Clean Java – Von Anfang an!

Anna Vasileva, Doris Schmedding

Technische Universität Dortmund, Deutschland
anna.vasileva | doris.schmedding@tu-dortmund.de

Zusammenfassung. In diesem Beitrag werden die Sensibilisierung der Entwickler für guten Code und die Integration von Qualitätsaspekten in einen Software-Entwicklungsprozess in der Informatik-Ausbildung vorgestellt. Für eine langfristige und erfolgreiche Verankerung des Themas Code-Qualität als Entwicklungsziel sind mehrere Vorgehensschritte notwendig. Neben den passend gewählten Werkzeugen zur statischen Codeanalyse, Metriken und Grenzwerte spielen eine hohe Motivation und Konzepte für die Beseitigung der gefundenen Mängel eine entscheidende Rolle für die Qualität des Codes und die Entwicklung eines besseren Programmierstils.

1 Einleitung

Das Software-Praktikum (SoPra) ist eine Bachelor-Veranstaltung, die in den ersten Semestern des Informatik-Studiums stattfindet. In der Realität sind die Studierenden, die diese Veranstaltung besuchen, zwischen 3. und 10. Semester. Im SoPra setzen die Studierenden in Software-Projekten die Inhalte der Software-Technik in die Praxis um. Erst im SoPra kommen die in den vorangehenden Veranstaltungen gelernten Techniken, Vorgehensmodelle und Konzepte der Programmiersprachen gemeinsam zum Einsatz. Die Studierenden setzen verschiedene Aspekte der Software-Entwicklung wie Planen, Modellieren mit UML, Programmieren und Testen um.

Die zufällig zusammengesetzten Achtergruppen bearbeiten gleichzeitig dieselben Aufgaben und werden dabei von Betreuern bzw. Betreuerinnen, die den Entwicklungsprozess gut kennen, unterstützt. Das Sopra wird dreimal im Jahr durchgeführt und es nehmen jeweils etwa 60-80 Studierende teil. Im einem SoPra werden jeweils zwei Projekte durchgeführt. Das erste Projekt ist meist eine Verwaltungsaufgabe. Im zweiten Projekt müssen die Studierenden oft ein Computerspiel realisieren. Das SoPra hat insgesamt einen Arbeitsumfang von etwa 180 Zeitstunden. Die Modellierung mit UML wird mit dem Tool Astah [1] durchgeführt. Zur Realisierung der Projekte wird die objektorientierte Programmiersprache Java eingesetzt. Die Programmierumgebung basiert auf Eclipse, in das einige nützliche Plugins integriert sind, wie z.B. Subclipse für den SVN-Zugriff und der WindowBuilder für die Gestaltung der grafischen Benutzungsschnittstelle.

Erst in der Zusammenarbeit im Team und der Umsetzung der Software-Entwicklungstechniken wird klar, wie wichtig die Lesbarkeit, die Verständlichkeit und die gute Wartbarkeit des entstehenden Programms sind. Trotzdem hat eine erste Messung (siehe Kapitel 3) mit Hilfe eines Werkzeugs zur statischen Code-Analyse gezeigt, dass für die Studierende die Qualität des Codes nicht im Fokus steht. Ein Grund dafür mag sein, dass in den vorangehenden Veranstaltungen der Schwerpunkt nicht auf

Code-Qualität liegt. Auch im SoPra versuchen die Studierenden in erster Linie für sie neue Technologien kennenzulernen und funktional korrekte Programme zu schreiben.

Die erste durchgeführte Analyse von studentischen Projekten [2] hat gezeigt, dass typische Mängel in folgenden Bereichen zu finden sind:

- Namensgebung und Java-Konventionen
- Komplexität und Länge der Methoden
- Verantwortlichkeit und Länge der Klassen

Mehr Details über die festgestellten Mängel sowie über die verwendete Metriken und Grenzwerte werden im Kapitel 2 vorgestellt.

1.1 Vorgehensweise

Um in der Lehrveranstaltung Software-Praktikum langfristig und nachhaltig für alle Jahrgänge von Studierenden die Erhöhung der Qualität des implementierten Codes zu gewährleisten, müssen in mehreren Iterationsschritten qualitätsverbessernde Maßnahmen in den Entwicklungsprozess integriert werden. Das nachfolgend vorgestellte Vorgehen orientiert sich an dem Plan-Do-Check-Act-Zyklus (PDCA) [3]. Gemäß des PDCA-Zyklus (siehe Abb. 1) wird eine Iteration geplant und durchgeführt. Danach werden die Ergebnisse überprüft und diskutiert. Anschließend folgt die nächste Iteration.

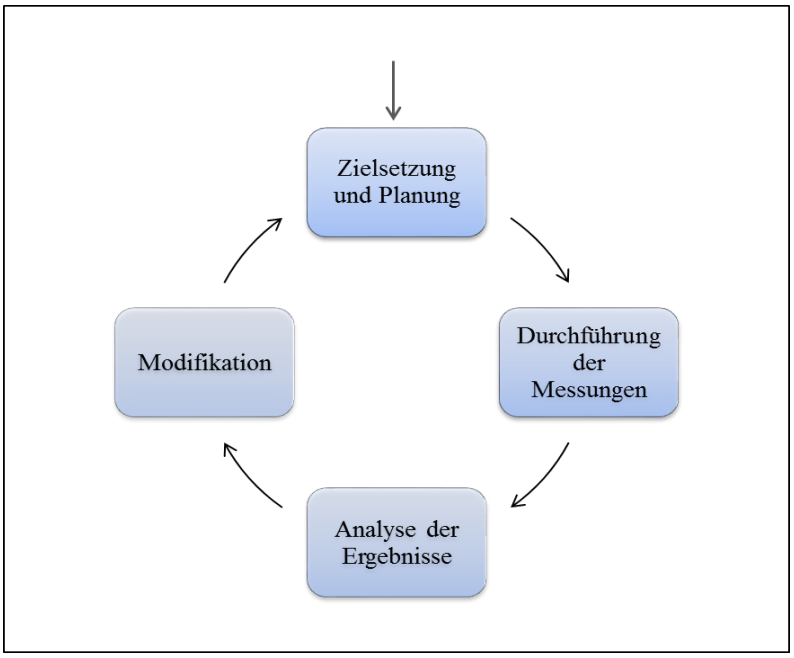

Abbildung 1: PDCA-Zyklus

In jedem Zyklus werden die Iterationsschritte wiederholt und die nächsten Ziele definiert. Wir beschreiben drei Zyklen, die zur Verbesserung der Codequalität durchgeführt wurden, bis erste gute Ergebnisse zu sehen waren.

Der Aufbau des Beitrags orientiert sich an dieser Vorgehensweise. In Kapitel 2 werden zunächst die Zielsetzung und Planung der ersten Iteration behandelt. Dann folgen in Kapitel 3 die Darstellung der ersten Messung, die Diskussion der Ergebnisse und die auf dieser Basis getroffenen Maßnahmen zur tieferen Verankerung des Themas Code-Qualität bei den studentischen Entwicklern. In der zweiten Iteration, die in Kapitel 4 vorgestellt wird, zeigen die getroffenen Maßnahmen leider keine positive Wirkung. Erst in der dritten Iteration, beschrieben in Kapitel 5, sind die neuen Maßnahmen erfolgreich. Es gelingt uns, das Thema Code-Qualität vom Anfang des Projekts an in den Fokus zu rücken. Der Beitrag schließt mit einem Fazit.

2 Zielsetzung und Planung

Gemäß der Goal-Question-Metrik-Methodik (GQM) [4] wurden als Erstes die Ziele für bessere Code-Qualität auf Basis typischer gefundener Mängel in studentischen Projekten definiert. Um diese Ziele zu erreichen, werden Fragen formuliert. Anschießend werden Metriken ausgewählt, mit deren Hilfe diese Fragen beantwortet werden können.

In seinem Buch *Clean Code* [5] präsentiert Robert Martin als erfahrener Software-Entwickler eine Fülle von typischen Qualitätsmängeln. Für unsere Veranstaltung wurden davon diejenigen ausgewählt, die häufig in studentischen Projekten vorkommen, die für die Studierenden leicht verständlich sind und die mit Tool-Unterstützung möglichst gut zu entdecken und zu beheben sind.

Für die Suche nach Mängeln in den studentischen Projekten wurde die Tool-unterstützte statische Code-Analyse angewendet. Diese können die Studierenden in der Implementierungsphase von Anfang an einsetzen, auch wenn der Code noch nicht kompilierbar ist. Tools zur statischen Code-Analyse bieten die Möglichkeit, gezielt nach verschieden Mängeln zu suchen und selbstdefinierte Grenzwerte zu verwenden. Selbstverständlich können die Messungen nicht völlig automatisiert durchgeführt werden. Der Code und insbesondere die Fundstellen müssen genau angeschaut werden, da solche Tools z.B. humorvolle oder sinnlose Bezeichner, die in studentischen Projekten häufig vorkommen, nicht finden können.

Das im SoPra verwendete Werkzeug zur statischen Code-Analyse ist PDM [6]. Aus den vielen zur Verfügung stehenden Tools zur statischen Code-Analyse, die sich teilweise sehr ähnlich sind, haben wir PMD gewählt, da PMD benutzerfreundlich und für Anfänger geeignet ist.

Auf Basis von typischen Defiziten in studentischen Projekten und den Ausbildungszielen im Bereich der Software-Qualität wurden für PMD eigene Regeln definiert [2]. Diese werden im XML-Format notiert und stehen allen GruppenbetreuerInnen sowie den Studierenden zur Verfügung. Wie bereits erwähnt, wurden die Ziele auf der Basis der Analyse der studentischen Projekte [2] und des Buches *Clean Code* [Mar09] festgelegt. Die gewählten Metriken und Grenzwerte orientieren sich am Projektumfang, den typischen Mängeln in studentischen Programmen sowie den in die Literatur [5, 6, 7, 8] vorgestellten Grenzwerten. In der

Literatur sind auch Statistik-basierte Grenzwerte zu finden, die z. B. nicht einfach übernommen werden können, da der in studentischen Projekten geringere Umfang und die geringere Erfahrung in Betracht gezogen werden müssen.

Die ausgewählten Qualitätsaspekte werden in die drei Gruppen Bezeichner, Methoden und Klassen eingeteilt. Angegeben sind jeweils außerdem die zur Messung verwendeten Metriken und die gewählten Grenzwerte.

2.1 Bezeichner

Eine gute Bezeichnerwahl ist von großer Bedeutung für die Lesbarkeit und die Verständlichkeit des Programmcodes. Die Analyse der studentischen Projekte hat gezeigt, dass die Studierende dazu neigen, kurze, wenig sinnvolle oder humorvolle Bezeichner zu wählen, was den Regeln für hohe Code-Qualität widerspricht.

Die Entdeckung der schlechten Bezeichner erfolgt mit Hilfe von PMD. Das Werkzeug prüft die Länge der Bezeichner. Da erfahrungsgemäß kurze Bezeichner inhaltlich nicht aussagekräftig sind, ließen wir die vier oder weniger Zeichen langen Bezeichner vom Tool herausfiltern.

Die Bezeichner müssen auch manuell, stichprobenartig kontrolliert werden. Auch wenn die Bezeichner gemäß der Regeln für die Länge gewählt wurden und die Java-Konventionen eingehalten werden, können sie dennoch sinnlos oder humorvoll sein, oder missverständliche Information liefern.

Auch die Einhaltung der Java-Konventionen zur Namensgebung kann von Werkzeugen zur statischen Code-Analyse größtenteils kontrolliert werden. PMD kann nicht erkennen, ob der Bezeichner einer Methode mit einem Verb anfängt, PMD kann aber z.B. feststellen, ob ein Klassenbezeichner mit einem Großbuchstaben beginnt.

2.2 Methoden

Die Länge der Methoden und die zyklomatische Komplexität werden ebenso mit dem Werkzeug zur statischen Code-Analyse untersucht. Aufgrund der von uns definierten Regeln erkennt PMD Methoden als mangelhaft, die länger als 40 LOC (Lines of Code) sind, wobei die Kommentare und die leere Zeilen mitgezählt werden. Nach Martin [5] dürfen die Methoden nicht länger als vier Zeilen sein, wobei jede Schleife nur eine Zeile Code beinhalten darf. McConnell [7] sagt, dass eine Methode zwischen 100 und 200 Zeilen haben darf. Der gewählte Grenzwert von 40 LOC ist für den relativ geringeren Umfang der Projekte und noch unerfahrene Entwickler akzeptabel und führt zu Methoden, die überschaubar sind, so dass Fehler schnell und einfach gefunden werden können.

Die Komplexität von Methoden als wichtiger Aspekt der Code-Qualität wird häufig in Form der zyklomatischen Komplexität gemessen. Diese lässt sich mit Hilfe von Tools zur statischen Code-Analyse leicht kontrollieren. Für die Analyse der studentischen Projekte wurde der Grenzwert von 10 von PMD übernommen [6]. Dieser darf nicht überschritten werden.

Neben der zyklomatischen Komplexität und der Länge der Methoden wurde die Anzahl der Parameter untersucht. Wenn die Anzahl der Parameter einer Methode vier überschreitet, dann meldet PMD nach den von uns definierten Regeln die Stelle als mangelhaft. Martin [5] sieht eine maximale Länge von drei vor, wobei drei Parameter

nur in Ausnahmefällen akzeptabel seien. PMD sieht in den mitgelieferten Regeln einen Maximalwert von 10 vor.

Eine Parameterliste darf nicht zu lang sein, weil diese schwer zu verstehen und zu benutzen wird. Außerdem ändern sich solche Listen beim Implementieren ständig. Die Nutzung vieler Parameter kann dazu führen, dass die Parameter des gleichen Typs verwechselt werden können. Lange Parameterlisten können auch zur Entstehung von redundantem Code führen. [9]

2.3 Klassen

Ebenso wie lange Methoden müssen auch lange Klassen vermieden werden, um eine hohe Übersichtlichkeit und eine gute Testbarkeit und Wiederverwendbarkeit zu erreichen. Die Länge der Klassen sowie die Anzahl der Klassen hängt laut [8] vom Projektumfang ab. Da der Projektumfang im SoPra nicht so hoch ist, haben wir in den SoPra-Regeln festgelegt, dass PMD eine Klasse als zu lang erkennen soll, wenn diese mehr als 400 LOC hat. Standardmäßig ist hierfür bei PDM 1000 Zeilen eingestellt [6].

Bei der Messung soll auch geprüft werden, 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 [8] übernommen.

PMD erkennt eine Gott-Klasse, wenn alle folgenden Kriterien 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 Rate 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 er höher als 0,33 ist, wird nach der Definition von Lanza und Marinescu [8] 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.

3 Die erste Messung

Die erste Messung fand im Ferien-SoPra im SS 14 statt. Die Studierenden wussten beim Programmieren nicht, dass der Programmcode untersucht wurde. Nach der Messung am Ende des SoPras wurden die festgestellten Mängel den Gruppen detailliert präsentiert. Die Gesamtergebnisse sind in Abbildung 2 dargestellt.

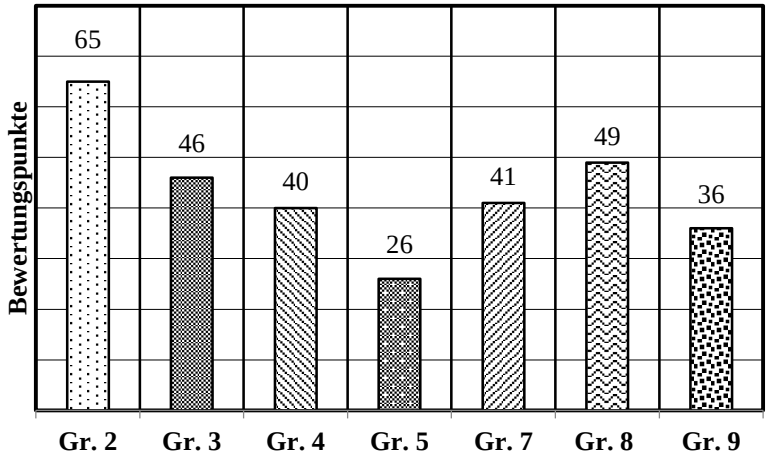

Abbildung 2: Gesamtergebnis im Freien-SoPra im Sommer 2014

Zu sehen sind die von den Gruppen jeweils insgesamt erreichten Punkte. Jede Gruppe startet mit einem Pluspunkte-Konto von 95 Punkten, aufgeteilt in die oben definierten drei Kategorien (Bezeichner, Methoden und Klassen) und „Sonstiges". Bei jedem gefundenen Verstoß wird für die entsprechende Kategorie ein Punkt abgezogen.

In Bezug auf den Qualitätsaspekt Bezeichnerwahl haben wir folgendes festgestellt. Wie bereits erwähnt, misst das Tool PMD die Länge der Bezeichner, über deren Aussagekraft das Tool keine Entscheidung treffen kann. Deshalb müssen die Fundstellen manuell geprüft werden. Wir haben Bezeichner gefunden, die zwar kurz, im Aufgabenkontext aber sinnvoll sind, wie z.B. "Zug" oder "Game" als Klassenbezeichner. Diese werden nicht als Mängel betrachtet. Bei anderen Bezeichnern der Art "m1", "m2" oder "p" fehlt die Aussagekraft, und sogar mit gutem Kontextwissen sind sie schlecht zu verstehen. Derartige Bezeichner werden gerne für Methodenparameter und lokale Variable verwendet.

Die erste Messung ergab außerdem, dass die Java-Namenskonventionen von den Studierenden weitgehend eingehalten wurden. In der Kategorie Namensgebung blieben deshalb trotz oft schlechter Bezeichnerwahl einige Punkte erhalten.

Auffällig war, dass einige Gruppen nur zwei bis drei Methoden mit zu hoher zyklomatischer Komplexität in ihrem Projekt hatten, wohingegen andere Gruppen sehr viele derartige Methoden entwickelt haben. Eine Gruppe hatte eine Methode mit einer zyklomatischen Komplexität von 67 geschrieben.

Die Methoden, die von PMD als zu komplex erkannt wurden, waren auch diejenigen, die offensichtlich zu lang waren. Im Programmcode der Gruppe 9 wurde vom PMD eine Methode mit 186 Zeilen und einer zyklomatischen Komplexität von 49 entdeckt.

Lange Klassen sind im SoPra eher selten, da der Projektumfang nicht so groß ist. Die erste Messung hat gezeigt, dass viele Klassen, die von PMD als zu lang erkannt wurden, auch als Gott-Klassen identifiziert wurden. Die längste Klasse hatte 992 LOC. Diese Klasse hat auch die Gott-Klasse-Regel von Lanza und Marinescu [8], die in Kapitel 2.3 vorgesellt ist, verletzt. WMC betrug 126, beim Grenzwert von 47, ATFD

war fast 10-fach größer als der Grenzwert und TCC erreichte etwa 0.042. In dieser Klasse wurden auch auskommentierter und unbenutzter Code gefunden, wie z.B. Parameter, Attribute sogar ganze Methoden. Diese „Baustellen“ wurden auch als Mängel notiert.

Zum Punktabzug haben auch die schlechte Sortierung der Komponenten in den Klassen sowie tief verschachtelte if-Anweisungen geführt.

3.1 Analyse der Ergebnisse

Im Rahmen der ersten Messung wurden viele der bei der Planung angenommenen Mängel [10] festgestellt, was für die passende Wahl der betrachteten Qualitätsaspekte, der Metriken und des Code-Analyse-Werkzeugs spricht. Da nicht nur die Auswahl der Metriken, sondern auch die der passenden Grenzwerte sehr wichtig für die Qualitätsmessung und die Erreichbarkeit der Ziele sind, liefert die Evaluation auch Information darüber, ob die Grenzwerte gut gewählt und erreichbar sind. Die Erreichbarkeit der Ziele ist eine Voraussetzung für die Akzeptanz der Messung, die Erhöhung der Motivation und zur Verbesserung des Programmierstils der Entwickler.

Ein Grund für die weitgehende Einhaltung der Java-Konventionen kann die Zusammenarbeit im Team beim Erstellen der UML-Modelle sein, aus denen die Java-Code-Rahmen generiert werden. Weiterhin stellten wir besonders viele einbuchstabige Bezeichner bei den Parametern in Programmteilen fest, die von Einzelpersonen erstellt wurden. Diese Programmteile wurden bis dahin keinem definierten Code-Review-Prozess unterzogen.

Nach unseren Erfahrungen in studentischen Projekten sind die langen Methoden auch diejenigen, die eine hohe zyklomatische Komplexität aufweisen.

Alle festgelegten Grenzwerte haben sich als geeignet erwiesen, da sie von Studierenden eingehalten werden konnten. Auch die ausgewählten Metriken haben sich aus unserer Sicht für das Ziel bewährt, die Lesbarkeit und Verständlichkeit des Codes zu steigern, da sie den Studierenden überwiegend gut vermittelbar waren.

3.2 Modifikationen

Die Ergebnisse der Messungen und damit die erreichte Code-Qualität wurden allerdings nicht als zufriedenstellend angesehen. Folgende Maßnahmen wurden deshalb getroffen:

- Das Thema Code-Qualität wird von Anfang an in den Ablauf den SoPras integriert.
- Die Messung wird in der Einführungsveranstaltung angekündigt.
- Zur weiteren Verankerung des Themas Code-Qualität wurden Diskussionen mit den Studierenden über ihre Ergebnisse nach dem ersten Projekt geplant.
- Außerdem wird als flankierende Maßnahme stärker auf das Thema Refactoring eingegangen. Den Studierenden wird in einer Präsentation gezeigt, wie man die in Eclipse vorhandenen Refactoring-Techniken einsetzt.
- Zusätzlich wird den Studierenden im SoPra-Wiki [11] Lernmaterial in Form von Tutorials zu den Themen PMD, Clean Code und Refactoring zur Verfügung gestellt.

Unter Refactoring [9] versteht man Techniken, die zur Verbesserung des Quellcodes zu verwenden sind, wobei aber die Funktionalität bestehen bleibt. Da man nicht davon ausgehen kann, dass studentischer Programmcode absolut mängelfrei ist, werden Refactoring-Techniken zu jedem von uns als besonders relevant betrachteten Qualitätsbereich eingeführt.

3.2.1 Refactoring für Bezeichner

Die Qualitätsaspekte Namensgebung und Java-Konventionen sind für die Studierende sehr einfach zu realisieren, weil diese gut nachvollziehbar und Mängel leicht zu beseitigen sind. Mit Hilfe der Refactoring-Technik *Rename* lassen sich alle Bezeichner schnell und ohne großen Aufwand verbessern. Ein Beispiel dafür ist, dass ein Mitglied der Siegergruppe 2 (s. Abb. 2) alle kurzen Bezeichner mit Hilfe von *Rename* vor der Messung ersetzt hat. Deswegen hat diese Gruppe in diesem Bereich die volle Punktzahl, wogegen andere Gruppen keinen Punkt behalten konnten.

3.2.2 Refactoring für Methoden

Die Lesbarkeit, die Komplexität und die Länge der Methoden können mit den Refactoring-Techniken *Extract Method* oder *Extract Local Valiable* verbessert werden. Durch *Extract Lokal Variable* wird ein Ausdruck durch eine neue lokale Variable ersetzt. Durch *Extract Method* wird ein Block von Anweisungen einer Methode in eine neue Methode ausgelagert. Diese Technik kann auch eingesetzt werden, um die Verständlichkeit des Codes zu erhöhen, z.B. wenn ein Teil der Methode gut kommentiert werden muss, damit die Funktionalität besser verstanden wird. [9]

Allerdings ist der Aspekt Refactoring für Methoden nicht so einfach umzusetzen. Ob die Funktionalität erhalten geblieben ist, muss deshalb durch Tests überprüft werden. Durch *Extract Method* entsteht ein neuer Sichtbarkeitsbereich für Instanz-Variable und Parameter. Wenn beispielsweise mehrere lokale Variable in dem Bereich, der zum Extrahieren markiert ist, deklariert und ihnen Werte zugewiesen werden, ist *Extract Method* nicht anwendbar.

Die Anzahl der Parameter lässt sich mit Hilfe von *Introduce Parameter Object* reduzieren. Auf dieser Weise wird eine neue Klasse erzeugt und mehrere Parameter einer Methode lassen sich durch ein Objekt dieser Klasse ersetzen. So wird aber nicht unbedingt eine bessere Lesbarkeit und Übersichtlichkeit gewährleistet, da durch das Erzeugen von neuen Klassen die gesamte Struktur geändert wird.

In studentischen Projekten sind auch Literale zu finden, meist in Form von konkreten Zahlenwerten in Bedingungen. Diese sind bei der Fehlersuche oder Wartung schwer zu verstehen, selbst wenn Kontextwissen vorhanden ist. Mit Hilfe von *Extract Constant*-Refactoring können die Zahlen einfach durch eine statische finale Variable mit aussagekräftigem Bezeichner ersetzt werden. Dadurch wird der Code verständlicher und auch wartungsfreundlicher.

3.2.3 Refactoring für Klassen

Die Beseitigung einer Gott-Klasse oder einer zu langen Klasse kann erreicht werden, indem durch das Verschieben von Methoden ein geeigneter Teil des Codes in eine

andere Klasse verschoben wird. Zur Reduzierung der Länge und der Komplexität der Klasse können *Extract Class* oder *Move* der richtige Weg sein. Das Aufspalten einer zu langen Klasse in zwei kann durch *Extract Class* erreicht werden. Das ist leider oft mit großem Aufwand verbunden, deshalb muss man sich fragen, ob sich der Aufwand lohnt [9].

Für die Sortierung der Komponenten in einer Klasse und die Beseitigung von unbenutztem Code gibt es keine speziellen Refactoring-Techniken. Diese Mängel müssen manuell behoben werden. *Extract Method* kann verwendet werden, um der redundanten Code zu beseitigen.

4 Die zweite Messung

Im zweiten Einsatz zur Integration der Qualitätsaspekte in den Entwicklungsprozess im WS 14/15 wurde das Thema „Clean Code“ eingeführt, die Messung von Anfang an angekündigt und erläutert, nach welchen Mängeln bei der Analyse gesucht wird und wieso. Tutorials zur statischen Code-Analyse und Refactoring wurden bereitgestellt.

4.1 Analyse der Ergebnisse

Die Ergebnisse der Messung sind im Vergleich zu denen im SS14 trotz aller getroffenen Maßnahmen immer noch nicht besser. Bei der Präsentation und Diskussion der Ergebnisse des ersten Projekts zeigten sich die Studierenden durchaus einsichtig. Dennoch waren auch die Ergebnisse des zweiten Projekts im WS 14/15, die in Abbildung 3 dargestellt sind, nicht besser als die im ersten Projekt und die des vorangehenden Software-Praktikums im ersten Zyklus (siehe Abb. 2).

Die knappe Zeit im Projekt wird offenbar lieber in das schöne Aussehen der Benutzungsschnittstelle, in komplexe Algorithmen und noch mehr Funktionalität als in statische Code-Analyse und Refactoring investiert. Weiterhin scheint bei den studentischen EntwicklerInnen das Ziel, eine hohe Code-Qualität zu erreichen, eine deutlich geringere Priorität als ein schönes User-Interface und die Funktionalität des Programms zu besitzen.

Es reicht offenbar nicht aus, auf die Einsicht der Studierenden zu vertrauen. Obwohl die Betreuer bereits für das Thema sensibilisiert waren, konnten sie bei den Gruppen die tägliche Code-Analyse und mehr Sorgfalt im zweiten Projekt nicht erreichen. Die Studierenden haben zwar eine Code-Analyse am Ende des Projekts durchgeführt, als es kaum noch Zeit mehr gab und viele Mängel sich nicht mehr so leicht, z. B. mit Hilfe von einfachen Refactorings, beseitigen ließen, so dass viele Mängel erhalten blieben.

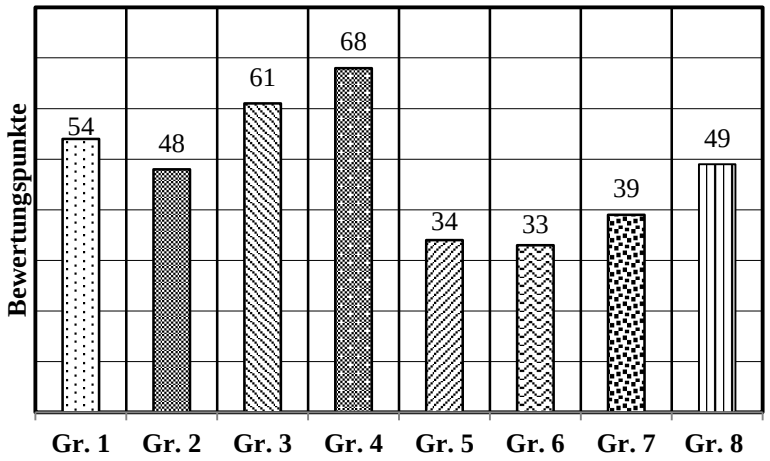

Abbildung 3: Gesamtergebnis im WS-SoPra 14/15

4.2 Modifikation

Die GruppenbetreuerInnen sollen frühzeitig mit den Gruppen zusammen Messungen der Code-Qualität durchführen und die Studierenden auf potenzielle Fehler und Mängel aufmerksam machen. Sie sollen mit den Gruppen diskutieren, wie man potentielle Mängel frühzeitig erkennen kann oder wie sich diese mit Hilfe von Refactorings beseitigen lassen, um die Motivation der Studierenden zu erhöhen.

Da es offenbar nicht ausreicht, über die Möglichkeit von Qualitätsmessungen und Maßnahmen zur Qualitätsverbesserung informiert zu sein, muss ein anderer Weg beschritten werden. Die Studierenden müssen "gezwungen" werden, die vorhandenen Tools und Techniken selbst auszuprobieren und anzuwenden.

5 Die dritte Messung

In der Einführungsveranstaltung wird das Thema Code-Qualität angesprochen. Die Studierenden führen eigenständig Messungen mit PMD und dem SoPra-Regelsatz durch. Die GruppenbetreuerInnen weisen wiederholt auf die Relevanz der Code-Qualität für das Projekt hin.

Nach dem ersten Projekt folgte eine Diskussion der Ergebnisse der durchgeführten Messungen mit den Gruppen und mit den BetreuerInnen. Vorschläge zur Vermeidung der festgestellten Mängel werden gesammelt.

Die Studierenden werden nach dem ersten und vor dem zweiten Projekt aufgefordert, die gefundenen Mängel mit Hilfe von Refactoring-Techniken zu beheben und einen Bericht darüber zu verfassen. Wenn das Verwenden von Refactoring nicht erfolgreich war, mussten die Studierenden dies schriftlich begründen. Viele Mängel insbesondere im Bereich der Namensgebung ließen sich problemlos z.B. durch das

Refactoring *Rename* beheben. Um die Länge der Methoden zu verkürzen und ihre Komplexität zu verringern konnte das Refactoring *Extract Method* oft erfolgreich eingesetzt werden. In manchen Fällen waren die Studierenden trotz ihrer Bemühungen zur Beseitigung der Mängel nicht in der Lage, dann konnten sie gut erklären, woran sie gescheitert waren.

Insbesondere zeigte sich in den zugesandten Berichten, dass die Studierenden trotz offensichtlicher Anstrengungen damit überfordert waren, die Gottklassen im Nachhinein zu beseitigen. Lange Parameterlisten wurden wie vorgesehen durch *Introduce Parameter Object* beseitigt. Die neu entstandene Struktur des Programmsystems wurde allerdings auch von den Studierenden kritisiert.

Insgesamt zeigte sich, dass die Mängel teilweise nur schwer behebbar waren und sich trotz Einsatz mehrerer Refactorings im Nachhinein nicht mit vertretbarem Aufwand beseitigen ließen.

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 dem Implementieren haben einige Gruppen versucht, der Regel vom Martin [5] zu folgen, dass eine Methode nur vier Zeilen lang sein sollte.

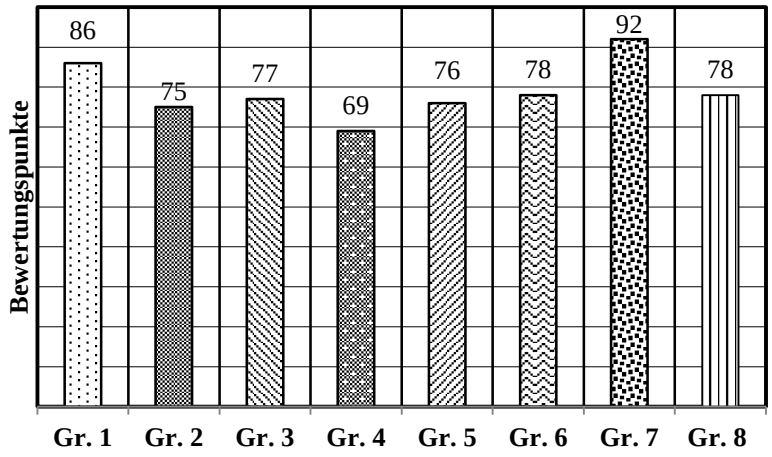

Abbildung 4: Gesamtergebnis im Freien-SoPra im Winter 2015

5.1 Analyse der Ergebnisse

Abbildung 4 stellt das Gesamtergebnis des zweiten Projekts im Ferien-SoPra nach WS 14/15 dar. Ein Vergleich der Werte in Abbildung 1 bis 4 zeigt, dass die besten Ergebnisse bei der vorerst letzten Iteration erzielt wurden. Besonders auffallend ist, dass die Ergebnisse aller Gruppen sehr hoch sind (zwischen 69 und 92).

Hervorzuheben ist, dass sich lange Parameterlisten nur noch bei Gruppe 4 finden. Die Hälfte der Gruppen konnte Methoden mit erhöhter zyklomatischer Komplexität vermeiden.

Aber fast alle Gruppen hatten immer noch Gott-Klassen. Das waren in der Regel die Klassen, die die Strategie der Spiele realisieren.

Mängel in den anderen Bereiche, die in Kapitel 2 vorgestellt worden sind, waren kaum noch zu finden.

5.2 Modifizieren

Die Auswertung der letzten durchgeführten Messung hat gezeigt, dass die Ergebnisse besser geworden sind, so dass wir von einer Verankerung des Themas Code-Qualität von Anfang der Entwicklung an durch die eingeführten didaktischen Maßnahmen ausgehen können. Dennoch sind noch einige Veränderungen in Bezug auf die Metriken notwendig.

Da die absoluten Methoden- und Klassenlängen gemessen wurden, gab es kritische Anmerkungen und Diskussionen zu den Messergebnissen. Dementsprechend besteht die Gefahr von adaptivem Verhalten, wie z. B. die Reduzierung der Kommentare. Um das zu vermeiden, wird bei der zukünftigen Messungen die Metrik NCSS (Non Commenting Source Statements) verwenden. Diese lässt Kommentare und die leere Zeilen unberücksichtigt.

Die Definition der Gott-Klassen ist für die Studierenden schwer zu verstehen und einmal entstandene Gott-Klassen sind durch Refactoring schwer zu beseitigen. Deshalb 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. Finden sich mehrere dieser Methoden in einer Klasse, so ist die Gefahr einer Gott-Klasse sehr groß.

6 Fazit

Eine langfristige Integration von Qualitätsaspekten konnte durch die Anwendung von didaktischen Maßnahmen in mehreren Iterationszyklen gewährleistet werden. Ein wichtiges Ergebnis ist, dass sich in einem modellbasierten Entwicklungsprozess eine hohe Code-Qualität mit vertretbarem Aufwand nur erreichen lässt, wenn dieses Ziel von den EntwicklerInnen bereits während der Modellierungsphase beachtet wird. Die Beseitigung von Mängeln am Ende der Implementierungsphase, wenn ein Tool zur statischen Code-Analyse auf die Mängel aufmerksam macht, ist nicht wirklich zielführend. Am Ende des Projekts ist einerseits die Zeit immer knapp und andererseits hat sich gezeigt, dass die Behebung mancher Mängel sehr aufwendig ist und Programmieranfänger überfordert. Komplexe Änderungen müssen durch die erneute Durchführung der Funktionstests überprüft werden. Komplexe Refactorings verändern die Struktur des Systems und können weitere Refactorings notwendig machen. So entstehen Kettenreaktionen, denen Programmieranfänger nicht gewachsen sind, und die Motivation der Studierenden leidet darunter.

Code-Qualität lässt sich also nur erreichen, wenn dieses Ziel von Anfang an verfolgt wird.

Literaturverzeichnis

[1] Astah, *http://astah.net/,* abgerufen am 09.07.2015.

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

[3] A. Syska, *Produktionsmanagement,* Gabler, 2006.

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

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

[6] PMD, *http://pmd.sourceforge.net/,* abgerufen am 09.07.2015.

[7] S. McConnell, Code Complete, Unterschleißheim: Microsoft Press, 2007.

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

[9] M. Fowler, *Refactoring - Wie Sie das Design vorhandener Software verbessern,* Köln: Addison-Wesley, 2000.

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

[11] SoPra, *https://sopra.cs.tu-dortmund.de/wiki/start,* abgerufen am 09.07.2015.

[12] Healt4J, *http://www.main-gruppe.de/cms/opencms,* abgerufen am 10.06.2015.

[13] Checkstyle, *http://checkstyle.sourceforge.net/,* abgerufen am 08.06.2015.

[14] N. Fenton und J. Bieman, Software Metrics: A Rigorous and Practical Approach, 2015: CRC Press.

[15] A. J. Riel, *Object-oriented design heuristics,* 1996: Addison-Wesley Publishing.

[16] D. Schmedding, *Ein Prozessmodell für das Software-Praktikum,* Zürich: SEUH, 2001.

[17] A. Syska, *Produktionsmanagement,* Gabler, 2006.