Anwendung maschinellen Lernens zur benutzeradaptiven Erkennung von Entwurfsmängeln in objektorientierter Software

Jochen Kreimer
Universität Paderborn — Institut für Informatik
Fachgebiet Programmiersprachen und Übersetzer
Fürstenallee 11, D-33102 Paderborn
jotte@uni-paderborn.de

Zusammenfassung

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.

Unser 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*.

Entwurfsmängel werden abhängig von der Sichtweise und dem Erfahrungsschatz des Suchenden unterschiedlich interpretiert. Wir kombinieren daher bekannte Verfahren zur Erkennung von Entwurfsmängeln auf der Basis von Metriken mit maschinellen Lernverfahren. Damit kann Entwurfsmangelerkennung individuellen Sichtweisen angepasst werden.

Wir präsentieren ein Werkzeug, das dieses Verfahren implementiert und zeigen die Ergebnisse einer ersten Fallstudie.[1]

1 Einleitung

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*“ [22] 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 von Programmen zu den wertvollen Techniken um die Qualität von Software zu verbessern. Dies wurde als *Software Inspection* von Fagan [10] [9] eingeführt und umfasst die manuelle Durchsicht des Quelltextes, des Entwurfs und der Dokumentation. In modernen agilen Software-Entwicklungsprozessen wie dem *Extreme Programming* [3] spielen sie eine wichtige Rolle zur Qualitätssicherung.

Unser Ziel ist es, Fehler im Entwurf von Software-Systemen zu erkennen und somit unverständliche, schwer erweiter- und änderbare Programmstrukturen zu vermeiden.

[1] Eine längere Fassung dieses Papiers wurde im Juli 2004 zur Begutachtung bei *Informatik – Forschung und Entwicklung (Springer-Verlag)* eingereicht.

1.1 Entwurfsmängel

Bei der Suche nach Entwurfsmängeln interessieren uns Programmstrukturen, die auf fehlerhaften Entwurf hindeuten. Diese Entwurfsmängel lassen sich einordnen zwischen *Anti Patterns* [5] und eher technischen Programmierfehlern.

Eine Vielzahl von Entwurfsmängeln finden sich in der Literatur als „*Design Heuristics*“ [22], „*Design Characteristics*“ [25] oder „*Bad Smells*“ [13]. 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 [13] beschreibt *Refactoring*-Transformationen, die die innere Struktur eines Programms anpassen ohne das von außen sichtbare Verhalten zu ändern. Dies erlaubt Programme zu bereinigen und zu vereinfachen. „*Bad Smells*“ sind Entwurfsmängel, die beschreiben, welche Programmstelle mit welchen Eigenschaften 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.

Entwurfsmängel werden informell jeweils nach einem bestimmten Muster beschrieben. Prägnante und metaphorische Namen erleichtern das Erlernen. Beispiele auffälliger Programmfragmente erläutern das jeweilige Problem. Häufig wird mit Indizien und Hinweisen auf Programmeigenschaften argumentiert. Einige Autoren formulieren Merksätze um die Anwendung zu erleichtern.

Die bisher bekannten Beschreibungen zu Entwurfsmängeln sind sehr wertvoll. Sie vermitteln das Wissen zum „guten“ objektorientierten Entwurf in handlichen Portionen. Dennoch ist es für einen unerfahrenen Entwickler oder Entwerfer schwierig, Entwurfsmängel zu erkennen. Es hängt jeweils von der Sichtweise und dem Erfahrungsschatz des Einzelnen ab, welche Mängel erkannt und wie diese erkannt werden. Häufig haben unterschiedliche Personen sehr unterschiedliche Auffassungen von dem selben Mangel.

Auch prägt das Anwendungsgebiet eines zu untersuchenden Programms die Sicht auf Entwurfsmängel. Man kann sich leicht vorstellen, dass z. B. in einem mathematisch/wissenschaftlich geprägtem Programm komplexe Algorithmen implementiert werden. Auf der Suche nach zu langen Methoden wird man hier viel längere Methoden zulassen als in einem stark Benutzerschnittstellenorientierten Programm, das ereignisgesteuert und an Bibliotheken geknüpft arbeitet.

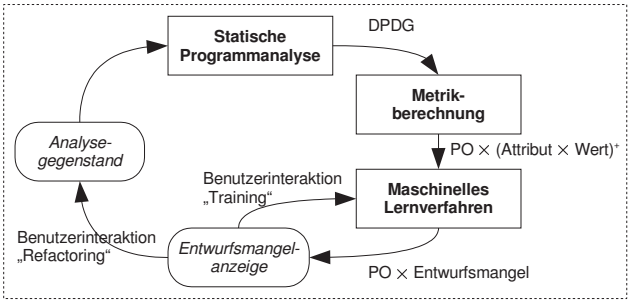

Abbildung 1: Grundkonzept zur Erkennung von Entwurfsmängeln.

Wir stellen daher in Abschnitt 2 ein Verfahren zur adaptiven Erkennung von Entwurfsmängeln vor. Wir kombinieren Erkennungsverfahren auf der Basis von Metriken mit maschinellen Lernverfahren. In Abschnitt 3 stellen wir das prototypische Werkzeug *It's Your Code* (IYC) vor, in dem das Verfahren implementiert wurde. Wir schließen mit den Ergebnissen einer ersten Fallstudie und liefern einen Überblick zu zukünftigen Arbeiten.

2 Adaptive Erkennung von Entwurfsmängeln

Um Entwurfsmängel automatisch zu erkennen, kombinieren wir objektorientierte Metriken (OO-Metriken) mit Verfahren des maschinellen Lernens.

Abbildung 1 zeigt das Grundkonzept zur automatischen und adaptiven Erkennung von Entwurfsmängeln.

Wir folgen dem Ansatz aus [16] und ordnen jedem Entwurfsmangel eine Menge von Programmeigenschaften zu, die wir durch objektorientierte Entwurfsmetriken ausdrücken. Häufig werden Größen-, Komplexitäts-, Kopplungs- oder Kohäsionsmetriken verwendet. Dieses Modell dient als Definition einer Strategie zur Erkennung jedes Entwurfsmangels.

Metriken werden durch statische Programmanalyse mit klassischen Methoden der Kontroll- und Datenflussanalyse sowie der abstrakten Interpretation ermittelt. Die Analyseergebnisse führen u. a. zu einem Programmabhängigkeitsgraphen der als abstraktes Modell des Analysegegenstandes dient und zur Metrikberechnung genutzt wird.

Die Messwerte von Programmobjekten werden dem Modell entsprechend 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 Programmobjekten zu erkennen.

2.1 Modellierung von Entwurfsmängeln

Für jeden Entwurfsmangel der erkannt werden soll, gehen wir von der informellen Beschreibung aus. Die Analysen dieser Beschreibungen geben Hinweise darauf, wie diese erkannt werden können. Häufig sind es Indizien oder vage beschriebene Programmstrukturen, die auf einen Entwurfsmangel hindeuten. Häufig werden Vorschläge gemacht, wie ein Entwurfsmangel z. B. durch *Refactoring*-Transformationen behoben werden kann. So bietet es sich auch an, nach Programmstellen zu suchen, auf die eine solche Transformation angewandt werden kann.

Bildung eines hypothetischen Modells. Das so entstandene eigene mentale Modell des Entwurfsmangels wird nun als Menge von Metriken ausgedrückt. Hierzu bildet man jedes gefundene Kriterium auf eine oder mehrere messbare Programmeigenschaften ab. Als Ergebnis erhält man für jeden Entwurfsmangel eine Menge von OO-Metriken, die diesen charakterisieren. An dieser Stelle existiert erst eine vage Vorstellung, welche Messwerte diese Metriken annehmen müssten, oder in welcher Kombination diese auftreten sollten. Auch die Relevanz einzelner Messwerte, d. h. ob diese geeignet sind, einen Entwurfsmangel mit ausreichender Genauigkeit zu charakterisieren, ist nichts bekannt. Es handelt sich also zunächst um ein hypothetisches Modell.

In der Literatur findet man eine unüberschaubare Menge von Metriken, unter anderem [6] [17] [11] [4] [12], aus denen wir eine Teilmenge für diese Zwecke ausgewählt bzw. neue ähnliche erstellt haben.

Im Folgenden entwickeln wir beispielhaft Modelle für die beiden Entwurfsmängel „Lange Methode“ und „Große Klasse“ aus [13].

Modell für eine „Lange Methode“.
Fowler argumentiert, dass Programme besonders lange leben, wenn sie besonders kurze Methoden haben. Als Schlüssel zu guten Methoden nennt Fowler möglichst gute Namen von Methoden. Als Indiz, wann eine Methode in kleinere zerteilt werden sollte, spricht er von Kommentaren. Immer dann wenn etwas kommentiert werden könnte, solle man daraus eine eigene Methode mit sprechendem Namen erstellen.

Eine lange Methode ist nicht etwa eine Methode mit besonders vielen Anweisungen, sondern eine Methode mit aufwändiger Kontrollstruktur. Hier bietet sich ein Komplexitätsmaß zur Charakterisierung an.

Fowler argumentiert weiter, dass viele Parameter und lokale Variable das Zerteilen einer Methode behindern. Hier schlägt Fowler mehrere *Refactoring*-Transformationen vor, die späteres Zerteilen erleichtern können.

Diese Kriterien lassen sich auf folgende Metriken abbilden:

- Das Größenmaß *„Anzahl der Anweisungen einer Methode“*.
- Das Komplexitätsmaß *„Komplexität einer Methode“*. Wir setzen hier das klassische Komplexitätsmaß von McCabe [17] ein.
- Das Größenmaß *„Anzahl der Parameter einer Methode“*.
- Das Größenmaß *„Anzahl lokaler Variablen einer Methode“*.

Modell für eine „Große Klasse“.
Eine Klasse sollte einem einzigen Zweck dienen. Häufig wird versucht einer Klasse mehrere Aufgaben aufzubürden. Solche Klassen entstehen mit der Zeit, wenn kleine Änderungen oder Funktionserweiterungen „heimlich“ in einer bestehenden Klasse durchgeführt werden. Solche Klassen entstehen auch, wenn ein prozeduraler Entwurf in einen objektorientierten überführt wird oder wenn Entwickler, die gewohnt sind, prozedural zu entwickeln, eine Hauptklasse mit aller Funktionalität und nur kleinen zugeordneten Helferklassen implementieren [22, S. 32ff.].

Fowler argumentiert, dass Klassen, die zu viel tun, oft zu viele Instanzvariablen besitzen. Außerdem kann es vorkommen, dass nicht immer alle Instanzvariablen genutzt werden. So zerfällt eine Klasse ganz natürlich in Teilklassen. Fowler argumentiert weiter, dass eine Klasse mit viel

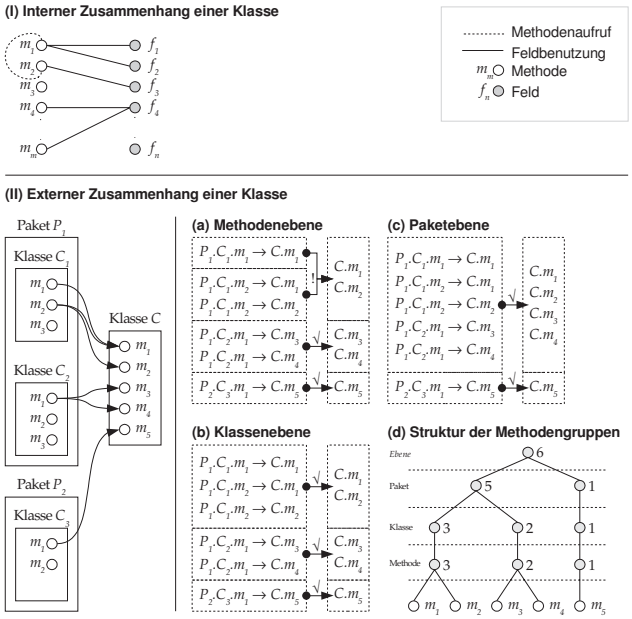

Abbildung 2: Kohäsions- und Kopplungsmaß einer Klasse.

Code verdächtig ist, auch duplizierten Code zu enthalten. Hier sollte Redundanz durch Extraktion langer Methoden in wiederverwendbare Methoden durchgeführt werden. Zuletzt betrachtet Fowler die Benutzung der Klasse. Wenn unterschiedliche Nutzer disjunkte Teilmengen der Methoden der Klasse nutzen, deutet dies auf eine zu prüfende Zerteilbarkeit hin.

Diese Kriterien lassen sich auf folgende Metriken abbilden:

- Das Größenmaß *„Anzahl der Instanzvariablen einer Klasse“*.
- Das Kohäsionsmaß *„Anzahl der internen Zusammenhangskomponenten“*.

  Abbildung 2-I zeigt die Methoden einer Klasse und deren benutzte Felder, sowie Methodenaufrufe innerhalb der Klasse. Es ergeben sich vier Zusammenhangskomponenten in die sich die Klasse vermutlich zerteilen ließe: $\{f_1, f_2, f_3, m_1, m_2\}, \{m_3\}, \{m_4, m_m, f_4\}$ und $\{f_n\}$.
- Das Komplexitätsmaß *„Median der Komplexitäten aller Methoden einer Klasse“*.
- Das Größenmaß *„Median der Anzahl der Anweisungen aller Methoden einer Klasse“*.
- Das Kopplungsmaß *„Anzahl der externen Zusammenhangskomponenten“*.

  Abbildung 2-II zeigt eine Klasse C, deren Methoden von verschiedenen Programmteilen aufgerufen werden. Je nach betrachteter Ebene ergeben sich drei bzw. zwei disjunkte Teilmengen der Methodenmenge. Diese Teilmengen liefern Hinweise für eine Zerteilung der Klasse.

2.2 Maschinelles Lernen

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 [18] [26] [14].

Die Vielzahl der beschriebenen Lernverfahren mit unterschiedlichsten Eigenschaften erschwert es, ein geeignetes Lernverfahren auszuwählen. Wir haben uns für die Erkennung von Entwurfsmängel zunächst auf das Lernen von Entscheidungsbäumen konzentriert. Ausschlaggebend für diese Entscheidung waren vor allem das einfache Verfahren, die gute Erklärbarkeit und eine hohe Konstruktionsgeschwindigkeit.

Konstruktion von Entscheidungsbäumen (C4.5) Entscheidungsbäume werden zur Klassifikation verwendet. Eingabe ist eine Menge von Kriterien bzw. Attributen. Ein Attribut ist ein Name-Wert-Paar, dessen Werte nominal oder numerisch sein können. Ein spezielles Attribut der Attributmenge spielt die Rolle des Zielattributes und beschreibt die gewünschte Klassifizierung der Menge von Attributen und deren konkrete Werte. Eine konkrete Ausprägung der Attributemenge, also Attribute mit Werten, wird Instanz genannt.

Eine Menge von Instanzen, die auch einen Wert für das Zielattribut enthalten dient als Trainingsmenge. Daraus wird ein Entscheidungsbaum 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 Teilbaumes. Eine detaillierte Beschreibung kann z. B. in [18, Kapitel 3] nachgelesen werden. Das hier eingesetzte *C4.5*-Verfahren geht auf [20][21] zurück.

Bisher nicht klassifizierte Instanzen, also Instanzen mit fehlendem Wert des Zielattributes, werden durch Interpretation des Entscheidungsbaumes klassifiziert. Dabei modellieren innere Knoten des Entscheidungsbaums die Entscheidungspunkte eines einzelnen Attributes. Es wird dann ausgehend von der Wurzzel an den jeweiligen Unterknoten verzweigt für den der aktuelle Attributwert passt. Blattknoten stellen das Ergebnis dar.

Ein Beispiel für eine Menge von Instanzen und ein daraus konstruierter Entscheidungsbaum ist in Abbildung 3 [15] zu sehen. Der konstruierte Entscheidungsbaum soll möglichst alle Trainingsbeispiele korrekt klassifizieren, garantiert ist dies jedoch nicht.

2.3 Adaptive Erkennung

Die Erkennung von Entwurfsmängeln ist ebenfalls ein Klassifizierungsproblem. Das zu erlernende Konzept ist also die Entscheidung, ob eine fragliche Programmstelle einen Entwurfsmangel enthält oder nicht.

Erkennung Jeder Entwurfsmangel wird durch eine Menge von OO-Metriken modelliert. Diese lassen sich als Attribute im Entscheidungsbaumverfahren nutzen. Das Zielattribut legt fest, ob der jeweilige Entwurfsmangel vorliegt oder nicht. Abbildung 3 zeigt beispielhaft eine Menge von Messwerten zu verschiedenen Klassen eines Programms und die Klassifizierung als Entwurfsmangel. Der konstruierte Entscheidungsbaum für diesen Entwurfsmangel wird genutzt, um zu prüfen, ob eine bisher unbekannte Programmstelle diesen Entwurfsmangel enthalten könnte.

Die Anzahl der Trainingsbeispiele und die Menge der verwendeten Attribute ist hier verkürzt dargestellt. Es sind nur die Anzahl der Felder, Methoden und Anweisungen aller Methoden der Klasse und nur 12 Instanzen dargestellt. Lernverfahren dieser Art konstruieren Entschei-

Nr.	*#Felder*	*#Meth.*	*#Anw.*	*Große Klasse*
1.	33	5	100	Ja
2.	28	29	87	Ja
3.	45	24	67	Ja
4.	21	27	95	Nein
5.	18	13	104	Nein
6.	13	5	83	Nein
7.	18	5	272	Ja
8.	5	2	301	Ja
9.	29	23	125	Ja
10.	67	23	125	Ja
11.	27	23	93	Nein
12.	32	8	113	Ja

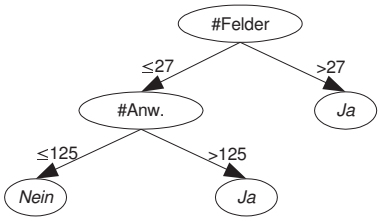

Abbildung 3: Verkürztes Beispiel von Messwerten zum Entwurfsmangel „Große Klasse“ und ein daraus konstruierter Entscheidungsbaum.

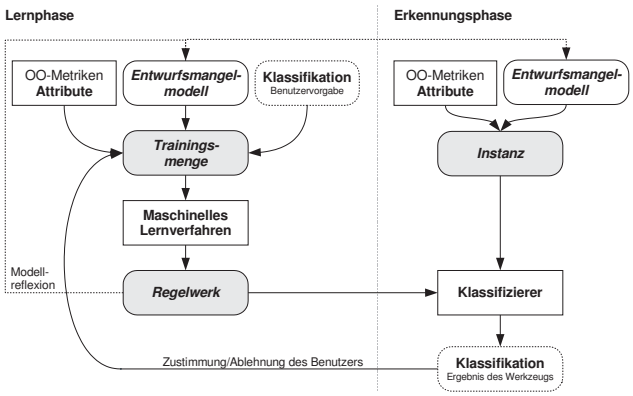

Abbildung 4: Lern- und Erkennungsphase mit Adaption des Entscheidungsbaumes und des Entwurfsmangelmodells.

dungsbäume mit hinreichend großer Trefferrate ab etwa 100 Trainingsbeispielen.

Es fällt auf, dass in diesem Modell die Anzahl der Felder einer Klasse als stärkstes Kriterium dient. Die Anzahl der Methoden wird nicht zur Entscheidung herangezogen. Diese vom Lernverfahren getroffenen Entscheidungen erlauben erste Rückschlüsse auf das gewählte Modell zur Erkennung von „großen Klassen“.

Adaption Liegt eine geeignete Trainingsmenge vor, so kann der daraus konstruierte Entscheidungsbaum für zukünftige Bewertungen von Programmstellen verwendet werden. Häufig ist aber nicht klar, ob die gewählten Attribute überhaupt geeignet sind, den jeweiligen Entwurfsmangel zu charakterisieren. Außerdem können die Trainingsbeispiele derart ungünstig gewählt sein, dass das gewünschte Spektrum von Ausprägungen dieses Mangels noch nicht abgedeckt wird. Das Verfahren soll daher ständig von weiteren Beispielen lernen können.

Abbildung 4 zeigt den aus zwei Phasen bestehenden Ansatz. 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 angewandt. Dies liefert die Information, ob der jeweilige Entwurfsmangel vorliegen könnte. Ist dies der Fall, wird dem Anwender die Programmstelle mit dem potentiellen Entwurfsmangel 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.

2.4 Erklärungskomponente

Der aktuelle Entscheidungsbaum für einen Entwurfsmangel kann sich laufend ändern. 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.

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.

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

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.

Bei der Entwicklung großer Software-Systeme werden i. d. R. Programmierrichtlinien festgelegt. Die Abwesenheit von Entwurfsmängeln könnte eine Richtlinie sein. Mit der automatischen Erkennung von Entwurfsmängeln lassen sich auch solche Richtlinien prüfen und einhalten. Damit sind Modelle von Entwurfsmängeln und Entscheidungsbäume als Wissensspeicher und zur Wissensweitergabe geeignet.

2.5 Modellreflexion

Lernverfahren, wie das hier verwendete *C4.5*, erreichen bei ausreichend großer Trainingsmenge eine sehr gute Anpassung. Zur Messung der Anpassungsleistung können z. B.

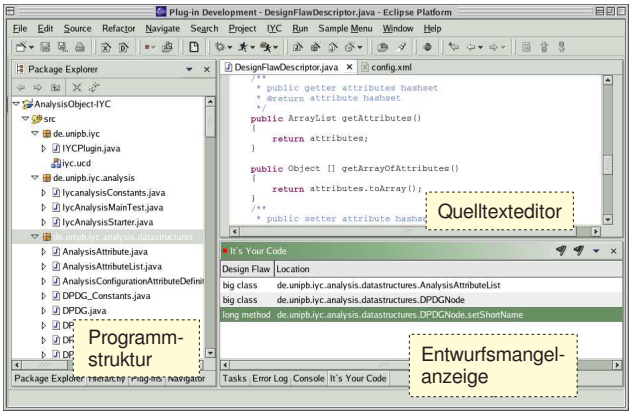

Abbildung 5: Bildschirmfoto IYC in *Eclipse*.

mehrfache Kreuzvalidierung eingesetzt werden. Dadurch 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 einem menschlichen Untersuchung nur wenige Abweichungen anzeigen. Gelingt dieses 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.

3 Werkzeug für *Eclipse*

Zur Evaluation des Konzeptes aus Abschnitt 2 haben wir das prototypische Werkzeug *„It's Your Code“* (IYC)[2] als Modul (*Plugin*) für die Java-Programmierumgebung der freien Entwicklungsumgebung *Eclipse* [8] entwickelt.

Eclipse bietet, wie viele andere Entwicklungsumgebungen auch, neben Projekt-, Versions- und *Make*-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 Selektion eines Programmobjektes, z. B. eines Paketes oder einer Klasse, kann im Kontextmenü die Suche nach Entwurfsmängeln gestartet werden. Die Ergebnisse werden als Liste von gefundenen potentiellen Entwurfsmängeln in einer eigenen Sicht präsentiert. Der Benutzer kann die Liste abarbeiten und einzelne Vorschläge ablehnen oder akzeptieren und damit das Lernverfahren weiter trainieren.

Abbildung 5 zeigt ein Bildschirmfoto von *Eclipse* mit der Anzeige des Quelltextes im rechten oberen Bereich, einer Sicht zur Navigation in der Java-Programmstruktur

[2]Wir haben den Namen gewählt, weil das Werkzeug lediglich Vorschläge für potentielle Entwurfsmängel macht. Der Benutzer muss zuletzt entscheiden, ob der Entwurfsmangel vorliegt oder nicht.

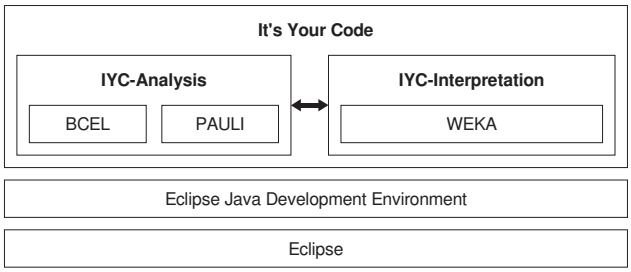

Abbildung 6: Architektur des Werkzeugs IYC.

Maß	*IYC*	*WEKA*
#Zeilen	4274	92615
#Klassen	91	597
#Methoden	765	7193
#Felder	283	3431

Abbildung 7: Größen der Analysegegenstände.

im linken Bereich und einer Liste von gefundenen Entwurfsmängeln im Bereich rechts unten.

Nicht zu sehen ist hier die Konfiguration von IYC. Benutzer können eigene Entwurfsmängel definieren oder bestehende ändern, indem sie einem Entwurfsmangel eine Menge von vorgefertigten Attributen zuordnen. Der aktuelle Trainingszustand läßt sich separat sichern und laden.

Abbildung 6 zeigt den Aufbau des Werkzeugs. Als Grundlage dient die *Eclipse*-Plattform mit den Modulen zur Java-Entwicklung. IYC besteht aus zwei Teilen: der Analyseteil *IYC-Analysis* berechnet OO-Metriken, der Interpretationsteil *IYC-Interpretation* implementiert die adaptive Erkennung mit Benutzerinteraktion.

Als maschinelles Lernverfahren setzen wir den *J48*-Klassifizierer — eine spezielle *C4.5*-Implementierung — aus der *WEKA*-Bibliothek [24] ein.

IYC-Analysis verwendet die *Bytecode Engineering Library* (BCEL) [7] [2] um auf den *Bytecode* des zu analysierenden Java-Programms zuzugreifen. *PAULI* [23] ist eine Bibliothek zur Programmanalyse von Java-Bytecode. Diese implementiert z. B. benötigte Kontroll- und Datenflussanalysen.

4 Evaluation

In einer ersten Fallstudie [15] haben wir die Leistung des Verfahrens mit Hilfe der prototypischen Implementierung untersucht.

Als Analysegegenstand dienen zwei Software-Systeme, die uns so gut bekannt waren, dass wir als menschliche Benutzer Entwurfsmängel beurteilen konnten. Dies sind unser Werkzeug *IYC* und das von uns eingesetzte *WEKA*-Paket. Abbildung 7 zeigt die Größen der beiden Systeme. Charakteristisch für beide Systeme ist ein großer Teil Implementierung der Benutzeroberfläche, sowie Verarbeitung von komplexen Datenstrukturen. Allerdings implementiert *WEKA* komplexe Algorithmen, währen *IYC* viele Aufgaben an Bibliotheken delegiert.

Obwohl die beiden Entwurfsmängel „Lange Methode“ und „Große Klasse“ mit jeweils nur 20 Beispielen trainiert wurden, ergab die *Leave-one-out*-Kreuzvalidierung eine Genauigkeit des Lernverfahrens von 95 bzw. 100 Prozent.

Abbildung 8 zeigt die entstandenen Entscheidungsbäume. Für die „Lange Methode“ reichte hier

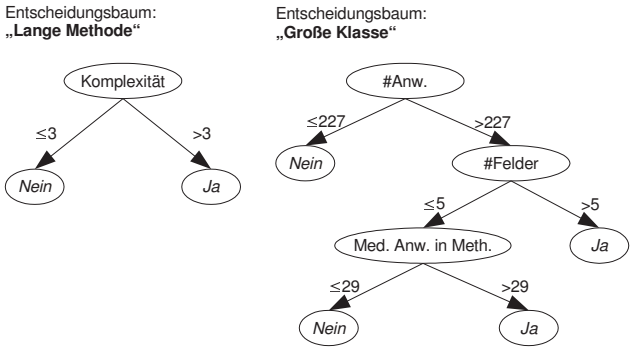

Abbildung 8: Beispiele für Entscheidungsbäume.

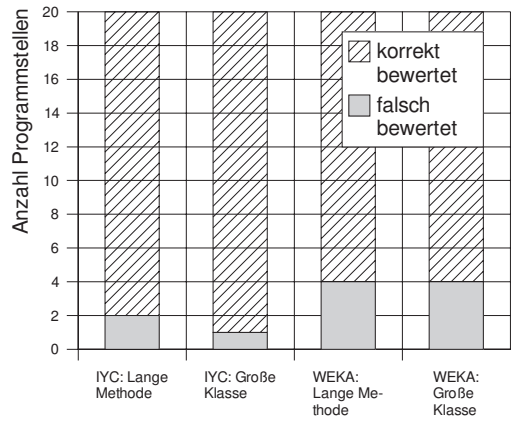

Abbildung 9: Leistungsvergleich zufällig ausgewählter manuell und automatisch bewerteter Programmstellen. Dargestellt sind die Anzahl der Programmstellen, die vom Werkzeug korrekt bzw. falsch klassifiziert wurden.

aus, die Komplexität nach McCabe zu betrachten. Andere Messwerte wurden hier vom Lernverfahren nicht zur Entscheidung benötigt. Bei der „Großen Klasse“ wurden drei der fünf Kriterien verwendet.

Die automatische Analyse der beiden Systeme ergab ähnliche Ergebnisse. Etwa 15 % der Methoden wurden als „Lange Methode“ und etwa 20 % der Klassen wurden als „Große Klasse“ identifiziert.

Im Leistungsvergleich mit der menschlichen Intuition wurden zufällig jeweils 20 Programmstellen ausgewählt und manuell bewertet. Jeweils 11 Programmstellen enthielten den jeweiligen Entwurfsmangel, 9 Programmstellen enthielten ihn nicht. Abbildung 9 zeigt, dass jeweils nur wenige automatische Klassifizierungen nicht mit der intuitiven Bewertung des Benutzers übereinstimmten.

Ausblick: Empirische Untersuchung Die ersten Testergebnisse sind ermutigend, reichen aber zu einer abschließenden Bewertung nicht aus. Wir planen daher eine Untersuchung unter empirisch-pädagogischen Gesichtspunkten durchzuführen.

Hierzu wird das prototypische Werkzeug weiterentwickelt und öffentlich zugänglich gemacht. Eine Online-Komponente soll es erlauben, Modelle und Entscheidungsbäume unterschiedlichster Nutzer, anonymisiert und Zustimmung vorausgesetzt, zu sammeln. Ein Nutzerprofil in Form einer Befragung über Programmierstil und -erfahrungen, Entwicklergruppengröße und -zusammensetzung, sowie Charakteristika des erstellten Software-Systems soll Aufschluss darüber geben, ob und welche Einflussfaktoren sich in unterschiedlichen Ausprägungen von Entwurfsmängeln widerspiegeln. Hierzu kann z. B. *Cluster*-Analyse [1] [19] eingesetzt werden.

Auf dieser Basis soll eine „Lerngruppe“ in Form eines öffentlicher Kataloges von Entwurfsmängeln, bestehend aus Modellen und Trainingsmengen bzw. Entscheidungsbäumen, entstehen.

5 Zusammenfassung und Ausblick

Wir haben ein Verfahren zur Erkennung von Entwurfsmängeln in objektorientierter Software vorgestellt. Da Entwurfsmängel sehr unterschiedlich interpretiert werden, passt sich dieses Verfahren dem speziellen Einsatzszenario an. Wir kombinieren bekannte Ansätze auf der Basis objektorientierter Metriken mit maschinellen Lernverfahren und schlagen ein adaptives und lernendes Verfahren vor.

Mit Hilfe einer prototypischen Implementierung haben wir eine erste erfolgreiche Fallstudie durchgeführt. Für eine abschließende Bewertung planen wir eine größere empirische Untersuchung.

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 unvollständige 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. Abwesenheit von Entwurfsmängeln kann als Qualitätsziel in Softwareprojekten vereinbart werden.
- Das Verfahren passt sich automatisch dem Analysegegenstand bzw. Benutzersichtweisen an. Anpassung ist jedoch nicht zwingend, nach einer Trainingsphase kann auch der Zustand gesichert und für ein Software-Projekt verbindlich festgelegt werden.

Ausblick Sobald Entwurfsmängel gefunden werden, sollten diese behoben werden. Hierzu kann der Software-Entwickler geeignete *Refactoring*-Transformationen anwenden. Ein Werkzeug könnte Hinweise liefern, welche und wo Transformationen sinnvoll eingesetzt werden könnten.

Wir können uns vorstellen, Entwickler bei dieser Tätigkeit zu beobachten und zusammen mit zuvor gefundenen Entwurfsmängeln und jeweiligen Programmeigenschaften zu protokollieren. Ein Lernverfahren könnte dies Beobachtungen erlernen um später selber geeignete Vorschläge machen zu können.

Der Einsatz von Entscheidungsbäumen zur Wissensrepräsentation ist ansprechend, da eine Erklärungskomponente direkt abgeleitet werden kann. Es bleibt zu prüfen, wie bewertet werden kann ob andere Lernverfahren, z. B. Bayes-Verfahren, für diesen Zweck besser geeignet wären.

In der Softwaretechnik könnten sicher noch zu anderen Zwecken maschinelle Lernverfahren als Metamethodik zur Wissensgenerierung verwendet werden.

Literatur

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

[2] The Apache Jakarta Projekt, `http://jakarta.apache.org/bcel`. *Byte Code Engineering Library (BCEL)*, 2004.

[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[4] Dirk Beyer, Claus Lewerentz, and Frank Simon. Impact of inheritance on metrics for size, coupling, and cohesion in object-oriented systems. *Lecture Notes in Computer Science*, 2006:1–??, 2001.

[5] William J. Brown, Raphael C. Malveau, Hays W. "Skip" McCormick III, and Thomas J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley, 1998.

[6] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[7] Markus Dahm. Byte code engineering. In *Java-Informations-Tage*, pages 267–277, 1999.

[8] Eclipse.org Consortium, `http://www.eclipse.org`. *Eclipse.org Main Page*, 2003.

[9] Micahel E. Fagan. Advances in software inspections. In *IEEE Trans. On Softw. Eng., 7 (12)*, pages 744–751, 1986.

[10] Michael E. Fagan. Design and code inspections and process control in the development of programs. Technical Report 00.2763, IBM, June 1976.

[11] Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.

[12] Norman Fenton and Shari Lawrence Pfleeger. *Software Metrics - A Rigorous and Practical Approach*. International Thomson Computer Press, London, 2 edition, 1996.

[13] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[14] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Stats. Springer, 2001.

[15] Mike Liebrecht. Adaptive Erkennung von Entwurfsmängeln in Java-Anwendungen. Master's thesis, Universität Paderborn, 2004.

[16] Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, University of Timisoara, 2002.

[17] Thomas J. McCabe. A complexity measure. In *Proceedings: 2nd International Conference on Software Engineering*, page 407. IEEE Computer Society Press, 1976. Abstract only.

[18] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[19] R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In Jorgeesh Bocca, Matthias Jarke, and Carlo Zaniolo, editors, *20th International Conference on Very Large Data Bases, September 12–15, 1994, Santiago, Chile proceedings*, pages 144–155, Los Altos, CA 94022, USA, 1994. Morgan Kaufmann Publishers.

[20] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1(1):81–106, 1986.

[21] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[22] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[23] Michael Thies. *Combining Static Analysis of Java Libraries with Dynamic Optimization*. Dissertation. Shaker Verlag, ISBN: 3-8322-0177-7, April 2001.

[24] Weka 3 — Data Mining with Open Source Machine Learning Software in Java, `http://www.cs.waikato.ac.nz/~ml/weka/`. *Eclips.org Main Page*, 2003.

[25] Scott A. Whitmire. *Object Oriented Design Measurement*. John Wiley & Sons, Inc., 1997.

[26] Ian H. Witten and Eibe Frank. *Data Mining*. Morgan Kaufmann, Los Altos, US, 2000.