Andreas Oberweis, Ralf Reussner (Hrsg.): Modellierung 2016,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2016 45

Vom Clean Model zum Clean Code

Anna Vasileva[1], Doris Schmedding[2]

Abstract: In diesem Beitrag wird der Zusammenhang zwischen Code-Qualität und UML-Modellen in einem Software-Entwicklungsprozess in der Informatik-Ausbildung vorgestellt. Es wird untersucht, welche der im Code sichtbar werdenden Mängel bereits im Modell erkannt werden können. Werkzeuge zur statischen Code-Analyse und Refactoring-Techniken unterstützen die Studierenden beim Entdecken und Beseitigen der Qualitätsmängel im Programm-Code. Eine Analyse der studentischen Projekte hat gezeigt, dass sich manche Code-Mängel im Nachhinein nur schwer beseitigen lassen. Aus diesem Grund müssen Qualitätsaspekte bereits beim Modellieren in Betracht gezogen werden. Frühzeitig erkannte Mängel lassen sich mit geringeren Kosten beseitigen als spät erkannte Defekte.

Keywords: Clean Code, Code-Qualität, Qualität von UML-Modellen, statische Code-Analyse, Software-Entwicklung, Metriken

1 Einleitung

Das Software-Praktikum (SoPra) ist eine Lehrveranstaltung, in der im Team Software-Entwicklungsprojekte durchgeführt werden. Es findet in den ersten Semestern des Informatik-Studiums statt und wird nach einer Programmier- und Software-Technik-Veranstaltung besucht. Im Rahmen des SoPras werden die bis dahin erlernten Methoden, grundlegende Prinzipien und Software-Entwicklungsprozesse in der Praxis eingesetzt.

Es werden zwei Varianten des SoPras angeboten – als reguläre Veranstaltung im Wintersemester und als Blockveranstaltung in der vorlesungsfreien Zeit im Winter und im Sommer. Der offizielle Stundenumfang beträgt in beiden Fällen vier Semesterwochenstunden. Im Ferien-SoPra treffen sich die Studierenden jeden Tag und arbeiten etwa 30 Stunden pro Wochen an den Projekten.

Die Teilnehmer sind Bachelor-Studierende, die in Gruppen mit jeweils acht Mitgliedern aufgeteilt sind. Jede Gruppe hat die Unterstützung eines Betreuers bzw. einer Betreuerin. In der beschränkten Zeit werden zwei Projekte durchgeführt. Etwa sechs bis zehn Gruppen bearbeiten gleichzeitig dieselben Projekte. In der Regel ist das erste Projekt ein Verwaltungsprogramm und das zweite Projekt ein Spiel.

Die Software-Entwicklung basiert auf einer vereinfachten Version des

[1] Technische Universität Dortmund, Fakultät für Informatik, Otto-Hahn-Str. 12, 44227 Dortmund, anna.vasileva@tu-dortmundd.de
[2] Technische Universität Dortmund, Fakultät für Informatik, Otto-Hahn-Str. 12, 44227 Dortmund, doris.schmedding@tu-dortmund.de

Wasserfallmodells von Royce [Ro70]. Dieses ist auch für Studierende ohne Erfahrung leicht verständlich sowie für kleine und übersichtliche Projekte gut geeignet. Jede Phase ist vordefiniert und wird streng befolgt. Die Betreuer führen am Ende jeder Phase Reviews durch. Mit einer Präsentation der Zwischenergebnisse wird die aktuelle Phase jeweils abgeschlossen.

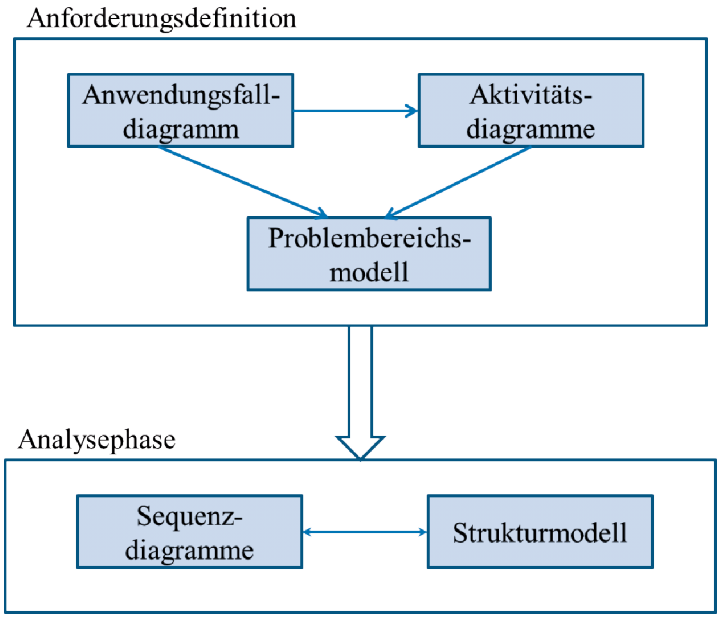

Abb. 1: UML-Modellierung

In der ersten Phase werden die Anforderungen definiert. Anschließend folgt die Analyse-Phase. In diesen beiden Phasen wird die Unified Model Language (UML) [Ru12] zur Modellierung eingesetzt. Abbildung 1 stellt die Abhängigkeit zwischen den in den einzelnen Phasen eingesetzten UML-Diagrammen dar. Darauf wird in Kapitel 3 näher eingegangen. Nach der Modellierung werden aus dem Strukturmodell einmalig Java-Code-Rahmen generiert. Das Programm wird implementiert und getestet. Dabei wird auch die Qualität des produzierten Codes überprüft. Am Ende der Implementierungsphase findet ein Produkttest durch das Entwicklerteam statt, in dem noch einmal überprüft wird, ob die in der Aufgabestellung geforderten Funktionalitäten von dem entwickelten Produkt erfüllt sind. Zum Abschluss des Projekts findet ein gegenseitiger Abnahmetest der teilnehmenden Gruppen statt.

Für die Implementierung wird die objektorientierte Programmiersprache Java eingesetzt. Die verwendete Entwicklungsumgebung ist Eclipse. Viele Plugins, z.B. für die Versions- und Zugriffskontrolle oder für die Entwicklung der grafischen Benutzeroberflächen, unterstützen die Studierenden zusätzlich.

Die Projekte haben einen Umfang von ca. 6000 Lines of Code (LOC) und umfassen ungefähr 30 Klassen ohne Berücksichtigung der Klassen der graphischen Benutzungsschnittstelle (GUI). Diese Klassen werden weitgehend mit Hilfe von GUI-Editoren erzeugt und bleiben deshalb in den nachfolgenden Betrachtungen unberücksichtigt.

Die Arbeit ist folgendermaßen aufgebaut. Nach der Einleitung gibt Kapitel zwei einen Überblick über die Motivation, die Ziele und die Probleme bei der Integration von Qualitätsaspekten in einen Software-Entwicklungsprozess. Im Kapitel „Modellierung mit UML“ wird genauer erläutert, welche UML-Diagramme von den Studierenden im Rahmen der ersten Phasen des Entwicklungsprozesses erstellt werden. In Kapitel vier und fünf wird vorgestellt, welche Maßnahmen bisher getroffen wurden, um die innere Qualität der erstellten Software zu verbessern. Anschließend folgt eine Erläuterung der Mängel, die bereits in der Modellierungsphase zu erkennen sind. Der Beitrag schließt mit einem Ausblick und einem Fazit.

2 Motivation

In den ersten Semestern des Informatik-Studiums lernen die Studierenden die Semantik der verwendeten Programmiersprachen, verschiedene Programmierparadigmen, die Modellierung mit UML, Entwurfsmuster und Software-Entwicklungsprozesse kennen. Im Zentrum der Ausbildung im Bereich Programmierung und Software-Entwicklung steht das Ziel, funktional korrekte Programme zu erstellen. Im Software-Praktikum ist bei der Abnahme der Projekte aber deutlich geworden, dass die innere Qualität der von den Studierenden erstellten Programme zu wünschen übrig lässt.

In mehreren Iterationen haben wir ein Konzept entwickelt, das der langfristigen Verankerung des Ziels „Hohe innere Software-Qualität“ im Software-Entwicklungsprozess dient. Unser Vorgehen basiert auf Untersuchungen zu typischen Qualitätsmängeln im Code von studentischen Projekten in der Lehrveranstaltung Software-Praktikum ([Sc15], [Re14]).

Die bisherigen Analysen der studentischen Projekte mit Hilfe eines Werkzeugs zur statischen Code-Analyse haben gezeigt, dass das Beseitigen von Mängeln, die bereits entstanden sind, häufig zeitaufwendig, sehr schwierig und für unerfahrene Entwickler manchmal auch unmöglich ist. Am Ende der Projekte bleibt keine Zeit für die Verbesserung der inneren Qualität der Programme. Der Schwerpunkt der Aktivitäten liegt in der Endphase des Projekts auf dem Aussehen, dem Hinzufügen von zusätzlichen Features und der funktionalen Korrektheit der vorhandenen Eigenschaften. Unser Ziel ist, dass die Studierenden bereits beim Modellieren mehr Wert auf die innere Qualität der Programme und nicht nur auf die funktionale Korrektheit legen.

3 Modellierung mit UML

In der ersten Phase des Software-Entwicklungsprozesses werden die Anforderungen erhoben. Ein Anwendungsfalldiagramm stellt die funktionalen Anforderungen in Form von Anwendungsfällen dar, die die Nutzer des Systems in unterschiedlichen Rollen ausführen können. Anschließend wird jeder Anwendungsfall näher beschrieben, indem der genaue Ablauf durch ein Aktivitätsdiagramm dargestellt wird.

Die Aktivitätsdiagramme sind wichtige Artefakte im Entwicklungsprozess. Beim Erstellen dieser Diagramme stellen die Studierenden zum ersten Mal fest, dass einige Anwendungsfälle sehr komplex sind. Andere Anwendungsfälle laufen anders ab, als die Studierenden gedacht haben. Zur genaueren Erläuterung der Aktivitätsdiagramme müssen die Studierenden eine textuelle Beschreibung der Anwendungsfälle erstellen, in denen Vor- und Nachbedingungen sowie Fehlerfälle beschrieben werden.

Der Entwicklungsprozess startet also mit einer ausführlichen Erhebung der funktionalen Anforderungen, auf die im weiteren Verlauf des Prozesses immer wieder zurückgegriffen wird. Neben der Modellierung der funktionalen Anforderungen wird ein Datenmodell, das so genannte Problembereichsmodell, erstellt, welches die Klassen des Problembereichs mit ihren Attributen und den Beziehungen untereinander darstellt.

In der Analysephase wird das Problembereichsmodell zum Strukturmodell ausgebaut, indem Steuerungsklassen ergänzt und Methoden hinzugefügt werden. Wir streben aus Gründen der Übersichtlichkeit und Verständlichkeit eine Dreischichtenarchitektur an, die sich an dem Model-View-Controller-Muster [Ga09] der Software-Entwicklung orientiert. Um zu überprüfen, ob im Strukturmodell alle Methoden und Beziehungen zwischen den Klassen vollständig erfasst sind, werden Sequenzdiagramme eingesetzt, die in der Regel die Anwendungsfälle aus dem Anwendungsfalldiagramm repräsentieren. Bei der Definition der Methoden im Strukturmodell werden die Signaturen der Methoden vollständig angegeben.

Im SoPra wird zum Modellieren mit UML das leichtgewichtige Werkzeug Astah [As15] verwendet. Dieses Tool ist sehr benutzerfreundlich und intuitiv bedienbar. Die Studierenden kommen damit gut zurecht. Das Strukturmodell und die Sequenzdiagramme sind, wie bereits erläutert, inhaltlich gekoppelt. Bei Astah liegt diesen beiden Diagrammtypen ein gemeinsames Datenmodell zugrunde. Bei der Erstellung eines Sequenzdiagramms stehen die bereits definierten Methoden zur Verfügung. Eine große Unterstützung stellt die Möglichkeit zur automatischen Generierung von Java-Code-Rahmen dar.

4 Code-Qualität

Die innere Qualität der Programme beruht an erster Stelle auf der Lesbarkeit und Verständlichkeit des Programm-Codes. Erst in der Zusammenarbeit im Team und wenn

die Zeit knapp ist, wird klar, dass eine hohe innere Qualität des Codes das Modifizieren und Testen vereinfacht und die Wartbarkeit des entstehenden Programms erhöht.

Mit Hilfe eines Werkzeugs zur statischen Code-Analyse können außerdem potentielle Fehler frühzeitig entdeckt werden.

Es wurden Messungen mit dem Tool zur statischen Code-Analyse PMD [PMD15] durchgeführt. Die erste Messung diente der Analyse des studentischen Programm-Codes und der Suche nach typischen Defekten. Beim Programmieren wussten die Studierenden anfangs nicht, dass Messungen durchführt werden und nach Mängeln gesucht wird.

Die Analyse der studentischen Projekte hat gezeigt, dass immer wieder ähnliche Mängel im Programmcode vorkommen. Diese lassen sich in folgende Bereiche aufteilen:

- Namensgebung und Einhaltung der Java-Konventionen,
- Komplexität und Länge der Methoden,
- Verantwortlichkeit, Länge und Komplexität der Klassen.

Die Anzahl der Metriken und der Tools, welche die innere Qualität vom Programm-Code messen, ist sehr groß. Gemäß der Goal-Question-Metrik-Methodik [Ba94] werden konkrete Ziele zur Verbesserung der Code-Qualität definiert. Die ausgewählten Metriken dienen dem Ziel, die Lesbarkeit, Verständlichkeit und Wartbarkeit des Codes zu erhöhen. Für unsere Lehrveranstaltung wurden Metriken und Grenzwerte ausgewählt, die für die Studierenden leicht verständlich sind [Re14]. Die Metriken helfen, die Mängel mit Tool-Unterstützung möglichst leicht zu entdecken und zu beheben. Tabelle 1 stellt die definierten Ziele, die verwendeten Metriken und Grenzwerte dar.

Bei der Auswahl der Metriken und der Festlegung der zu erreichenden Grenzwerte wurden der Projektumfang sowie die Erfahrungen der Studierenden berücksichtigt. Darauf aufbauend wurde für das ausgewählte Tool zur statischen Code-Analyse PMD ein SoPra-spezifischer Regelsatz definiert. Dieser wird im XML-Format im SoPra-Wiki [SP15] zur Verfügung gestellt.

Da ein Werkzeug zur statischen Code-Analyse nicht über die Aussagekraft eines Bezeichners entscheiden kann, müssen die Bezeichner auch manuell kontrolliert werden. Mit Hilfe von PMD kann nur die Länge der Bezeichner geprüft werden. Jedoch können auch Bezeichner, die länger als fünf Zeichen sind, sinnlos sein, die Java-Konventionen nicht einhalten oder missverständliche Information liefern.

Ziel	Metrik	Grenzwert	Erkannt durch
Einhaltung der Java-Konventionen	Naming Conventions	-	PMD + Manuelle Prüfung
Sinnvolle Bezeichner	Länge der Bezeichner	5 Zeichen	PMD + Manuelle Prüfung
Übersichtliche und gut lesbare Methoden	Zeilenlänge der Methoden	40 Zeilen	PMD
	Parameteranzahl der Methoden	4	PMD
	Zyklomatische Komplexität	10	PMD
Übersichtliche und gut lesbare Klassen	Zeilenlänge der Klassen	400 Zeilen	PMD
	Toter Code	-	PMD
	Gott-Klasse	WMC[3] > 47, ATFD[4] > 5, TCC[5] < 0,33	PMD

Tab. 1: SoPra-Regeln

5 Bisherige Vorgehensweise zur Qualitätsverbesserung

Eine erste Messung ohne Vorankündigung hat gezeigt, dass die ausgewählten Metriken [Re14] und Grenzwerte für die Studierenden grundsätzlich erreichbar und passend zum Projektumfang gewählt sind. Dennoch wiesen die Projekte erhebliche Mängel auf.

In mehreren Iterationen, Durchführungen des Software-Praktikums, wurde daran gearbeitet, das Thema Code-Qualität in den Ablauf des Software-Entwicklungsprozesses langfristig zu integrieren und die Qualitätsergebnisse der Gruppen zu verbessern [Sc15]. Die Iterationen orientieren sich an dem Plan-Do-Check-Act-Zyklus (siehe Abbildung 2). In jeder Iteration wurden mit Hilfe von PMD-Messungen durchgeführt. Diese wurden analysiert und anhand der Ergebnisse wurden Änderungen am didaktischen Vorgehen vorgenommen, die die Integration der Qualitätsaspekte zusätzlich unterstützen. Diese Änderungsmaßnahmen werden nachfolgend erläutert.

[3] Weighted Method Count
[4] Access To Foreign Data
[5] Tight Class Cohesion

Bereits in der Einführungsveranstaltung zum Praktikum wird Code-Qualität thematisiert. Zum Thema Clean Code [Ma09] und Refactoring [Fo99] werden Tutorials im SoPra [SP15] zum Selbststudium bereitgestellt.

Die Studierenden führen in der Implementierungsphase eigenständig Messungen mit PMD und dem zur Verfügung gestellten SoPra-Regelsatz durch. Die GruppenbetreuerInnen weisen wiederholt auf die Relevanz der Code-Qualität für das Projekt hin und versuchen gemeinsam mit ihrer Gruppe bessere Lösungen zu finden.

Trotz all dieser Maßnahmen wurde festgestellt, dass es offenbar nicht ausreicht, über die Möglichkeit von Qualitätsmessungen und Maßnahmen zur Qualitätsverbesserung informiert zu sein. Deswegen musste ein anderer Weg beschritten werden. Nach dem ersten und vor dem zweiten Projekt folgte eine Diskussion der Ergebnisse der durchgeführten Messungen mit jeder Gruppe und deren BetreuerIn. Die Studierenden wurden in der letzten Iteration aufgefordert, die gefundenen Mängel mit Hilfe von Refactoring-Techniken [Fo99] zu beheben und einen Bericht darüber zu verfassen. Wenn das Verwenden von Refactoring nicht erfolgreich war, mussten die Studierenden dies schriftlich begründen.

Ein Vergleich der Ergebnisse mehrerer Sopra-Durchläufe zeigt, dass diese Maßnahme endlich dazu geführt hat, dass die Ergebnisse im zweiten Projekt deutlich besser wurden. Besonders auffallend ist, dass es in der Kategorie Bezeichner fast keine Verstöße gegen die Bezeichnerwahl gab. Das liegt auch daran, dass die Studierenden diese Mängel bereits beim Modellieren erkennen konnten.

Viele Mängel, insbesondere im Bereich der Namensgebung, lassen sich problemlos, z.B. durch das Refactoring *Rename,* beheben. Um die Länge der Methoden zu verkürzen und ihre Komplexität zu verringern, kann das Refactoring *Extract Method* oft erfolgreich eingesetzt werden. In manchen Fällen waren die Studierenden trotz ihrer Bemühungen nicht in der Lage, die Mängel zu beseitigen. Jedoch konnten sie gut erklären, woran sie gescheitert waren.

Diese praktische eigene Erfahrung der Studierenden und der Einsatz dieser didaktischen Methoden haben dazu geführt, dass sich die Entwicklerteams bereits in der Modellierungsphase des zweiten Projektes Gedanken über eine gute Bezeichnerwahl und die Vermeidung von langen Methoden, langen Parameterlisten und Gott-Klassen gemacht haben. Bei der Implementierung haben einige Gruppen versucht, der Regel vom Martin [Ma09] zu folgen. Diese besagt, dass eine Methode nur vier Zeilen lang sein sollte.

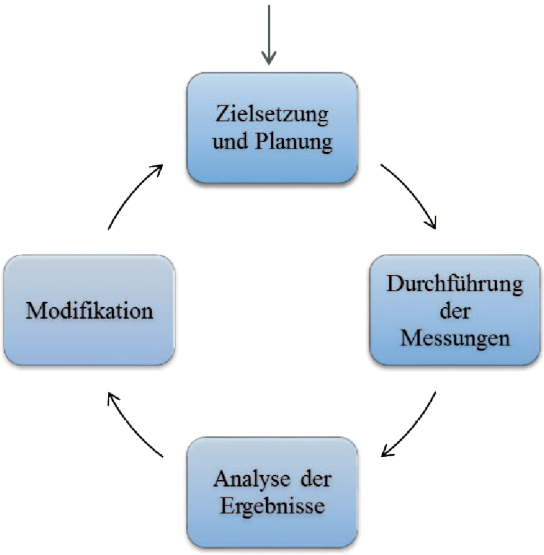

Abb. 2: PDCA-Zyklus

6 Entdeckung von Mängeln beim Modellieren

Frühzeitig erkannte Mängel sind einfacher zu beheben, als wenn diese bereits fest im Programm-Code verankert sind. Basierend auf unseren Untersuchungen [Sc15] wurden die Mängel analysiert, die bereits im Modell zu finden sind, und nach Hinweisen gesucht, die im Modell auf spätere mögliche Defekte im Programm-Code hindeuten. Diese Mängel werden in den folgenden Unterkapiteln vorgestellt.

6.1 Bezeichner

Die Wahl guter Bezeichner ist von großer Bedeutung für die Lesbarkeit des Programmcodes. Die Studierenden neigen dazu, kurze und nicht aussagekräftige Bezeichner zu wählen. Häufig wird auch Humor verwenden, der die allgemeine Verständlichkeit des Codes nicht unterstützt.

In Abbildung 3 wird eine Steuerungsklasse gezeigt, die im Rahmen eines Verwaltungsprojekts modelliert wurde. Die Studierenden musste ein Software-Produkt für die Organisation einer Cocktail-Bar erstellen. Mit Hilfe des Programms sollten Cocktail-Rezepte und ein Vorrat an Zutaten verwaltet werden. Die vorgestellte Klasse beinhaltet die Methode „getMaxMix“, die ein Beispiel für eine schlechte

Bezeichnerwahl darstellt. Der Name der Methode hält zwar die Java-Konventionen ein, ist aber inhaltlich nicht aussagekräftig. Sogar mit gutem Kontextwissen ist die Bedeutung schlecht zu verstehen.

In der ersten Iteration wurde festgestellt, dass die Java-Konventionen schon bei der Namensgebung in den UML-Diagrammen nicht eingehalten wurden. Da, wie bereits erwähnt, mit Hilfe von Astah aus dem Strukturmodell die Java-Code-Rahmen der Klassen und Methoden generiert werden, werden die schlecht gewählten Bezeichner aus dem Modell automatisch in den Java-Code übernommen.

Die Experimente haben gezeigt, dass die Qualitätskategorie Namensgebung sehr einfach und verständlich für die Studierenden ist. Mit Hilfe von Refactoring können ohne weitere Probleme und Nebenwirkungen die gefundenen Mängel schnell beseitigt werden. In der letzten Iteration haben die Studierenden nach der Einführung des Beseitigungszwangs bereits beim Modellieren auf die Bezeichnerwahl geachtet. Verstöße gegen die Java-Konventionen wurden deshalb im Programm-Code kaum noch gefunden.

Ein Grund für die hohe Qualität bei der Namensgebung für aus dem Modell übernommene Bezeichner in der letzten Iteration kann die Zusammenarbeit im Team beim Erstellen der UML-Modelle, aus denen die Java-Code-Rahmen generiert werden, sein. Hinzu kommen die von den Betreuern durchgeführten Reviews.

Besonders viele zu kurze Bezeichner stellten wir in Programmteilen fest, die von Einzelpersonen in der Implementierungsphase erstellt wurden. Diese Programmteile wurden bis dahin keinem definierten Code-Review-Prozess unterzogen.

6.2 Methoden

Als Qualitätsmerkmale der Methoden werden die Länge der Parameterlisten, die Länge der Methoden und ihre Komplexität betrachtet.

Mit Hilfe von Refactoring-Techniken, z.B. *Extract Method* [Fo99], kann die Lesbarkeit der Methoden verbessert werden, indem ihre Länge und Komplexität verringert werden. Diese Verbesserungen können aber nicht immer oder nicht so einfach umgesetzt werden. Ob nach dem Refactoring die Funktionalität erhalten geblieben ist, muss regelmäßig durch Tests überprüft werden. Die korrekte Beseitigung derartiger Mängel kostet viel Zeit.

Die Anzahl der Parameter lässt sich mit Hilfe von Refactoring reduzieren. Durch *Introduce Parameter Object* eine neue Klasse erzeugt und mehrere Parameter einer Methode lassen sich durch ein Objekt dieser Klasse ersetzen. Diese Methodik hat jedoch auch Nachteile. Die Lesbarkeit und die Übersichtlichkeit der Programme werden nicht unbedingt verbessert, da durch das Erzeugen von neuen Klassen die gesamte Struktur im Nachhinein verändert wird.

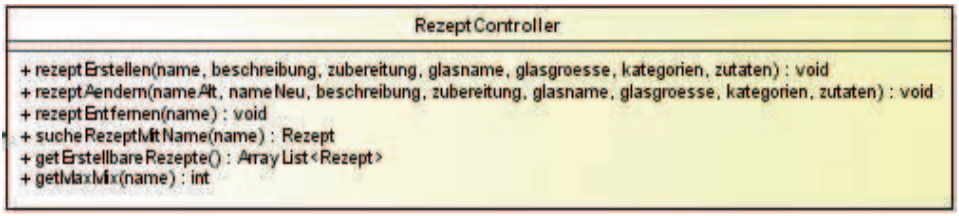

Abb. 3: Klasse in einem SoPra-Projekt

Die Länge der Parameterliste lässt sich sehr gut bereits im Modell überprüfen. Die vorgestellte Klasse in Abbildung 3 beinhaltet zwei Methoden mit sehr langen Parameterlisten. Laut unserer SoPra-Regeln (s. Tabelle 1) dürfen Methoden maximal vier Parameter übergeben bekommen. Eine Parameterliste darf nicht zu lang sein, weil sie schwer zu verstehen und zu benutzen ist. Insbesondere Boolesche Variablen, die wie Schalter funktionieren, sollten nach den Regeln von Robert Martin [Ma09] vermieden werden. Stattdessen sollten zwei Methoden angelegt werden, um die Komplexität zu verringern.

Die Wahrscheinlichkeit, dass Methoden mit sehr langen Parameterlisten auch zu umfangreich sind und eine hohe zyklomatische Komplexität besitzen, ist hoch. Die übergebenen Parameterwerte müssen überprüft werden. In unserem Beispiel muss beim Erstellen eines Rezepts überprüft werden, ob die Eingabe z.B. keine Sonderzeichen beinhaltet, kein leerer String übergeben wird oder ein Rezept mit dem Namen bereits hinzugefügt wurde. Diese Überprüfungen erhöhen die Komplexität und die Länge zusätzlich. Erfahrungsgemäß sind die Methoden, die mehr als 40 LOC haben, auch diejenigen, die von PMD als zu komplex erkannt werden.

Die Komplexität und die Länge der Methoden kann nicht erkannt werden, wenn nur das Klassendiagramm betrachtet wird, da ein Klassendiagramm nur die Struktur des Systems und nicht das Verhalten darstellt.

Zu Beginn eines Projekts werden in den Aktivitätsdiagrammen die Abläufe der Anwendungsfälle modelliert. Diese Anwendungsfälle entsprechen den Methoden in den Steuerungsklassen. Einen ersten Eindruck von der Komplexität eines Anwendungsfalls erhält man, wenn die Verzweigungen im Aktivitätsdiagramm betrachtet werden.

In der nächsten Phase, der Analysephase, erfolgt eine bereits implementierungsnähere Modellierung der Anwendungsfälle als Interaktion der Objekte in Sequenzdiagrammen. So wird das Verhalten der Methoden modelliert. Aus einem Sequenzdiagramm lässt sich zum einen ablesen, welche Teilaufgaben an andere Objekte delegiert werden, aber auch, welche Entscheidungen im Objekt selbst getroffen werden.

Beide Diagrammarten sind Teil der Verhaltensmodellierung und stellen Skizzen der Abläufe dar. Sie liefern nur Indizien auf zu hohe Komplexität. Anhand dieser Diagramme kann aber geschätzt werden, wie hoch ungefähr die zyklomatische Komplexität einer Methode sein könnte.

6.3 Klassen

Für die Studierenden war es besonders schwierig, die bereits entstandenen Mängel in dieser Kategorie durch Refactoring auf dem Programm-Code zu beseitigen.

Bei der Messung mit dem Tool zur statischen Code-Analyse wird geprüft, ob eine Klasse zu viel Verantwortung übernimmt, man spricht dann von einer Gott-Klasse. Die Definition von Gott-Klassen und die Kriterien, die die Entdeckung unterstützen, wurden von Lanza und Marinescu [LM06] übernommen.

PMD erkennt eine Gott-Klasse, wenn alle folgenden Kriterien (vgl. Tab. 1) verletzt sind:

- Die Summe der zyklomatischen Komplexität aller Methoden einer Klasse (WMC - Weighted Method Count) darf den Grenzwert von 47 nicht überschreiten.
- Die Anzahl der direkten Zugriffe einer Klasse auf die Attribute anderer Klassen (ATFD – Access To Foreign Data) darf nicht höher als 5 sein.
- Die dritte Metrik (TCC - Tight Class Cohesion) misst die Anzahl der direkt gekoppelten public-Methoden einer Klasse geteilt durch die maximale Anzahl der Verbindungen der Methoden. Dieser Wert sollte 0,33 nicht unterschreiten. Nur wenn dieser höher als 0,33 ist, wird nach der Definition von Lanza und Marinescu [LM06] die gewünschte Kohäsion der Methoden gewährleistet. Zwei Methoden sind als verbunden anzusehen, wenn sie auf die gleichen Instanzvariablen einer Klasse zugreifen. Innerhalb einer Klasse sollten die Methoden eine hohe Kohäsion besitzen. Andernfalls kann man die Methoden leicht auf zwei Klassen aufteilen.

Aus den durchgeführten Diskussionen mit den Studierenden und ihren Berichten ist bekannt, dass die Definition der Gott-Klassen für die Studierenden schwer zu verstehen ist. Einmal entstandene Gott-Klassen sind durch Refactoring schwer zu beseitigen. Die Berichte in der letzten Iteration zeigten, dass die Studierenden trotz offensichtlicher Anstrengungen damit überfordert waren, z.B. Gott-Klassen im Nachhinein zu beseitigen. Aus diesem Grund ist es notwendig, dass zu lange Klassen und zu komplexe Methoden bereits bei der Modellierung vermieden werden. Oft lässt sich schon bei der Modellierung erkennen, dass eine Klasse oder eine Methode zu viel Verantwortung übernimmt. Die Klasse in Abb. 3 ist ein Beispiel für eine Gott-Klasse, die bereits in der Modellierungsphase erkennbar ist.

Controller-Klassen, die für die Umsetzung der Anwendungsfälle verantwortlich sind, neigen häufig dazu, sich zu komplexen Klassen zu entwickeln. Die Modell-Klassen sind meist unproblematisch.

Besonders häufig haben wir Gott-Klassen in den Spiel-Programmen gefunden, wenn simulierte Gegner mit verschiedenen Spielstärken implementiert werden sollten. Oft ist es in diesen Klassen auch duplizierter Code zu finden, weil die verschiedenen Stärken ähnlich realisiert sind. Außerdem sind auskommentierte Methoden zu finden, die z.B. die schnellere Berechnung der möglichen Züge unterstützen sollten. „Duplizierter Code“

ist ein Mangel, der erst in der Implementierungsphase entsteht und im Modell meist noch nicht vorhergesehen werden kann. Sequenzdiagramme mit ähnlichen Abläufen könnten ein Indiz für diesen Mangel sein.

„Toter Code“ ist ein weiterer Mangel, auf den man bei den Klassen achten muss. Viele Methoden werden aus dem Strukturmodell automatisch generiert, von denen einige in der Implementierungsphase nicht benutzt werden. In Spielprogrammen ist es oft so, dass die Studierenden beim Modellieren des simulierten Gegners z.B. nicht wissen, welcher Algorithmus passend zu dem Spiel sein kann. Aus diesem Grund werden Methoden hinzugefügt, die sich später als überflüssig herausstellen. Derartige Mängel stören nicht beim Kompilieren, so dass sie nicht betrachtet und beseitigt werden, obwohl die Beseitigung einfach ist. „Toter Code“ ist ein Mangel, der erst in der Implementierungsphase entsteht und den es im Modell noch nicht gibt.

Wie bereits erwähnt, kann die zyklomatische Komplexität der Methoden nicht aus dem Klassendiagramm abgelesen werden. Dafür wird eine nähere Erläuterung der Methoden benötigt, welche die Aktivitäts- und Sequenzdiagramme liefern.

Die Aktivitäts- und die Sequenzdiagramme zeigen, wie kompliziert die Methoden sein können. Diese Erkenntnis muss beim Modellieren des Strukturmodells berücksichtigt werden, so dass nicht zu viele komplexe Methoden in einer Klasse zusammengefasst werden sollten. Erfahrungsgemäß ist, wenn zwei oder mehrere derartigen Methoden in einer Klasse zu finden sind, die Gefahr einer Gott-Klasse sehr groß (siehe Beispiel in Abb. 3).

Bei der Kontrolle der Code-Qualität am Ende des Projekts hat sich unsere Vermutung bestätigt. Für die Klasse „RezeptController“ hat PMD, abgesehen davon, dass die Anzahl der Methoden zu hoch war, für die Methode „rezeptErstellen“ eine zyklomatische Komplexität von 17 und für „rezeptAendern“ von 20 gemessen. Diese Methoden hatten auch ca. 65 LOC, was unseren Grenzwert von 40 LOC überschreitet. Außerdem war diese Klasse eine Gott-Klasse mit WMC 49, ATFD 18 und TCC 0.0.

Auch Klassen, die zu viele Attribute haben, können kritisch sein und sollten rechtzeitig betrachtet werden. Viele Attribute führen dazu, dass die Parameterlisten, z.B. im Konstruktor, lang werden. Eine derartige Klasse sollte in mehrere Klassen aufgeteilt werden.

Lange Klassen sind im SoPra eher selten, da der Projektumfang nicht so groß ist. Die längste Klasse, die wir gefunden haben, hatte 992 LOC. Diese war auch eine Gott-Klasse. WMC betrug 126, ATFD war 10-fach größer als der Grenzwert und TCC erreichte nur 0.042. In dieser Klasse wurden auch auskommentierter und duplizierter Code gefunden.

An den vorgestellten Beispielen ist erkennbar, dass die betrachteten Mängel voneinander abhängig sind. Eine Klasse wird zu lang, wenn sie entweder zu viele Methoden hat oder/und die Methoden zu lang sind. Wenn die Methoden zu lang sind, sind sie

erfahrungsgemäß auch zu komplex, dann kann sich die Klasse auch zu einer Gott-Klasse entwickeln. Die Komplexität der Methoden kann oft anhand der Anzahl der Parameter der Methoden, der Komplexität des zugehörigen Aktivitäts- und Sequenzdiagramms vorhergesehen werden (siehe Abbildung 4). Diese Abhängigkeit lässt sich von Anfang an kontrollieren, indem die Modell-Merkmale überprüft werden.

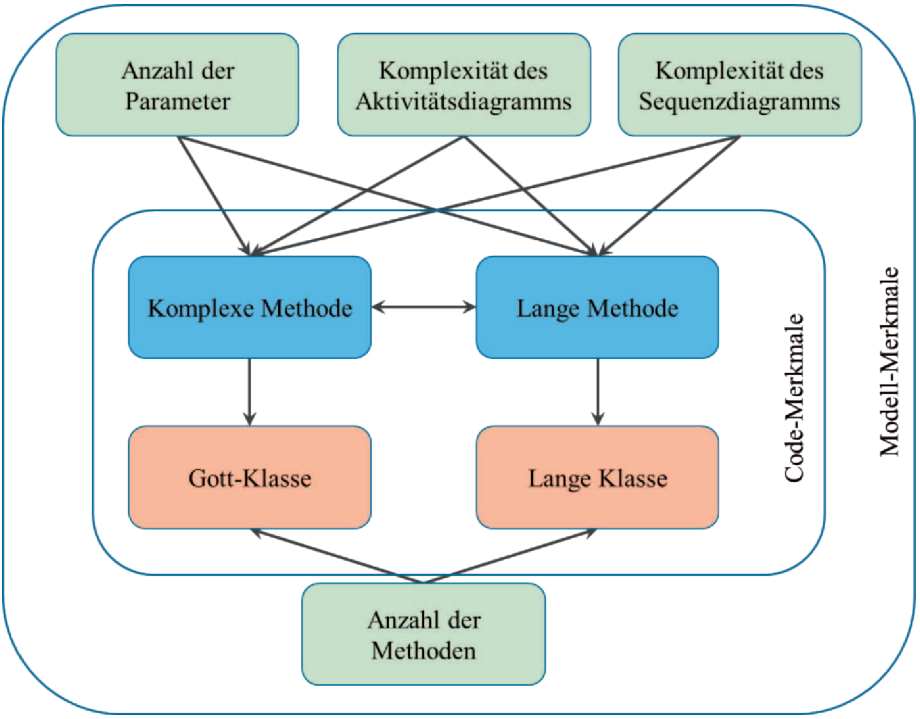

Abb. 4: Abhängigkeit zwischen den verwendeten Metriken

7 Ausblick

Wir wollen, dass die Studierenden lernen, die in den UML-Modellen abgebildete Komplexität richtig einzuschätzen, um von vornherein zu lange und zu komplexe Klassen zu vermeiden. Das ist das nächste Ziel, das wir uns für die Verbesserung unserer Lehrveranstaltung gesetzt haben.

Unsere bisherigen Arbeiten zur Code-Qualität ([Sc15], [Va15]) basieren auf den meist verwendeten Metriken im Bereich der objektorientierten Programmierung. Diese wurde von McCabe [Mc76] und Chidamber und Kemerer (CK-Metriken) [CK94] für Code-Qualität vorgestellt und analysiert. Die für unsere Lehrveranstaltung gewählten Grenzwerte und Bewertungskriterien haben sich als passend zum Umfang unserer

Projekte und der Erfahrungen der Studierenden erwiesen, so dass auch die Motivation nicht verletzt wurde. Für die Verankerung der Code-Qualität im Entwicklungsprozess hat sich der PDCA-Zyklus als erfolgreich erwiesen.

In der Modifikationsphase (s. Abbildung 2) haben wir in jeder Iteration die Reaktion der Studierenden berücksichtigt und passende Änderungen vorgenommen. In den zukünftigen Iterationen sollen Regeln, die auf den oben dargestellten Überlegungen zur gegenseitigen Beeinflussung der Qualitätsmetriken basieren und somit die Untersuchung der Qualität der UML-Modelle ermöglichen, verwendet werden.

Durch automatisches Analysieren der UML-Diagramme und der von Astah erstellten Code-Rahmen wollen wir die Studierende auf potentielle Fehler, die beim Modellieren übersehen wurden, aufmerksam machen.

Tang und Chen [TC02] stellen ein Werkzeug vor, das UML-Modelle ohne Berücksichtigung des Quellcodes überprüft. Das Tool basiert auf den CK-Metriken. Die betrachteten UML-Diagramme sind Klassen-, Aktivitäts- und Kommunikationsdiagramme und werden mit Hilfe des Tools Ration Rose erstellt. Tang und Chen nehmen an, dass alle Methoden in einer Klasse ähnliche Komplexität haben. Aus diesem Grund verwenden sie zu ihren Messungen die Metrik WMC1 statt WMC für alle Methoden im Modell. Nach der Definition von WMC1 hat jede Methode eine Komplexität von 1, d.h. es wird für die Komplexität der Klassen die Anzahl der Methoden pro Klasse gemessen. Das scheint erfahrungsgemäß ein sehr ungenauer und ungeeigneter Schätzwert zu sein. Wir streben die Entwicklung einer besseren Metrik für die Komplexität der Methoden im Modell auf Basis der zur Verfügung stehenden Aktivitäts- und Sequenzdiagramme an.

8 Fazit

In diesem Beitrag wurde eine langfristige Integration von Qualitätsaspekten für Programm-Code durch didaktische Maßnahmen vorgestellt. Am Ende der Implementierungsphase lassen sich die Mängel im Programm-Code mit Hilfe von Tools zur statischen Code-Analyse entdecken. Diese sind aber schwer zu beseitigen, besonders wenn die Zeit eher in das bessere Aussehen des Programms investiert wird. Komplexe Änderungen führen zur Entstehung von Kettenreaktionen, denen Programmieranfänger nicht gewachsen sind. Darunter leidet an erster Stelle die Motivation der Studierenden. Dies soll vermieden werden. In Rahmen eines modellbasierten Entwicklungsprozesses lässt sich hohe Code-Qualität von Anfang des Projektes an integrieren. Aus diesem Grund werden neben der Thematisierung der Code-Qualität in den Einführungsveranstaltungen, der Messung mit PMD und dem Beseitigungszwang die Modelle näher analysiert, um Mängel frühzeitig zu erkennen und zu beseitigen. Wir haben Merkmale im Modell identifiziert, die direkt zu Mängeln im Programm-Code führen, wie schlecht gewählte Bezeichner und lange Parameterlisten. Andere Merkmale liefern uns Hinweise darauf, welche Methoden und Klassen zu lang und zu komplex

werden könnten.

Tom DeMerko sagt, dass die Alternative zur Fehlerbeseitigung die Fehlerlosigkeit ist [De04]. Deswegen beschäftigen wir uns mit der Integration von Qualität Im Rahmen eines Software-Entwicklungsprozesses von Anfang an.

Literaturverzeichnis

[As15] Astah, http://astah.net/, abgerufen am 21.10.2015.

[Ba94] Basili, V. R., Caldiera, G., Rombach, H. D.: The Goal Question Metric Paradigm, In: Encyclopedia of Software Engineering - 2 Volume Set, John Wiley & Sons, 1994.

[CK94] Chidamber, S. R., Kemerer, C. F.: A Metrics Suits for Object Oriented Design. IEEE Transaction on Software Engineering, 20(6), 476-493, June 1994

[De04] DeMarco, T: "Was man nicht messen kann, kann man nicht kontollieren", mitp-Bonn, 2004.

[Fo99] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA., 1999

[Ga09] Gamma, E., Helm, R., Johnson, R.: Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software. Addison-Wesley; 2009

[LM06] Lanza, M., Marinescu, R., Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the design of Object-Oriented Systems, Springer, 2006.

[Ma09] Martin, R. C.: Clean Code, Prentice Hall, 2009.

[Mc76] McCabe, Thomas: A complexity measure. In: IEEE Transaction on Software Engineering, Nr. 4, S. 308-320, 1976.

[PMD15] PMD, https://pmd.github.io/, abgerufen am 21.10.2015.

[Re14] Remmers, J., Code-Qualität im Software-Praktikum, Bachelorarbeit, Fakultät für Informatik, TU Dortmund, 2014.

[Ro70] Royce, W. W.: Managing the Development of Large Software Systems: Concepts and Techniques - WESCON Conference Proceeding, August 1970

[Ru12] Rupp, C., Queins, S. und die SOPHISTen, Nürnberg. UML 2 glasklar: Praxiswissen für die UML-Modellierung, Carl Hanser Verlag, München, 2012.

[Sc15] Schmedding, D., Vasileva, A., Remmers, J.: Clean Code - ein neues Ziel im Software-Praktikum, SEUH 2015, Dresden, 2015

[SP15] Software-Praktikum, https://sopra.cs.tu-dortmund.de/wiki/, abgerufen am 21.10.2015.

[TC02] Tang, M. H., Chen, M. H.: Measuring OO Design Metrics from UML, UML 2002, Springer-Verlag Berlin Heidelberg, 2002

[Va15] Vasileva, A., Schmedding, D.: Integration von Qualitätsaspekten in einen Entwicklungsprozess, In Proceeding of Metrikon 2015, Köln, 2015