Clean Code – ein neues Ziel im Software-Praktikum

Doris Schmedding, Anna Vasileva, Julian Remmers, Technische Universität Dortmund

doris.schmedding | anna.vasileva | julian.remmers@tu-dortmund.de

Zusammenfassung

In diesem Beitrag wird ein Konzept vorgestellt, wie in der Informatik-Ausbildung die Sensibilität für guten Code erhöht werden soll. Das Software-Praktikum bietet einen geeigneten Rahmen, dieses Ausbildungsziel umzusetzen. Wir zeigen Code-Defizite, die sich in studentischen Projekten häufig finden. Es wird ein Tool vorgestellt, das bei der Entdeckung unterstützt, und Refactorings präsentiert, mit denen die Studierenden die selbst gefundenen Defizite leicht beheben können.

Motivation

In den ersten Semestern der universitären Ausbildung liegt der Schwerpunkt darauf, die Konzepte der verwendeten Programmiersprachen kennen zu lernen und funktional korrekte Programme zu schreiben. Dies wird zumindest an der Fakultät für Informatik an der TU Dortmund aufgrund der großen Anzahl der Studienanfänger weitgehend durch automatisierte Tests überprüft. In der Software-Technik-Vorlesung liegt der Schwerpunkt auf der Modellierung, Mustern [Gamma 2004] und dem Vorgehen in Software-Entwicklungsprojekten, weniger auf der Programmierung.

Erst in einem studentischen Software-Entwicklungsprojekt kommen die verschiedenen Aspekte der Software-Entwicklung wie Prozessmodell, Modellierung und Programmierung gemeinsam zum Einsatz und fügen sich zu einem Ganzen. Die Bedeutung von Lesbarkeit und Verständlichkeit des Codes kann sich den Studierenden aber erst wirklich erschließen, wenn mit mehreren Studierenden gleichzeitig in einem komplexen Projekten gearbeitet wird und die Studierenden anhand von selbst produzierten Beispielen eigene evtl. auch leidvolle Erfahrungen beim Lesen fremden Codes sammeln. Auch wenn die Wartbarkeit von Programmen ein wichtiges Ziel in der Software-Entwicklung darstellt, lässt sie sich im universitären Kontext, in dem studentische Projekte nur eine kurze Lebensdauer haben, nur schwer vermitteln.

Im Software-Praktikum (SoPra) in unserer Fakultät führen acht BA-Studierende gemeinsam Projekte durch. Etwa sechs bis 10 Gruppen bearbeiten gleichzeitig die gleichen Aufgaben. In einem Blockpraktikum in der vorlesungsfreien Zeit dauert eine Projekt drei Wochen. Es werden nacheinander in 6 Wochen zwei Projekte durchgeführt.

Wir verwenden die Programmiersprache Java. Zur Modellierung setzen wir UML mit dem Tool Astah ein. Aus dem Modell werden Java-Code-Rahmen generiert. Als Entwicklungsumgebung benutzen wir Eclipse mit dem SVN-Plugin Subclipse. Die Dokumentation mit JavaDoc und JUnit-Tests sind wichtige Bestandteile unseres Entwicklungsprozesses und inzwischen ebenso wie alle oben genannten Tools und Methoden allgemein akzeptiert. Am Ende eines Projekts erfolgt die Abnahme und Bewertung der Projekte gegenseitig durch die Gruppen. Die Ergebnisse der Bewertung werden im Plenum diskutiert.

Auch wenn man nach drei Wochen nicht erwarten kann, ein perfekt funktionierendes Programm zu erhalten, wird die funktionale Korrektheit als Ziel in der Software-Entwicklung von keinem der Beteiligten in Frage gestellt. Für die Studierenden ist in ihrem eigenen Projekt und bei der Bewertung der Ergebnisse der anderen Gruppen auch das Aussehen des Programms sehr wichtig, insbesondere bei Spielen. Bisher überhaupt nicht im Fokus stand die innere Qualität der Programme. Das wollen wir ändern. Unsere Vision ist, dass der Qualitäts-Check nach einer Änderung genauso selbstverständlich wie der JUnit-Test wird.

Vorgehensweise

Dazu identifizieren wir zunächst einmal typische Defizite in studentischen Projekten [Remmers 2014]. Aus der Masse von Qualitätsmängeln, die Martin [Martin 2008] beschreibt, wählen wir diejenigen aus, die

- für Studierende leicht verständlich sind,

- häufig in Studenten-Projekten vorkommen,
- leicht und möglichst mit Tool-Unterstützung erkennbar sind und
- mit Tool-Unterstützung einfach zu beheben sind.

Ab wann ein Qualitätsmangel vorliegt, ist eine Frage des persönlichen Anspruchs an die Qualität. Wenn man mit einem Tool Qualitätsmessungen durchführt, lassen sich Grenzwerte angeben, was noch toleriert wird und ab wann ein Defekt vorliegen soll. Wir geben jeweils an, für welche Grenzen wir uns entschieden haben und diskutieren ihre Sinnhaftigkeit.

Ein erster Einsatz unserer Messinstrumente und Grenzwerte im Software-Praktikum in der vorlesungsfreien Zeit nach dem Sommersemester 2014 diente zur Evaluation, ob die angenommenen Mängel tatsächlich mit Hilfe des von uns ausgewählten Tools festzustellen sind und in welchem Umfang (siehe "Ergebnisse unserer Messungen") sie auftreten. Am Ende des SoPras erhielten die Gruppen jeweils einen Bericht über die festgestellten Mängel.

In einem zweiten Einsatz im SoPra im WS 14/15 ist geplant, bereits in der Einführungsveranstaltung das Thema Code-Qualität anhand einiger Folien einzuführen und die Qualitätsmessung anzukündigen. Den Studierenden stehen im Wiki des Software-Praktikums Tutorials zum Thema Clean Code [Martin 2008], statische Code-Analyse und Refactoring [Fowler1994] für das Selbststudium zur Verfügung. Außerdem wird bei der Eclipse-Einführung gezeigt, wie einfache Refactorings durchgeführt werden. Im Rahmen der Reflexion der Erfahrungen im ersten Projekt [Schmedding 2011] soll jede Gruppe nach dem ersten Projekt einen Bericht über die festgestellten Mängel erhalten und eine Diskussion über die möglichen Ursachen und Folgen der Mängel geführt werden. Wir erwarten im zweiten Projekt eine Verbesserung der Code-Qualität.

Durch die Beschränkung auf zunächst einmal wenige Mängel und den Einsatz eines Tools zur Qualitätskontrolle, jeweils gekoppelt mit Hinweisen, wie man den Mangel mit Tool-unterstützten Refactorings einfach beheben kann, erhoffen wir uns eine große Akzeptanz unserer Qualitätsoffensive.

Clean Code

In [Matin 2008] werden sehr viele Qualitätsmängel beschrieben, die sehr ambitioniert sind und sich den Studierenden zum Teil nur schwer vermitteln lassen, z. B. dass eine Methode maximal vier Zeilen lang sein soll. Jedenfalls erschienen uns Martins Qualitätsanforderungen für unsere Studierenden völlig unerreichbar, so dass wir einige erreichbare Regeln [Remmers 2014] definiert haben, die von den SoPra-TeilnehmerInnen nicht verletzt werden sollten.

Die Regeln wurden in die folgenden drei Gruppen eingeteilt:

- **Bezeichner** sollen verständlich sein und die Java-Konventionen einhalten. Humor ist bei der Wahl der Bezeichner zu vermeiden!
- **Methoden** sollen nicht zu komplex sein, eine bestimmte Maximallänge nicht überschreiten und wegen der Vertauschungsgefahr nicht zu viele Parameter haben.
- **Klassen** sollen nicht zu lang sein, keinen toten Code enthalten und keine Gott-Klassen sein.

Gott-Klassen oder Gott-Objekte [Riel 1996] gehören zu den sogenannten Anti-Patterns und widersprechen dem „Single-Responsibility"-Prinzip [Martin 2008]**.** Als Gott-Klassen werden Klassen bezeichnet, deren Methoden zu komplex sind, die auf zu viele Attribute anderer Klassen zugreift und deren Methoden nur eine geringe oder gar keine Kohäsion besitzen.

Außerdem haben wir nach dem Anti-Pattern „Magische Werte" („Magic Values") und nach Literalen anstelle von Konstanten in Bedingungen suchen lassen, da sie die Lesbarkeit und Wartbarkeit der Bedingungen verletzen.

Statische Programmanalyse

Wir setzen die statische Programmanalyse hier im Software-Praktikum zur Aufdeckung von Qualitätsmängeln ein. Dazu verwenden wir das Eclipse-Plugin PMD[1]. PMD prüft die Einhaltung einer sehr umfangreichen Menge von vordefinierten Regeln, die jeweils unter bestimmten Oberbegriffen zusammengefasst sind. Manche der von PMD festgelegten Qualitätsregeln empfinden wir als zu streng, z. B. die Forderung, dass eine Methode nur ein „return statement" enthalten soll. Dagegen sollten unserer Meinung nach manche Grenzwerte wegen des relativ kleinen Projektumfangs und der begrenzten Projektlaufzeit niedriger liegen.

[1] http://sourceforge.net/projects/pmd/files/pmd-eclipse/

Der Benutzer kann unter `Preferences->PMD` einzelne Regeln oder Regelgruppen an- oder abwählen. Außerdem besteht die Möglichkeit, eine spezielle Regelmenge in Form einer XML-Datei (siehe Anhang B) zu laden. Für diesen Weg haben wir uns entschieden, um alle Studierenden mit den gleichen auf die Lehrveranstaltung zugeschnittenen Qualitätsmaßstäben zu versorgen. Wie bereits erläutert, haben wir für die Ausbildung im Software-Praktikum eigene Grenzwerte definiert. Auch Spillner und Linz empfehlen in [Spillner 2005], beim erstmaligen Einsatz eines Analysetools die Grenzwerte so zu wählen, dass die Akzeptanz eines derartigen Tools nicht gefährdet wird.

Wir haben PMD als Analysetool ausgewählt, weil es flexibel ist und gut in unsere Arbeitsumgebung passt. Außer PMD können die Studierenden auch das Eclipse-Plugin FindBugs[2] zur statischen Code-Analyse benutzen, das ebenfalls Bestandteil der vorbereiteten Entwicklungsumgebung ist.

Im SoPra-Wiki wird für das Selbststudium gezeigt, wie man PMD (und FindBugs) verwendet und eigene Regeln einbindet. Die im SoPra eingesetzten PMD-Regeln sind in Anhang B im XML-Format aufgelistet. Die gewählten konkreten Grenzwerte für die eingesetzten Zählmetriken kann man dem Anhang A entnehmen. Die Wahl der Grenzwerte wird jeweils im Zusammenhang mit den Ergebnissen der Messungen erläutert. Einige Grenzwerte haben wir von PMD übernommen, für andere haben wir unserem Ausbildungsumfeld angepasste Werte gewählt. Das Kriterium für die Erkennung einer möglichen Gott-Klasse wird im Zusammenhang mit den Messergebnissen (siehe Klassen) vorgestellt und diskutiert.

Nicht alle für das Software-Praktikum als wichtig ausgewählten Regeln des Clean Codes lassen sich automatisch überprüfen. Kein Tool kann feststellen, ob ein Bezeichner sinnvoll gewählt wurde. Als Indiz für eine sorgfältige Wahl dient uns die Länge eines Bezeichners, die wir messen können.

Aber wie Spillner und Linz [Spillner 2005] sind wir der Ansicht, dass sich Qualitätsstandards und Konventionen nur dann einführen und beibehalten lassen, wenn geeignete Prüfwerkzeuge vorhanden und leicht zu bedienen sind, da ein nicht automatisch überprüfbares Regelwerk nur als bürokratischer Ballast empfunden wird.

Refactoring

Sogenannte „Refactorings“ [Fowler 1999], Verbesserungen bestehenden Codes, werden eingesetzt, um die mit PMD gefundenen Mängel zu beseitigen. Wichtig ist, dass die Funktionalität erhalten bleiben muss, was durch ständige Tests überprüft wird. Viele Refactorings sind in unserer Entwicklungsumgebung Eclipse bereits standartmäßig integriert, so dass sie über das Kontextmenü leicht zu erreichen und anzuwenden sind.

Mit Hilfe von Tutorials im SoPra-Wiki für das Selbststudium wird gezeigt, welche Refactoring-Techniken die Studierenden einsetzen können, um die mit PMD (und FindBugs) gefundenen Mängel zu beseitigen. Ähnlich wie Martin beschreibt Fowler in [Fowler 1999] eine Fülle von Verbesserungen (~70), aus denen wir diejenigen ausgewählt haben, die wir für unsere Studierenden für praktikabel und gut nachvollziehbar halten. Diejenigen, die sich mit Tool-Unterstützung umsetzen lassen, sind die folgenden:

- **Rename** wird zur Verbesserung der Namensgebung eingesetzt und kann auf alle Bezeichner angewendet werden. Da alle Aufrufe mit geändert werden, können die Studierenden gut die Mächtigkeit des Werkzeugs erkennen.
- **Extract Method** kann eingesetzt werden, wenn eine Methode zu lang oder zu komplex geworden ist. Das Werkzeug legt für den markierten Bereich eine neue Methode an und ersetzt ihn durch den Aufruf der Methode.
- **Extract Local Variable** kann verwendet werden, wenn der Code aufgrund einer langen Aufrufkette nur noch schwer lesbar ist. Dann wird für die markierte Aufrufkette automatisch eine neue lokale Variable angelegt.
- Mit **Introduce Parameter Object** wird eine überlange Parameterliste durch ein Parameterobjekt ersetzt. Zusätzlich wird eine neue Klasse mit den früheren Parametern als Attributen angelegt, auf Wunsch auch mit entsprechenden Zugriffsmethoden.
- Mit **Extract Constant** kann eine Zahl in einer Bedingung in eine Konstante umgewandelt werden, um die Wartbarkeit und Lesbarkeit des Codes zu erhöhen.
- **Move** verschiebt eine Komponente einer Klasse innerhalb einer Klasse, bei Bedarf auch in eine andere Klasse, sofern die ursprüngliche Klasse eine Referenz auf die neue Klasse besitzt. Dann werden in Falle von Methoden ihre Aufrufe entsprechend geändert.

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

Alle anderen Mängel sollten manuell, also ohne Eclipse-Refactoring-Tool, behoben werden. Das ist z. B. die Beseitigung von totem Code kein Problem. Von der Anwendung von **Extract Class** in Eclipse raten wir ab, da dessen Funktionalität nicht der in [Fowler1999] beschriebenen entspricht. **Extract Class** wird eingesetzt, um eine zu komplex oder zu lang geratene Klasse zu teilen. Auch **Extract Method** kann nicht immer voll automatisch vom Tool durchgeführt werden. Wenn in der zu extrahierenden Anweisungsfolge mehrere Werte geändert werden, stößt die Code-Transformation von Eclipse an ihre Grenzen, die man auf jeden Fall berücksichtigen sollte.

Eine wirkliche Verbesserung des Codes erhält man in der Regel erst, wenn man eine Kombination von Refactoring-Techniken anwendet, z. B. erst **Extract Class** und danach **Move**. Selbstverständlich muss nach jeder Veränderung geprüft werden, ob die verlangte Funktionalität noch gegeben ist und ob die Mängel wirklich beseitigt wurden oder neue Mängel entstanden sind.

Ergebnisse unserer Messungen

In diesem Kapitel stellen wir die Ergebnisse unser Messungen vor, die wir durchgeführt haben, ohne dass die Studierenden zuvor informiert waren, dass wir die Codequalität analysieren würden.

Aufgabe in diesem Projekt war die Realisierung eines Computerspiels, das Brettspiel Cartagena[3] in den Varianten Jamaika und Tortuga. Das Spiel sollte mit folgenden Anforderungen realisiert werden:

- Drei verschiedene simulierte Gegner,
- Rücknahme der Züge bis zum Spielbeginn,
- Speichern und Laden angefangener Spiele,
- Highscore-Liste,
- Tipp für unerfahrene SpielerInnen,
- Editor für die freie Gestaltung des Spielfelds,
- Einlesen einer Spielsituation im vorgegebenen Format.

Am Software-Praktikum haben 8 Gruppen mit je 8 Mitgliedern teilgenommen. Alle Gruppen haben funktionstüchtige Spielprogramme produziert, die die oben aufgelisteten Anforderungen erfüllt haben. Da eine Gruppe (Gruppe 6) nicht unseren SVN-Server sondern Git[4] zur Versionsverwaltung benutzt hat, konnten wir keine Messung vornehmen. Eine Gruppe (Gruppe 1) ist nicht zustande gekommen. Abb. 1 und Abb. 2 zeigen deshalb den Umfang der Projekte von sieben Gruppen.

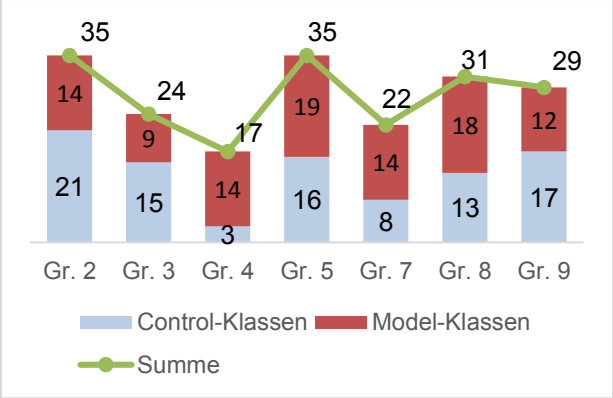

Abb. 1: Anzahl der Klassen

Die GUI-Klassen haben wir bei der Umfangssmessung nicht mit berücksichtigt, da sie zwar sehr umfangreich sind, ihr Programmierstil aber weitgehend von der Game-Engine (Slick2D oder LibGDX) geprägt ist. Wir interessieren uns mehr für die Teile der Programme, welche nach der Modellierung mit UML selbst entwickelt wurden.

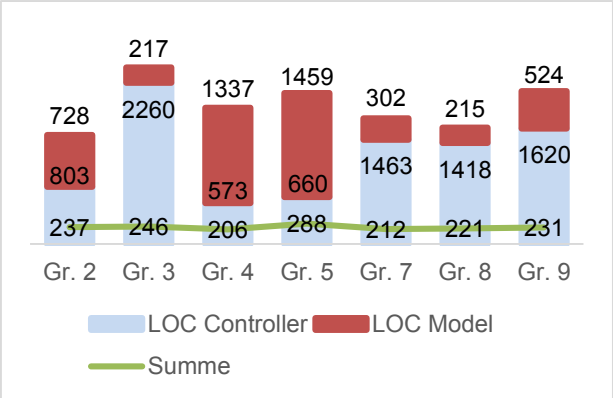

Abb. 2: LOC differenziert nach Model- und Control-Klassen und Anzahl der Methoden insgesamt

Man sieht, dass die Projekte sowohl in der Anzahl der Klassen (Abb. 1), zwischen 17 und 35, als auch im Umfang (Abb.2), zw. 1531 und 2477 LOC (Lines of Code), und Methodenanzahl, zw. 206 und 288, stark schwanken.

Da wir das MVC-Muster [Gamma u.a. 2004] für die Strukturierung der Software vorgeben, wird in Abb. 1 und Abb. 2 zwischen Model- und Control-Klassen unterschieden. Die beiden Gruppen 4 und 5 haben eher dem objektorientierten Paradigma folgend die Programmlogik den Klassen des Problembereichs, den Model-Klassen, zugeordnet, während die Model-Klassen Gruppen 3, 7, 8 und 9 fast nur Datenhaltung betreiben. Das wird in Abb. 2 gut deutlich: Die Model-Klassen von Gruppe 4 und

[3] http://de.wikipedia.org/wiki/Cartagena_(Spiel)

[4] http://git-scm.com/

5 sind deutlich umfangreicher als ihre Control-Klassen und die Model-Klassen der übrigen Gruppen. Wir werden später sehen, dass, wie zu erwarten war, die einzelnen Model-Klassen der anderen Gruppen, die rein der Datenhaltung dienen, nur kurz und wenig komplex sind. Das gilt auch für die Methoden dieser Klassen. Die Gruppe 2 hat während des Projektverlaufs ihre Strategie verändert, von zunächst „ganz dummen" Datenklassen, zu Model-Klassen mit mehr Logik. Daher rührt das relativ ausgeglichene Bild vom Umfang von Model- und Control-Klassen in Abb. 2.

Am Ende der Projektlaufzeit von drei Wochen haben wir ein Code-Review vorgenommen, indem wir die verfügbaren Projekte der Gruppen und die von PMD angezeigten Defekte kritisch analysiert und dokumentiert haben. Dazu mussten wir die Projekte aus dem jeweiligen SVN-Repository herunterladen und die Messungen mit PMD und den SoPra-spezifischen Regeln durchführen. Die Stellen, an denen PMD Defekte gefunden hat, haben wir uns genauer angesehen und selbst beurteilt, ob es sich um einen relevanten Verstoß handelt. Pro Gruppe haben wir zu Zweit etwa eine Stunde benötigt.

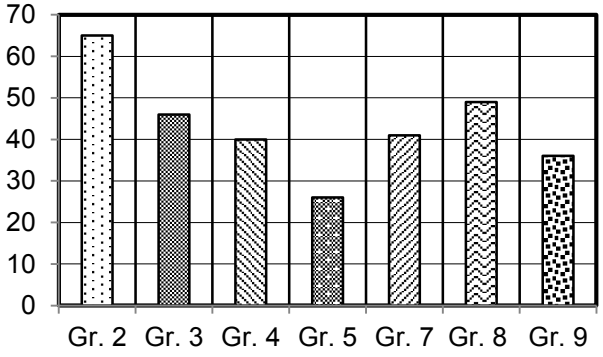

Abb. 3: Gesamtergebnis

Zum Projektabschluss bewerten die Gruppen gegenseitig ihre Projekte anhand eines von uns vorgegebenen Schemas, das in erster Linie die geforderte Funktionalität und die Benutzungs-freundlichkeit berücksichtigt. Analog dazu wurde ein Bewertungsschema (siehe Anhang A) mit Punkten entwickelt, um die Gruppe (mit Schokolade) zu belohnen, die den nach unseren Erkenntnissen besten Code geschrieben hat. Abb. 3 zeigt die Summe der jeweils erreichten Punkte. Gruppe 2, die Siegergruppe, hat 65 von 75 möglichen Punkten erreicht. Zusätzlich bekam jedes Entwicklungsteam einen Bericht, welche Mängel in ihrem Projekt besonders typisch und häufig waren.

Nachfolgend werden die verschiedenen Aspekte diskutiert, die in die Bewertung eingegangen sind. Für jede Kategorie gab es maximal 10, Punkte pro gefundenen Verstoß wurde ein Punkt abgezogen. Da wir davon ausgegangen sind, dass extrem lange Klassen im SoPra selten sind, gab es dafür nur maximal 5 Punkte.

Namensgebung

Bei der Suche nach Defekten in der Namensgebung wurden verschiedene Aspekte unterschieden (siehe Abb. 4):

- Nicht-Einhaltungen der Java-Konventionen bei der Bezeichnerwahl. Das kann von PMD festgestellt werden.
- Sinnvolle oder nicht sinnvolle Bezeichner erkennt PMD natürlich nicht. Wir suchen nach besonders kurzen Bezeichnern (min. 5 Zeichen), schauen uns die Fundstellen an und entscheiden über die Verständlichkeit. Bezeichner „t" oder „x" haben bei uns keine Chance und führen zu Punktabzug. Wir akzeptieren dagegen Bezeichner aus unserem Problembereich wie „Game" oder „Zug".

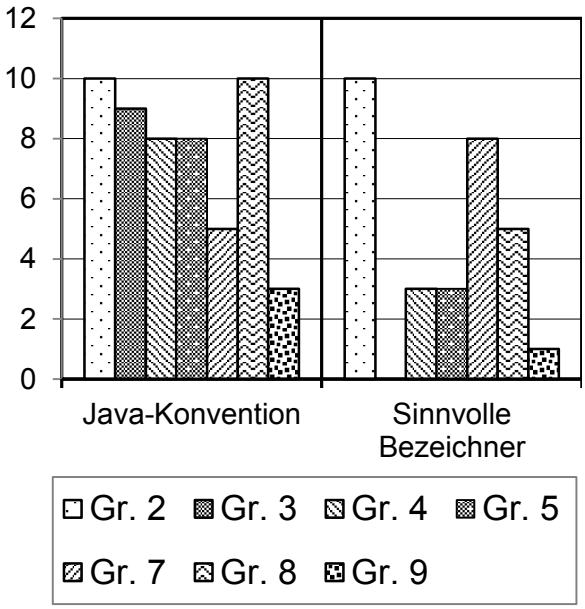

Abb. 4: Namensgebung

Bei Gruppe 2 haben wir keine Defekte in der Namensgebung gefunden, da ein Gruppenmitglied den Code überarbeitet hat. Wir können daraus schließen, dass sich Verstöße bei der Namensgebung leicht mit PMD erkennen und mit den Refactoring-Techniken von Eclipse beseitigen lassen.
Die Java-Konventionen werden von allen Gruppen weitgehend eingehalten. Wir vermuten, dass es daran liegt, dass viele Bezeichner aus dem UML-Modell stammen, das einem mehrfachen Review-Prozess unterzogen wird. Besonders viele einbuchstabige Bezeichner findet man bei den Parametern.

Auffällig waren Unterstriche in Methodennamen in Gruppe 9, die nicht den Java-Konventionen entsprechen (Abb. 4). Sie haben wohl ihre Ursache in dem zuvor besuchten C-Kurs.

Methoden

Bei Methoden haben wir ihre Länge, die Anzahl der Parameter und ihre zyklomatische Komplexität untersucht.

Eine Methode sollte maximal 40 Zeilen lang sein. Bei PMD ist 100 als maximale Länge von Methoden eingestellt. Martin akzeptiert nur 4 Zeilen. Wir denken aber, dass die Studierenden etwa 40 Zeilen noch überschauen können. Außerdem wollten wir die Messlatte auch nicht zu hoch bzw. zu niedrig setzen.

Wie man sieht (Abb. 5), hat es keine Gruppe geschafft, ohne zu lange Methoden auszukommen. Die längste gefundene Methode mit 186 Zeilen hat die Gruppe 9 produziert. Die Gruppe 8 hat so viele lange Methoden geschrieben, dass in diesem Bereich keine Punkte erhalten geblieben sind.

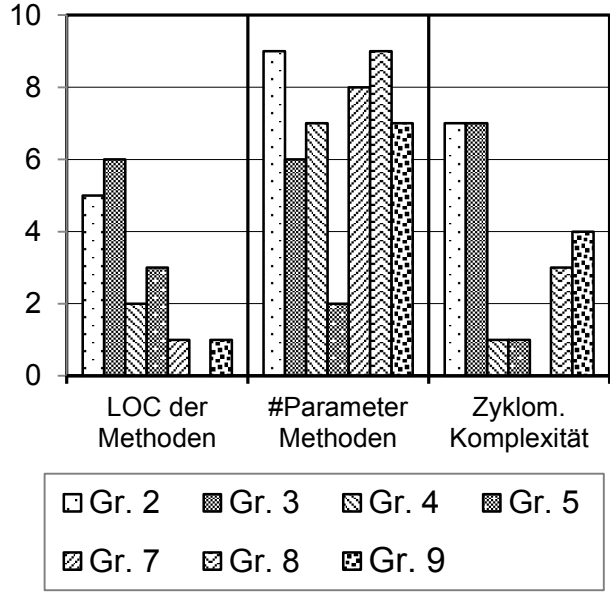

Abb. 5: Methoden

Die extrem langen Methoden besitzen in der Regel auch eine hohe zyklomatische Komplexität. Die längste Methode (von Gruppe 9) hatte eine zyklomatische Komplexität von 49. Als mangelhaft werden Methoden ab einer Komplexität von 10 gewertet. Das entspricht dem von McCabe [McCabe 1976] selbst vorgeschlagenen Wert. Die Methode mit der höchsten zyklomatischen Komplexität von 67 ist „loadGame“ von Gruppe 4.

Auch die von Eclipse generierten equals-Methoden von Klassen sind schwer zu lesen und werden von PMD als zu komplex eingestuft.

Bei der genaueren Betrachtung der besonders komplexen Methoden sieht man deutlich, wie die Entwickler mit der Komplexität „gekämpft“ haben. Man findet Ausgaben auf die Konsole, auskommentierte Programmzeilen, Kommentare zur eigenen Orientierung und Fragen („Was soll ich damit?“). Keine Gruppe hat es geschafft, ohne zu komplexe Methoden auszukommen.

Eine Parameterliste wird als zu lang angesehen, wenn sie fünf oder mehr Parameter beinhaltet. Der Defekt, sehr lange Parameterlisten zu verwenden, ist in den Gruppen unterschiedlich häufig vorgekommen. Im Programm von Gruppe 5 fand sich eine Methode mit 7 Parametern. Das war der höchste gefundene Wert.

Klassen

Wir betrachten Klassen als zu groß, wenn sie mehr als 400 Zeilen lang sind. Voreingestellt bei PMD sind 1000 Zeilen, was bei einem Projektumfang von etwa 2000 LOC im Software-Praktikum viel zu viel ist und in keiner Weise den Ideen von Clean Code entspricht.

Wie bereits erwähnt, haben wir nur 5 Pluspunkte als Guthaben für diese Rubrik verteilt. Alle Gruppen konnten mindestens 2 Punkte behalten (siehe Abb. 6), so dass unsere Vermutung bestätig wurde, dass sehr große Klassen im SoPra selten sind. Gruppe 2, Siegerin der Gesamtwertung und eine der Gruppen mit den meisten Klassen (35), hat überhaupt keine überlange Klasse produziert.

Die zu langen Klassen sind auch die, die von PMD als Gott-Klassen identifiziert werden. Gott-Klassen sind Klassen, die zu viel wissen und sich in zu viele Dinge einmischen.

PMD setzt in der Regel “GodClass“[5] eine Strategie zur Erkennung um, welche in [LMD06] beschrieben wurde. Nach dieser Strategie kann eine Gott-Klasse durch die Berechnung der drei Werte WMC, ATFD und TCC erkannt werden:

- WMC (Weighted Method Count) ist die Summe der zyklomatischen Komplexitäten aller Methoden einer Klasse. Diese darf nach der Regel „GodClass“ den Grenzwert von 47 nicht überschreiten.

[5] Die Regel „GodClass“ wird in folgender Java-Klasse definiert: http://pmd.sourceforge.net/pmd-5.1.2/xref/net/sourceforge/pmd/lang/java/rule/design/GodClassRule.html

- ATFD (Access To Foreign Data) ist die Anzahl der direkten Zugriffe einer Klasse auf Attribute anderer Klassen. Diese darf nach der Regel „GodClass“ nicht höher als 5 sein.
- TCC (Tight Class Cohesion) ist für eine Klasse als Quotient definiert, die Anzahl an Methodenpaaren, die auf mindestens ein gemeinsames Attribut oder eine lokale Methode (dieser Klasse) zugreifen, die also eng miteinander verbunden sind, durch die Gesamtzahl alle möglichen Beziehungen zwischen den Methoden [LMD06]. Entsprechend kann TCC Werte zwischen 0 und 1 annehmen. Der berechnete Wert darf nach der Regel „GodClass“ 0,33 nicht unterschreiten. Ansonsten liegt die gewünschte Kohäsion der Methoden nicht vor und man kann die Klasse relativ leicht aufteilen.

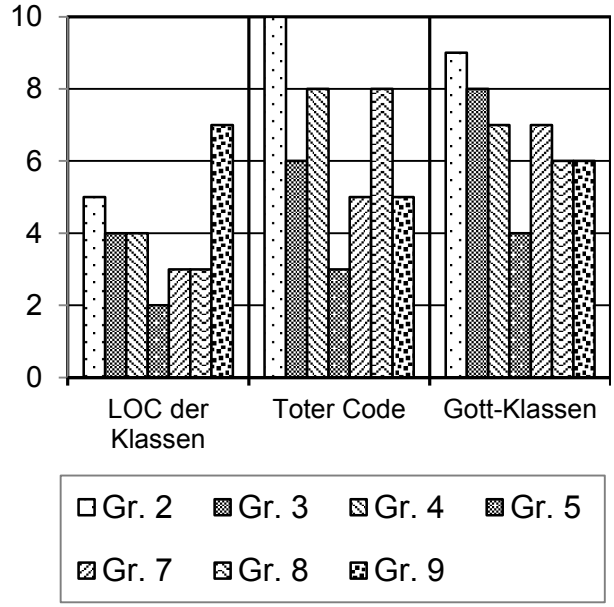

Abb. 6: Klassen

Wenn eines der Kriterien verletzt ist, wird eine Gott-Klasse vermutet. Bei uns waren meist zwei oder alle Kriterien betroffen. Die längste Klasse „GameController“ mit 992 LOC ist auch eine mögliche Gott-Klasse.

Schaut man sich die Projekte im Detail an, stellt man fest, dass es immer die gleichen Klassen sind, die als mögliche Gott-Klassen erkannt werden. Vier von sieben Gruppen haben nur eine Klasse für das Einlesen der Spielsituation in dem vorgegebenen Format angelegt, die den Inhalt der Datei einliest und abhängig vom gelesenen String viele unterschiedliche Objekte erzeugt. Oft hat diese Klasse dann sogar nur eine Methode wie die bereits als komplexeste Methode vorgestellte Methode „loadGame“ von Gruppe 4 (siehe Methoden).

Auch die Klassen, die jeweils den simulierten Computergegner realisieren, werden in fünf von sieben Gruppen als potentielle Gott-Klassen erkannt. In diesen Klassen wird die Spielsituation analysiert und ausprobiert, welcher Zug zu einer neuen, besonders vorteilhaften Spielsituation führt. Die Gruppe 5 hat für jede der geforderten KI-Stärken eine eigene Klasse angelegt, wobei alle drei als Gott-Klassen erkannt werden.

Außerdem sind die Klassen gefährdet, die den Zug des Menschen auf Zulässigkeit prüfen, alle möglichen Züge bestimmen und den Zug durchführen. Diese Klassen heißen „Game“, „GameController“, „Board“ oder „BoardController“, „SpielerController“ oder „Human“. Damit sind schon alle als Gott-Klassen erkannte Klassen genannt. Alle Gruppen haben sich mindestens eine Gott-Klasse geleistet (siehe Abb. 6).

Im Gegensatz zu den Control-Klassen sind bei fast allen Gruppen die Model-Klassen in Ordnung. Was nicht weiter erstaunlich ist, wenn man sich daran erinnert, dass es sich hier meist um einfache Datenhaltungsklassen handelt. Nur bei Gruppe 5, eine der Gruppen mit einem eher objektorientierten Ansatz, finden sich auch im Modell mögliche Gott-Klassen, z. B. die Klasse „Board“ mit WMC=112, ATFD=51, TCC=0.098.

Zusammenfassend lässt sich festhalten, dass bei uns spezielle Klassen dafür prädestiniert zu sein scheinen, sich zu Gott-Klassen zu entwickeln, nämlich diejenigen mit den algorithmisch anspruchsvollsten Aufgaben.

Auch wenn Bezeichner wie „…Manager“ oder „…Controller“ Hinweise auf potentielle Gott-Klassen liefern sollen, hat bei uns die Gruppe 5 die meisten potentiellen Gott-Klassen produziert, obwohl sie einen Ansatz mit mehr und umfangreicheren Model- als Control-Klassen gewählt hatte.

Auch das Muster, dass Gott-Klassen gern aus „entarteten“ Mediator-Klassen [Gamma u.a. 2004] entstehen, haben wir so bei uns nicht gefunden. Mediator-Klassen werden von den Studierenden häufig eingesetzt, um die Arbeit der verschiedenen Controller zu koordinieren. Sie heißen z. B. „MainController“. Bei uns scheint eher die algorithmische Komplexität der Auslöser für Gott-Klassen zu sein. Wir finden potentielle Gott-Klassen da, wo die algorithmisch anspruchsvolleren Aufgaben zu lösen sind. Es gelingt vielen Studierenden offenbar nicht, ein komplexes Problem in mehrere überschaubarere Teilprobleme zu zerlegen.

Toten Code haben wir in Form von unbenutzten Methoden, Parametern und Attributen gefunden. Es gab leere Methoden, die als Code-Rahmen aus dem UML-Modell generiert wurden, aber sich dann wohl doch als überflüssig herausgestellt haben. Wir haben auskommentierten Programmcode und sogar zwei unbenutzte Klassen gefunden. Die Projektlaufzeit ist mit drei Wochen zu knapp für den letzten Feinschliff oder die „Baustellen" geraten in Vergessenheit, weil sie ja beim Kompilieren keine Fehler melden und die Ausführung des Programms nicht stören.

Weitere Defekte

In den Spielprogrammen haben wir sehr häufig Zahlen in Bedingungen gefunden, z. B. für die maximale Anzahl der Mitspieler und die Anzahl der Teilschritte, aus denen ein Zug besteht. In einem Team, das sich wochenlang mit genau diesem Spiel beschäftigt hat, mag ein derartiger Ausdruck absolut verständlich erscheinen. Ohne sehr gutes Kontextwissen aus der Aufgabenstellung und den Spielregeln sind solche Code-Stellen nicht zu verstehen.

Ursache dafür scheint uns zu sein, dass gute Vorbilder fehlen. Vielmehr ist die Formulierung derartiger Bedingungen mit Literalen statt Konstanten in anderen Vorlesungen und Übungen schon aus Platzgründen üblich. Den Studierenden wird die Bedeutung von lesbarem Code erst dann klar, wenn sie in fremdem Code nach Fehlern suchen müssen.

Da Java die Anordnung der Komponenten einer Klasse nicht wirklich einschränkt, findet sich in manchen Klassen eine lustige Abfolge von Methoden und Variablen. Dass die Deklaration nicht wahllos erfolgen sollte, sondern insbesondere in umfangreichen Klassen einer vorgegebenen Ordnung entsprechen sollte, wird nicht vermittelt und erst beim Studium fremden Codes klar.

Diskussion der Ergebnisse

Wir haben relativ viele Mängel gefunden. Die Studierenden wussten allerdings beim Programmieren nicht, dass wir danach suchen würden.

Die Probleme bei der Namensgebung und die Verwendung von Literalen rühren wahrscheinlich daher, dass diese in den vorangehenden Vorlesungen und Übungen nicht thematisiert wurden. In den Vorlesungsfolien und beim Anschrieb der Übungsaufgaben werden aus Platz- und Zeitgründen besonders kurze Bezeichner verwendet. Man zeigt ja auch immer nur einen kleinen Ausschnitt. Die Studierenden sind somit überhaupt nicht vorbereitet auf die Notwendigkeit von aussagekräftigen Bezeichnern und die Vermeidung von Zahlen in Bedingungen zur Verbesserung der Wartbarkeit in umfangreicheren Projekten.

Der Zusammenhang zwischen der Länge einer Methode und ihrer Komplexität ist leicht nachzuvollziehen. Gruppen, die besonders viele lange Methoden geschrieben haben, hatten auch viele komplexe Methoden. Im Umkehrschluss kann man für die Studierenden die Empfehlung ableiten, keine langen Methoden zu schreiben. Lange Methoden sollten rechtzeitig aufgespalten werden. Derartiger Code ist auch besser zu testen.

Die Beseitigung von langen Parameterlisten durch das Refactoring „Introduce Parameter Object" ist durchaus kritisch zu sehen, da bei seiner Anwendung eine Klasse entsteht, die wiederum einen Konstruktor mit gleicher Parameterliste hat. Auch die Vermeidung von vielen Parametern durch Arraylisten oder Ähnlichem ist keine wirkliche Lösung, da dieser Code nicht besser lesbar ist. Vielmehr müssen von vornherein andere, kleinere Methoden entworfen werden, die Teilaufgaben lösen.

Die Länge einer Klasse scheint ein relativ leicht verständliches und leicht zu erkennendes Kriterium und ein gutes Indiz zu sein, um eine Klasse zu identifizieren, die Gefahr läuft, sich zu einer Gott-Klasse zu entwickeln. Die Regel für die Gott-Klassen mag für Anfänger schwierig zu durchschauen sein, aber „zu lang" kann wirklich jeder begreifen und im Auge behalten. Oft kann man beim Entwurf bereits erkennen, dass eine Klasse sehr viele Methoden bekommen soll. Dann könnten die Gruppenbetreuer rechtzeitig den Tipp geben, die Klasse zu zerlegen. Insbesondere, wenn ein Mangel an Kohäsion der Methoden vorliegt, sollte die Aufspaltung kein Problem sein.

Das Refactoring von einmal entstandenen Gott-Klassen ist relativ kompliziert. In dem Entwicklerforum „Stackoverflow" findet man auf die Frage nach dem Refactoring von Gott-Klassen als Antwort[6]: Das ist wie Jenga spielen. Wenn man an der falschen Stelle was wegnimmt, bricht alles zusammen.

[6] http://stackoverflow.com/questions/14870377/how-do-you-refactor-a-god-class

Das vorliegende Datenmaterial ist zu wenig umfangreich, als dass man vermutete Zusammenhänge, wie „Wenige Klassen führen zu mehr Gott-Klassen." oder „Mehr Control-Klassen führen zu mehr Gottklassen.", belegen könnte. In unseren Beispielen sind die Gottklassen in der Regel sehr lang.

Auch die interessante Frage, ob das MVC-Muster mit seiner Aufteilung in die Model- und die Control-Schicht die Entstehung von Gott-Klassen begünstigt, lässt sich (noch) nicht beantworten. Die Entwicklung eines Projekts gemäß MVC-Muster ohne Gott-Klassen wird angestrebt und sollte durchaus möglich sein.

Neben den Fragen danach, warum die Studierenden so viele Defekte in ihren Code einbauen und wie sie beseitigt oder vermieden werden können, stellt sich die Frage, wie die Studierenden auf die Kritik reagieren. Viele Studierende zeigen sich durchaus einsichtig und es ergeben sich konstruktive Diskussionen, wie man die Defekte hätte vermeiden können, z. B. zum Thema große Klassen oder schlechte Bezeichnerwahl. Daneben gibt es kritische Anmerkungen von den Studierenden zu den gewählten Regeln und Grenzwerten, z. B. zur maximalen Länge der Parameterliste und zu sehr kurzen Parameterbezeichnern. Uns ist natürlich klar, dass die Grenzwerte für die Messung relativ willkürlich gewählt sind und dass man darüber diskutieren kann. Grundsätzlich begrüßen wir auch diese Diskussionen über das Vorgehen, weil damit unser Ziel, Sensibilisierung für das Thema Codequalität, erreicht wird. Diskussionen über die Punktevergabe gibt es nicht. Sie scheint als fair empfunden zu werden.

Wie geht es weiter?

Die gewählten Grenzwerte haben sich bewährt und können beibehalten werden. Martin formuliert in [Martin 2008] sehr strenge Qualitätsmaßstäbe, denen selbst erfahrene Entwickler nur schwer genügen können. Da wir die Studierenden nicht völlig demotivieren wollen, haben bewusst erreichbare Ziele gesetzt. Mit den von uns gewählten Werten haben wir offenbar Grenzwerte gefunden, die die Studierenden in den meisten Fällen einhalten und nur selten verletzen.

Die Idee, bei der Bewertung von einem Guthaben von 10 Punkten bzw. 5 Punkten bei jedem Verstoß einen Punkt abzuziehen, halten wir weiterhin für didaktisch richtig. Da die Kategorien unabhängig voneinander betrachtet werden und es keine negativen Punkte gibt, haben selbst ganz viele Missgriffe in einer Kategorie, z. B. bei der Bezeichnerwahl, nicht den völligen Verlust aller Punkte zur Folge. In einer anderen Kategorie kann der Punkteverlust wieder ausgeglichen werden.

Im WS14/15 wollen wir Code-Qualität von Beginn an stärker thematisieren (siehe Vorgehensweise). Die Tutorials zu Clean Code und Refactoring sind fertig gestellt. PMD steht mit dem SoPra-Regelwerk als Eclipse-Plugin zur Verfügung. Wir hoffen, dass die Studierenden damit in der Lage sind, Mängel zu erkennen, und viele Mängel so beheben können.

Auch die Gruppenbetreuer sind aufgrund der im letzten SoPra durchgeführten Messungen und der Diskussion der Ergebnisse mehr für das Thema Code-Qualität sensibilisiert. Sie müssen den Gruppen dabei helfen, umfassendere Mängel wie potentielle Gott-Klassen oder zu lange Parameterlisten möglichst schon in der Entwurfsphase zu vermeiden, indem sie wie wir ein Gespür dafür entwickeln, wo sie entstehen könnten.

Zusammenfassung

Wir stellen vor, wie wir das Thema „Code-Qualität" in der Lehrveranstaltung Software-Praktikum eingeführt haben. Dabei gehen wir im Sinne des Blended Learning vor. Aus angeleiteten Lernprozessen und durch Selbststudium mit Hilfe von im Wiki bereitgestellten Unterlagen erarbeiten sich die Studierenden das Thema selbst. Wir haben für Anfänger angemessene Aspekte der Code-Qualität und geeignete Grenzwerte für die Messinstrumente festgelegt. Diese Vorbereitungen für ein erfolgreiches Selbststudium stellen wir hier vor. Durch die handlungsorientierte Vermittlung im Rahmen eines Software-Projekts soll den Studierenden die Bedeutung von Code-Qualität für die Software-Entwicklung leicht verständlich werden.

Durch das ständige Feedback des Tools erfahren die Studierenden, wie nahe sie dem Ziel von hoher Code-Qualität bereits gekommen sind und wo noch Mängel bestehen. Den Studierenden wird in den Tutorials und in einer Live-Demo von Eclipse gezeigt, wie sie welche Mängel leicht mit welchen Refactoringschritten beheben können und wo sie manuell eingreifen müssen. Durch die Diskussion mit den Dozenten und untereinander sollte den Studierenden die Bedeutung der Mängel klar werden.

Auf keinen Fall wollen wir die Bewertung zu strikt durchführen, beispielsweise die Vergabe des Scheins von der Einhaltung der Qualitätsgrenzen abhängig machen, da wir die Gefahr von adaptivem Verhalten sehen und verhindern wollen. Der Inhalt, die Funktionalität des Programms, ist immer noch wichtiger als die Form.

Literatur

Fowler, M. (1999): Refactoring: Improving the Design of Existing Code. Addison-Wesley, Reading, MA.

Gamma, Erich; Helm, Richard; Johnson, Ralph E.; Vlissides, John (2004): *Entwurfsmuster*. Elemente wiederverwendbarer objektorientierter Software. Addison Wesley, München.

Lanza, Michele; Marinescu, Radu; Ducasse, Stephan (2006): Object-oriented metrics in practice, Springer.

Martin, R. C. (2008): Clean code: a handbook of agile software craftsmanship. Pearson Education.

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

Remmers, J.R. (2014): Clean Code – Code-Qualität im Software-Praktikum. BA-Arbeit, Fakultät für Informatik, TU Dortmund.

Riel, Arthur J. (1996): Object-oriented design heuristics. Addison-Wesley Publishing Company.

Schmedding, D. (2011): Teamentwicklung in studentischen Projekten. Software Engineering im Unterricht der Hochschulen (SEUH) 2011, München.

Spillner, Andreas; Linz, Tilo (2005): Basiswissen Softwaretest. dpunkt.verlag, Heidelberg.

Anhang A: Bewertungsschema

Kriterium	**Überprüfung durch**	**Bewertung**
Einhaltung der Java-Konventionen bei der Namensgebung	PMD	Max. 10 Punkte, pro Verstoß 1 Punkt Abzug
Sinnvolle Bezeichner	Stichproben, manuell	Max. 10 Punkte, pro Verstoß 1 Punkt Abzug
Zeilenlänge der Methoden <= 40	PMD	Max. 10 Punkte, pro zu lange Methode 1 Punkt Abzug
Parameteranzahl der Methoden <= 4	PMD	Max. 10 Punkte, pro Methode mit zu vielen Parametern 1 Punkt Abzug
Zyklomatische Komplexität der Methoden <= 10	PMD	Max. 10 Punkte, pro Verstoß 1 Punkt Abzug
Zeilenlänge der Klassen <= 400	PMD	Max. **5 Punkte**., pro zu lange Klasse 1 Punkt Abzug
Toter Code	PMD	Max. 10 Punkte, pro Verstoß ein Punkt Abzug
Gott-Klasse	PMD	Max. 10 Punkte, pro gefundene Gott-Klasse 1 Pkt. Abzug

Anhang B: SoPra-spezifische Regelmenge

```
<?xml version="1.0"?>
<ruleset name="SoPra"
        xmlns="http://pmd.sourceforge.net/ruleset/2.0.0"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://pmd.sourceforge.net/ruleset/2.0.0 http://pmd.source-
forge.net/ruleset_2_0_0.xsd">
        <description>
                Diese Regelmenge wurde zur Nutzung in der Lehrveranstaltung "SoPra" der TU Dort-
mund erstellt.
                Sie besteht ausschließlich aus Standardregeln.
        </description>
<!-- Namensgebung-->
        <rule ref="rulesets/java/naming.xml/ShortVariable"/>
        <rule ref="rulesets/java/naming.xml/ShortMethodName"/>
        <rule ref="rulesets/java/naming.xml/ShortClassName"/>
        <rule ref="rulesets/java/naming.xml/VariableNamingConventions"/>
        <rule ref="rulesets/java/naming.xml/MethodNamingConventions"/>
        <rule ref="rulesets/java/naming.xml/ClassNamingConventions"/>
        <rule ref="rulesets/java/naming.xml/PackageCase"/>
        <rule ref="rulesets/java/naming.xml/AvoidDollarSigns"/>
        <rule ref="rulesets/java/naming.xml/SuspiciousConstantFieldName"/>
        <rule ref="rulesets/java/controversial.xml/AvoidLiteralsInIfCondition"/>
<!-- Methoden-->
        <rule ref="rulesets/java/codesize.xml/CyclomaticComplexity"/>
        <rule ref="rulesets/java/codesize.xml/ExcessiveMethodLength">
                <properties>
                        <property name="minimum" value="40"/>
                </properties>
        </rule>
        <rule ref="rulesets/java/design.xml/AvoidDeeplyNestedIfStmts"/>
        <rule ref="rulesets/java/codesize.xml/ExcessiveParameterList">
                <properties>
                        <property name="minimum" value="4"/>
                </properties>
        </rule>
<!-- Klassen-->
        <rule ref="rulesets/java/design.xml/FieldDeclarationsShouldBeAtStartOfClass"/>
        <rule ref="rulesets/java/codesize.xml/ExcessiveClassLength">
                <properties>
                        <property name="minimum" value="400"/>
                </properties>
        </rule>
        <rule ref="rulesets/java/design.xml/GodClass"/>
<!-- Unused Code-->
        <rule ref="rulesets/java/unusedcode.xml/UnusedPrivateField"/>
        <rule ref="rulesets/java/unusedcode.xml/UnusedLocalVariable"/>
        <rule ref="rulesets/java/unusedcode.xml/UnusedPrivateMethod"/>
        <rule ref="rulesets/java/unusedcode.xml/UnusedFormalParameter"/>
</ruleset>
```