mathsf{Ziel}\}$ mit z. B. dem Wertebereich $W_z = \{\mathsf{wahr}, \mathsf{falsch}\}$ oder auch $W_z = \{\mathsf{ja}, \mathsf{vielleicht}, \mathsf{nein}\}$, dient dem Lernverfahren zur Anzeige und Aufnahme der Klassifizierung als Entwurfsmangel. Die Kardinalität der Instanzmenge ist gleich der Kardinalität der Programmvermessung.

Lernverfahren
Ein Lernverfahren benötigt zum Erlernen und Klassifizieren von Instanzen keine zugehörige Programmstelle. D. h. dem Lernverfahren werden Trainingsinstanzen zur Bearbeitung gegeben, die nur die Attribute enthalten.

Wenn einzelne Programmstellen klassifiziert werden, dann ordnet man dem Ergebnis die gleiche Programmstelle zu, die auch der Eingabeinstanz zugeordnet war. Man erhält somit als Ergebnis des Klassifizierers alle Informationen zur Entwurfsmangelanzeige EA:

$$\mathrm{EA} \subset P \times E \times W^z,$$

nämlich die Programmstelle, den fraglichen Entwurfsmangel und eine Zuordnung, ob der Entwurfsmangel vorliegt.

Der Einsatz des Lernverfahrens und die sich daraus ergebenden Konsequenzen werden im nun folgenden Abschnitt skizziert.

7.2.1. Einsatz eines Lernverfahrens

Lernverfahren bieten die Chance der „leichtgewichtigen“ Benutzerinteraktion. D. h. der Benutzer wird lediglich regelmäßig nach seiner Meinung befragt. Die Arbeit, daraus Schlüsse zu ziehen, wird dem Benutzer aber weitestgehend abgenommen.

Die Abbildung 7.2 zeigt den aus zwei Phasen bestehenden Einsatz des Lernverfahrens. In der Trainingsphase werden Programmstellen ausgewählt, für die einzeln manuell entschieden wird, ob der betrachtete Entwurfsmangel vorliegt oder nicht. Die Messwerte dieser Programmstellen bilden — zusammen mit der Entwurfsmangelentscheidung — die Trainingsmenge, aus der sich ein initialer Entscheidungsbaum konstruieren lässt.

In der Erkennungsphase gibt der Anwender vor, welche Systemteile analysiert werden sollen. Hier werden alle bisher unbekannten Programmstellen vermessen und alle Entscheidungsbäume der betrachteten Entwurfsmängel darauf ange-

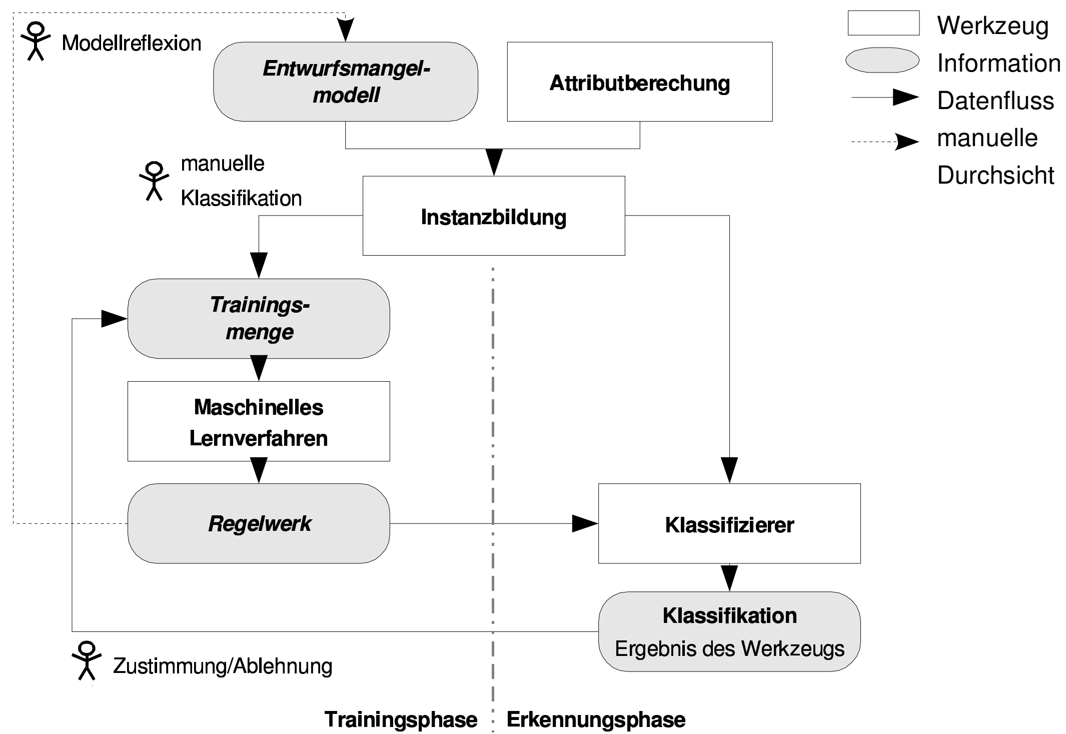

Abbildung 7.2.: Lern- und Erkennungsphase mit Adaption des Entscheidungsbaumes und des Entwurfsmangelmodells.

wandt. Jede Programmstelle, die als mangelbehaftet klassifiziert wurde, wird dem Anwender genannt.

Der Anwender prüft nun einzelne Fälle und akzeptiert den gefundenen Entwurfsmangel oder er lehnt ihn ab. In beiden Fällen kann die Programmstelle als weiteres Beispiel der Trainingsmenge hinzugefügt werden. Ignoriert der Anwender den Fund, bleibt die Trainingsmenge unverändert.

Trainings- und Erkennungsphase sind voneinander unabhängig. Auch während der Erkennung kann der Anwender weiterhin auffällige Programmstellen in die Trainingsmenge aufnehmen.

7.2.2. Erklärungskomponente

Dem Anwender fällt es u. U. schwer, zu erkennen, warum eine Programmstelle als mangelbehaftet erkannt wurde.

Zur Erklärung kann dem Anwender der Entscheidungsbaum präsentiert und zusammen mit den Messwerten zur aktuell betrachteten Programmstelle der Pfad durch den Entscheidungsbaum visualisiert werden. Der Anwender kann so die Entscheidung nachvollziehen.

Hierzu kann er zunächst die eigene Vorstellung mit dem Entscheidungsbaum abgleichen. Unter Umständen gibt es Attribute, die der Anwender für sehr aussagekräftig hält, die aber nicht, oder nur weit entfernt von der Wurzel des Baumes auftreten. Attribute, die ein hohes Gewicht an der Entscheidung haben, sind per Konstruktion in der Wurzel und in deren Nähe untergebracht.

Häufig führt eine Kante vom Wurzelknoten oder wurzelnahen Knoten direkt auf einen Blattknoten. Damit ist das Attribut eines solchen Knotens als K.O.-Kriterium zu verstehen. Dies verdeutlicht dem Anwender zusätzlich die Bedeutung des Attributes für die Entscheidung.

Der Anwender könnte feststellen, dass die bisherigen Trainingsbeispiele noch nicht das geplante Spektrum des Entwurfsmangels abdecken. Oder aber die Analyse des Anwenders ergibt, dass das eingesetzte Modell den Entwurfsmangel nicht angemessen modelliert.

7.2.3. Modellreflexion

Lernverfahren, wie das hier verwendete *C4.5*, erreichen bei ausreichend großer Trainingsmenge eine sehr gute Anpassung. Durch standardisierte Validierungsverfahren erhält der Anwender einen Eindruck, wie sicher das Lernverfahren klassifiziert.

Für die Anwendung zur Erkennung von Entwurfsmängeln ist die Leistung des Verfahrens allerdings an der Leistung der menschlichen Intuition zu messen. D. h. ein gut trainiertes System sollte im Vergleich zu einer menschlichen Untersuchung

nur wenige Abweichungen anzeigen. Gelingt das nicht, entspricht das Modell des Entwurfsmangels vermutlich nicht der Intuition des Anwenders. Eine Anpassung des Modells ist nötig.

Da zu jedem Trainingsbeispiel die Programmstelle bekannt ist, kann auch noch nachträglich, ohne Verlust der Trainingsmenge, das Modell angepasst werden. Unter der Voraussetzung, dass das Programm nicht verändert wurde, können neue Messwerte zu den Programmstellen berechnet werden. Zusammen mit den bereits bekannten Klassifizierungen ergibt sich die ursprüngliche Trainingsmenge für ein angepasstes Modell.

7.3. Zusammenfassung

Die Entwurfsmängel eines Programmes werden erkannt, indem Teile des Programms statisch analysiert und aus den Ergebnissen Metriken berechnet werden. Entsprechend der Modelle für Entwurfsmängel werden die Metrikwerte zusammen mit einem Klassifizierungsvorschlag einem maschinellen Lernverfahren übergeben, das die Beispiele analysiert und daraus für jeden Entwurfsmangel einen Entscheidungsbaum konstruiert.

Der Entscheidungsbaum wird verwendet um unbekannte Programmstellen anhand ihrer Metrikwerte zu charakterisieren. Desweiteren dient der Entscheidungsbaum (bzw. die zugrunde liegende Beispielmenge) als Wissensspeicher und wird als Erklärungskomponente und zur Modellreflexion genutzt.

8. Evaluation

Inhalt

8.1. Übersicht **96**
8.2. Kriterien der Evaluation **97**
8.2.1. Effektivität 97
8.2.2. Effizienz 99
8.3. Untersuchungsmethoden **100**
8.3.1. Voruntersuchung 100
8.3.2. Messung der Effektivität 106
8.3.3. Messung der Effizienz 127
8.4. Werkzeugunterstützung **129**
8.4.1. Das Werkzeug IYC 130
8.4.2. Architektur 130
8.4.3. Benutzung 131
8.5. Fallstudie **136**
8.5.1. Untersuchungsteilnehmer 136
8.5.2. Analysegegenstand 137
8.5.3. Ergebnisse der Voruntersuchung 137
8.5.4. Ergebnisse der Effektivitätsmessungen 137
8.6. Zusammenfassung **138**

8.1. Übersicht

Die Evaluation des vorgestellten Verfahrens soll zeigen, dass dieses grundsätzlich und auch praktisch geeignet ist, Entwurfsmängel zu entdecken. Hierzu werden Bewertungskriterien vorgestellt, die zu einer umfassenden Evaluation des Ansatzes geeignet sind.

Entwurfsmängel lassen sich nicht eindeutig fraglichen Programmstellen zuordnen. Das Urteil einzelner Software-Entwickler, -Architekten oder Test-Ingenieure fiele vermutlich stark verschieden aus. Eine Bewertung des Ansatzes ist jedoch nur durchführbar, wenn zu einer Menge von Programmstellen bereits bekannt ist, ob und welche Entwurfsmängel diese darstellen. Diese Information ist nur durch das Urteil von Testpersonen oder einer Expertengruppe zu erheben.

Ich setze daher Methoden der empirischen Sozialforschung ein, um trotz, aber auch mit Hilfe des Faktors „Mensch“ gesicherte Ergebnisse präsentieren zu können. Forschungsprozesse der Empirie folgen sehr engen strukturellen Vorgaben, an denen ich mich hier ebenfalls orientiere [10].

So ist zunächst das zu untersuchende Problem zu beschreiben. Dies führt zu einer zu prüfenden Forschungshypothese. Ein Untersuchungsplan legt fest, mit welchen Methoden Daten erhoben, aufbereitet und ausgewertet werden sollen. Dies wird Operationalisierung der Forschungshypothese genannt[1]. Methoden zur Datenerhebung sind z. B. einfache Zählungen, aber auch Tests, Befragungen, Beobachtungen und physiologische Messungen. Ich werde hier vorwiegend die Methode des Urteilens verwenden. Dabei wird eine Testperson gebeten, zu einem Sachverhalt eine persönliche Einschätzung abzugeben. Die methodische Aufarbeitung des Datenmaterials erlaubt, die Forschungshypothese zu beurteilen. Ggf. erfolgt eine Prüfung durch mathematische Signifikanztests, die erlauben, zufällige Tendenzen im Datenmaterial, die zu falschen Schlüssen führen würden, von signifikanten Tendenzen zu unterscheiden. Damit lässt sich abschließend die Forschungshypothese bestätigen oder widerlegen.

Es gibt allerdings auch Evaluationskriterien, die nicht an ein menschliches Urteil zu knüpfen sind und unabhängig bewertet werden können. Im Forschungsbereich des *Information Retrieval* [76][36] findet man hierzu standardisierte Verfahren, die den unabhängigen Vergleich mehrerer Systeme erlauben.

Das *Information Retrieval* beschäftigt sich mit der Suche von Dokumenten in Datenbanken. Beispiele sind Suchsysteme für Literaturdatenbanken oder Suchmaschinen für Web-Seiten. Solche Systeme verarbeiten Nutzeranfragen und liefern relevante Dokumente als Antwort zurück. Die zur Klassifizierung der fraglichen Dokumente eingesetzten Modelle sind denen des maschinellen Lernens sehr ähnlich.

[1] Vergleiche auch *Ziel-Frage-Metrik*-Ansatz in Abschnitt 5.2.4 auf Seite 58.

Die Erkennung von Entwurfsmängeln kann ebenfalls als *Information Retrieval* angesehen werden. Die Datenbank ist der Analysegegenstand, also hier die Implementierung eines Software-Systems; die enthaltenen Dokumente sind Programmstellen; und die Suchanfrage wird hier nicht z. B. als Folge von Suchworten formuliert, sondern basiert auf Modellierung und Training einzelner Entwurfsmängel.

Im folgenden Abschnitt 8.2 werden zunächst die Kriterien und empirischen Forschungsfragen dieser Evaluation vorgestellt. Die benötigten Methoden, um relevante Daten zu diesen Fragen zu erheben, werden in Abschnitt 8.3 detailliert beschrieben. Als Messinstrument dient die prototypische Implementierung eines Werkzeugs zur Entwurfsmangelerkennung, das in Abschnitt 8.4 skizziert wird. Vollständige empirische Untersuchungen benötigen neben geeigneten Testpersonen vor allem viel Zeit, die im Rahmen dieser Arbeit nicht zur Verfügung stand. Es wurde daher eine Fallstudie durchgeführt, deren Ergebnisse in Abschnitt 8.5 beschrieben sind. Dieses Kapitel schließt mit einer zusammenfassenden Bewertung und einem Ausblick im Abschnitt 8.6.

8.2. Kriterien der Evaluation

Die Evaluation des Ansatzes lässt sich zweiteilen. Zum Einen ist die Effektivität des Verfahrens zu bewerten, zum Anderen spielt die Effizienz eine wichtige Rolle.

Die Effektivität wird benutzerorientiert definiert und beschreibt die Erkennungsleistung des Verfahrens hinsichtlich verschiedener Kriterien. So ist z. B. die Balance zwischen Präzision und Vollständigkeit einer Suche und die Anpassungsleistung zu bewerten.

Im Rahmen der Effizienz werden Kriterien untersucht, welche die Benutzbarkeit des Verfahrens beleuchten. Im wesentlichen werden der Ressourcenverbrauch (Zeit, Speicher) und Kosten untersucht.

8.2.1. Effektivität

Der Nutzen eines Systems zur Erkennung von Entwurfsmängeln wird vor allem in der *Erkennungsleistung* gemessen. D. h. bei der Klassifizierung von Programmstellen sollen keine Fehler auftreten.

Die Erkennungsleistung beschreibt, wie erfolgreich Programmstellen klassifiziert werden. Hierzu werden Kennwerte wie z. B. Präzision und Vollständigkeit ermittelt. Wenn das Verfahren besonders präzise arbeitet, dann weist es keine oder nur wenige Programmstellen aus, die keinen Entwurfsmangel darstellen. Ein vollständiges Ergebnis bedeutet, dass keine Entwurfsmängel unentdeckt bleiben. Idealerweise wünscht man sich sowohl eine hohe Präzision als auch eine hohe Vollständigkeit.

Kenngrößen wie Präzision oder Vollständigkeit können nur dann erhoben werden, wenn die vorhandenen Entwurfsmängel vorher bekannt sind. Hier ist man auf das Urteil einer Testperson oder eines Expertengremiums angewiesen, das die „korrekten" Klassifizierungsergebnisse vorgibt. Diese werden im Rahmen einer Voruntersuchung ermittelt (siehe Abschnitt 8.3.1, Seite 100).

Die Erkennungsleistung des Verfahrens hängt von der aktuellen Konfiguration, also den Modellen für Entwurfsmängel und deren Trainingsmengen ab. Es ergeben sich daher differenzierte Fragen zur Bewertung der Erkennungsleistung:

Reproduktionsleistung Eine Menge von Programmstellen wird zusammen mit der Klassifizierung durch eine Testperson als Trainingsmenge verwendet. Lässt man anschließend das Verfahren in der gleichen Menge von Programmstellen suchen, so beobachtet man idealerweise, wie genau die gelernten Programmstellen wiedergefunden werden. Gemessen wird hier die Leistung des Lernverfahrens das Gelernte wiederzugeben.

Transferleistung Wenn man erfahren möchte, wie gut das Lernverfahren das Konzept eines Entwurfsmangels erlernt hat, dann verwendet man nur einen Teil der Trainingsmenge zum Lernen und beobachtet, ob ungelernte Programmstellen ebenfalls korrekt klassifiziert werden.

Beobachtet man eine hohe Erkennungsleistung, so lässt sich daraus schließen, dass sowohl Modell als auch das Spektrum der Trainingsmenge den Entwurfsmangel treffend charakterisieren. Bei niedriger Erkennungsleistung bleibt unklar, ob das Spektrum der Trainingsmenge nicht ausreicht, also zu wenig Trainingsbeispiele verwendet wurden, oder ob das Modell unpassend ist.

Trainingsleistung Das Entwurfsmangelmodell ist erst dann anzupassen, wenn ausgeschlossen werden kann, dass die Transferleistung nicht durch Erweiterung des Trainingsspektrums verbessert werden kann. Hierzu wird eine Effektanalyse eingesetzt, um die Vollständigkeit der Trainingsmenge zu bestimmen.

Anpassungsleistung Mit steigendem Umfang der Trainingsmenge sinkt der Einfluss neu hinzugefügter Trainingsbeispiele auf den konstruierten Entscheidungsbaum. Man erwartet zwar eine weiterhin hohe Reproduktionsleistung, aber eine sinkende Transferleistung. Diesem Phänomen versucht man mit gewichteten Trainingszugängen entgegen zu wirken. Damit könnte der Benutzer erzwingen, dass eine bestimmte neu aufzunehmende Programmstelle einen Effekt zeigt. So dass sich wenigstens der Entscheidungsbaum ändert oder besser, dass erzwungen wird, dass das neue Trainingsbeispiel

korrekt klassifiziert wird und somit nicht unter die (kleine) Menge falsch klassifizierter Trainingsinstanzen fällt.

Im Abschnitt 8.3 werden Verfahren vorgestellt, mit denen die verschiedenen Leistungsmaße bestimmt werden. Aus der differenzierten Leistungsbewertung ergeben sich direkt folgende Fragestellungen:

Wie werden Reproduktions- und Transferleistung beeinflusst? Es wird vermutet, dass bei hoher Transferleistung die Reproduktionsleistung ebenfalls sehr hoch ist. Denn eine Reproduktionsleistung lässt sich einfacher erreichen, da das geplante Spektrum zu erkennender Entwurfsmängel hierfür nicht abgedeckt werden muss. Es reicht aus, einen kleinen Ausschnitt der Wirklichkeit getreu wiederzugeben.

Es ist zu erwarten, dass die Reproduktionsleistung unabhängig vom Umfang der Trainingsmenge ist. Die Transferleistung jedoch sollte sich mit steigender Trainingsleistung verbessern.

Wie korrelieren Trainings- und Anpassungsleistung? Eine hohe Trainingsleistung sagt nichts über den Umfang der Trainingsmenge aus. Wenige Trainingsinstanzen, die gut über das Spektrum des Entwurfsmangels verteilt sind, erreichen eine ebenso gute Trainingsleistung wie viele Instanzen unter denen es Paare von gleichen oder leicht verschiedenen gibt.

Bei großem Umfang der Trainingsmenge wird eine geringere Anpassungsleistung erwartet, da dort eine einzelne Trainingsinstanz ggf. keine Änderung des Entscheidungsbaumes hervorruft.

In der empirischen Forschung wird die informelle Formulierung der Forschungsfragen umgesetzt in einen Plan zur Erhebung und Auswertung von Daten. Dies wird Operationalisierung der Forschungshypothese genannt und wird hier in der Beschreibung der Fallstudie in Abschnitt 8.5 fortgeführt.

8.2.2. Effizienz

Verfahren und Werkzeuge, deren korrekte Ergebnisse nicht mit vertretbarem Zeit- und Kostenaufwand erbracht werden können, sind in der Praxis schwer einsetzbar. Gemessen wird daher auch die Effizienz des Verfahrens verglichen mit der Effizienz, die eine manuelle Suche nach Entwurfsmängeln aufweist.

Effizienz automatischer Klassifizierung Der Zeitaufwand setzt sich zusammen aus der Zeit, die für Programmanalysen und für Konstruktion und Interpretation von Entscheidungsbäumen benötigt werden.

Um zu zeigen, dass das Verfahren in der Praxis einsetzbar ist, wird zusätzlich der Speicherbedarf erhoben. Dieser wird getrennt nach kurzzeitigem und langfristigem Speicherbedarf betrachtet.

Die Kosten werden hier nicht berücksichtigt.

Effizienz manueller Klassifizierung Die Leistung einer Testperson bemisst sich in der Zeit, die benötigt wird um eine Programmstelle hinsichtlich eines Entwurfsmangels zu beurteilen. Dabei ist die Korrektheit zweitrangig. Um allerdings zu verhindern, dass eine Testperson das Ergebnis rät, statt die jeweilige Programmstelle zu untersuchen, werden nur Ergebnisse von Testpersonen berücksichtigt, die eine bessere Leistung als die Zufallsbewertung liefern.

Die Korrektheit lässt sich wiederum nur mit dem Urteil anderer Testpersonen beurteilen. So dass hier wieder die Ergebnisse einer Voruntersuchung benötigt werden (siehe Abschnitt 8.3.1).

8.3. Untersuchungsmethoden

Im Folgenden werden Evaluationsmethoden beschrieben und Messwerte definiert. Diese werden eingesetzt um die Fragen aus dem vorherigen Abschnitt 8.2 zu beantworten.

Zunächst wird die Voruntersuchung beschrieben, deren Ziel es ist, eine gesicherte Grundmenge von Programmstellen mit zugeordneten Entwurfsmängeln zu erhalten. Danach werden Methoden zur Beurteilung der Erkennungsleistung beschrieben. Die Beschreibung einfacher Effizienzmaße beendet den Abschnitt.

8.3.1. Voruntersuchung

Alle Bestandteile der Evaluation sind auf Analysegegenstände angewiesen, in denen enthaltene Entwurfsmängel bekannt sind. Hierzu wird das Urteil von Testpersonen benötigt, die Analysegegenstände manuell bewerten.

Eine einzelne Testperson reicht hier nicht aus. Diese ließe nur die eigene Sichtweise einfließen. Die Bewertung soll daher von einem Expertengremium vorgenommen werden. Experten und Nicht-Experten werden anhand einer Wissensstandserhebung unterschieden. Als Ergebnis steht anschließend ein Referenzkatalog von Entwurfsmängeln zur Verfügung.

8.3.1.1. Wissensstandserhebung

Testpersonen, die Entwurfsmängel suchen sollen, entwickeln unterschiedliche Strategien für ihre Aufgabe. Dies liegt vor allem im Wissensstand des Einzelnen

begründet. Vereinfacht gesprochen lässt sich ein Spektrum vom Anfänger bis zum Experten in folgenden Kategorien beschreiben:

- *Entwurfsmangel.* Einer Person ist das Konzept Entwurfsmangel bereits bekannt und diese hat auch bereits häufiger erfolgreich Entwurfsmängel erkannt. Eine andere Person kennt Entwurfsmängel nicht oder hat nur am Rande davon gehört.

- *Analysegegenstand.* Man unterscheidet Personen, die das zu untersuchende Programm bereits kennen oder nicht kennen. Ist das Programm unbekannt, muss die Person dieses zunächst verstehen oder sich wenigstens einen Überblick über die wichtigsten Zusammenhänge innerhalb und außerhalb der fraglichen Programmstellen aneignen.

- *Anwendungsgebiet.* Die spezielle Kenntnis des Anwendungsgebietes, für welches das analysierte Programm entwickelt wird, beeinflusst die Sicht auf Entwurfsmängel. Aus der Erfahrung entstehen spezielle Implementierungsstrategien, die bereits erkannte Nachteile anderer Lösungen zurückdrängen. So verblasst die Erscheinung eines Entwurfsmangels mitunter gegenüber der wohl durchdachten spezifischen Lösung in einem Anwendungsgebiet.

- *Programmiererfahrung.* Langjährige Software-Entwickler entwickeln ein Gespür für „gute" Lösungen. Sie haben i. d. R. häufiger die Erfahrung machen müssen, das ihre Programme kommenden Anforderungsänderungen nicht gewachsen waren und angepasst werden mussten. Sie halten daher die Balance zwischen stark spezialisierten und generalisierten Lösungen. Diese Art Weitblick fehlt dem unerfahrenen Software-Entwickler.

- *Arbeitsweise.* Arbeitet eine Gruppe von Entwicklern am gleichen Programm, dann haben diese einen wesentlich höheren Kommunikationsbedarf als alleine arbeitende Entwickler, die z. B. den Auftrag haben, eine bestimmte Komponente anhand einer Spezifikation zu fertigen. Kommunikation führt zu schnellen Lernerfolgen, dadurch dass regelmäßig Fehler und Ideen diskutiert werden. Gruppen entwickeln somit auch ein weitaus festeres Grundverständnis von Entwurfsmängeln.

- *Rahmenbedingungen.* In der industriellen Software-Entwicklung spielt die reine Verbesserung von Software-Strukturen mitunter keine wichtige Rolle. Teure Entwicklerzeit kann nur für Fehlerbehebung und Funktionserweiterungen bereitgestellt und nicht für Investitionen in die (ungewisse) Zukunft geopfert werden. Zeitdruck ist demnach ein schlechter Begleiter für die Suche nach Entwurfsmängeln.

Anmeldung zur »It's Your Code« Online-Entwurfsmangel-Datenbank

Benutzerkennung: ______________________ **Passwort:** ________________________

Frage					
1. Kennen Sie das Konzept »Entwurfsmangel« oder »Bad Smell«?	Ja, häufig eingesetzt				Nein, nie davon gehört
2. Wieviele Entwurfsmängel haben Sie bereits gefunden und behoben?	> 40	> 20	> 10	einige wenige	bisher keine
3. Das Software-System, indem Sie Entwurfsmängel suchen...	wurde von mir (mit)entwickelt				ist mir völlig unbekannt
4. Wie lange entwickeln bzw. begutachten Sie schon Software?	> 10 Jahre	> 5 Jahre	> 3 Jahre	> 1 Jahr	> 0 Jahre
5. Arbeiten Sie in der Gruppe? Wie groß ist Ihre Gruppe?	> 30 Pers.	> 15 Pers.	> 5 Pers.	> 1 Pers.	ich arbeite alleine
6. Wieviel Ihrer Arbeitszeit verwenden Sie um Entwurfsmängel zu suchen und zu beheben?	> 80 %	> 40 %	> 10 %	> 0 %	0 %
7. In welcher/m Branche/Anwendungsgebiet arbeiten Sie?	Betriebswirtschaftl. Software	Wissenschaftl. Software	Multimedia Software	Betriebssystem Software	anderes: _________

Abbildung 8.1.: Fragebogen zur Ermittlung der relevanten Kenntnisse und Rahmenbedingungen eines Entwurfsmangelsuchenden.

Es soll herausgefunden werden, wie schnell und wie sicher Entwurfsmängel im Allgemeinen manuell erkannt werden können. Ein Fragebogen soll hierzu das nötige Bild von einer Testperson vermitteln. Abbildung 8.1 zeigt den Fragebogen, der sich aus den Überlegungen zum Wissensstand ergibt.

8.3.1.2. Konstruktion eines Referenzkatalogs

Ich bezeichne als Referenzkatalog eine Menge von Programmen, in denen Programmstellen als Entwurfsmangel markiert sind. Dieser Katalog repräsentiert eine allgemein anerkannte Sicht auf Entwurfsmängel und deren Ausprägungen. Der Referenzkatalog erlaubt, mehrere Erkennungsverfahren zu vergleichen, indem sie die Programme des Katalogs nach Entwurfsmängeln durchsuchen. Ein Vergleich mit dem Sollergebnis aus dem Katalog kann zur Leistungsbewertung des Verfahrens herangezogen werden. Ein lernendes Verfahren kann mit diesem Referenzka-

talog trainiert bzw. konfiguriert werden. Diese kann als Startkonfiguration für ein solches Werkzeug ausgeliefert werden.

Ein solcher Katalog kann nicht entstehen, indem eine Person oder Personengruppe Programmbeispiele sammelt und Entwurfsmängel nach eigener Sichtweise markiert. Solche einzelnen Urteile über Programmstellen sind nicht als Referenz geeignet. Es wird daher das *„Law of Categorial Judgement“* [86] eingesetzt, das Kategoriezuordnungen mehrerer Testpersonen oder mehrerer sukzessiver Erhebungen mittelt und der jeweils wahrscheinlichsten Kategorie zuordnet.

„Law of Categorial Judgement“

Man geht davon aus, dass menschliche Empfindungen bei der Beurteilung dargebotener Objekte nicht immer gleich sind, sondern um den „wahren“ Wert oszillieren. Diese Werte werden als normalverteilt angenommen. Das gilt auch, wenn das Objekt nicht mehrmals vom gleichen Betrachter, sondern von mehreren Betrachtern beurteilt wird.

Man lässt nun Objekte, also hier die Fragestellung, ob eine gegebene Programmstelle einen gegebenen Entwurfsmangel darstellt, einer Rangkategorie zuordnen. Die Rangkategorien könnten z. B. in fünf Kategorien aufgeteilt sein: (1.) der Entwurfsmangel liegt sicher vor, (2.) der Entwurfsmangel liegt vor, (3.) es ist unsicher, ob der Entwurfsmangel vorliegt, (4.) der Entwurfsmangel liegt eher nicht vor, (5.) der Entwurfsmangel liegt sicher nicht vor.

Abbildung 8.2 zeigt ein Beispiel, in dem 20 Experten gebeten wurden, vier Programmstellen PS bezüglich eines vorgegebenen Entwurfsmangels E zu beurteilen ($PS_i^{[E]}$). Die Experten geben jeweils eine der fünf Kategorien als ihr Urteil ab. Tabelle (a) zeigt, wie häufig eine Kategorie für eine der fünf Fragestellungen gewählt wurde. Bildet man die relativen Häufigkeiten, so erhält man Tabelle (b). Die kumulierte Tabelle (c) erhält man, indem man für jede Kategorie i die Summe der relativen Häufigkeiten der Kategorien $1 \ldots i$ bildet. Kategorie 5 erhält dann immer den Wert 1.

Im letzten Schritt werden die kumulierten relativen Häufigkeiten durch deren z-Werte der Standardnormalverteilung[2] ersetzt. Der z-Wert gibt den Abzissenwert der Normalverteilung an, der den angegebenen Flächenanteil abschneidet. Die Kategorie 5 wird nicht mehr gebraucht, da die relative Häufigkeit hier 1 ist (und damit $z \to \infty$). Die Spaltenmittelwerte der einzelnen Kategorien werden als Kategoriegrenzen zur jeweils nächsten Kategorie verwendet. Abbildung 8.3 zeigt die kumulierte Standardnormalverteilung mit Kategoriegrenzen als vertikal gestrichelte Linien. Die einzelnen Kategorien sind in runden Klammern notiert.

[2] Die verwendete Standardnormalverteilung kann eingeschränkt werden. Hier wurde der Bereich von 0.1 % – 99.9 % verwendet, sodass Häufigkeitswerte außerhalb des Bereichs auf die entsprechende Grenze gesetzt werden. Das verhindert, dass als z-Werte ∞ oder $-\infty$ auftreten und die Berechnungen der arithmetischen Mittel behindern.

(a) **Absolute Häufigkeiten**

Frage	Kat. 1	Kat. 2	Kat. 3	Kat. 4	Kat. 5
$PS_1^{[E]}$?	1	0	5	8	6
$PS_2^{[E]}$?	2	1	6	9	2
$PS_3^{[E]}$?	2	3	10	4	1
$PS_4^{[E]}$?	8	5	3	3	1

(b) **Relative Häufigkeiten**

Frage	Kat. 1	Kat. 2	Kat. 3	Kat. 4	Kat. 5
$PS_1^{[E]}$?	0.05	0.00	0.25	0.40	0.30
$PS_2^{[E]}$?	0.10	0.05	0.30	0.45	0.10
$PS_3^{[E]}$?	0.10	0.15	0.50	0.20	0.05
$PS_4^{[E]}$?	0.40	0.25	0.15	0.15	0.05

(c) **Kumulierte relative Häufigkeiten**

Frage	Kat. 1	Kat. 2	Kat. 3	Kat. 4	Kat. 5
$PS_1^{[E]}$?	0.05	0.05	0.30	0.70	1
$PS_2^{[E]}$?	0.10	0.15	0.45	0.90	1
$PS_3^{[E]}$?	0.10	0.25	0.75	0.95	1
$PS_4^{[E]}$?	0.40	0.65	0.80	0.95	1

(d) **Normierte Häufigkeiten und Kategorisierung**

Frage	Kat. 1	Kat. 2	Kat. 3	Kat. 4	$\overline{X}$	Ausp.	Kat.	Antw.
$PS_1^{[E]}$?	-1.645	-1.645	-0.524	0.524	-0.8224	0.731	(4)	FALSE
$PS_2^{[E]}$?	-1.282	-1.036	-0.126	1.282	-0.2905	0.199	(3)	FALSE
$PS_3^{[E]}$?	-1.282	-0.674	0.674	1.645	0.0908	-0.183	(3)	FALSE
$PS_4^{[E]}$?	-0.253	0.385	0.842	1.645	0.6546	-0.746	(2)	TRUE
$\overline{Y}$	-1.12	-0.743	0.217	1.27	$-\mathbf{0.0919}$			

Abbildung 8.2.: Beispiel kumulierter und kategorisierter Urteile.

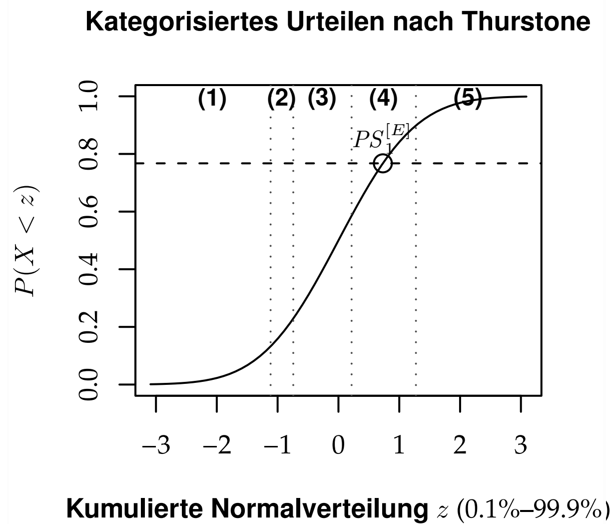

Abbildung 8.3.: Beispiel kumulierter Normalverteilung mit kategorisierter Fragestellung aus Abb. 8.2.

Eine Fragestellung wird jetzt anhand des Mittelwertes der z-Werte eingeordnet. Die Ausprägung ist die Differenz zwischen Zeilenmittelwert $\overline{\overline{X}}$ und mittlerer Kategoriegrenze (hier $\overline{\overline{X}} = -0.0919$). Damit ist z. B. für Frage $PS_1^{[E]}$ die Ausprägung $-0.0919 - (-0.8224) = 0.731$ und wird daher in Kategorie 4 eingeordnet. Diese Fragestellung ist auch in Abbildung 8.3 eingezeichnet.

Wenn die Fragestellung in die Kategorien 1 oder 2 eingeordnet wird, so wird sie mit „Ja, der Entwurfsmangel liegt vor“, andernfalls mit „Nein, der Entwurfsmangel liegt nicht vor“ beantwortet.

Delphi-Methode

Die Delphi-Methode ist eine besondere Form der Befragung, in der alle Beteiligten an einer strukturierten Gruppenkommunikation teilnehmen. Das Ziel ist es, aus Einzelbeiträgen Lösungen komplexer Probleme zu erarbeiten. Die Delphi-Methode ist ein Begutachtungsverfahren.

Hier besteht das Problem, für eine Menge von Programmstellen festzulegen, ob ein Entwurfsmangel vorliegt oder nicht. Dazu wird eine Gruppe von Testpersonen gebeten, wie im vorherigen Abschnitt, ein kategorisiertes Urteil zu einer Programmstelle bezüglich eines Entwurfsmangels abzugeben. Zusätzlich wird jede Testperson gebeten, das Urteil schriftlich zu begründen. Hier können besonders

auffällige Merkmale, Indizien, o. ä. aufgeführt werden.

In einer zweiten Befragung werden den Testpersonen die gleichen Programmstellen zur Prüfung vorgelegt. Die Urteile der anderen Testpersonen mit den schriftlichen Begründungen liegen ebenfalls vor, sodass die Testperson daran das eigene Urteil messen kann. Ggf. erkennt die Testperson Fehler in der eigenen Einschätzung oder sieht die eigene Ansicht bestätigt. In selteneren Fällen beharrt eine Person auf dem eigenen Standpunkt, obwohl er von vielen nicht geteilt wird.

Die so gefestigten kategorisierten Urteile der einzelnen Testpersonen werden wiederum gemittelt und bilden einen qualitativ hochwertigen Referenzkatalog.

8.3.2. Messung der Effektivität

Die Effektivität wird durch etablierte Verfahren aus dem *Information Retrieval* bewertet. Diese werden für die aktuellen Bedürfnisse angepasst.

Die Effektivität des Systems ist am Nutzen für den jeweiligen Benutzer zu messen [76]. Hierzu wird experimentell untersucht, ob eine Menge von Anfragen präzise und vollständige Ergebnisse liefert. Eine solche Bewertung ist nur dann möglich, wenn das korrekte Ergebnis aus der Voruntersuchung bekannt ist und mit den Antworten des Systems verglichen wird.

Im Folgenden wird das grundlegende Konzept der *Relevanz* vorgestellt, auf dem alle weiteren hier beschriebenen Analysen beruhen. Alle Verfahren werden anhand von Beispielmessreihen beschrieben und verdeutlicht.

Relevanz

Das Konzept der Relevanz beschreibt, ob eine Programmstelle einen fraglichen Entwurfsmangel darstellt. In der klassischen Variante nimmt die Relevanz genau zwei Werte an: relevant und nicht relevant. Die Relevanz wird aus der Voruntersuchung gewonnen und modelliert das subjektive Urteil des Expertengremiums oder, im einfachen Fall, einer Testperson. Hier wird die Relevanz wie folgt definiert:

Es sei jeweils eine Menge von Testpersonen T, Programmstellen PS und Entwurfsmängel EM gegeben. Die Relevanz wird dann durch die folgende Funktion beschrieben:

$$r_t(ps, em) \ \subseteq \ PS \times EM \rightarrow \{0, 1\}$$

Im Folgenden wird vereinfacht geschrieben:

$$r_{t,em}(ps) = r_t(ps, em)$$

Wobei $t \in T$ eine Testperson, $em \in EM$ ein Entwurfsmangel und $ps \in PS$ eine betrachtete Programmstelle ist.

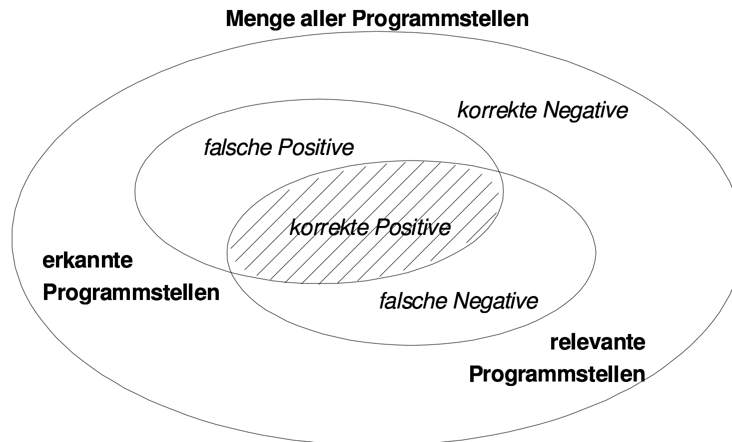

Abbildung 8.4.: Einordnung von Programmstellen in die Erkennungssituation.

Die Relevanzfunktion $r_{t,em}(ps)$ ergibt 1, wenn die Testperson t in der Programmstelle ps den Entwurfsmangel em erkennt und sonst 0.

Die Umkehr der Relevanzfunktion für den Funktionswert 1 liefert die Menge aller Programmstellen, die von der Testperson als Entwurfsmangel bezeichnet wird:

$$PS_t = r_{t,em}^{-1}(1) \subseteq PS$$

Metriken zur Leistungsbewertung

Die Erkennung von Entwurfsmängeln ordnet jeder Programmstelle ebenfalls einen boolschen Wert zu: die fragliche Programmstelle stellt den Entwurfsmangel dar, oder nicht. Es ist zu messen, ob die automatisch als Entwurfsmangel erkannten Programmstellen der Vorstellung einer Testperson entsprechen. Relevanz und Erkennung teilen die Menge der Programmstellen in Teilmengen auf, die in Abbildung 8.4 dargestellt sind.

Als *Positive* werden Programmstellen bezeichnet, die den betrachteten Mangel darstellen. *Negative* enthalten den Mangel nicht. Falsche Positive bzw. Negative sind dann Programmstellen, die vom Werkzeug falsch klassifiziert wurden; korrekte Positive bzw. Negative hat das System nach der Vorstellung des Benutzers eingeordnet.

Ziel ist es, die Schnittmenge der vom Benutzer als relevant bezeichneten Programmstellen, d. h. die den Mangel darstellen, und die Menge der automatisch vom Werkzeug ausgewiesenen Programmstellen zu maximieren.

Tabelle 8.1 zeigt alle Kenngrößen und deren exakte Berechnung. Die Spalten zeigen, ob eine Testperson die fraglichen Programmstellen für relevant hält oder nicht. Die Zeilen stellen das Urteil des Klassifizierers dar. Hiervon ausgehend wur-

		Testperson	
		relevant (p)	nicht relevant (n)
Klassifizierer	erkannt (P)	*True Positives (TP)*	*False Positives (FP)*
	nicht erkannt (N)	*False Negatives (FN)*	*True Negatives (TN)*

Die einzelnen Maße sind wie folgt definiert:

$$\begin{aligned} TP &= |PS_{.,em} \cap PS_{t,em}| \\ FP &= |PS_{.,em} \cap \overline{PS_{t,em}}| \\ FN &= |\overline{PS_{.,em}} \cap PS_{t,em}| \\ TN &= |\overline{PS_{.,em}} \cap \overline{PS_{t,em}}| \end{aligned}$$

Tabelle 8.1.: Übersicht der Kenngrößen der Klassifizierungsleistung.

Bezeichner	**Berechnung**	Kommentar
True Positive Rate	$P(P\|p) = TPR = \frac{TP}{TP+FN}$	**„Recall“**, „Vollständigkeit“, „Sensitivity“, „Positive Accuracy“
False Negative Rate	$P(N\|p) = FNR = \frac{FN}{TP+FN}$	„Positive Error“
	$TPR + FNR = 1$	
True Negative Rate	$P(N\|n) = TNR = \frac{TN}{TN+FP}$	**„Specificity“**, „Genauigkeit“, „Neg. Accuracy“
False Positive Rate	$P(P\|n) = FPR = \frac{FP}{TN+FP}$	„Negative Error“, „Ungenauigkeit“
	$TNR + FPR = 1$	
Postitive Predictive Value	$P(p\|P) = PPV = \frac{TP}{TP+FP}$	**„Precision“**
Negative Predictive Value	$P(n\|N) = NPV = \frac{TN}{TN+FP}$	
Macro-Avarage	AVG(TPR, TNR)	AVG is arith., geometr. oder anderer Mittelwert
Break-Even	$\frac{\text{Precision}+\text{Recall}}{2} = \frac{PPV+TPR}{2}$	
F-Measure	$\frac{\text{Precision}*\text{Recall}}{\text{Break}-\text{Even}} = \frac{2*PPV*TPR}{PPV+TPR}$	

Tabelle 8.2.: Übersicht der Metriken zur Klassifizierungsleistung.

den verschiedene Metriken definiert. Auf einige wird im Späteren detailliert eingegangen. Tabelle 8.2 zeigt eine Übersicht der wichtigsten Maße. Dabei bedeutet die bedingte Wahrscheinlichkeit $P(x|y)$ die Wahrscheinlichkeit von x unter der Bedingung von y. So ist z. B. $P(P|p)$ die Wahrscheinlichkeit, dass der Klassifizierer eine Programmstelle korrekt als Entwurfsmangel erkennt, wenn man die Menge der von der Testperson als Entwurfsmangelbehaftet beurteilten Programmstellen zugrunde legt. Die einzelnen Rechenregeln ergeben sich aus den Regeln zur Berechnung bedingter Wahrscheinlichkeiten.

Zu den wichtigsten Metriken gehören *Präzision (Precision) P*, *Genauigkeit (Specificity) S* und *Vollständigkeit (Recall) R*. Sie dienen als Grundlage für eine Reihe weiterer Metriken.

Die *Präzision (Precision)* gibt an, wieviele der ausgewiesenen Programmstellen auch von der Testperson als relevant eingeordnet wurden. Das anteilig definierte Präzisionsmaß hat den Wert 1, wenn alle gefundenen Mängel relevant sind und den Wert 0, wenn keine ausgewiesene Programmstelle relevant ist.

Die *Genauigkeit (Specificity)* verhält sich komplementär zur Präzision. Sie gibt an, wieviele nicht ausgewiesene Programmstellen auch von einer Testperson nicht als Mangel erkannt wurden. Dies kann auch als Präzision der Nichterkennung verstanden werden.

Die *Vollständigkeit (Recall)* gibt an, wieviele, der von der Testperson als relevant bezeichneten Programmstellen, auch vom Klassifizierer ausgewiesen wurden. Wenn das Verfahren alle vorhandenen relevanten Programmstellen präsentiert, ist der Wert der Vollständigkeit 1, wird keine Programmstelle erkannt, ist der Wert 0.

Kumulierte Metriken

Zur Berechnung von z. B. Präzision und Vollständigkeit muss die Menge der relevanten Programmstellen bekannt sein. Hierzu müssen alle Programmstellen des Analysegegenstandes manuell bzgl. der betrachteten Entwurfsmängel bewertet werden. Dies ist in großen Software-Systemen nicht durchführbar.

Man beschränkt sich daher auf eine ggf. kleine Teilmenge der Programmstellen. Eine Suche nach Entwurfsmängeln liefert dann eine zufällig geordnete Folge von $i = 1 \dots n$ Programmstellen als Ergebnis. Damit sind die tatsächlichen Werte für nicht erkannte Programmstellen nicht verfügbar. In Tabelle 8.1 ist das die untere Zeile mit den Größen *True Negatives (TN)* und *False Negatives (FN)*.

Die Folge von Programmstellen wird in n Teilmengen PS_i mit $i = 1 \dots n$ zerlegt. Dabei enthält die Teilmenge PS_i genau die ersten i Elemente der geordneten Ergebnisfolge. Die Werte für TN und FN sind dann 0, da keine Programmstellen als Entwurfsmangelfrei ausgewiesen wurden.

Die einzelnen Kennwerte werden dann für jede Teilmenge PS_i berechnet. P_i beschreibt wieviele der ersten i Programmstellen relevant sind; und R_i beschreibt,

i	**Programmstelle**	**Relevanz** $r_t(i)$	**Vollständigkeit** R_i	**Präzision** P_i
1	a_1	1	$\frac{1}{21} = 0.0476$	$\frac{1}{1} = 1.000$
2	a_2	0	$\frac{1}{21} = 0.0476$	$\frac{1}{2} = 0.500$
3	a_3	1	$\frac{2}{21} = 0.0952$	$\frac{2}{3} = 0.667$
4	a_4	1	$\frac{3}{21} = 0.1429$	$\frac{3}{4} = 0.750$
5	a_5	0	$\frac{3}{21} = 0.1429$	$\frac{3}{5} = 0.600$
6	a_6	1	$\frac{4}{21} = 0.1905$	$\frac{4}{6} = 0.667$
7	a_7	1	$\frac{5}{21} = 0.2381$	$\frac{5}{7} = 0.714$
8	a_8	0	$\frac{5}{21} = 0.2381$	$\frac{5}{8} = 0.625$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
27	a_{27}	1	$\frac{18}{21} = 0.8571$	$\frac{18}{27} = 0.667$
28	a_{28}	1	$\frac{19}{21} = 0.9048$	$\frac{19}{28} = 0.679$
29	a_{29}	1	$\frac{20}{21} = 0.9524$	$\frac{20}{29} = 0.690$
30	a_{30}	1	$\frac{21}{21} = 1.0000$	$\frac{21}{30} = 0.700$
		Arith. Mittel $\overline{x}$	0.505	0.681

Tabelle 8.3.: Beispiel für geordnete P/R-Berechnungen.

wieviele relevante Programmstellen unter den ersten i betrachteten Programmstellen zu finden sind.

Als Beispiel werden $n = 30$ Programmstellen $\{a_1, \ldots, a_{30}\} \subseteq PS$ betrachtet, die vom System bei Anfrage eines Entwurfsmangels ausgewiesen wurden. Die manuelle Bewertung einer Testperson ergibt allerdings, dass nicht alle Programmstellen relevant sind. Tabelle 8.6 zeigt einen Auszug der 30 Programmstellen mit der Bewertung einer Testperson und den zugehörigen kumulierten Präzisions- und Vollständigkeitswerten. Insgesamt sind 21 relevante Programmstellen enthalten.

Für $i = 3$ ist dann in dem Beispiel

$$P_3 = \frac{|\{a_1, a_2, a_3\} \cap \{a_1, a_3\}|}{3} = \frac{2}{3}$$

$$R_3 = \frac{|\{a_1, a_2, a_3\} \cap \{a_1, a_3\}|}{21} = \frac{2}{21}$$

Trägt man diese Werte in ein Diagramm ein, so erhält man die Darstellung in Abbildung 8.5. Die relevanten Programmstellen sind schwarz und die nicht relevanten grau gefärbt. Wird eine relevante Programmstelle in die Berechnung aufgenommen, schreitet die Vollständigkeit voran, bis schließlich alle relevanten Pro-

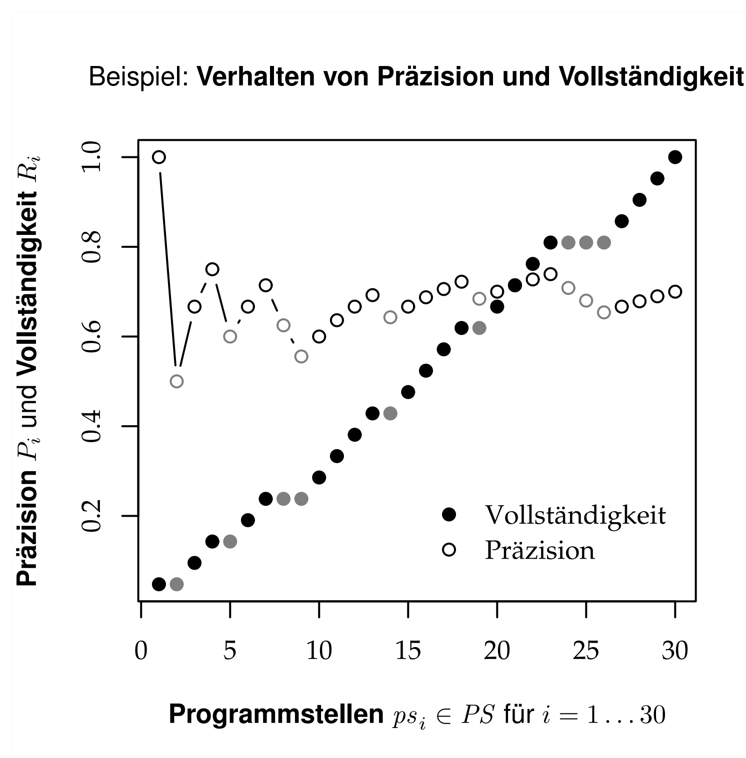

Abbildung 8.5.: Beispiel einer Messreihe mit Präzisions- und Vollständigkeitswerten.

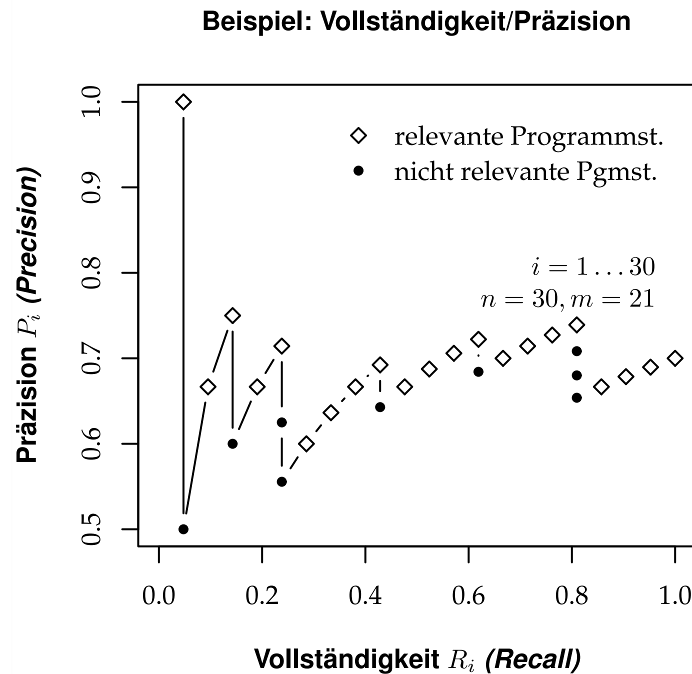

Abbildung 8.6.: Beispiel eines kumulierten P/R-Diagramms.

grammstellen Teil der Betrachtung sind und damit der Wert 1 erreicht wird. Die Präzisionswerte sinken bei nicht relevanten und steigen bei relevanten Programmstellen. Bei wenigen betrachteten Programmstellen haben falsch ausgewiesene eine hohe Auswirkung auf den Präzisionswert; je mehr Programmstellen betrachtet werden, um so geringer wird der Einfluss.

Typischerweise wird die Abhängigkeit zwischen Vollständigkeit und Präzision beobachtet. Hierzu werden die Einzelwerte als Folge von Tupeln (R_i, P_i) betrachtet und in ein Diagramm gezeichnet. Siehe Abbildung 8.6.

Zur Bewertung eines Klassifizierers reichen die beschriebenen einzelnen Metriken nicht aus. So können z. B. Präzision und Vollständigkeit nicht einzeln zur Bewertung verwendet werden. Liefert das System einfach alle Programmstellen als Antwort, so ergibt sich für die Güte der Vollständigkeit der Wert 1. Ebenso könnte das System exakt eine relevante Programmstelle ausweisen und damit die Präzisionsgüte 1 erhalten. Man verwendet daher die Kombination mehrerer Metriken und erhält zum Einen *Precision/Recall*- und zum Anderen *ROC*-Analysen.

8.3.2.1. Precision/Recall-Analyse

Ein System verhält sich ideal, wenn sowohl Präzision als auch Vollständigkeit einen hohen Wert erreichen. In der Praxis wird die Güte eines Systems allerdings abhängig vom Benutzer bewertet. So wünscht sich ein Benutzer eine hohe Präzision, d. h., es sollen wenige oder keine Programmstellen irrtümlich als Entwurfsmangel erkannt werden. Andere Benutzer wünschen sich eine hohe Vollständigkeit, d. h. es soll jede Programmstelle gemeldet werden, die einen Entwurfsmangel enthalten könnte.

Man möchte verschiedene Modelle zur Erkennung von Entwurfsmängeln bezüglich ihrer Erkennungsleistung beurteilen und vergleichen. Hierzu trägt man die Messreihen der einzelnen Modelle in ein gemeinsames P/R-Diagramm. Das bessere Modell ist weiter rechts oben angeordnet; dort sind Präzision und Vollständigkeit maximal.

Aussagekräftige Modellvergleiche erhält man nur, wenn mehrere Messreihen unter verschiedenen Randbedingungen und von unterschiedlichen Testpersonen durchgeführt werden. Unterscheiden sich die Randbedingungen, weil z. B. die Anzahl der Programmstellen variiert, dann ist ein direkter Vergleich im Diagramm nicht erlaubt.

Normierung und Interpolation

Man stellt daher die Vergleichbarkeit her, indem man das Diagramm bezüglich der Vollständigkeit normiert und beschränkt. Hierzu wählt man feste Punkte auf der Vollständigkeitsskala und bestimmt die zugehörigen (i. d. R. nicht vorhandenen) Präzisionswerte durch ein Interpolationsverfahren. Hier werden die elf Werte $\{0, 0.1, \ldots, 1\}$ von 0 bis 1 in 0.1-Intervallen für die Vollständigkeit festgelegt. Als Wert für die Präzision wird der erste Wert genommen, dessen zugehöriger Vollständigkeitswert größer oder gleich dem jeweiligen Intervall-Startwert ist. Abbildung 8.7 zeigt das P/R-Diagramm aus Abb. 8.6 mit der zugehörigen Interpolation.

Trägt man mehrere interpolierte P/R-Diagramme in ein Koordinatensystem, so lassen sich diese augenscheinlich nach ihrer Erkennungsleistung sortieren. Abbildung 8.8 zeigt die bekannte Beispielmessreihe und die schlechteste sowie beste Messreihe. Die schlechteste Messreihe enthält in den ausgewiesenen Programmstellen keine Relevante. Die beste Messreihe enthält ausschließlich relevante Programmstellen. Da dies eine kumulierte Messreihe ist, ist die Aussage über die Vollständigkeit schwach.

Da nur für die ausgewiesenen Programmstellen die Relevanz durch eine Testperson bestimmt wird, ergibt sich zwangsläufig, dass die Messreihe mit einer Vollständigkeit von 1 endet. Für die weitere Betrachtung wird die Vollständigkeit, da ohnehin normiert, außer Acht gelassen.

Mit dem Mittelwert der Präzision erhält man schließlich ein eindimensionales

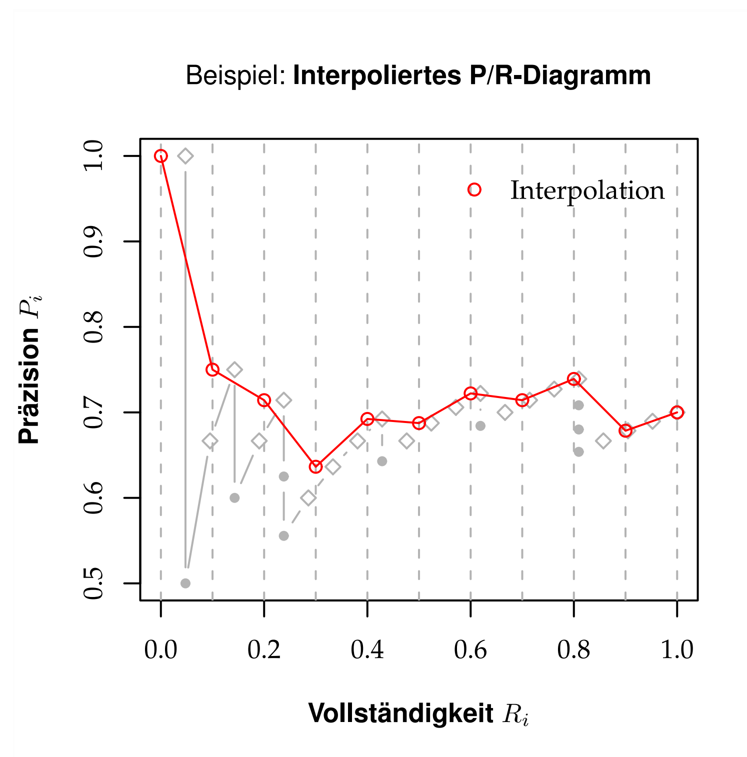

Abbildung 8.7.: Beispiel eines kumulierten P/R-Diagramms mit zugehöriger Normierung und Interpolation

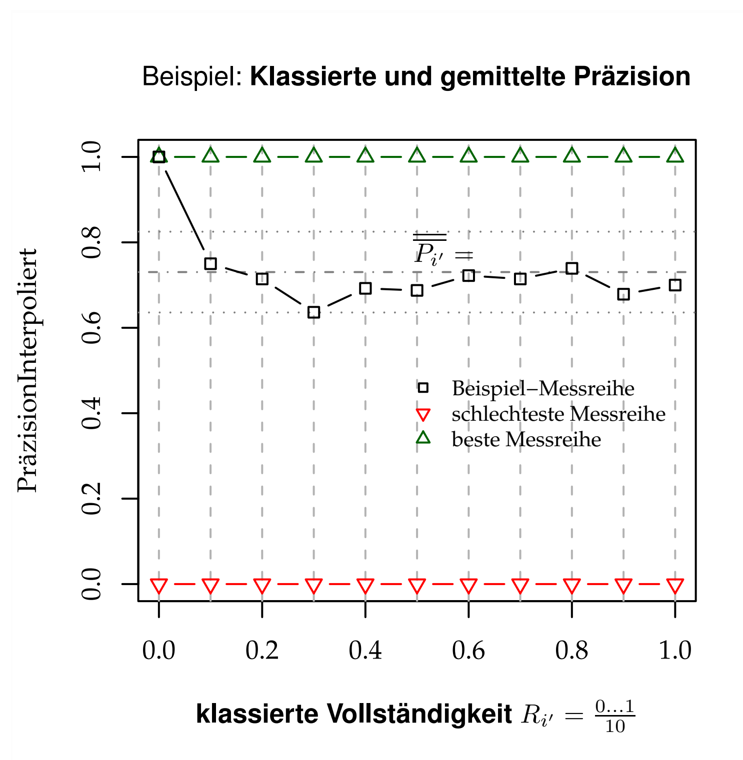

Abbildung 8.8.: Beispiel eines kumulierten und interpolierten P/R-Diagramms.

Es werden 10 Vorlesungen betrachtet. 5 Vorlesungen werden von jeweils 99 Studierenden besucht und 5 von je nur einer Person.
Wieviel Studierende besuchen im Durchschnitt die Vorlesungen?

Makrobewertung: Betrachtet man alle $n = 10$ *Vorlesungen* und die Anzahl A_i der Studierenden in der Vorlesung i, so ergibt:

$$\overline{x} = \frac{1}{n}\sum_{i}^{n} A_i = \frac{5*1+5*99}{10} = 50$$

Mikrobewertung: Betrachtet man alle $n = 5*99+5*1$ *Studierenden* und die Anzahl A_i der Hörer in derselben Vorlesung wie der Studierende i, so ergibt:

$$\overline{x} = \frac{1}{n}\sum_{i}^{n} A_i = \frac{(5*99)*99+(5*1)*1}{5*99+5*1} = 98.02$$

Das heißt, in einer Vorlesung sind im Mittel 50 Hörer anwesend; aber ein Studierender sitzt mit durchschnittlich 98.02 Hörern in einer Vorlesung.

Abbildung 8.9.: Beispiel zur Makro- und Mikromittelwertberechnung aus [76].

Maß, mit dem sich die Modelle bezüglich ihrer Leistung in eine Rangfolge bringen lassen. Für die Mittelwertberechnung unterscheidet man Benutzer- und Systemsicht bzw. Makro- und Mikrobewertung.

Makro- und Mikrobewertung

Bei der Makrobewertung wird das arithmetische Mittel der Präzisionswerte P_i für $i = 1 \ldots N$ bestimmt. Dabei können mehrere Messreihen mit insgesamt N Präzisionswerten berücksichtigt werden:

$$P_U = \frac{1}{N}\sum_{i}^{N} P_i = \frac{1}{N}\sum_{i}^{N} \frac{|PS_{.,em} \cap PS_{t,em}|}{|PS_{.,em}|}$$

Die Mikrobewertung berechnet den Mittelwert wie folgt:

$$P_S = \frac{\sum_i^N |PS_{.,em} \cap PS_{t,em}|}{\sum_i^N |PS_{.,em}|}$$

Ein viel zitiertes Beispiel aus [76] ist in Abbildung 8.9 zu sehen und verdeutlicht den Unterschied zwischen Makro- und Mikrobewertung.

i	**Vollständigkeit** R_i	**Präzision** $P_i^{(1)}$	**Präzision** $P_i^{(2)}$
1	0.0	$\frac{1}{1} = 1.000$	$\frac{0}{0} = 0.000$
2	0.1	$\frac{15}{15} = 0.714$	$\frac{1}{1} = 0.333$
3	0.2	$\frac{29}{29} = 0.725$	$\frac{2}{2} = 0.500$
4	0.3	$\frac{44}{44} = 0.733$	$\frac{3}{3} = 0.500$
5	0.4	$\frac{58}{58} = 0.744$	$\frac{4}{4} = 0.333$
6	0.5	$\frac{72}{72} = 0.720$	$\frac{4}{4} = 0.333$
7	0.6	$\frac{87}{87} = 0.719$	$\frac{5}{5} = 0.333$
8	0.7	$\frac{101}{101} = 0.721$	$\frac{6}{6} = 0.300$
9	0.8	$\frac{116}{116} = 0.725$	$\frac{7}{7} = 0.333$
10	0.9	$\frac{130}{130} = 0.730$	$\frac{8}{8} = 0.286$
11	1.0	$\frac{144}{144} = 0.720$	$\frac{8}{8} = 0.286$
	Umfang n	200	30
	Relevante Programmst. m	144	8
	Makromittel P_U	0.75	0.322
	Mikromittel P_S	0.725	0.32
	Gesamtumfang n	230	
	Relevante Pgst. gesamt m	152	
	Makromittel P_U	0.536	
	Mikromittel P_S	0.677	

Tabelle 8.4.: Vergleich zwischen Makro- und Mikrobewertung

Betrachtet man mehrere Messreihen, so wird bei der Makrobewertung der Umfang der einzelnen Antwortmengen nicht berücksichtigt. Jede Messreihe geht gleichgewichtig in die Berechnung ein. Deshalb wird diese Bewertung auch *nutzungsorientiert* genannt. Im Gegensatz gehen bei der Mikrobewertung große Messreihen stärker in die Bewertung ein als kleine Messreihen. Diese wird *systemorientierte* Bewertung genannt.

Die zuvor durchgeführte Normierung der Vollständigkeit hat auf die Berechnung der Mittelwerte keine normierende Wirkung. Denn die Größen der Messreihen sind noch erhalten, lediglich die Anzahl der Messpunkte ist reduziert.

Tabelle 8.4 zeigt zwei Messreihen mit unterschiedlichem Umfang. Innerhalb einer Messreihe unterscheiden sich Makro- und Mikrobewertung nicht nennenswert. Lediglich Fehler bei kleiner Vollständigkeit gehen bei der Makrobewertung stärker ein. Die Präzision der kleineren Messreihe ist deutlich schlechter als die der größeren Messreihe; z. B. weil unterschiedliche Testpersonen die Relevanzen

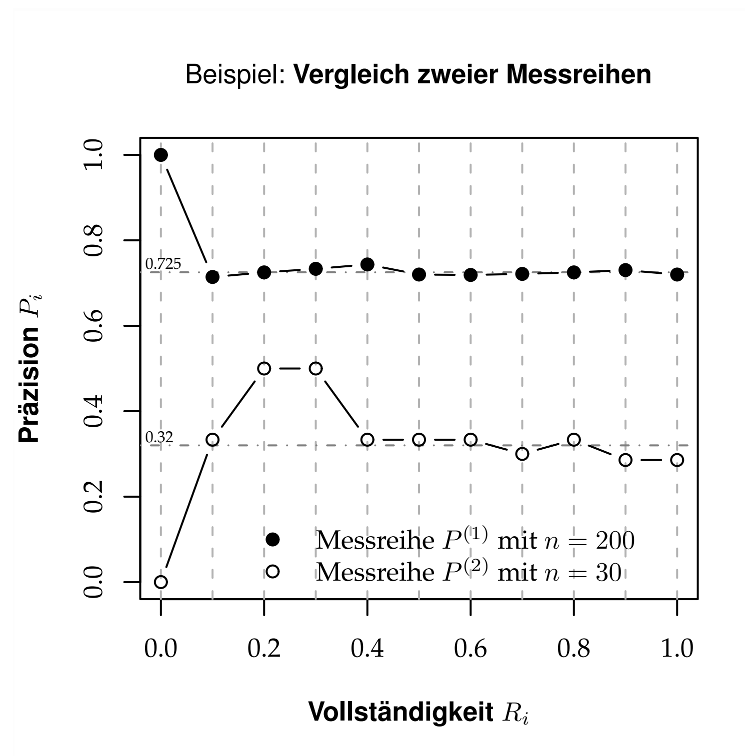

Abbildung 8.10.: Vergleich zwischen Makro- und Mikrobewertung

festgelegt haben oder verschiedene Erkennungsmodelle verwendet wurden. Abbildung 8.10 zeigt die zugehörigen P/R-Diagramme. Der jeweilige Mikromittelwert P_S ist durch eine horizontale Linie kenntlich gemacht; am Rand steht der tatsächliche Wert.

Die Mittelwertberechnung beider Messreihen zeigt nun einen deutlichen Unterschied zwischen Makro- und Mikrobewertung. Wegen der schwächeren Berücksichtigung der kleinen Messreihe liegt der Präzisionswert deutlich höher.

Die Bewertung von Modellen kann als Systembewertung aufgefasst werden. Daher wird hier die Mikrobewertung favorisiert. Anhand des Mikromittelwertes kann nun die Güte einer Messreihe bewertet werden.

Statistische Bewertung

Messreihen werden zunächst mit Hilfe des Mikromittelwertes in eine Rangfolge gebracht. Der statistische Signifikanztest dient der Bestätigung der Hypothese. Ist der Mittelwert der Präzisionswerte einer Messreihe höher als der Mittelwert einer anderen Messreihe, so wird diese als „besser“ bezeichnet. Analog gibt es „gleiche“ und „schlechtere“ Messreihen.

Die beobachteten Unterschiede zwischen den Präzisionswerten einer Messreihe könnten rein zufällig sein. Es ist daher nötig, zusätzlich mit Hilfe eines statistischen Signifikanztests festzustellen, ob die Unterschiede wesentlich sind.

Die folgenden Nullhypothesen H_0 mit ihren Gegenhypothesen H_1 beschreiben jeweils die Zusammenhänge zwischen zwei Messreihen bzw. Stichproben X und Y. Es werden die Wahrscheinlichkeiten, dass $X > Y$ bzw. $X < Y$ ist, in Beziehung gesetzt.

(a) Die Messreihe X ist *gleich* der Messreihe Y; kurz: $X \approx Y$.

$$H_0 : P(X > Y) = P(X < Y),\ H_1 : P(X > Y) \neq P(X < Y)$$

(b) Die Messreihe X ist *besser als* die Messreihe Y; kurz: $X \gg Y$.

$$H_0 : P(X > Y) \geq P(X < Y),\ H_1 : P(X > Y) < P(X < Y)$$

(c) Die Messreihe X ist *schlechter als* die Messreihe Y; kurz: $X \ll Y$.

$$H_0 : P(X > Y) \leq P(X < Y),\ H_1 : P(X > Y) > P(X < Y)$$

Es werden zwei Stichproben miteinander verglichen, über deren statistische Eigenschaften (z. B. Verteilung) nichts bekannt ist. Daher kommen nur nichtparametrische Signifikanztests für unabhängige Stichproben in Frage [5]. Statt des einfachen Vorzeichentests verwendet man den *Wilcoxon-Rangsummentest* [93][45] (in

i	**Vollständigkeit** R_i	**Präzision** $P_i^{(2)}$	**Präzision** $P_i^{(3)}$	**Präzision** $P_i^{(4)}$
1	0.0	0.000	0.000	0.000
2	0.1	0.333	0.333	0.500
3	0.2	0.500	0.500	0.667
4	0.3	0.500	0.500	0.750
5	0.4	0.333	0.333	0.667
6	0.5	0.333	0.333	0.417
7	0.6	0.333	0.300	0.400
8	0.7	0.300	0.333	0.350
9	0.8	0.333	0.286	0.381
10	0.9	0.286	0.310	0.321
11	1.0	0.286	0.310	0.321
	Umfang n	30	30	30
	Relevante Programmst. m	8	9	9
	Makromittel P_U	0.322	0.322	0.434
	Mikromittel P_S	0.32	0.321	0.386

Tabelle 8.5.: Beispiel für drei minimal verschiedene Messreihen.

Abbildung 8.11 wird der Test skizziert). Der häufig verwendete *t*-Test ist nicht geeignet, da die Unterschiede zwischen den Messreihen normalverteilt sein müssen.

Wie bei Signifikanztests üblich, testet man die Gegenhypothese H_1. Wird diese abgelehnt kann die zugehörige Nullhypothese H_0 angenommen werden. Es werden nun alle drei Hypothesen (a)–(c) getestet. Der Test gibt jeweils das Konfidenzniveau an, mit der die Nullhypothese angenommen bzw. verworfen wird. Hierzu legt man fest, welche Irrtumswahrscheinlichkeit α zugelassen werden soll. I. d. R. wählt man für α die Werte 0.01, 0.05 oder 0.10. Das zugehörige Konfidenzniveau ist definiert als $1 - \alpha$. Wenn die Konfidenz eines Tests oberhalb des festgelegten Wertes $1 - \alpha$ liegt, wird die Hypothese angenommen, andernfalls verworfen.

In dem nun folgenden Beispiel wird die bereits bekannte Messreihe $P^{(2)}$ mit Umfang $n = 30$ wiederverwendet. Hierzu werden zwei weitere Messreihen $P^{(3)}$ und $P^{(4)}$ konstruiert, die nur minimal verbesserte Relevanzwerte aufweisen. Bei $P^{(3)}$ erhält die 29. Programmstelle die Relevanz 1 statt 0. Bei $P^{(4)}$ wird die Relevanz der 2. Programmstelle von 0 auf 1 geändert. In Tabelle 8.5 sind die normierten Messreihen und in Abbildung 8.12 die zugehörigen P/R-Diagramme zu sehen.

Da in $P^{(4)}$ sehr früh eine zusätzliche relevante Programmstelle eingeht, weicht die Präzision bereits für $i = 0.1$ nach oben ab. Solange $i < 0.6$ bleiben die Präzisionswerte von $P^{(2)}$ und $P^{(3)}$ paarweise gleich. Dann sorgt die Interpolation dafür, dass $P^{(3)}$ für $i = 0.6$ einen geringeren Wert als $P^{(2)}$ erhält, obwohl bisher alle Relevanzen gleich lauteten. Das liegt daran, dass $P^{(3)}$ eine relevante Programmstelle mehr hat und sich daher die Vollständigkeitsskala wegen $m^{(3)} = m^{(2)} + 1$ leicht links orientiert. Bis zum Ende der Messung erreicht $P^{(3)}$ dann wieder einen leicht

Wilcoxon-Rangsummentest [93]

Der Wilcoxon-Rangsummentest ist für zwei Stichproben X und Y gleichen Umfangs gedacht, die paarweise verglichen werden. I. d R. handelt es sich bei den beiden Stichproben um „vorher“ und „nachher“ Werte.
Beispiel: $X = (-1, 5, -2, -3, 2, 3, 2)$ und $Y = (2, -3, -2, 1, 1, -4, 1)$.
Hypothese: $H_0 : P(X > Y) = P(X < Y),\ H_1 : P(X > Y) \neq P(X < Y)$

1. Bilde den Betrag der Differenz jedes Paares und streiche Nullwerte: $|X - Y| = (|-3|, 8, 0, |-4|, 1, 7, 1) = (3, 8, \epsilon, 4, 1, 7, 1)$.

2. Ordne den absoluten Differenzen eine Rangzahl zu. Die Rangzahl gibt die Position eines Elementes in der geordneten Folge an. Treten Werte mehrfach auf, werden deren Rangzahlen addiert, durch die Anzahl gleicher Werte geteilt und unter diesen verteilt: $(3_{[3]}, 8_{[6]}, \epsilon, 4_{[4]}, 1_{[1.5]}, 7_{[5]}, 1_{[1.5]})$.

3. Negiere die Ränge jedes Elements i für das $X_i - Y_i < 0$ gilt: $(3_{[-3]}, 8_{[6]}, \epsilon, 4_{[-4]}, 1_{[1.5]}, 7_{[5]}, 1_{[1.5]})$.

4. Berechne die Summe der positiven Ränge T_+ und die Summe der Beträge der negativen Ränge T_-:
$T_+ = 6 + 1.5 + 5 + 1.5 = 14$ und $T_- = |-3| + |-4| = 7$.

5. Bestimme den Wert W als den kleineren Wert von T_+ und T_-; und die Anzahl N der Elemente $\neq \epsilon$: $W = 7$ und $N = 6$.

6. Für $N > 50$ kann die Hypothese mit dem Z-Test geprüft werden. Für kleine N kann der kritische Wert für W in der Tabelle für das jeweils gewünschte Konfidenzniveau nachgeschlagen werden.

N	5	6	7	8	9	10	11	$\cdots$
$\alpha = 0.10$	0	2	3	5	8	10	13	
$\alpha = 0.05$				3	5	8	10	
$\vdots$								$\ddots$

Hier ist z. B. für das Signifikanzniveau $\alpha = 0.10$ und $N = 6$ der kritische Wert $W_k = 2$. Wenn $W \leq W_k$ ist, dann wird H_0 verworfen und H_1 akzeptiert. $\Rightarrow$ Hier wird H_0 beibehalten.

Abbildung 8.11.: Beschreibung des *Wilcoxon-Rangsummentests*.

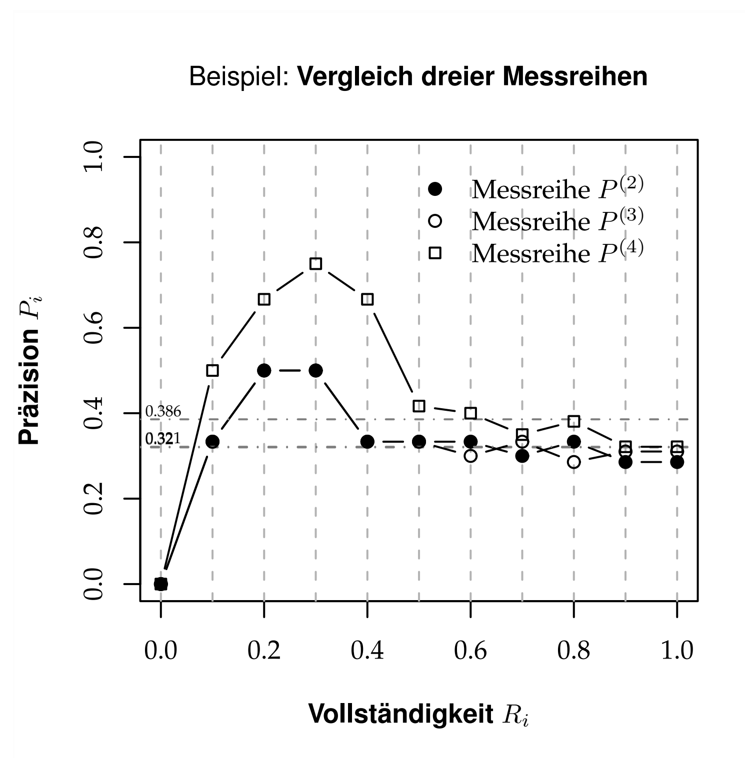

Abbildung 8.12.: P/R-Diagramme für drei minimal verschiedene Messreihen.

höheren Wert als $P^{(2)}$, da die zusätzliche relevante Programmstelle noch eingeht.

Intuitiv betrachtet, weisen die Messreihen $P^{(2)}$ und $P^{(3)}$ also kaum Unterschiede auf; $P^{(4)}$ hingegen erzielt wegen seiner höheren Präzision bei noch kleinem Stichprobenumfang eine höhere Leistung.

Als erstes Vergleichskriterium für Messreihen wurden bereits die Makro- und Mikromittelwerte genannt. Diese stimmen mit der intuitiven Bewertung überein. Für $P^{(4)}$ wird die bereits angesprochene Differenz zwischen den Mittelwertarten deutlich. Intuitiv ist nicht ersichtlich, warum die Präzision im Mittel über 33% größer ist als bei $P^{(2)}$ und $P^{(3)}$ sein sollte.

In diesem Beispiel unterscheidet sich der Mikromittelwert zwischen $P^{(2)}$ und $P^{(3)}$ nur um genau 0.001. Es wird dennoch geschlossen, dass $P^{(2)}$ schlechter als $P^{(3)}$ ist. Die Differenzen der Mittelwerte dieser beiden mit $P^{(4)}$ deuten deutlich auf eine höhere Leistung von $P^{(4)}$ hin. Tabelle 8.6 zeigt die paarweisen Vergleiche der drei Messreihen. In der zweiten Spalte sind die Schlüsse aufgeführt, die durch Vergleich der Mikromittelwerte gezogen wurden.

Der Vergleich zwischen $P^{(2)}$ und $P^{(3)}$ wurde bereits als Problemfall erkannt. Der statistische Signifikanztest wird daher zur Klärung benötigt. Die Tabelle 8.6 enthält weiterhin für jeden Vergleich die Wilcoxon-Testergebnisse für alle Hypothesen; sie sind jeweils nach der Konfidenz geordnet aufgeführt. Der Test ist sich demnach am sichersten bei der Hypothese, dass beide Messreihen gleiche Präzision liefern. Danach folgt die Hypothese, dass $P^{(3)}$ schlechter ist und die geringste Wahrscheinlichkeit weist der Test dem Schluss des Mikrovergleichs zu. Allerdings erreicht keiner der Tests das angestrebte Konfidenzniveau von 95%.

Das $P^{(3)}$ schlechter ist als $P^{(4)}$ bestätigt der Test sehr sicher. Es ist die einzige Hypothese, dessen Konfidenz fast 1 erreicht; die anderen Hypothesen werden mit Konfidenzen von fast 0 abgelehnt. Der Vergleich zwischen $P^{(2)}$ und $P^{(4)}$ bestätigt die Vermutung des Mikrovergleichs ebenfalls.

8.3.2.2. ROC-Analyse

Receiver Operating Characteristics (ROC) wurden zuerst zur Beurteilung von Radargeräten im zweiten Weltkrieg verwendet. Später folgte der Einsatz zur Analyse des Ansprechverhaltens von Transistoren und in der medizinischen Diagnostik.

Während P/R-Analysen die Präzision und Vollständigkeit gegenüber stellen, wird bei der ROC-Analyse die Präzision gegenüber der Ungenauigkeit betrachtet (vergleiche Tabelle 8.2).

Analog zur P/R-Analyse werden hier kumulierte Messreihen der Werte Präzision *(True Positive Rate)* und Ungenauigkeit *(1−Specificity, False Positive Rate)* gegenüber in ein Diagramm eingetragen. Eine Normierung findet nicht statt. Die einzelnen Werte werden durch eine treppenförmige Kurve verbunden.

Vergleich	Mikromittelwerte	Wilcoxon-Test	Konfidenz
$P_i^{(2)}$ mit $P_i^{(3)}$	$P_i^{(2)} \ll P_i^{(3)}$	$P_i^{(2)} \approx P_i^{(3)}$	0.892 (unsicher)
		$P_i^{(2)} \gg P_i^{(3)}$	0.658 (unsicher)
		$P_i^{(2)} \ll P_i^{(3)}$	0.446 (unsicher)
$P_i^{(3)}$ mit $P_i^{(4)}$	$P_i^{(3)} \ll P_i^{(4)}$	$P_i^{(3)} \ll P_i^{(4)}$	0.998 (sicher)
		$P_i^{(3)} \approx P_i^{(4)}$	0.00589 (unsicher)
		$P_i^{(3)} \gg P_i^{(4)}$	0.00294 (unsicher)
$P_i^{(2)}$ mit $P_i^{(4)}$	$P_i^{(2)} \ll P_i^{(4)}$	$P_i^{(2)} \ll P_i^{(4)}$	0.998 (sicher)
		$P_i^{(2)} \approx P_i^{(4)}$	0.658 (unsicher)
		$P_i^{(2)} \gg P_i^{(4)}$	0.446 (unsicher)

$$\Longrightarrow (P_i^{(2)} \approx P_i^{(3)}) \ll P_i^{(4)}$$

Vergleichsoperatoren: $\approx$ etwa gleich, $\ll$ schlechter als, $\gg$ besser als.

Tabelle 8.6.: Beispiel für den paarweisen Vergleich von drei Messreihen.

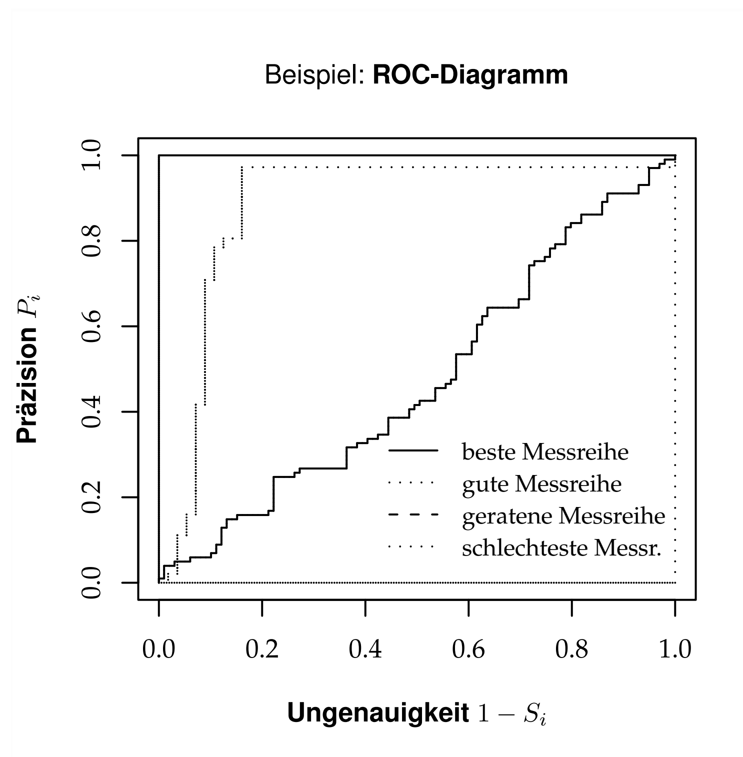

Abbildung 8.13.: Beispiel eines ROC-Diagramms.

Maß	$P^{(2)}$	$P^{(3)}$	$P^{(4)}$
Mikromittel	0.320	0.321	0.386
Makromittel	0.322	0.322	0.434
AUC	0.585	0.508	0.651

Tabelle 8.7.: Vergleich der Kenngrößen aus P/R- und ROC-Analyse.

Man erhält eine geschlossene Kurve, indem man die Werte $(0, 0)$ und $(1, 1)$ der beiden trivialen Klassifizierer hinzufügt. Diese sind:

1. Der triviale Klassifizierer, der keinen Entwurfsmangel ausweist. D. h. es wurde nichts gefunden, aber auch kein Fehler gemacht. Damit haben sowohl Präzision als auch Ungenauigkeit den Wert 0.

2. Der triviale Klassifizierer, der einfach alle Programmstellen als Entwurfsmangel ausweist. D. h. alle Positiven wurden gefunden aber auch alle Negativen fälschlich ausgewiesen. Damit sind beide Werte gleich 1.

Abbildung 8.13 zeigt vier Messreihen. Die beste Messreihe enthält keine Fehler, sodass die Präzision stetig steigt, die Ungenauigkeit aber bei 0 bleibt. Wegen der künstlichen Start- und Endpunkte beschreibt die Kurve den rechten Winkel links oben. Der schlechteste Klassifizierer enthält nur Fehler, sodass die Ungenauigkeit ständig steigt, die Präzision aber bei 0 bleibt. Diese Kurve beschreibt den rechten Winkel rechts unten.

Ein ratender Klassifizierer bildet eine Kurve aus, die der Diagonalen sehr ähnlich ist. Ein guter Klassifizierer sollte daher deutlich oberhalb der Diagonalen liegen.

Als direktes Maß für die Güte eines Klassifizierers wird die Fläche unter der Kurve *(Area Under Curve (AUC))* verwendet. Der Schlechteste hat daher den Wert $AUC = 0$, der Beste den Wert $AUC = 1$ und der ratende Klassifizierer ungefähr den Wert $AUC \approx 0.5$. Der gute Klassifizierer in Abbildung 8.13 hat den Wert $AUC = 0.884$.

8.3.2.3. Vergleich zwischen P/R- und ROC-Analyse

Betrachtet man noch einmal die drei minimal verschiedenen Messreihen aus Tabelle 8.5 und Abbildung 8.12 mit Hilfe der ROC-Analyse, so ergeben sich die ROC-Diagramme in Abbildung 8.14. In Tabelle 8.7 sind die Kenngrößen gegenüber gestellt.

Die AUC-Werte bringen die Messreihen in folgende Rangfolge:

$$P^{(3)} \ll P^{(2)} \ll P^{(4)}$$

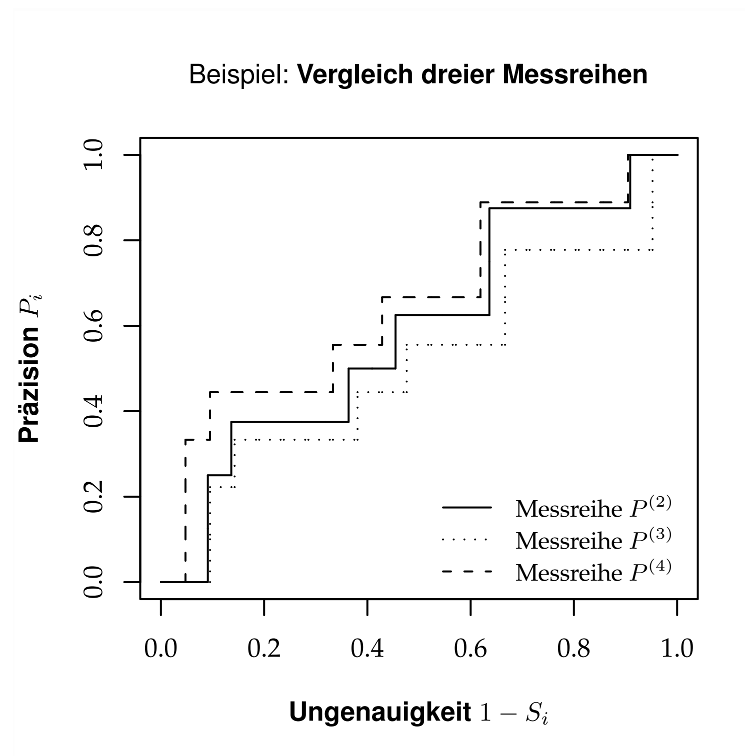

Abbildung 8.14.: ROC-Analyse für drei minimal verschiedene Messreihen.

Erstaunlich ist vor allem, dass $P^{(3)} \ll P^{(2)}$ ist. Betrachtet man die ROC-Diagramme, so sieht man, dass $P^{(4)}$ frühzeitig wegen der zusätzlichen relevanten Programmstelle nach oben abweicht. $P^{(3)}$ bleibt wenig später unterhalb von $P^{(2)}$. Weil $P^{(3)}$ insgesamt eine zusätzliche relevante Programmstelle ausweist, ist die relative Präzision zunächst geringer und die relative Ungenauigkeit zunächst höher.

Konsequenzen
Messreihen, die sich nur leicht unterscheiden, sind schwer zu bewerten. Weder P/R- noch ROC-Analyse erlauben es, eine eindeutige Rangfolge der Messreihen aufzustellen. Der Wilcoxon-Test hat hierfür bereits den Hinweis geliefert, indem im Beispiel kein signifikanter Unterschied zwischen $P^{(2)}$ und $P^{(3)}$ festgestellt werden konnte.

Grundsätzlich erlauben aber sowohl P/R- als auch ROC-Analyse eine anschauliche Bewertung von Messreihen und damit der Erkennungsleistung eines Klassifizierers bzw. einer Konfiguration aus Modell und Trainingsmenge. In Zweifelsfällen kann der Wilcoxon-Test hinzugezogen werden.

Kumulierte Messreihen werden in diesem Anwendungsfall benötigt, da die vollständige manuelle Relevanzerhebung unmöglich ist. Nicht etwa, weil einzelne Programme zu groß wären, um sie manuell zu untersuchen, sondern weil sonst *alle* Programme untersucht werden müssten. Es kann daher nur eine Stichprobe erhoben werden.

Durch kumulierte Metriken werden kleine leistungsstarke Messreihen großen und im Mittel sogar leicht besseren Messreihen vorgezogen. Dies entspricht auch der Strategie, dass Benutzer ggf. kleine Programmteile für ihre Suche auswählen. Die Erkennungsleistung soll hier besonders gut sein und darf daher in der Bewertung etwas stärker gewichtet werden.

8.3.3. Messung der Effizienz

Effizienzkriterien beschreiben, mit welchem technischen, zeitlichen und Kostenaufwand Entwurfsmängel gefunden werden können. Der in dieser Arbeit eingesetzte Prototyp soll experimentell zeigen, dass das Konzept tragfähig ist. Die Effizienz wird daher nur der Vollständigkeit halber betrachtet und kann Hinweise zur Verbesserung der Implementierung liefern.

Laufzeitbedarf Ein automatisches System sollte selbstverständlich Ergebnisse schneller liefern können als die manuelle Suche. Laufzeit und kurzzeitiger Speicherbedarf fallen für folgende Aktionen an:

- Metrikberechnung (durch statische Programmanalyse und Kumulation und Verrechnung der Analyseergebnisse).

- Interpretation von Entscheidungsbäumen.
- Konstruktion von Entscheidungsbäumen.

Dabei werden Metrikberechnungen sowohl bei manuellem Training als auch bei automatischer Suche benötigt. Im ersten Fall schließt sich dann die Konstruktion, im zweiten Fall die Interpretation von Entscheidungsbäumen an.

Die benötigte Zeit für Metrikberechnungen hängt von der Größe des Analysegegenstandes (Anzahl Instruktionen oder Größe der Klassendatei) und des strukturellen Umfangs (Summe der Anzahl der Programmobjekte und der Anzahl der Beziehungen zwischen den Programmobjekten) ab. Man weist daher zwei Maße aus:

1. Die Berechnungszeit (in Sekunden) gegenüber der Anzahl der Instruktionen.
2. Die Berechnungszeit gegenüber dem strukturellen Umfang.

Die Interpretation sollte im Vergleich zur Konstruktion von Entscheidungsbäumen wenig Zeit beanspruchen. Beide hängen vom Umfang der Trainingsmenge und der Anzahl der am jeweiligen Modell beteiligten Metriken bzw. Kriterien ab. Es wird daher notiert:

1. Die Interpretationszeit eines Entscheidungsbaumes gegenüber der Anzahl der Kriterien im Modell.
2. Die Konstruktionszeit eines Entscheidungsbaumes gegenüber der Anzahl der Kriterien im Modell und dem Umfang der Trainingsmenge.

Speicherbedarf Der Speicherbedarf muss getrennt untersucht werden: Speicher, der nur kurzzeitig, z. B. zur Programmanalyse, benötigt wird und Speicher, der länger verwendet wird (z. B. zur Speicherung von Entscheidungsbäumen). Analog zur Betrachtung der Laufzeit werden folgenden Maße für den kurzzeitigen Speicherbedarf ausgewiesen:

1. Der kurzzeitige maximale Speicherbedarf zur Metrikberechnung gegenüber dem strukturellen Umfang des Analysegegenstandes.
2. Der kurzzeitige maximale Speicherbedarf der Interpretation eines Entscheidungsbaumes gegenüber der Anzahl der Kriterien im Modell.
3. Der kurzzeitige maximale Speicherbedarf der Konstruktion eines Entscheidungsbaumes gegenüber der Anzahl der Kriterien im Modell und dem Umfang der Trainingsmenge.

Der ständige Speicherbedarf lässt sich schwer messen. In allen Bereichen wird Speicher für Verwaltungsaufgaben verwendet. Für eine grobe Abschätzung genügt es hier folgendes Maß zu erheben:

1. Der maximale ständige Speicherbedarf gegenüber der Anzahl der definierten Entwurfsmängel.

Kosten Die Kosten der Suche werden hier nicht weiter betrachtet. Im Allgemeinen bezieht man hier Kosten der Rechner, Energieverbrauch, Infrastruktur- und Personalkosten mit ein. Die Kosten des konkurrierenden Verfahrens, hier die manuelle Suche, sind gegenüber zu stellen. Für diesen Forschungsprototyp werden keine Kostendaten erhoben.

Die erhobenen Daten erlauben, das System hinsichtlich der Kriterien *Benutzbarkeit* und *Skalierbarkeit* zu beurteilen. Das System ist benutzbar, wenn die benötigte Zeit zur Beurteilung einer fraglichen Programmstelle wesentlich weniger Zeit benötigt als eine Testperson. Das System ist skalierbar, wenn es keine Eingabegrößen (Umfänge der Analysegegenstände oder der Trainingsmengen) gibt, die Laufzeit- oder Speicherbedarf exponentiell ansteigen lassen.

Um die Benutzbarkeit zu bewerten, werden die Gesamtlaufzeiten der Anwendungsfälle „Entwurfsmangel suchen" und „Entwurfsmangel trainieren" berechnet. Mittelwert, Standardabweichung, Minimal- und Maximalwerte geben einen Überblick zum Laufzeitverhalten des Systems.

Einen groben Eindruck vom Nutzen des Systems erhält man, wenn man die Zeit, die eine Testperson zur Entdeckung eines Entwurfsmangels benötigt, gegenüber stellt.

Die Skalierbarkeit wird bewertet, indem Abhängigkeiten zwischen den Eingabegrößen und dem Laufzeit- bzw. Speicherbedarf untersucht werden. Stellt man diese gegenüber in einem Koordinatensystem als Punktwolke dar, so lassen sich Abhängigkeitstendenzen ablesen. Diese können, sofern erforderlich, mit Methoden der statistischen linearen und nichtlinearen Korrelationstests geprüft werden.

Zwischen Geschwindigkeit und Speicherbedarf gibt es Gestaltungsspielraum. So ist es nicht nötig alle Trainingsmengen und Entscheidungsbäume permanent im Speicher zu belassen. Die Trainingsmengen könnten ebensogut ausgelagert werden. Sie würden dann bei jeder Verwendung geladen und ein Entscheidungsbaum daraus konstruiert. Dies führt zu erhöhter Laufzeit.

8.4. Werkzeugunterstützung

Zum Zwecke der Evaluierung wurde ein Werkzeug für Java-Programme implementiert. Es wurde dabei auf umfangreiche bestehende Bibliotheken und Werk-

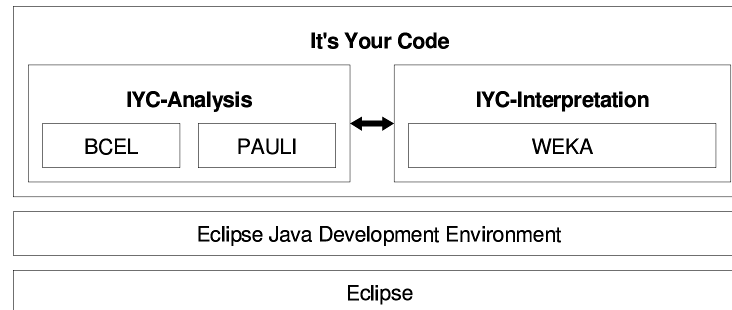

Abbildung 8.15.: Übersicht der Architektur von IYC.

zeugsysteme zurückgegriffen, sodass ein verwendbares Analysewerkzeug für Entwurfsmängel entstand.

8.4.1. Das Werkzeug IYC

Das Werkzeug *It's Your Code* (IYC) wurde als *Plugin* für die Entwicklungsumgebung *Eclipse* konzipiert. Der Name wurde gewählt, um nicht den Eindruck zu erwecken, es handle sich um ein Werkzeug, das als unfehlbares Orakel auftritt. Sofern der Benutzer das Werkzeug selber trainiert, reflektiert es die eigenen Ideen, Erfahrungen und das Anwendungsgebiet des Benutzers. Dieser analysiert und begutachtet somit sein Programm aus eigenen Gesichtspunkten.

Analysegegenstand ist Java-Bytecode. Dieser enthält alle nötigen Informationen, die zum Aufbau des Programmabhängigkeitsgraphen und weiterer statischer Programmanalysen benötigt werden. Die Analyse des Java-Quelltextes hätte den Nachteil, dass die Analyse auf die Programmteile beschränkt wäre, die im Quelltext vorliegen und zudem vorweggenommene Analysen des Übersetzers zusätzlich durchgeführt werden müssten; z. B. die Namensanalyse. Die analysierten Softwaresysteme verwenden i. d. R. die Java-Laufzeitbibliothek und andere Bibliotheken. Diese liegen nicht im Quelltext vor. Würde man die Informationen über die Benutzung dieser Programmteile ausblenden, ergäbe sich ein verzerrtes Bild der Nutzungsbeziehungen.

8.4.2. Architektur

Abbildung 8.15 zeigt einen Überblick über die Architektur des IYC-Werkzeuges. Eingebettet in *Eclipse* lässt es sich aber auch unabhängig davon betreiben. In der eingebetteten Form dient die *Eclipse*-Plattform mit der Java-Entwicklungskomponente als Basis.

IYC besteht dann aus den Teilen *IYC-Analysis*, der die statische Programmanalyse von Java-Bytecode implementiert, und *IYC-Interpretation*, der Mechanismen zur Konfigurationsverwaltung und Steuerung des Lernverfahrens enthält.

Eclipse [28] ist frei verfügbar und gilt als Referenz moderner Entwicklungsumgebungen. Wie viele andere Umgebungen auch bietet *Eclipse* neben Projekt-, Versions- und *Build*-Verwaltung auch Quelltexteditoren und verschiedene Sichten auf das gerade geladene Projekt. In solchen Sichten navigiert man z. B. in der Dateistruktur des Projektes oder am Beispiel Java in der Paket- und Klassenstruktur.

Das *IYC*-Modul klinkt sich in die Benutzerschnittstelle solcher Sichten ein. Bei Auswahl eines Programmobjektes, z. B. eines Paketes oder einer Klasse, kann im Kontextmenü die Suche nach Entwurfsmängeln gestartet werden.

Als maschinelles Lernverfahren wird der *J48*-Klassifizierer — eine spezielle *C4.5*-Implementierung [73] — aus der *WEKA*-Bibliothek [90] eingesetzt.

IYC-Analysis verwendet die *Bytecode Engineering Library* (BCEL) [24] [3], um auf den *Bytecode* des zu analysierenden Java-Programms zuzugreifen. *PAULI* [85] ist eine Bibliothek zur Programmanalyse von Java-*Bytecode*. Diese implementiert z. B. benötigte Kontroll- und Datenflussanalysen.

In einer XML-notierten Datenstruktur wird die Konfiguration des Werkzeugs gespeichert (die Syntaxbeschreibung ist in Anhang C zu finden). Diese enthält einen Definitionsteil, indem alle implementierten Attribute/Metriken aufgeführt sind. Zudem wird für jeden Entwurfsmangel definiert, welche Attribute verwendet werden. Außerdem enthält die Konfiguration alle Trainingsmengen, aus denen die Entscheidungsbäume konstruiert werden. Ebenfalls enthalten sind Datenstrukturen, die zur Kommunikation zwischen Interpretations- und Analyseteil dienen. Der Analyse wird das zu analysierende Programmobjekt und der Analysekontext mitgeteilt. Die Analyse liefert die berechneten Metriken zurück.

8.4.3. Benutzung

IYC ist vorwiegend für den interaktiven Betrieb konzipiert worden; für die Evaluation können aber auch automatische Läufe durchgeführt werden.

Interaktiver Betrieb

Abbildung 8.16 zeigt ein Bildschirmfoto von *Eclipse* mit der Anzeige des Quelltextes im rechten oberen Bereich, einer Sicht zur Navigation in der Java-Programmstruktur im linken Bereich und einer Liste von gefundenen Entwurfsmängeln im Bereich rechts unten.

Die IYC-Funktionen sind an den jeweils üblichen Stellen der *Eclipse*-Oberfläche integriert. Im Einzelnen sind dies Konfigurationsdialoge, Wahl einer Programmstelle und Durchführung von Training und Suche sowie Anzeige von vorgeschla-

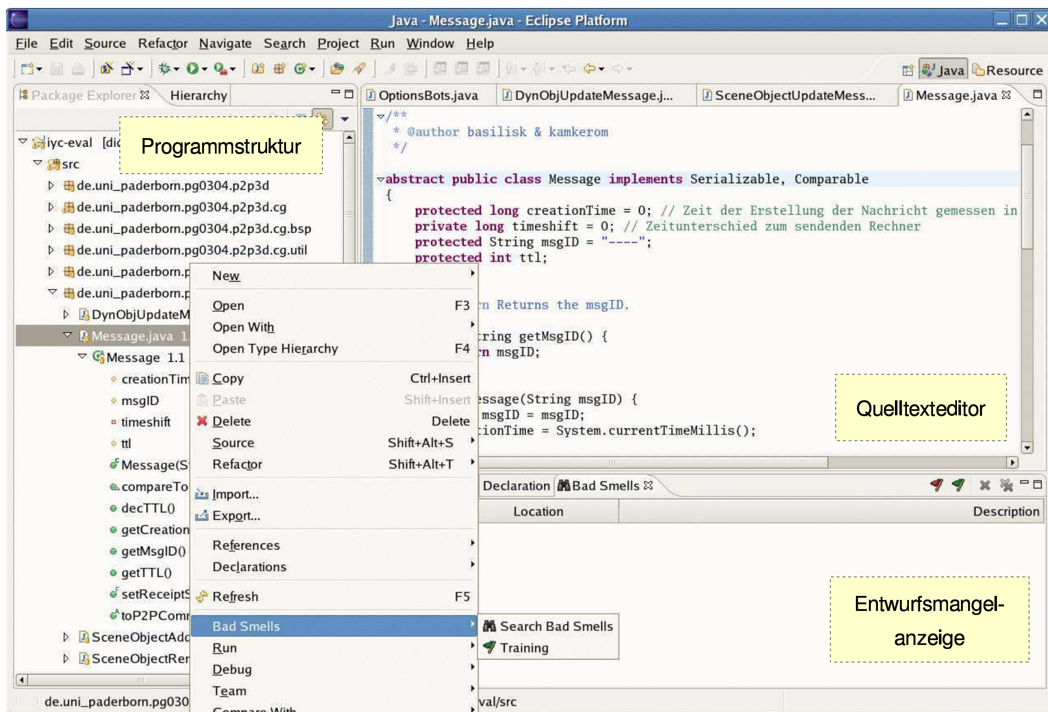

Abbildung 8.16.: Bildschirmfoto der Benutzeroberfläche von *Eclipse*

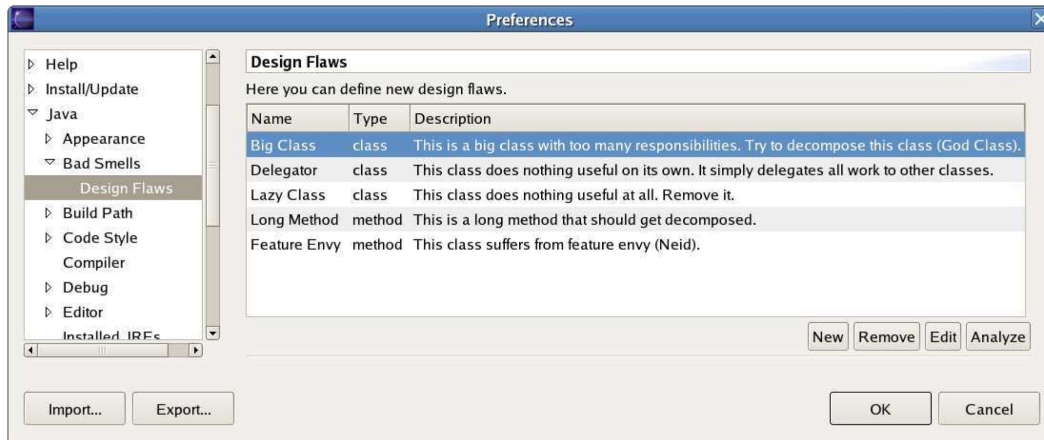

Abbildung 8.17.: Konfiguration von Entwurfsmängeln in IYC

genen Entwurfsmängeln.

Konfiguration

Innerhalb des allgemeinen *Preferences*-Baumes von *Eclipse* befindet sich ein Ast für Java-spezifische Einstellungen. Hier wurde ein Zweig für *Bad Smells* ergänzt.

Abbildung 8.17 zeigt den Dialog, der alle modellierten Entwurfsmängel aufführt. Jedem ist die Art der Programmstelle und ein Kommentar zugeordnet. Weitere können ergänzt oder nicht benötigte entfernt werden. Die Schaltfläche „Analyse“ führt zur Anzeige von statistischen Informationen über die Güte des Lernverfahrens und des aktuellen Entscheidungsbaumes, sowie zur Statistik der Attributnutzung. Diese Informationen können zur Modellreflexion genutzt werden.

Man kann mit mehreren parallelen IYC-Konfigurationen arbeiten. Die jeweilige XML-Datei kann geladen, verwendet und unter anderen Namen gespeichert werden. Der Konfigurationdialog hierzu ist in der Abbildung nicht zu sehen.

Abbildung 8.18 zeigt die Modellierung eines einzelnen Entwurfsmangels. Je nach zugeordneter Art einer Programmstelle, stehen verschiedene Mengen von vorgefertigten Attributen zur Verfügung, die zur Modellierung verwendet werden können.

Die Menge der Attribute ist vorgegeben und entspricht den aktuell implementierten Metriken der statischen Programmanalyse. Jedem Attribut ist ein Typ und eine Beschreibung zugeordnet. Als Typen sind primitive Typen, wie ganze Zahlen und Fließkommazahlen und boolsche Werte, und ein komplexer Aufzählungstyp erlaubt.

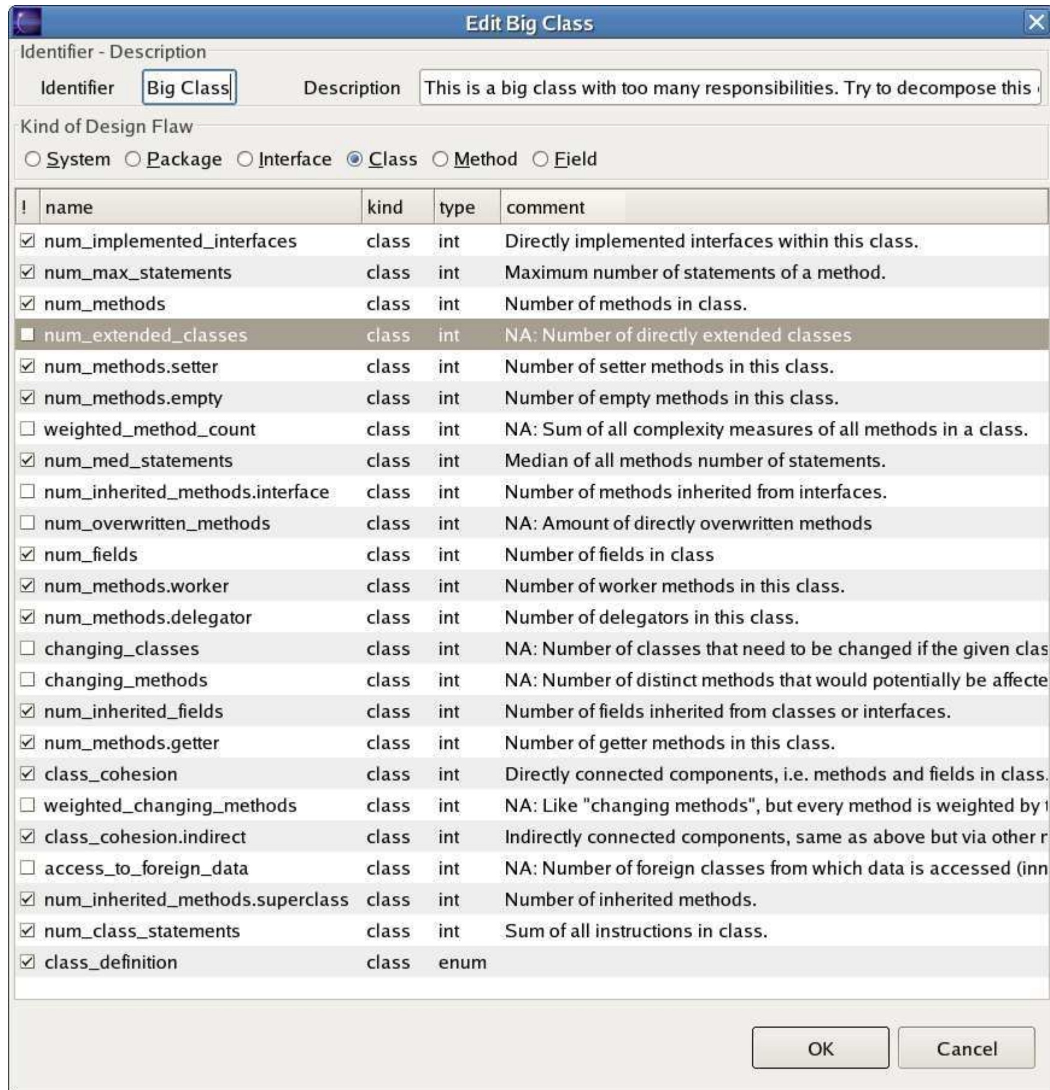

Abbildung 8.18.: Konfiguration und Modellierung eines Entwurfsmangels in IYC

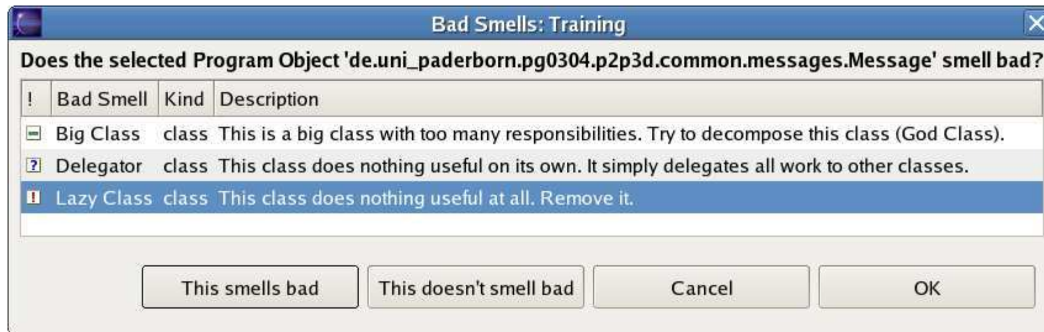

Abbildung 8.19.: Training einer Programmstelle in IYC

Training und Suche

In Abbildung 8.16 sah man bereits ein geöffnetes Kontextmenü. Hier wurde in der Ansicht der Programmstruktur eine Klasse ausgewählt. Im Menü stehen nun Suche und Training von Entwurfsmängeln zur Verfügung.

Betätigt man die Suche, so werden das gewählte Programmobjekt und alle enthaltenen Programmobjekte analysiert. Die Programmanalyse liefert die resultierenden Metrikwerte aller Programmobjekte, sodass alle Klassifizierer, die der jeweiligen Programmobjektart zugeordnet sind, diese Werte interpretieren. Werden Entwurfsmängel gefunden, so erscheinen sie in der Liste der Entwurfsmangelvorschläge.

Betätigt man das Training, so erscheint der Dialog aus Abbildung 8.19 er zeigt alle konfigurierten Entwurfsmängel, deren Art gleich der Art der einzelnen gewählten Programmstelle ist. Der Benutzer legt für jeden Mangel fest, ob dieser hier vorliegt oder er enthält sich. Die Programmstelle wird als Trainingsinstanz bei jedem Mangel aufgenommen, zu denen eine Entscheidung getroffen wurde.

Entwurfsmangelanzeige

Entwurfsmängel werden ähnlich wie Fehlermeldungen des Übersetzers angezeigt. Abbildung 8.16 zeigt unten rechts ein Feld, in dem eine Liste angezeigt wird, die jeweils den Bezeichner des Entwurfsmangels, den Namen der Programmstelle und eine ggf. optionalen Kommentar enthält. Der Kommentar zu einer Programmstelle kann aus der statischen Programmanalyse stammen. Diese liefert zusätzliche Erklärungen, wenn besonders auffällige Programmstrukturen entdeckt werden.

Aktiviert (Doppelklick) der Benutzer einen Eintrag, so zeigt die Entwicklungsumgebung die betreffende Programmstelle an. Der Benutzer ist dann aufgefordert diese manuell zu inspizieren.

Der Benutzer ignoriert den Fund, indem er nichts weiter unternimmt. Andernfalls äußert er sich jeweils zustimmend oder ablehnend zu einem Eintrag. Hierzu stehen Schaltflächen und Kontextmenüs zur Verfügung.

Stapelbetrieb
Zu Zwecken der Evaluation war es notwendig neben der interaktiven Benutzung auch automatische Trainings- und Suchphasen ablaufen zu lassen.

Gesteuert werden diese Abläufe durch die XML-Konfigurationsdatei. Fehlen Metrikwerte beim Laden der Datei, so werden diese nachträglich berechnet und ergänzt. Fehlt zusätzlich das Zielattribut, so wird der Klassifizierer nach der Trainingsphase gebeten dieses zu bestimmen.

Damit lassen sich beliebige Trainings- und Vergleichsinstanzen zusammenstellen und auswerten.

8.5. Fallstudie

Im Rahmen dieser Arbeit standen weder genügend Zeit noch geeignete Personen zur Verfügung um eine umfangreiche empirische Studie durchzuführen.

Zur Bewertung des Ansatzes wurde eine Fallstudie durchgeführt. Hier werden die in Abschnitt 8.3 beschriebenen Methoden zur Leistungsmessung und -bewertung eingesetzt um einen Teil der Bewertungskriterien aus Abschnitt 8.2 zu bestimmen.

Im Folgenden werden zunächst die Untersuchungsteilnehmer und der Analysegegenstand vorgestellt. Im Rahmen der Voruntersuchung wird ein Referenzkatalog erstellt, der für die folgenden Analysen genutzt wird. Die Fallstudie endet mit den Ergebnissen der Reproduktions- und Transferleistung.

8.5.1. Untersuchungsteilnehmer

Im Rahmen dieser Fallstudie konnten zwei Untersuchungsteilnehmer gewonnen werden, die unabhängig voneinander eine vorgegebenes Software-System manuell inspizierten und nach Entwurfsmängeln durchsuchten.

Nach Auswertung des Fragebogens (siehe Abb. 8.1) ergeben sich Gemeinsamkeiten aber auch Unterschiede der Vorkenntnisse:

Beide kannten das Konzept Entwurfsmangel und haben es bereits häufig eingesetzt. In einer Diskussionsphase erkannten beide, dass sie eine sehr ähnliche Sichtweise haben und jeweils bereits mehr als 40 Mängel entdeckt und behoben hatten.

Als der wesentliche Unterschied zwischen den beiden Teilnehmern ergab sich, dass der Eine das zu untersuchende Software-System bereits kannte und mitentwickelt hatte, wobei dem Anderen das System unbekannt war.

Kenngröße	Wert
Anzahl Zeilen	11502
Anzahl Pakete	13
Anzahl der Klassen	117
Anzahl der Methoden	1023
Anzahl der Attribute	720

Tabelle 8.8.: Kenngrößen des Analysegegenstandes.

8.5.2. Analysegegenstand

Der gewählte Analysegegenstand war das Software-System „P2P-3D-Rendering" — ein verteiltes System zu Berechnung von dreidimensionalen Bildern.

Dieses System wurde ausgewählt, weil deren einzelne Teile sehr unterschiedliche Themengebiete implementieren. Neben algorithmischen Anteilen zur Berechnung von 3D-Szenen, sind Anteile zur graphischen Benutzerschnittstelle, Netzwerkkommunikation und komplexe Datenstrukturen implementiert worden. Eine kleine Test-Umgebung in Form von *JUnit*-Tests ist zudem vorhanden.

Um den Quelltext beurteilen zu können, mussten die Teilnehmer sich in den Quelltext einlesen. Da das System nicht besonders groß ist, gab es die Chance hierzu. Die wichtigsten Kenngrößen des Systems sind in Tabelle 8.8 dargestellt.

8.5.3. Ergebnisse der Voruntersuchung

Das Programm wurde von beiden Teilnehmern inspiziert und nach Entwurfsmängeln durchsucht. Um zu einer gemeinsamen Bewertung zu gelangen wurde das *Thurstone*-Verfahren eingesetzt (siehe Abschnitt 8.3.1.2, S. 102). Für jeden gefundenen Entwurfsmangel wurde auf einer Skala von $1 - 5$ festgehalten, wie sicher sich die Person ist. 1 bedeutet, dass der Mangel ganz sicher vorliegt; 5 bedeutet, dass der Mangel sicher nicht vorliegt. Wenn sich die Person unsicher ist, kann 3 gewählt werden.

Die Tabelle 8.9 zeigt die Anzahl der gefunden Mängel nachdem beide Urteile kumuliert wurden. Dabei wurde ein Entwurfsmangel angenommen, wenn er mit einer Sicherheit von ≤ 2 bestimmt wurde.

8.5.4. Ergebnisse der Effektivitätsmessungen

Die Effektivitätsmessung wurde gemäß Abschnitt 8.3.2 vorgenommen.

Dabei wurden für die Reproduktionsleistung jeweils alle Programmstellen dem Verfahren bekannt gemacht und im zweiten Schritt zur Klassifizierung erneut vor-

Entwurfsmangel	Anzahl
„Große Klasse“	20
„Faule Klasse“	28
„Lange Methode“	119
„Neid“	78
„Vermittler“	9

Tabelle 8.9.: Anzahl der manuell erkannten Entwurfsmängel des Analysegegenstandes.

gelegt.

Für die Transferleistung wurde die Trainingsmenge in 10 Teilemengen zerlegt und jeweils eine Teilmenge der Trainingsmenge ausgenommen und nicht dem Verfahren bekannt gemacht. Anschließend wurden die unbekannten Programmstellen automatisch beurteilt. Dies wurde für alle Teilmengen wiederholt und die Ergebnisse kumuliert (siehe Abschnitt 8.3.2.1).

Analog zur manuellen Beurteilung von Programmstellen wurden das Lernverfahren beauftragt in den Stufen $1-5$ zu klassifizieren. Dies erlaubt es, die ausgewiesenen Programmstellen entsprechend zu sortieren und somit die am sichersten gefundenen Entwurfsmängel zuerst zu benennen.

Die Abbildungen 8.21 bis 8.27 zeigen jeweils die gemessene Reproduktions- und Transferleistung für einzelne Entwurfsmängel. Trotz der kleinen Trainingsmengen (vergleiche Tabelle 8.8 und Tabelle 8.9) sind sehr gute Erkennungsleistungen zu erkennen. Dies ist ein ermutigendes Ergebnis.

Der Entwurfsmangel „Vermittler“ wurde wegen der geringen Anzahl gefundener Programmstellen in dieser Auswertung nicht weiter betrachtet.

8.6. Zusammenfassung

In diesem Kapitel wurden Kriterien beschrieben, die geeignet sind, den gewählten Ansatz zu evaluieren. Neben der Effektivität, welche die Erkennungsleistung misst, spielt die Effizienz für den Einsatz in einem interaktiven Werkzeug eine Rolle.

Für die Effektivitätsmessungen wurden Methoden aus der empirischen Sozialforschung vorgeschlagen. Diese erlauben es, die Ergebnisse eines automatischen Werkzeugs anhand der manuellen Beurteilung eines Expertengremiums zu bewerten.

Die Fallstudie hat in einem kleinen Rahmen gezeigt, dass das Verfahren geeignet ist auch in unbekannten Programmstellen Entwurfsmängel zu finden. Eine ab-

schließende Bewertung steht noch aus und konnte im Rahmen dieser Arbeit nicht erbracht werden.

Ein wesentlicher offener Punkt ist dabei die Frage nach dem Nutzen von Modellen für einzelne Entwurfsmängel. Man könnte auch den reichhaltigen Fundus von Metriken in der Literatur verwenden und das Lernverfahren alleine entscheiden lassen, wie Entwurfsmängel erkannt werden. Gegen diesen Ansatz spricht allerdings die Sprachvorliebe, die durch Modelle von Entwurfsmängeln gebildet werden. Außerhalb dieses Korsetts kann das Lernverfahren keine Entwurfsmängel finden.

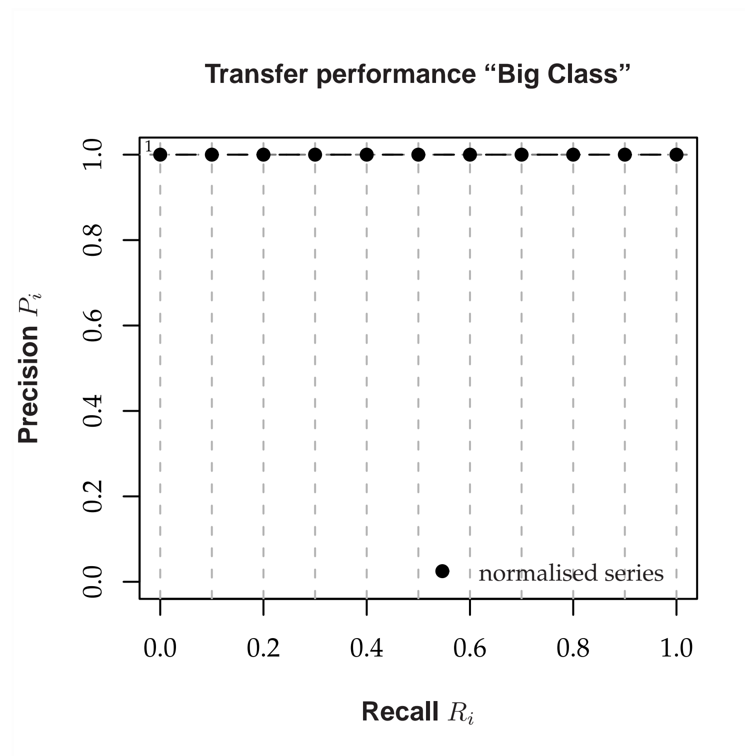

Abbildung 8.20.: Messung der Reproduktionsleistung für die „Große Klasse".

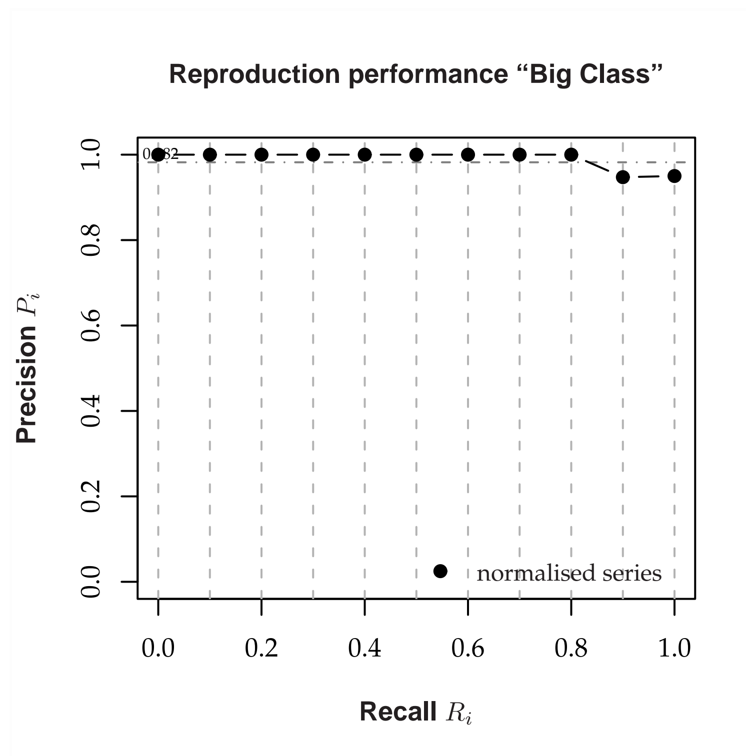

Abbildung 8.21.: Messung der Transferleistung für die „Große Klasse“.

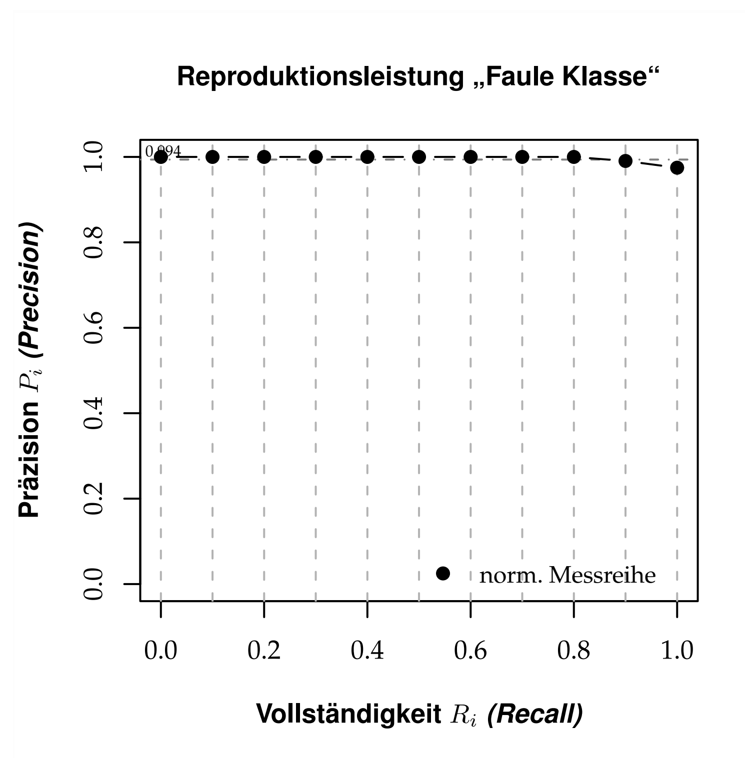

Abbildung 8.22.: Messung der Reproduktionsleistung für die „Faule Klasse“.

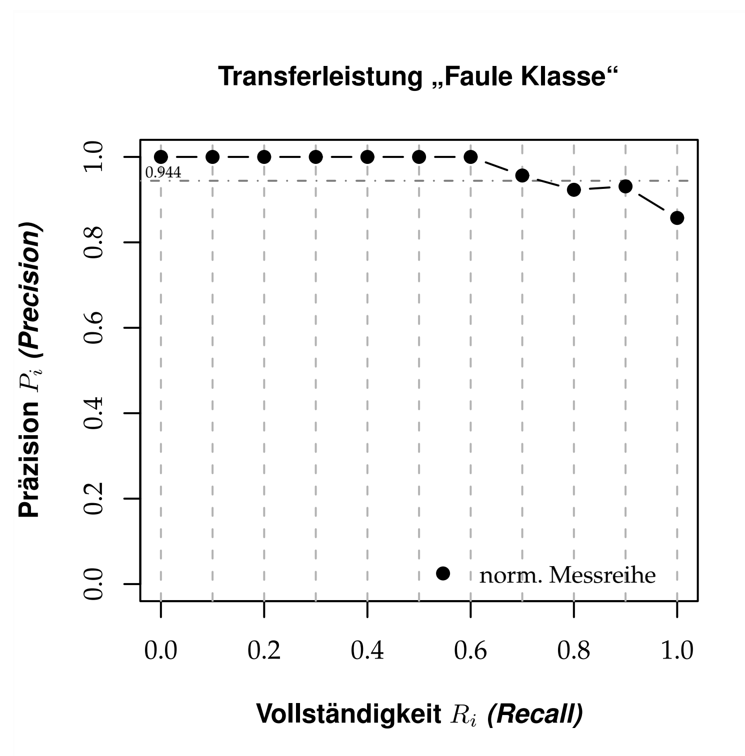

Abbildung 8.23.: Messung der Transferleistung für die „Faule Klasse“.

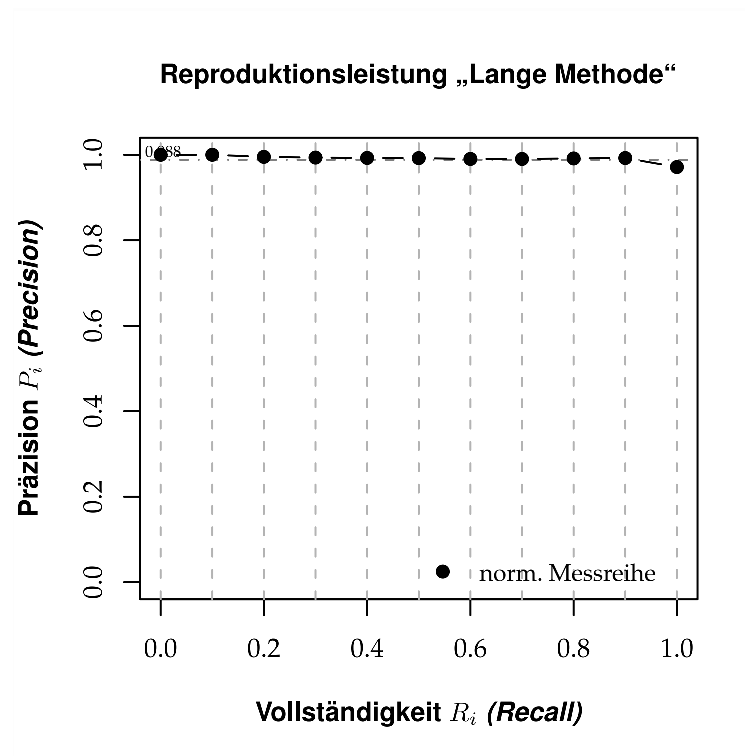

Abbildung 8.24.: Messung der Reproduktionsleistung für die „Lange Methode“.

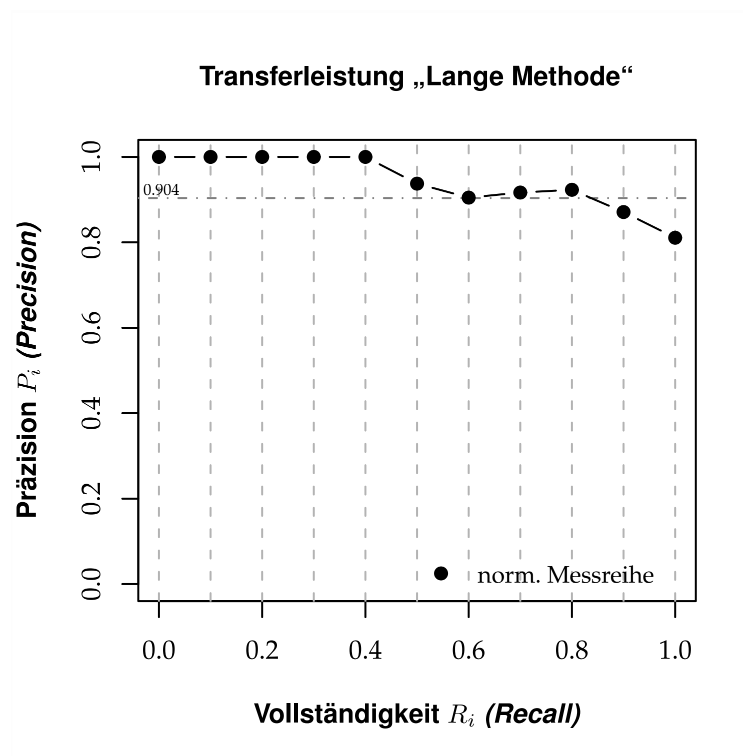

Abbildung 8.25.: Messung der Transferleistung für die „Lange Methode“.

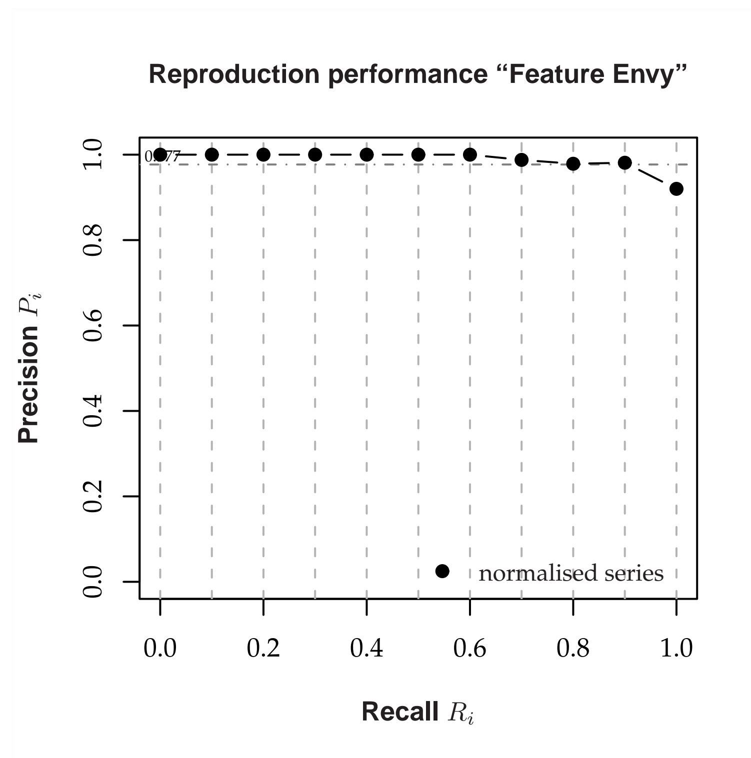

Abbildung 8.26.: Messung der Reproduktionsleistung für die „Neid“.

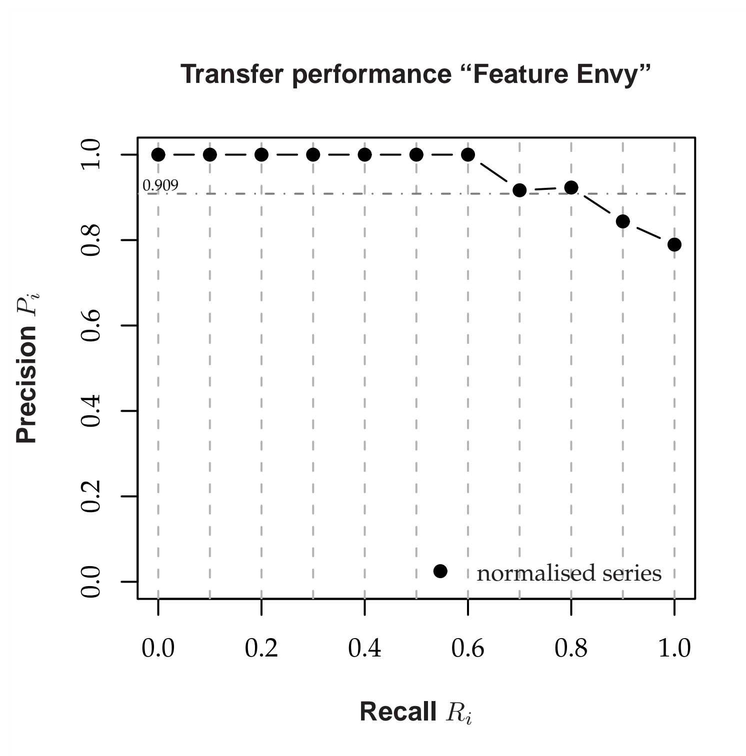

Abbildung 8.27.: Messung der Transferleistung für die „Neid“.

9. Fazit

Inhalt

9.1. Ausgangspunkt 150
9.2. Beiträge und Ergebnisse 151
9.3. Ausblick 152

9.1. Ausgangspunkt

Der Ausgangspunkt dieser Arbeit ist die Konstruktion großer Software-Systeme. Deren Wartung und Weiterentwicklung geschieht im Team. Da niemand den vollständigen Überblick über das gesamte System hat, ist es nötig klare Programmstrukturen einzusetzen, die sich an bekannten Konzepten orientieren. Ein Beispiel sind Entwurfsmuster. Diese erleichtern die Kommunikation innerhalb der Entwicklergruppe und schärfen die Sprechweise über das Software-System. Das Wissen und die effektive Kommunikation innerhalb des Teams ist ihre größte Leistung.

Entwurfsmuster, Schichten- und Komponentenarchitekturen prägen dieses Kommunikationsgerüst schon heute. Es finden aber auch immer mehr Muster für Architekturfehler und Entwurfsmängel Einzug in die klassischen Gespräche auf den Fluren und in den Kaffeeküchen.

Entwurfsmängel beschreiben den intuitiven Eindruck — ein „Bauchgefühl" — das Software-Entwickler und -Architekten dazu treibt, einen als problematisch erkannten Ausschnitt des Systems zu überarbeiten. Hierfür stehen erprobte Mittel zur Verfügung. Durch *Refactoring*-Transformationen lassen sich mit geringem Risiko spröde Programmstrukturen glätten und für veränderte Anforderungen anpassen oder auf zukünftige Pläne vorbereiten. Testbibliotheken dämpfen des Risiko zusätzlich ab. Software-Systeme sind aber nicht nur Programmtext, sondern auch in erheblichem Umfang zusätzliche Information: Datenbankschemata, Konfigurationsdateien und Spezifikationsdokumente. Dies alles im Einklang zu behalten ist Aufgabe der Software-Ingenieure. Ihr Gespür, ihre Erfahrung und ihr Urteilsvermögen bildet das Rückgrat eines Software-Projektes.

Werkzeuge leisten hier wertvolle Beiträge zur Qualitätssicherung. Automatische Testbibliotheken prüfen, ob bereits vorhandene Eigenschaften unverändert erhalten bleiben. Werkzeuge zur Programmanalyse ermitteln Speicherlöcher oder finden Muster gängiger Programmierfehler. Es liegt daher nahe, neben der Ebene der Programmierung auch den Entwurf und die Architektur zu bewerten.

Erich Gamma erwiderte einmal nach einem Vortrag auf die Frage, ob der Einsatz von Entwurfsmustern zum Projekterfolg führt, dass er genau so viele gescheiterte Projekte gesehen habe, die nie etwas von Entwurfsmustern gehört hatten, wie Projekte, die diese intensiv einsetzten. Guter Entwurf ist etwas sehr Individuelles, das sich inzwischen in Referenzarchitekturen für spezielle Anwendungsgebiete niederschlägt.

Bei der Suche nach Entwurfsmängeln hilft es daher nur wenig, konkrete Programmstrukturen zu untersuchen. Eine solche Nahaufnahme lässt den Suchenden am sprichwörtlichen „Wald mit lauter Bäumen" verzweifeln. Erst das Portrait macht Zusammenhänge sichtbar.

9.2. Beiträge und Ergebnisse

In dieser Arbeit habe ich einen neuen Ansatz vorgestellt, mit dem Entwurfsmängel in bestehenden Software-Systemen gefunden werden können. Der Ansatz kombiniert Techniken der statischen Programmanalyse, der objektorientierten Entwurfsmetriken und des maschinellen Lernens. Ein wesentlicher Aspekt ist die Anpassungsfähigkeit des Verfahrens, sowohl an spezielle Wissensgebiete, als auch an Vorstellungen und Erfahrungen von Personen. Für die Evaluation habe ich daher Verfahren der empirischen Sozialforschung vorgeschlagen.

Im Einzelnen enthält diese Arbeit folgende Beiträge:

- Ich beschreibe ein Verfahren, um von vagen Symptom-Beschreibungen zu konkreten messbaren Programmstrukturen zu gelangen. Daraus ergibt sich für jeden Entwurfsmangel ein Modell.
- Für einige Entwurfsmängel habe ich Beispielmodelle entwickelt, um den Ansatz zu evaluieren.
- Ich setze Verfahren der Programmanalyse ein um ein Portrait des Systems zu zeichnen. Diese Art der Analyse beruht auf einfachen Techniken der statischen Programmanalyse. Als Repräsentation der Analyseergebnisses konstruiere ich einen Programmabhängigkeitsgraphen und verwende die relationalen Algebra um diesen auszuwerten. Mit diesen Mitteln lässt sich einfach das vollständige Spektrum der Entwurfsmetriken berechnen.
- Maschinelle Lernverfahren, hier die Konstruktion von Entscheidungsbäumen, setze ich als Werkzeug ein, um lose gekoppelte Information (hier Messwerte von Metriken) zu Wissen zu verdichten. Allerdings kann ein Lernverfahren dies alleine nicht leisten. Konsolidiertes Wissen entsteht erst durch eine zusätzliche empirische Untersuchungen, die den Ansatz „im Großen“ bewertet.
- Ich habe Verfahren der empirischen Sozialforschung für die Evaluierung des Ansatzes vorgeschlagen. Es fand eine erste Validierung anhand einer überschaubaren Fallstudie statt.

Die Fallstudie hat für die dort betrachteten Entwurfsmängel gezeigt, dass sehr guten Erkennungsleistungen erzielt werden können. Gleichwohl ist zu beachten, dass eine umfangreiche empirische Untersuchung aussteht.

Der Nutzen des Verfahrens ist vielfältig. Auch unerfahrene Entwurfsmangel-Suchende können geeignete Modelle und Entscheidungsbäume zur Erkennung eigener Entwurfsmängel erstellen. Durch das eingesetzte Lernverfahren wird automatisch Expertenwissen aus Beispielen konstruiert. Fehlerhafte oder unvollständi-

ge Modellierung eines Entwurfsmangels kann durch Analyse des Entscheidungsbaumes erkannt und nachträglich korrigiert werden.

Wiederholtes und zeitsparendes Anwenden der automatischen Erkennung erlaubt nachhaltige Qualitätsverbesserung eines Software-Systems. Dabei ist die automatische Anpassung nicht zwingend nötig. Nach einer Trainingsphase kann auch der Zustand gesichert und für ein Software-Projekt verbindlich festgelegt werden. Die Abwesenheit von Entwurfsmängeln kann somit als Qualitätsziel in Software-Projekten vereinbart werden.

Prinzipiell kann das Verfahren nicht nur auf objektorientierte Programme angewandt werden. Das vorgestellte Werkzeug ist allerdings auf Java-Programme spezialisiert; der Anschluss von Extraktoren für Programmeigenschaften und Metriken anderer Sprachen ist grundsätzlich möglich.

Ebenso ist der Ansatz nicht nur auf Entwurfsmängel beschränkt. Auch andere Programmeigenschaften, die auf vagen Ideen beruhen, sind auf diesem Weg beschreibbar. Hierzu gehören auch viele Kriterien eines allgemeinen Software-Qualitätsbegriffs, wie man sie im ISO-Standard 9126 [80] findet.

9.3. Ausblick

Aus der Blickrichtung der Empirie sind einige Fragen offen geblieben: Welchen Einfluss hat das Anwendungsgebiet auf die Erkennung von Entwurfsmängeln? Sind Entwurfsmängel ein allgemeines Konzept oder bildet jedes Anwendungsgebiet seine eigene Vorstellung von Entwurfsmängeln aus? Wie groß ist der Einfluss des lehrenden Benutzers im Vergleich zum Einfluss des Entwurfsmangelmodells?

Für ergänzende Untersuchungen könnten interessierte Freiwillige beteiligt werden. Hierzu würde das prototypische Werkzeug weiterentwickelt und öffentlich zugänglich gemacht. Eine Online-Komponente könnte es erlauben, Modelle und Entscheidungsbäume verschiedener Nutzer (anonymisiert) zu sammeln. Ein Nutzerprofil in Form einer Befragung (siehe Abschnitt 8.3.1.1) könnte Aufschluss darüber geben, ob und welche Einflussfaktoren sich in unterschiedlichen Ausprägungen von Entwurfsmängeln widerspiegeln. Auf dieser Basis könnte eine „Lerngruppe" in Form eines öffentlicher Kataloges von Entwurfsmängeln, bestehend aus Modellen und Trainingsmengen bzw. Entscheidungsbäumen, entstehen.

Sobald Entwurfsmängel gefunden werden, sollten diese behoben werden. Hierzu können geeignete *Refactoring*-Transformationen eingesetzt werden. Ein Werkzeug könnte Hinweise liefern, welche und wo Transformationen sinnvoll eingesetzt werden könnten. Hierzu könnte man Entwickler bei dieser Tätigkeit beobachten und zusammen mit zuvor gefundenen Entwurfsmängeln und jeweiligen Programmeigenschaften protokollieren. Ein Lernverfahren könnte dies Beobachtungen erlernen um später selber geeignete Transformation vorzuschlagen.

Anhang A.

„Design Heuristics“ nach Riel

Die folgenden Merksätze fassen die Ideen zum „guten“ objektorientierten Entwurf von Riel [74] zusammen.

Chapter 2 *Classes and Objects: The Building Blocks of the Object-Oriented Paradigm*

2.1 All data should be hidden within its class.

2.2 Users of a class must be dependent on its public interface, but a class should not be dependent on its users.

2.3 Minimize the number of messages in the protocol of a class.

2.4 Implement a minimal public interface which all classes understand (e.g. operations such as copy (deep versus shallow), equality testing, pretty printing, parsing from a ASCII labeling, etc.).

2.5 Do not put implementation details such as common-code private functions into the public interface of a class.

2.6 Do not clutter the public interface of a class with things that users of that class are not able to use or are not interested in using..

2.7 Classes should only exhibit nil or export coupling with other classes, i.e. a class should only use operations in the public interface of another class or have nothing to do with that class.

2.8 A class should capture one and only one key abstraction.

2.9 Keep related data and behavior in one place.

2.10 Spin off non-related information into another class (i.e. non-communicating behavior).

2.11 Be sure the abstraction that you model are classes and not simply the roles objects play.

Chapter 3 Topologies of Action-Oriented Vs. Object-Oriented Applications

3.1 Distribute system intelligence horizontally as uniformly as possible, i.e. the top level classes in a design should share the work uniformly.

3.2 Do not create god classes/objects in your system. Be very suspicious of an abstraction whose name contains Driver, Manager, System, or Subsystem.

3.3 Beware of classes that have many accessor methods defined in their public interface, many of them imply that related data and behavior are not being kept in one place.

3.4 Beware of classes which have too much non-communicating behavior, i.e. methods which operate on a proper subset of the data members of a class. God classes often exhibit lots of non-communicating behavior.

3.5 In applications which consist of an object-oriented model interacting with a user interface, the model should never be dependent on the interface. The interface should be dependent on the model.

3.6 Model the real world whenever possible. (This heuristic is often violated for reasons of system intelligence distribution, avoidance of god classes, and the keeping of related data and behavior in one place).

3.7 Eliminate irrelevant classes from your design.

3.8 Eliminate classes that are outside the system.

3.9 Do not turn an operation into a class. Be suspicious of any class whose name is a verb or derived from a verb. Especially those which have only one piece of meaningful behavior (i.e. do not count sets, gets, and prints). Ask if that piece of meaningful behavior needs to be migrated to some existing or undiscovered class.

3.10 Agent classes are often placed in the analysis model of an application. During design time, many agents are found to be irrelevant and should be removed.

Chapter 4 The Relationships Between Classes and Objects

4.1 Minimize the number of classes with which another class collaborates.

4.2 Minimize the number of message sends between a class and its collaborator.

4.3 Minimize the amount of collaboration between a class and its collaborator, i.e. the number of different messages sent.

4.4 Minimize fanout in a class, i.e. the product of the number of messages defined by the class and the messages they send.

4.5 If a class contains objects of another class then the containing class should be sending messages to the contained objects, i.e. the containment relationship should always imply a uses relationship.

4.6 Most of the methods defined on a class should be using most of the data members most of the time.

4.7 Classes should not contain more objects than a developer can fit in his or her short term memory. A favorite value for this number is six.

4.8 Distribute system intelligence vertically down narrow and deep containment hierarchies.

4.9 When implementing semantic constraints, it is best to implement them in terms of the class definition. Often this will lead to a proliferation of classes in which case the constraint must be implemented in the behavior of the class, usually, but not necessarily, in the constructor.

4.10 When implementing semantic constraints in the constructor of a class, place the constraint test in the constructor as far down a containment hierarchy as the domain allows.

4.11 The semantic information on which a constraint is based is best placed in a central third-party object when that information is volatile.

4.12 The semantic information on which a constraint is based is best decentralized among the classes involved in the constraint when that information is stable.

4.13 A class must know what it contains, but it should never know who contains it.

4.14 Objects which share lexical scope, i.e. those contained in the same containing class, should not have uses relationships between them.

Chapter 5 The Inheritance Relationship

5.1 Inheritance should only be used to model a specialization hierarchy.

5.2 Derived classes must have knowledge of their base class by definition, but base classes should not know anything about their derived classes.

5.3 All data in a base class should be private, i.e. do not use protected data.

5.4 Theoretically, inheritance hierarchies should be deep, i.e. the deeper the better.

5.5 Pragmatically, inheritance hierarchies should be no deeper than an average person can keep in their short term memory. A popular value for this depth is six.

5.6 All abstract classes must be base classes.

5.7 All base classes should be abstract classes.

5.8 Factor the commonality of data, behavior, and/or interface as high as possible in the inheritance hierarchy.

5.9 If two or more classes only share common data (no common behavior) then that common data should be placed in a class which will be contained by each sharing class.

5.10 If two or more classes have common data and behavior (i.e. methods) then those classes should each inherit from a common base class which captures those data and methods.

5.11 If two or more classes only share common interface (i.e. messages, not methods) then they should inherit from a common base class only if they will be used polymorphically.

5.12 Explicit case analysis on the type of an object is usually an error, the designer should use polymorphism in most of these cases.

5.13 Explicit case analysis on the value of an attribute is often an error. The class should be decomposed into an inheritance hierarchy where each value of the attribute is transformed into a derived class.

5.14 Do not model the dynamic semantics of a class through the use of the inheritance relationship. An attempt to model dynamic semantics with a static semantic relationship will lead to a toggling of types at runtime.

5.15 Do not turn objects of a class into derived classes of the class. Be very suspicious of any derived class for which there is only one instance.

5.16 If you think you need to create new classes at runtime, take a step back and realize that what you are trying to create are objects. Now generalize these objects into a class.

5.17 It should be illegal for a derived class to override a base class method with a NOP method, i.e. a method which does nothing.

5.18 Do not confuse optional containment with the need for inheritance, modelling optional containment with inheritance will lead to a proliferation of classes.

5.19 When building an inheritance hierarchy try to construct reusable frameworks rather than reusable components.

Chapter 6 Multiple Inheritance

6.1 If you have an example of multiple inheritance in your design, assume you have made a mistake and prove otherwise.

6.2 Whenever there is inheritance in an object-oriented design ask yourself two questions: 1) Am I a special type of the thing I'm inheriting from? and 2) Is the thing I'm inheriting from part of me?

6.3 Whenever you have found a multiple inheritance relationship in a object-oriented design be sure that no base class is actually a derived class of another base class, i.e. accidental multiple inheritance.

Chapter 7 The Association Relationship

7.1 When given a choice in an object-oriented design between a containment relationship and an association relationship, choose the containment relationship.

Chapter 8 Class Specific Data and Behavior

8.1 Do not use global data or functions to perform bookkeeping information on the objects of a class, class variables or methods should be used instead.

Chapter 9 Physical Object-Oriented Design

9.1 Object-oriented designers should never allow physical design criteria to corrupt their logical designs. However, very often physical design criteria is used in the decision making process at logical design time.

9.2 Do not change the state of an object without going through its public interface.

Anhang B.

„Bad Smells“ nach Fowler

Im Folgenden sind die *„Bad Smells“* von Fowler [37] dargestellt. Es sind der Bezeichner, eine Kurzbeschreibung und empfohlene *Refactoring*-Transformationen aufgeführt.

Smells Within Classes

Comments Should only be used to clarify “why” not “what”. Can quickly become verbose and reduce code clarity. (Extract Method, Rename Method, Introduce Assertion)

Long Method The longer the method the harder it is to see what it is doing. (Extract Method, Replace Temp with Query, Introduce Parameter Object, Preserve Whole Object, Replace Method with Method Object)

Long Parameter List Don’t pass in everything the method needs; pass in enough so that the method can get to everything it needs. (Replace Parameter with Method, Preserve Whole Object, Introduce Parameter Object)

Duplicated Code (Extract Method, Pull Up Field, Form Template Method, Substitue Algorithm)

Large Class A class that is trying to do too much can usually be identified by looking at how many instance variables it has. When a class has too many instance variables, duplicated code cannot be far behind. (Extract Class, Extract Subclass)

Type Embedded in Name Avoid redundancy in naming. Prefer “schedule.add(course)” to “schedule.addCourse(course)”. (Rename Method)

Uncommunicative Name Choose names that communicate intent; pick the best name for the time, change it later if necessary. (Rename Method)

Inconsistent Names Use names consistently. (Rename Method)

Dead Code A variable, parameter, method, code fragment, class, etc is not used anywhere (perhaps other than in tests). (Delete the code)

Speculative Generality Don't over-generalize your code in an attempt to predict future needs. If you have abstract classes that aren't doing much use "Collapse Hierarchy". Remove unnecessary delegation with "Inline Class". Methods with unused parameters: "Remove Parameter". Methods named with odd abstract names should be brought down to earth with "Rename Method".

Smells Between Classes

Primitive Obsession Use small objects to represent data such as money (which combines quantity and currency) or a date range object. (Replace Data Value with Object, Replace Type Code with Class, Replace Type Code with Subclasses, Replace Type Code with State/Strategy; If you have a group of fields that should go together, use Extract Class If the primitives occur in a param lists use Introduce Parameter Object When working with an array consider Replace Array With Object).

Data Class Classes with fields and getters and setters and nothing else (aka, Data Transfer Objects - DTO). Move in behavior with Move Method.

Data Clumps Clumps of data items that are always found together. Turn the clumps into an object with "Extract Class". Then continue the refactoring with "Introduce Parameter Object" or "Preserve Whole Object".

Refused Bequest Subclasses don't want or need everything they inherit. The Liskov Substitution Principle (LSP) says that you should be able to treat any subclass of a class as an example of that class. Most of the time that's fine, just don't use what you don't need. Occassionally you'll need to create a new sibling class and use "Push Down Method" and "Push Down Field". The smell is worst if a subclass is reusing behavior but does not want to support the interface of the superclass. In this case use "Replace Inheritance with Delegation".

Inappropriate Intimacy Two classes are overly entertwined. (Move Method, Move Field, Change Bidirectional Association to Unidirectional Association, Extract Class, Hide Delegate, Replace Inheritance with Delegation)

Lazy Class Classes that aren't doing enough should be refactored away. (Collapse Hierarchy, Inline Class)

Feature Envy Often a method that seems more interested in a class other than the one it's actually in. In general, try to put a method in the class that contains most of the data the method needs. (Extract Method, Move Method).

Message Chains This is the case in which a client has to use one object to get another, and then use that one to get to another, etc. Any change to the intermediate relationships causes the client to have to change. (Hide Delegate, Extract Method, Move Method)

Middle Man When a class is delegating almost everything to another class, it may be time to refactor out the middle man. "Remove Middle Man" if only a few methods aren't doing much, use "Inline Method You" could also consider turning the middle man into a subclass with "Replace Delegation with Inheritance".

Divergent Change Occurs when one class is commonly changed in different ways for different reasons. Any change to handle a variation should change a single class. Identify everything that changes for a particular cause and use "Extract Class" to put them all together.

Shotgun Surgery The opposite of "Divergent Change". A change results in the need to make a lot of little changes in several classes. Use "Move Method" and "Move Field" to put all the changes into a single class. Often you can use "Inline Class" to bring a whole bunch of behavior together.

Parallel Inheritance Hierarchies A special case of "Shotgun Surgery". Every time you make a subclass of one class, you also have to make a subclass of another. Use "Move Method" and "Move Field" to combine the hierarchies into one.

Anhang C.

Konfiguration des IYC-Werkzeuges

Die Konfiguration des IYC-Werkzeuges wird in einer XML-Datenstruktur gespeichert. Dessen syntaktischer Aufbau ist hier in Form einer *Document Type Definition* (DTD) [95] aufgeführt.

iyc.dtd

```
<!--
#=============================================================
# IYC Document Type Definition
# Author: Jochen Kreimer <jotte@upb.de>
#=============================================================
-->

<!DOCTYPE iyc SYSTEM "platform:/plugin/de.unipb.iyc_1.0.0/xml/iyc.dtd">

<!-- === Root-Element ============================================ -->

<!ELEMENT iyc (ana_cfg?, ana_con?, ana_res?, ml_cfg?)>

<!-- === Analysis Configuration ================================== -->

<!ELEMENT ana_cfg (attrdefs)>

<!-- === Analysis Context ======================================== -->

<!ELEMENT ana_con (analyse, universe)>
<!ELEMENT analyse (pobj+)>
<!ELEMENT universe (path+)>

<!-- === Analysis Result ========================================= -->

<!ELEMENT ana_res (result*)>
<!ELEMENT result (pobj, attrs)>

<!-- === Machine Learning Configuration ========================== -->

<!ELEMENT ml_cfg (flaw*)>
<!ATTLIST ml_cfg
           name CDATA #REQUIRED
          dest_attr ID #REQUIRED
```

```
35          protocol (yes|no) "no">
   <!ELEMENT flaw (kind, comment, attrs, training?)>
   <!ATTLIST flaw
             name CDATA #REQUIRED>
   <!ELEMENT training (inst*)>
40 <!ELEMENT inst (pobj?,attr*)>

   <!-- =============================================================
            COMMON DEFINITIONS
        ============================================================= -->
45
   <!-- === Attribute Definition ==================================== -->

   <!ELEMENT attrdefs (attrdef*)>
   <!ELEMENT attrdef (kind, type, category?, comment?)>
50 <!ATTLIST attrdef
             name ID #REQUIRED
             available (yes|no) "yes">

   <!-- === Attribute Usage ========================================= -->
55
   <!ELEMENT attrs (attr*)>
   <!ELEMENT attr (comment?)>
   <!ATTLIST attr
              name IDREF #REQUIRED
60           value CDATA #IMPLIED
             expected CDATA #IMPLIED>

   <!-- === Attribute Type/+Properties ============================== -->

65 <!ELEMENT type (enum*)>
   <!ATTLIST type
             name (int|float|boolean|enum) #REQUIRED>
   <!ELEMENT enum EMPTY>
   <!ATTLIST enum
70           name CDATA #REQUIRED>
   <!ELEMENT category EMPTY>
   <!ATTLIST category
             name CDATA #REQUIRED>

75 <!-- === Programobject Definition ================================ -->

   <!ELEMENT pobj (kind?)>
   <!ATTLIST pobj
             name CDATA #REQUIRED>
80
   <!-- === Programobject Kind ====================================== -->

   <!ELEMENT kind EMPTY>
   <!ATTLIST kind
```

```
          name (system|package|class|field|method) #REQUIRED>

<!-- === Comment ================================================= -->

<!ELEMENT comment (#PCDATA)>

<!-- === Path ==================================================== -->

<!ELEMENT path EMPTY>
<!ATTLIST path
          name CDATA #REQUIRED>

<!-- === DONE ==================================================== -->
```

Anhang D.

Verzeichnisse

Literaturverzeichnis

[1] ABREU, F. und R. BRITO: *Objectoriented Software Engineering: Measuring and Controlling the Development Process*, 1994.

[2] ABREU, FERNANDO BRITO E, GEERT POELS, HOUARI A. SAHRAOUI und HORST ZUSE: *Quantitative Approaches in Object-Oriented Software Engineering*. In: J. MALENFANT, S. MOISAN, A. MOREIRA (Herausgeber): *ECOOP 2000 Workshops*, Band LNCS 1964 der Reihe *ECOOP 2000 Workshops*, Seiten 93–103. Springer-Verlag Berlin Heidelberg, 2000.

[3] THE APACHE JAKARTA PROJEKT, `http://jakarta.apache.org/bcel`: *Byte Code Engineering Library (BCEL)*, 2004.

[4] BACON, DAVID FRANCIS: *Fast and Effective Optimization of Statically Typed Object-Oriented Languages*. Doktorarbeit, Mai 17 1997.

[5] BAMBERG, G. und F. BAUR: *Statistik*. Oldenbourg, München, 2001.

[6] BECK, KENT: *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[7] BEYER, DIRK, CLAUS LEWERENTZ und FRANK SIMON: *Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems*. Lecture Notes in Computer Science, 2006:1–??, 2001.

[8] BEYER, DIRK, ANDREAS NOACK und CLAUS LEWERENTZ: *Efficient Relational Calculation for Software Analysis*. IEEE Transactions on Software Engineering, 31(2):137–149, 2005.

[9] BOOCH, GRADY: *Object Oriented Design with Application*. Benjamin/Cummings, Redwood City, 1991.

[10] BORTZ, J.: *Lehrbuch der empirischen Forschung für Sozialwissenschaftler*. Springer-Verlag, Berlin, Heidelberg, New York, Tokyo, 1984.

[11] BRIAND, LIONEL C., SANDRO MORASCA und VICTOR R. BASILI: *Property-Based Software Engineering Measurement*. IEEE Transactions on Software Engineering, 22(1):68–86, Januar 1996.

[12] BRIAND, LIONEL C., SANDRO MORASCA und VICTOR R. BASILI: *Response to: Comments on "Property-Based Software Engineering Measurement: Refining the Additivity Properties"*. IEEE Transactions on Software Engineering, 23(3):196–197, März 1997.

[13] BRÖHL, A. P. und W. DRÖSCHEL: *Das V-Modell - Der Standard für die Softwareentwicklung mit Praxisleitfaden*. Oldenbourg Verlag, München, 1993.

[14] BROWN, WILLIAM J., RAPHAEL C. MALVEAU, HAYS W. "SKIP" MCCORMICK III und THOMAS J. MOWBRAY: *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley, 1998.

[15] BUDD, TIMOTHY: *Understanding object-oriented programming with Java*. Addison-Wesley, Reading, MA, USA, 2000.

[16] CHIDAMBER, SHYAM R. und CHRIS F. KEMERER: *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, 20(6):476–493, Juni 1994.

[17] CINNEIDE, M. und P. NIXON: *Composite refactorings for java programs*, 2000.

[18] CINNEIDE, MEL O und PADDY NIXON: *Program Restructuring to Introduce Design Patterns*. In: *ECOOP Workshops*, Seiten 79–80, 1998.

[19] COAD, P. und M. MAYFIELD: *Java-inspired design: use composition, rather than inheritance*. American Programmer, 10(1):22–31, Januar 1997.

[20] COAD, P., D. NORTH und M. MAYFIELD: *Object Models: Strategies, Patterns, and Applications*. Yourdon Press, Upper Saddle River, 2 Auflage, 1997.

[21] COAD, PETER und EDWARD YOURDON: *Object-Oriented Analysis*. Prentice Hall, London, 2 Auflage, 1991.

[22] CODD, E. F.: *Relational Completeness of Database Sublanguages*. In: RUSTIN, R. (Herausgeber): *Data Base Systems*. Prentice-Hall, New Jersey, 1972.

[23] DAGPINAR, MELIS und JENS H. JAHNKE: *Predicting Maintainability with Object-Oriented Metrics - An Empirical Comparison*. In: DEURSEN, ARIE VAN, ELENI STROULIA und MARGARET-ANNE D. STOREY (Herausgeber): *WCRE*, Seiten 155–164. IEEE Computer Society, 2003.

[24] DAHM, MARKUS: *Byte Code Engineering*. In: *Java-Informations-Tage*, Seiten 267–277, 1999.

[25] DEAN, JEFFREY, DAVID GROVE und CRAIG CHAMBERS: *Optimization of Object-Oriented Programs Using Static Class Hierarchy Analysis.* In: *European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[26] DIJKSTRA, E. W.: *Goto statement considered harmful*. Communications of the ACM, 11(3):147–148, Mai 1968.

[27] DR. BARRY BOEHM, RICARDO VALERDI, JO ANN LANE und A. WINSOR BROWN: *COCOMO Suite Methodology and Evolution*. CrossTalk - The Journal of Defense Software Engineering, April 2005.

[28] ECLIPSE.ORG CONSORTIUM, `http://www.eclipse.org`: *Eclipse.org Main Page*, 2003.

[29] EMDEN, EVA VAN und LEON MOONEN: *Java Quality Assurance by Detecting Code Smells*. In: *Proceedings of the 9th Working Conference on Reverse Engineering*. IEEE Computer Society Press, Oktober 2002.

[30] ERNI, K. und C. LEWERENTZ: *Applying Design-Metrics to Object-Oriented Frameworks*, 1996.

[31] ETZKORN, LETHA, CARL DAVIS und WEI LI: *A Practical Look at the Lack of Cohesion in Methods Metric*. Journal of Object-Oriented Programming, 11(5):27–34, September 1998.

[32] FAGAN, MICAHEL E.: *Advances In Software Inspections*. In: *IEEE Trans. On Softw. Eng., 7 (12)*, Seiten 744–751, 1986.

[33] FAGAN, MICHAEL E.: *Design and Code Inspections and Process Control in the Development of Programs*. Technical Report 00.2763, IBM, Juni 1976.

[34] FENTON, NORMAN: *Software Measurement: A Necessary Scientific Basis*. IEEE Transactions on Software Engineering, 20(3):199–206, März 1994.

[35] FENTON, NORMAN und SHARI LAWRENCE PFLEEGER: *Software Metrics - A Rigorous and Practical Approach*. International Thomson Computer Press, London, 2 Auflage, 1996.

[36] FERBER, R.: *Information Retrieval*. dpunkt.verlag, Heidelberg, 2003.

[37] FOWLER, MARTIN, KENT BECK, JOHN BRANT, WILLIAM OPDYKE und DON ROBERTS: *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[38] GAMMA, ERICH, RICHARD HELM und RALPH JOHNSON UND JOHN VLISSIDES: *Design Patterns: Elements of reusable object-oriented software*. Addison–Wesley, 1986. ISBN: 0–201–63361–2.

[39] GROUP, THE STANDISH: *CHAOS Chronicles V3.0*. Technischer Bericht, Standish Group, `http://www.standishgroup.com/chaos/toc.php`, 2003.

[40] GROVE, DAVID PAUL: *Effective interprocedural optimization of object-oriented languages*. Doktorarbeit, University of Washington, 1998.

[41] HAMMER, CHRISTIAN und GREGOR SNELTING: *An improved slicer for Java*. In: *Proceedings of the ACM-SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, Seiten 17–22. ACM Press, 2004.

[42] HASTIE, T., R. TIBSHIRANI und J. FRIEDMAN: *The Elements of Statistical Learning*. Stats. Springer, 2001.

[43] HENDERSON-SELLERS, BRIAN: *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.

[44] HITZ, MARTIN und BEHZAD MONTAZERI: *Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective*. Software Engineering, 22(4):267–271, 1996.

[45] HOLLANDER, M. und D.A. WOLFE: *Nonparametric Statistical Methods*. John Wiley and Sons, 1973.

[46] *20th International Conference on Software Maintenance (ICSM 2004), 11-17 September 2004, Chicago, IL, USA*. IEEE Computer Society, 2004.

[47] JOHNSON, S.: *Lint, a C Program Checker*, 1978.

[48] KOHLER, GERD, HEINRICH RUST und FRANK SIMON: *Assessment of Large Object-Oriented Software Systems: A Metrics Based Process*. In: DEMEYER, SERGE und JAN BOSCH (Herausgeber): *Object-Oriented Technology (ECOOP'98 Workshop Reader)*, Band 1543, Seiten 250–251. Springer-Verlag, 1998.

[49] KOSCHKE, RAINER und DANIEL SIMON: *Hierarchical Reflexion Models*. In: *10th Working Conference on Reverse Engineering*, Seite 36. IEEE, 2003.

[50] LACHMANN, CARSTEN: *Statische Programmanalyse zur Erkennung von Entwurfsmängeln in Java-Anwendungen*. Diplomarbeit, Universität Paderborn, 2004.

[51] LAKSHMANAN, K. B., S. JAYAPRAKASH und P. K. SINHA: *Properties of Control-Flow Complexity Measures*. IEEE Transactions on Software Engineering, 17(12):1289–1295, Dezember 1991.

[52] LANZA, MICHELE: *Object-Oriented Reverse Engineering — Coarse-grained, Fine-grained, and Evolutionary Software Visualization*. Doktorarbeit, University of Berne, Mai 2003.

[53] LIEBERHERR, KARL J., IAN HOLLAND und ARTHUR J. RIEL: *Object-oriented programming: An objective sense of style*. In: *Object-Oriented Programming, Systems, Languages and Applications*, Nummer 11 in *23*, Seiten 323–334, San Diego, CA, September 1988. A short version of this paper appears in *IEEE Computer Magazine*, June 1988, Open Channel section, pages 78-79.

[54] LORENZ, M. und J. KIDD: *Object-Oriented Software Metrics*. Prentice Hall, Englewood Cliffs, 1994.

[55] MÄNTYLÄ, MIKA, JARI VANHANEN und CASPER LASSENIUS: *Bad Smells - Humans as Code Critics*. In: *ICSM2004* [46], Seiten 399–408.

[56] MARINESCU, RADU: *Measurement and Quality in Object-Oriented Design*. Doktorarbeit, University of Timisoara, 2002.

[57] MARINESCU, RADU: *Detection Strategies: Metrics-Based Rules for Detecting Design Flaws*. In: *ICSM2004* [46], Seiten 350–359.

[58] MCCABE, THOMAS J.: *A Complexity Measure*. In: *Proceedings: 2nd International Conference on Software Engineering*, Seite 407. IEEE Computer Society Press, 1976. Abstract only.

[59] MITCHELL, TOM M.: *Machine Learning*. McGraw-Hill, New York, 1997.

[60] MOHAPATRA, SANJAY und B. MOHANTY: *Defect Prevention through Defect Prediction: A Case Study at Infosys*. In: *ICSM*, Seiten 260–272, 2001.

[61] MORGAN, R.: *Building an Optimizing Compiler*. Digital_Press, 1998.

[62] MUCHNICK, STEVEN S.: *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 2000.

[63] MURPHY, GAIL C., DAVID NOTKIN und KEVIN SULLIVAN: *Software reflexion models: bridging the gap between source and high-level models*. In: *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, Seiten 18–28. ACM Press, 1995.

[64] MYERS, GLENFORD J.: *Software Reliability: Principles and Practices*. John Wiley & Sons, New York, 1976.

[65] MÄNTYLÄ, MIKA, JARI VANHANEN und CASPER LASSENIUS: *A Taxonomy and an Initial Empirical Study of Bad Smells in Code*. In: *ICSM '03: Proceedings of the International Conference on Software Maintenance*, Seite 381, Washington, DC, USA, 2003. IEEE Computer Society.

[66] OPDYKE, WILLIAM F.: *Refactoring Object-Oriented Frameworks*. Doktorarbeit, University of Illinois at Urbana-Champaign, 1992.

[67] OTTENSTEIN, KARL J. und LINDA M. OTTENSTEIN: *The program dependence graph in a software development environment*. ACM SIGPLAN Notices, 19(5):177–184, Mai 1984.

[68] PETROSKI, HENRY: *The Hybris of Extreme Engineering*. Scientific American, 2(3):94–130, April 1999.

[69] PHILIPPS, J. und B. RUMPE: *Roots of Refactoring*, 2001.

[70] POETZSCH-HEFFTER, ARND: *Konzepte objektorientierter Programmierung*. Springer, 2000.

[71] POLO, MACARIO, MARIO PIATTINI und FRANCISCO RUIZ: *Using Code Metrics to Predict Maintenance of Legacy Programs: A Case Study.* In: *ICSM*, Seiten 202–208, 2001.

[72] QUINLAN, J. R.: *Induction of decision trees*. Machine Learning, 1(1):81–106, 1986.

[73] QUINLAN, J. R.: *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[74] RIEL, ARTHUR J.: *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[75] ROMBACH, H. DIETER: *Practical benefits of goal-oriented measurement*. In: *Proceedings of the Annual Workshop of the Centre for Software Reliability*, Seiten 217–235. Elsevier, September 1990.

[76] SALTON, G. und M. J. MCGILL: *Introduction to Modern Information Retrieval*. McGraw-Hill, Tokio, 1983.

[77] SAYYAD SHIRABAD, J. und T.J. MENZIES: *The PROMISE Repository of Software Engineering Databases*. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[78] SIEDERSLEBEN, JOHANNES: *Moderne Softwarearchitektur*. dpunkt Verlag, 2004.

[79] SIMON, FRANK, FRANK STEINBRUCKNER und CLAUS LEWERENTZ: *Metrics Based Refactoring*. In: *CSMR*, Seiten 30–38, 2001.

[80] STANDARDIZATION, ISO INTERNATIONAL ORGANIZATION FOR: *ISO/IEC TR 9126-1: Software engineering, Product quality, Part 1: Quality model*, Jun 2001.

[81] STANDARDIZATION, ISO INTERNATIONAL ORGANIZATION FOR: *ISO/IEC TR 9126-2: Software engineering, Product quality, Part 2: External metrics*, July 2003.

[82] STOREY, MARGARET-ANNE D., KENNY WONG und HAUSI A. MULLER: *Rigi: A Visualization Environment for Reverse Engineering*. In: *International Conference on Software Engineering*, Seiten 606–607, 1997.

[83] SUN MICROSYSTEMS INC., `http://java.sun.com/j2se/1.5.0/docs/api/`: *Java 2 API Documentation*, 2005.

[84] SUN MICROSYSTEMS INC., `http://java.sun.com/docs/books/tutorial/reflect/`: *The Reflection API*, 2005.

[85] THIES, MICHAEL: *Combining Static Analysis of Java Libraries with Dynamic Optimization*. Dissertation. Shaker Verlag, ISBN: 3-8322-0177-7, April 2001.

[86] THURSTONE, L. L.: *A law of comparative judgement*. Psychological Review, 34:273–286, 1927.

[87] TIAN, JIANHUI und MARVIN V. ZELKOWITZ: *A Formal Program Complexity Model and Its Application*. The Journal of Systems and Software, 17(3):253–266, März 1992.

[88] UNIVERSITÄT PADERBORN, `http://studinfo.uni-paderborn.de`: *Das StudInfo-System*, 2005.

[89] WALKINSHAW, N., M. ROPER und M. WOOD: *The Java System Dependence Graph*. In: *3rd IEEE International Workshop on Source Code Analysis and Manipulation*, September 2003.

[90] *Weka 3 — Data Mining with Open Source Machine Learning Software in Java*. `http://www.cs.waikato.ac.nz/~ml/weka/`, 2003.

[91] WEYUKER, ELAINE J.: *Evaluating Software Complexity Measures.* IEEE Transactions on Software Engineering (TSE), 1988.

[92] WHITMIRE, SCOTT A.: *Object Oriented Design Measurement*. John Wiley & Sons, Inc., 1997.

[93] WILCOXON, F.: *Individual comparisons by ranking methods.* Biometrics, 1:80–83, 1945.

[94] WITTEN, IAN H. und EIBE FRANK: *Data Mining*. Morgan Kaufmann, Los Altos, US, 2000.

[95] WORLD WIDE WEB CONSORTIUM, `http://www.w3.org/XML/1998/06/xmlspec-report-v21.htm`: *XML Specification for Document Type Definitions (DTD)*, 1998.

[96] ZIMMERMANN, THOMAS, PETER WEISSGERBER, STEPHAN DIEHL und ANDREAS ZELLER: *Mining Version Histories to Guide Software Changes*. In: *26th International Conference on Software Engineering (ICSE 2004)*, Seiten 563–572, 2004.

[97] ZUSE, H.: *Software Complexity: Measures and Methods*. Walter de Gruyter, Berlin, 1 Auflage, 1991.

[98] ZUSE, HORST: *Geschichte der Programmiersprachen*. Technischer Bericht, Technische Universität Berlin, 1999.

Abbildungsverzeichnis

2.1. Entwicklung der Programmiersprachen — Einteilung in Generationen . . . 10
2.2. V-Modell zur Konstruktion und Qualitätssicherung von Software-Systemen . . . 14

3.1. Abgrenzung: Entwurfsmängel zwischen AntiPattern und Programmierfehlern . . . 30
3.2. Beispiele für die „Große Klasse“ und „Datenklasse“. . . . 33
3.3. Beispiel aus Abb. 3.2 nach *Refactoring*-Transformationen. . . . 34
3.4. Beispiel für den Entwurfsmangel „Lange Methode“. . . . 35
3.5. Problem-orientierte Klassifizierung von Entwurfsmängeln [65]. . . 39

4.1. Hypothesenraum des konzeptionellen Lernens . . . 44
4.2. Beispiel für einen Entscheidungsbaum des Konzeptes „Tisch“. . . . 46
4.3. Berechnung von Entropie und Nutzen. . . . 48
4.4. Propagation der Trainingsmenge bei der Konstruktion des Entscheidungsbaums. . . . 48

5.1. Ebenen des *Ziel-Frage-Metrik*-Ansatzes [75]. . . . 58
5.2. Klassenabhängigkeitsgraph . . . 61
5.3. Aufrufgraph zur Analyse des externen Zusammenhangs. . . . 62

6.1. Beziehungen zwischen Entitäten des Programmabhängigkeitsgraphen. . . . 76
6.2. Beispiel eines Programmabhängigkeitsgraphen [50]. . . . 77
6.3. Relationales Schema zur Abfrage des Programmabhängigkeitsgraphen . . . 79

7.1. Grundkonzept zur adaptiven Erkennung von Entwurfsmängeln. . 87
7.2. Lern- und Erkennungsphase mit Adaption des Entscheidungsbaumes und des Entwurfsmangelmodells. . . . 91

8.1. Fragebogen zur Ermittlung der relevanten Kenntnisse und Rahmenbedingungen eines Entwurfsmangelsuchenden. . . . 102
8.2. Beispiel kumulierter und kategorisierter Urteile. . . . 104

8.3. Beispiel kumulierter Normalverteilung mit kategorisierter Fragestellung aus Abb. 8.2. 105
8.4. Einordnung von Programmstellen in die Erkennungssituation. . . . 107
8.5. Beispiel einer Messreihe mit Präzisions- und Vollständigkeitswerten. 111
8.6. Beispiel eines kumulierten P/R-Diagramms. 112
8.7. Beispiel eines kumulierten P/R-Diagramms mit zugehöriger Normierung und Interpolation 114
8.8. Beispiel eines kumulierten und interpolierten P/R-Diagramms. . . 115
8.9. Beispiel zur Makro- und Mikromittelwertberechnung aus [76]. . . . 116
8.10. Vergleich zwischen Makro- und Mikrobewertung 118
8.11. Beschreibung des *Wilcoxon-Rangsummentests*. 121
8.12. P/R-Diagramme für drei minimal verschiedene Messreihen. 122
8.13. Beispiel eines ROC-Diagramms. 124
8.14. ROC-Analyse für drei minimal verschiedene Messreihen. 126
8.15. Übersicht der Architektur von IYC. 130
8.16. Bildschirmfoto der Benutzeroberfläche von *Eclipse* 132
8.17. Konfiguration von Entwurfsmängeln in IYC 133
8.18. Konfiguration und Modellierung eines Entwurfsmangels in IYC . . 134
8.19. Training einer Programmstelle in IYC 135
8.20. Messung der Reproduktionsleistung für die „Große Klasse“. 140
8.21. Messung der Transferleistung für die „Große Klasse“. 141
8.22. Messung der Reproduktionsleistung für die „Faule Klasse“. 142
8.23. Messung der Transferleistung für die „Faule Klasse“. 143
8.24. Messung der Reproduktionsleistung für die „Lange Methode“. . . 144
8.25. Messung der Transferleistung für die „Lange Methode“. 145
8.26. Messung der Reproduktionsleistung für die „Neid“. 146
8.27. Messung der Transferleistung für die „Neid“. 147

Tabellenverzeichnis

8.1. Übersicht der Kenngrößen der Klassifizierungsleistung. 108
8.2. Übersicht der Metriken zur Klassifizierungsleistung. 108
8.3. Beispiel für geordnete P/R-Berechnungen. 110
8.4. Vergleich zwischen Makro- und Mikrobewertung 117
8.5. Beispiel für drei minimal verschiedene Messreihen. 120
8.6. Beispiel für den paarweisen Vergleich von drei Messreihen. 124
8.7. Vergleich der Kenngrößen aus P/R- und ROC-Analyse. 125
8.8. Kenngrößen des Analysegegenstandes. 137
8.9. Anzahl der manuell erkannten Entwurfsmängel des Analysegegenstandes. 138