Clean Code Development - Grundlagen und Anwendung

Ing. Stefan Kert

BACHELORARBEIT

Nr. S1310307019

eingereicht am

Fachhochschul-Bachelorstudiengang

Software Engineering

in Hagenberg

im Juli 2016

Diese Arbeit entstand im Rahmen des Gegenstands

Gegenstand??

im

Semester??

Betreuer:

Dipl.Ing.(FH) Dr. Josef Pichler

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 14. Juli 2016

Ing. Stefan Kert

Inhaltsverzeichnis

Kurzfassung

Für die Softwareentwicklung ist es von enormer Bedeutung, Programme so zu schreiben, dass diese auch nach vielen Iterationen noch gut zu warten und gut verständlich sind. Um dies zu ermöglichen ist es von sehr großer Bedeutung, dass bereits beim Schreiben des Quelltextes darauf geachtet wird, dass dieser *clean* ist. Dazu haben sich in den letzten Jahren verschiedene Vorgehensweisen etabliert, um Regeln für das Schreiben von *Clean Code* zu definieren. Die wohl bedeutendste Arbeit auf diesem Gebiet hat Robert C. Martin mit seinem Buch Clean Code [12] geleistet. Er beschreibt in diesem Buch, was für ihn Clean Code bedeutet, welche Vorteile es hat Clean Code zu schreiben und wie man auf sehr einfache Art und Weise besseren Quelltext produzieren kann.

Diese Bachelorarbeit nimmt *Clean Code* [12] zur Basis, um die Grundsätze von Clean Code Development aufzuarbeiten und diese zu erläutern. Dabei wird die Arbeit in mehrere Abschnitte gegliedert die sich unter anderem mit den Grundlagen, der Anwendung und dem alltäglichen Gebrauch von CCD beschäftigen. In den Grundlagen sollte ein Eindruck vermittelt werden, wofür CCD steht und welche Vorteile dessen Anwendung bringt, sowie der Ursprung und die Vorreiter von CCD. Dabei wird unter anderem darauf eingegangen, worum es sich bei Clean Code handelt und was Clean Code für einige der bekanntesten Programmierer wie z.B. Grady Booch und Michael Feathers bedeutet. Dabei werden diese Meinungen, welche Robert C. Martin für sein Buch Clean Code [12] eingeholt hat verwendet um die Grundlagen von Clean Code zu erläutern. In den Grundlagen werden weiters Möglichkeiten aufgezeigt, wie einige der CCD Regeln automatisch oder manuell überprüft werden, sowie Tools, welche ein leichteres Anwenden dieser Regeln ermöglicht.

Im zweiten Teil der Arbeit werden einige der Regeln, welche im ersten Teil erwähnt werden, beispielhaft gezeigt und erläutert, welche Probleme durch Anwendung dieser Regeln behoben werden.

Der dritte Teil wird sich in erster Linie mit einer persönlichen Meinung zu CCD und dem Gebrauch im Alltag beschäftigen.

Abstract

In software development, it is extremely important to write programs so that they are easy understandable and maintainable, even after many iterations. To make this possible, it is very important to assure that the source code is already kind of *clean* when it is written. For this purpose there have been found a few best practices to define rules for writing *clean Code*. One of the most significant work in this field is done by Robert C. Martin with his book *Clean Code* [12]. In this book he describes his approach to write clean code, the benefits of writing clean code and many simple rules that are really helpful for simply writing better code.

This thesis takes *Clean Code* [12] as basic to address the principles of Clean Code Development and explain them. The thesis is splitted into severel sections which deal with the basics, the application and the everyday use of CCD. In the basic section an impression should be conveyed what CCD is and the benefits of its use, as well as the origin and the pioneer of CCD. For this purpose it is explained what clean code means and also what clean code means to some of the most famous programmers such as Grady Booch and Michael Feathers. These opinions, that Robert C. Martin collected for his book Clean Code [12], are used then to explain the basics of clean code. Further to the basics of clean code there are mentioned some possiblities to check some of the rules automatically or manually and also some tooles that make it easier to apply these rules.

In the second part some of the rules which are mentioned in the first part are exemplified. Every example also contains a description, which problems are resolved by applying these rules.

The third part will deal primarily with a personal opinion about CCD and the use in everyday life.

Kapitel 1

Einleitung

1.1 Problemstellung

Ein sehr zentraler Aspekt der Softwareentwicklung ist die Notwendigkeit, Software ständig zu ändern. Dies kann aus verschiedenen Gründen nötig werden. Ein Kunde möchte ein neues Feature implementiert haben, bei einem anderen Kunden ist ein Fehler aufgetreten der behoben werden sollte. Ein weiterer Kunde beschwert sich über die sehr schlechte Performance einer Funktionalität. Alle diese Fälle haben eines gemeinsam; sie erfordern eine Änderung der bestehenden Software. Die Gesetze, warum sich Software ändert, wurden von Manny Lehman [10] bereits hinlänglich beschrieben.

Wenn man in der glücklichen Lage ist ein neues Produkt zu entwickeln ist es oft noch sehr leicht weitere Funktionalität hinzuzufügen, aber Neuentwicklungen, sogenannte *Greenfield Applications* sind sehr selten der Fall. Meistens muss die Funktionalität zu einer bereits vorhandenen, meist schon länger entwickelten, Codebasis hinzugefügt werden. In vielen Fällen kommen jetzt zahlreiche Probleme zum Tragen: Es wurde beim ersten Entwurf der Software nicht daran gedacht, dass diese Feature benötigt wird, daher ist es durch die gewählte Architektur nur sehr schwer möglich die gewünschte Funktionalität hinzuzufügen. Dies kann wiederum dazu führen, dass versucht wird einen Workaround zu finden, sodass die Funktionalität trotzdem realisiert werden kann, jedoch führt dies zu einer weiteren Verunstaltung des Quellcodes und er wird dadurch noch schwerer zu lesen und zu warten.

Diese verschlechterte Lesbarkeit und Wartbarkeit führt dazu, dass Änderungen sehr viel Zeit benötigen und nur sehr selten vorgenommen werden. Im schlimmsten Fall ist irgendwann der Punkt erreicht, an dem die Software nicht mehr weiterentwickelt werden kann, da jede Änderung unvorhersehbare Seiteneffekte haben könnte. Dies führt schlussendlich wohl oder übel zum Tod der Software, den Software die sich nicht weiterentwickeln lässt wird früher oder später von besserer Software abgelöst. Dieses Problem wird auch als *Legacy Code Problem* bezeichnet und ist hinlänglich bekannt und von vielen Autoren

beschrieben z.B. Michael Feathers [3].

Die ständige Verschlechterung der Codequalität lässt sich außerdem durch die von James Q. Wilson und George L.Kelling [24] aufgestellte Theorie der zerbrochenen Fenster erklären. Bei dieser Theorie geht es darum, welche Folgen ein zerbrochenes Fenster in einem Gebäude nach sich zieht insofern dieses nicht repariert wird. Die Autoren erläutern dabei, dass durch die Tatsache, dass es aussieht als ob sich niemand um den Zustand dieses Gebäudes kümmern würde, weitere Zerstörung nach sich ziehen wird. Robert C. Martin ([12]) verwendet diese Theorie und bringt sie in Verbindung mit der Softwareentwicklung. Dabei beschreibt er einen sehr ähnlichen Fall wie der eines zerbrochenen Fensters. Bei einem Programmabschnitt, sei es eine Klasse, eine Methode oder ähnliches, die bereits unsauber programmiert und schlecht lesbar ist, wird bei zukünftigen Änderungen auch nicht darauf geachtet werden den Code sauber zu gestalten. Es werden immer mehr unsaubere Codeteile hinzugefügt, was schlussendlich von unsauberen Code zu nicht mehr wartbarem Code führen kann. Dieses Problem existiert in allen Teilen der Programmierung. Bei Projekten, bei denen nicht auf einen sauber gestalteten Code geachtet wurde, wird in Zukunft auch nur selten darauf geachtet.

1.2 Lösungsvorschlag

Um diese Probleme zu vermeiden, sollten Regeln und Grundsätze gefunden werden, wie gut lesbarer bzw. wartbarer Code gestaltet werden kann. Dabei ist es sehr wichtig darauf zu achten, dass der Code meist nicht nur von einer Person bearbeitet und gelesen wird, sondern von vielen verschiedenen Personen. Wenn darauf geachtet wird denn Code so zu gestalten, dass er verständlich, gut lesbar und gut strukturiert ist, dann ist es auch nach einigen Wochen, Monaten und Jahren noch gut möglich zu verstehen, was der Zweck der betrachteten Codestelle ist. In den letzten Jahren hat sich zu diesem Thema eine Strömung ergeben welche sich als Clean Code Development (im folgenden CCD) bezeichnet. Geprägt wurde diese Bewegung in erster Linie von Robert. C. Martin und seinem Buch Clean Code [12]. Er beschreibt dabei, dass umso genauer beim Schreiben des Quellcodes darauf geachtet wurde, diesen lesbar zu gestalten, desto leichter wird es für zukünftige Leser des Quellcodes diesen zu verstehen und auch zu ändern.

1.3 Ziele

Das Hauptziel der Arbeit besteht darin, zu zeigen, wie CCD dabei unterstützen kann, besseren Quellcode zu schreiben. Dem Leser sollte am Ende dieser Arbeit klar sein, dass CCD für alle Stakeholder eines Softwareprojektes Vorteile bietet. Für den Kunden ergibt sich der Vorteil, dass gewünschte Änderungen leichter

eingearbeitet werden können, da durch die verbesserte Lesbarkeit und Struktur eine Änderung bzw. ein Hinzufügen von Funktionalitäten sehr stark erleichtert wird. Für das Unternehmen ergeben sich dadurch ähnliche Vorteile wie für den Kunden. Dadurch, dass es leichter ist Funktionalitäten hinzuzufügen, ist das Unternehmen wettbewerbsfähiger, da man Änderungen am Produkt oder Verbesserungen leichter einbauen kann. Für die ProgrammiererInnen wird die Arbeit durch Clean Code sehr stark vereinfacht, da durch die bessere Lesbarkeit ein Einarbeiten in den Quellcode um einiges leichter wird. Ein weiteres Ziel dieser Arbeit ist es zu zeigen, dass es selbst durch kleine Schritte möglich ist, eine enorme Verbesserung des Quellcodes zu erreichen.

Die Arbeit ist in folgende Abschnitte gegliedert:

- Abschnitt 1 beschäftigt sich in erster Linie mit allgemeinen Aspekten des CCD. Es wird in diesem Abschnitt geklärt, wofür Clean Code Development steht, was Clean Code bedeutet, welche Möglichkeiten es gibt Clean Code zu schreiben und zu überprüfen. Weiters wird auch kurz auf die Entstehung von Clean Code Development eingegangen und Tools und Möglichkeiten gezeigt, wie Clean Code Development in den Alltag einfließen kann.
- In Abschnitt 2 werden einige der in Abschnitt 1 beschriebenen Clean Code Kriterien beispielhaft angewandt. Diese Beispiele sind zu großen Teilen aus Open Source Frameworks und sollen zeigen, wie die Lesbarkeit durch wenige Änderungen im Quellcode verbessert werden kann. Dabei wird immer ein Ist-Stand gezeigt, welcher eines der Clean Code Kriterien missachtet. Danach sollte geklärt werden, welche Probleme dadurch entstehen können und schlussendlich wird gezeigt wie ein Clean Code Beispiel aussehen könnte und welche Vorteil diese bietet.
- Abschnitt 3 beschäftigt sich mit CCD im Alltag einer Firma und mit persönlichen Erfahrungen die im Zusammenhang mit CCD gemacht wurden. Dieser Abschnitt ist eine persönliche Einschätzung, wie wertvoll CCD für den alltäglichen Gebrauch ist.

Kapitel 2

Clean Code Development

2.1 Was ist Clean Code?

Um zu klären was genau Clean Code ist hat Robert C. Martin mehrere bekannte Softwareentwickler befragt und deren Zitate in seinem Buch Clean Code [12] aufgelistet. Im folgenden sollen die wichtigsten Punkte aus den verschiedenen Zitaten wiedergegeben werden, da sie sehr zentrale Aussagen zum Thema Clean Code darstellen. Die folgenden Zitate sind dem Buch Clean Code [12] entnommen.

Bjarne Stroustrup

Der Erfinder von C++ Bjarne Stroustrup spricht davon, dass für ihn Clean Code folgende Punkte erfüllen sollte:

- Elegant
- Effizient
- Gradlinige Logik
- Minimale Abhängigkeiten
- Vordefinierte Strategie für Fehlerbehandlung
- Leistungsverhalten optimieren

Bjarne Stroustrup spricht mit diesen Punkten schon einige sehr zentrale Aspekte des CCD an. Dabei spricht er davon, dass Code elegant sein sollte. Robert C. Martin teilt diese Meinung und geht dabei noch einen Schritt weiter und schreibt davon, dass guter Code einen Ausdruck des Wohlgefallens auf das Gesicht des Lesers zaubern sollte. Ein weiterer sehr wichtiger Punkt den Bjaren Stroustrup erwähnt, der jedoch nicht direkt mit CCD zusammenhängt, ist es das Leistungsverhalten zu optimieren. Wie bereits in Abschnitt 1.1 kurz erläutert, kann es durch eine schlechte Performance dazukommen, dass es zu einem späteren Zeitpunkt dazu kommt, dass ein Workaround für diese Performanceproblem gefunden werden muss.

Grady Booch

Grady Booch ist einer der Autoren des Buches *Object-Oriented Analysis and Design with Applications* [9] und hält sich mit seinem Zitat zu Clean Code sehr kurz. Er beschriebt, dass sich Clean Code wie wohlgeschriebene Prosa lesen sollte, ähnlich wie ein gutes Buch. Der Quellcode sollte keine Überraschungen für den Leser bringen und auch nicht den eigentlichen Zweck des Codes verschleiern. Wie auch Bjarne Stroustrup schreibt Grady Booch davon, dass Code geradlinig geschrieben sein sollte, wobei er sich dabei explizit auf Kontrollstrukturen bezieht.

Dave Thomas

Für Dave Thomas, den Gründer von Object Technology International, sind folgende Punkte ausschlaggebend für Clean Code:

- Lesbar und gut zu verbessern von **anderen** ProgrammiererInnen
- Vorhandensein von Unit- und Acceptance Tests
- Bedeutungsvolle Namen
- Eine Lösung für eine Aufgabe
- Klares und minimales API

Hier sollte man sehr stark den Punkt, dass der Code lesbar und gut zu verbessern sein muss hervorheben. Vor allem die Tatsache, dass er dies auch für andere ProgrammiererInnen sein sollte ist von sehr hoher Bedeutung. Dieser Punkt ist sehr zentral im CCD und erstreckt sich auch über alle Teilbereiche von CCD. Im Gegensatz zu Bjarne Stroustrup erwähnt er auch das Vorhandensein von Unit- und Acceptance Tests. Auch dieser Punkt ist ein sehr zentraler im CCD, da es ohne Tests nur sehr aufwendig möglich ist die Funktionalität des Codes zu überprüfen. Durch die schwerere Überprüfbarkeit der Funktionalität des Codes wird sehr oft davor zurückgescheut diesen zu ändern, was wiederum dazu führt, dass er seltener verbessert wird.

Michael Feathers

Der Autor des Buches *Working Effectively with Legacy Code* [3] Michael Feathers beschreibt in seiner Meinung zu Clean Code, dass es für ihn eine alles übergreifende Qualität gibt die andere Punkte überragt ([12, Seite 36]):

> Sauberer Code sieht immer so aus, als wäre er von jemanden geschrieben worden, dem dies wirklich wichtig war. Es fällt nichts ins Auge, wie man den Code verbessern könnte. Alle diese Dinge hat der Autor des Codes bereits selbst durchdacht; und wenn Sie versuchen, sich Verbesserungen vorzustellen, landen Sie wieder an der Stelle, an der Sie gerade sind.

Dieses Zitat gibt sehr gut wieder, welchen Wert guter Code haben kann. Michael Feathers streicht dabei einen sehr wichtigen Punkt im Zusammenhang mit Clean Code heraus: Man sollte beim Lesen des Codes bemerken, dass es dem Autor wichtig war und dass sich der Autor Gedanken darüber gemacht hat während er diesen geschrieben hat. Der Code sollte mit Sorgfalt geschrieben werden.

Ron Jeffries

Ein weiteres Zitat ist von Ron Jeffries ([7], [6]). Er fasst dabei die für ihn wichtigsten Punkte folgendermaßen zusammen:

- Es werden alle Tests bestanden
- Er enthält keine Duplizierungen
- Es werden die Architektur bzw. Designentscheidungen gut wiedergegeben
- Die Anzahl von Klassen und Methoden werden minimiert
- Ausdruckskraft

Wie bereits in anderen Zitaten zu Clean Code erwähnt ist die Ausdruckskraft ein sehr zentraler Aspekt des CCD. Dabei geht es zum Beispiel um die Auswahl bedeutungsvoller Namen. Ron Jeffries erwähnt dabei, dass er die von ihm ausgewählten Namen mehrfach ändert bis er die finale Version hat. Weiters zeigt er eine sehr wichtige Möglichkeit auf, die sich durch die ständige Weiterentwicklung der modernen Entwicklungsumgebungen wie Visual Studio und Eclipse aufgetan hat. Es ist sehr einfach möglich Namen zu ändern ohne, dass viele manuelle Änderungen vorgenommen werden müssen. Eine Namensänderung erfolgt über einen einfachen Befehl und die Entwicklungsumgebung übernimmt das Umbenennen aller Referenzen auf diesen Namen. Ein weitere Möglichkeit die Ron Jeffries im Zusammenhang mit modernen Entwicklungsumgebungen erwähnt ist das Extrahieren von Methoden. Falls eine Methode mehr als einen Zweck erfüllt und aufgespalten werden muss ist dies mit dem *Extract Method* Befehl in modernen Entwicklungsumgebungen sehr einfach. Durch diese Möglichkeiten wird das Refactoring von Quellcode sehr stark vereinfacht, wodurch es schlussendlich leichter möglich ist Clean Code zu schreiben. Im Abschnitt 2.4 wird noch näher auf die Möglichkeiten mit modernen Tools in der Softwareentwicklung eingegangen. Ron Jeffries schließt seine Aussage mit folgendem Zitat ab, welches die einzelnen von ihm angesprochenen Punkte sehr gut zusammenfasst ([12, Seite 38]):

> Reduzierung der Duplizierung, Steigerung der Ausdruckskraft und frühere Formulierung einfacher Abstraktionen machen für mich sauberen Code aus.

Ward Cunningham

Das Zitat von dem Erfinder des Wikis [23] und Fit [15], sowie dem Miterfinder des eXtreme Programming und treibende Kraft hinter den Design Patterns. Ward Cunningham sollte hier wortwörtlich zitiert werden, da sein Zitat die Grundlagen von Clean Code sehr gut wiedergibt ([12, Seite 39]):

> Sie wissen, dass Sie an sauberem Code arbeiten, wenn jede Routine, die Sie lesen, ziemlich genau so funktioniert, wie Sie es erwartet haben. Sie können den Code auch »schön« nennen, wenn er die Sprache so aussehen lässt, als wäre sie für das Problem geschaffen worden.

Dieses Zitat fasst sehr schön zusammen was Clean Code bedeutet. Code der genau so funktioniert, wie man es erwartet. Eine sehr einfache Regel die jedoch meist nicht sehr einfach umzusetzen ist. Meist ist ein Indikator für das Missachten dieser Regel die Notwendigkeit einen erklärenden Kommentar zu schreiben. Code der genau so funktioniert, wie man es erwartet wird nahezu keine Kommentare benötigen, da es keine unerwarteten Überraschungen für den Leser geben wird. Robert C. Martin streicht besonders den Teil der Aussage hervor, der sich auf die Schönheit des Codes bezieht und ergänzt folgenden Satz ([12, Seite 40]):

> Es ist nicht die Sprache, die ein Programm einfach aussehen lässt. Es ist der Programmierer, der die Sprache einfach aussehen lässt.

Er bezieht sich dabei auf die Ausrede vieler ProgrammiererInnen, dass die Sprache nicht für die Probleme konzipiert wurde, welche sie lösen sollte. Es ist jedoch die Aufgabe den Code so zu gestalten, dass es so aussieht, als wäre genau dies der Fall.

2.2 Smells und Heuristiken

Robert C. Martin beschreibt in seinem Buch *Clean code* [12] eine Liste von Smells und Heuristiken. Dabei handelt es sich bei Smells um Probleme, die beim Schreiben von Code auftreten können und Heuristiken sind Lösungen für bekannte Probleme. In den folgenden Abschnitten werden diese in Listen gruppiert aufgelistet. Es wird dabei nicht zwischen Smells und Heuristiken unterschieden.

Kommentare

- Name
- Ungeeignete Informationen
- **Überholte Kommentare**
- **Redundante Kommentare**
- Schlecht geschriebene Kommentare
- **Auskommentierter Code**

Umgebung

- Name
- Ein Build erfordert mehr als einen Schritt
- Test erfordern mehr als einen Schritt

Funktionen

- Name
- **Zu viele Argumente**
- Output-Argumente
- **Flag-Argumente**
- Tote Funktionen

Allgemein

- Name
- Mehrere Sprachen in einer Quelldatei
- Offensichtliches Verhalten ist nicht implementiert
- Falsches Verhalten an den Grenzen
- Übergangene Sicherungen
- **Duplizierung**
- Auf der falschen Abstraktionsebene codieren
- Basisklasse hängt von abgeleiteten Klassen ab

- Zu viele Informationen
- Toter Code
- Vertikale Trennung
- Inkonsistenz
- Müll
- Künstliche Kopplung
- Funktionsneid
- Selektor-Argumente
- Verdeckte Absicht
- Falsche Zuständigkeit
- Fälschlich als statisch deklarierte Methode
- Aussagekräftige Variablen verwenden
- Funktionsname sollte die Aktion ausdrücken
- Den Algorithmus verstehen
- Logische Abhängigkeiten in physische umwandeln
- **Polymorphismus statt If/Else oder Switch/Case verwenden**
- Konventionen beachten
- Magische Zahlen durch benannte Konstanten ersetzen
- Präzise sein
- Struktur ist wichtiger als Konvention
- Bedingungen einkapseln
- Negative Bedingungen vermeiden
- Eine Aufgabe pro Funktion!
- Verborgene zeitliche Kopplungen
- Keine Willkür
- Grenzbedingungen einkapseln
- In Funktionen nur eine Abstraktionsebene tiefer gehen
- Konfigurierbare Daten hoch ansiedeln
- Transitive Navigation vermeiden

Namen

- Name
- **Deskriptive Namen wählen**
- Namen sollten der Abstraktionsebene entsprechen
- Möglichst die Standardnomenklatur verwenden
- Eindeutige Namen
- Lange Namen für große Geltungsbereiche

- Codierungen vermeiden
- Namen sollten Nebeneffekte beschreiben

Tests

- Name
- Unzureichende Tests
- Ein Coverage-Tool verwenden
- Triviale Tests nicht überspringen
- Ein ignorierter Test zeigt eine Mehrdeutigkeit auf
- Grenzbedingungen testen
- Bei Bugs die Nachbarschaft gründlich testen
- Das Muster des Scheiterns zur Diagnose nutzen
- Hinweise durch Coverage-Patterns
- Testen sollten schnell sein

Zusätzliche Smells und Heuristiken

Robert C. Martin hat einige in seinem Buch beschriebenen Kriterien für Clean Code in seiner abschließenden Liste nicht aufgelistet. Diejenigen Kriterien, die in den Anwendungsbeispielen in Kapitel 3 gezeigt werden, sind hier noch aufgelistet.

- **Keine Null zurückgeben**
- **Ausnahmen sind besser als Fehler-Codes**
- **Duplizierung boolscher Ausdrücke**
- **Duplizierung von Testcode**
- **Try/Catch - Blöcke extrahieren**

2.3 Die Pfadfinder-Regel

Eine der grundlegendsten Regeln im CCD ist die sogenannte Pfadfinder-Regel. Robert C. Martin definiert diese wie folgt ([12, Seite 43]):

> Hinterlasse den Campingplatz sauberer, als du ihn gefunden hast.

Auf die Arbeit eines Programmierers/einer Programmiererin umgemünzt bedeutet dieser Leitsatz, dass jeder Quellcode, den man betrachtet verbessert werden sollte. Dies muss nicht unbedingt in einer Änderung der Struktur resultieren. Es reicht meist schon aus, wenn für eine Variable einen besseren Namen vergeben, oder einen unnötigen Kommentar entfernt wird. Robert C.

Martin schreibt weiters, dass sich durch dieses Vorgehen ein für die Softwareentwicklung ungewöhnlicher Trend ergibt. Der Quellcode wird über die Zeit besser.

2.4 Nutzen moderner Werkzeuge zur Softwareentwicklung

Etwas das Robert C. Martin sehr oft in seinem Buch kritisiert, ist die Tatsache, dass viele ProgrammiererInnen die modernen Werkzeuge, die ihnen zur Verfügung stehen nicht nutzen und daher unnötige Arbeiten immer wieder erledigen. Dabei wurde bereits im Abschnitt 3.2.1 erläutert, welche Probleme entstehen, wenn die vorhandenen Werkzeuge wie z.B. Versionsverwaltungssysteme nicht korrekt eingesetzt werden.

2.4.1 Testen, Erzeugen und Verteilen von Software

In der modernen Softwareentwicklung spricht man oft von *Continous Delivery*. Dabei geht es um die azyklische Auslieferung von Software. Es wird dabei eine Pipeline eingerichtet, welche die einzelnen Aufgaben, welche für die Auslieferung von Software nötig sind ausführt. In dem Buch *Continuous Delivery* von Eberhard Wolff erläutert der Autor seine Gedanken zum automatisieren des Prozesses vom Erstellen der Software, über das Testen bis zur Auslieferung dieser. Dies sollte für das Team möglichst einfach möglich sein und einmalig eingerichtet, für jeden zugänglich sein, sodass man möglichst schnell eine Rückmeldung erhält, ob die aktuellen Änderungen Probleme verursacht haben. Dies sollte für das Team über eine sogenannte *Continous Integration* Komponente möglich sein.

Der automatische Prozess des Erzeugens und des Testens sollte jedoch nicht nur über die *Continous Integration* Komponente ermöglicht werden. Es sollte für jeden Programmierer/jede Programmiererin durch wenige Schritte möglich sein, die Software zu Erzeugen und die Tests auch auszuführen. Dies sollte möglichst durch einen einzigen Befehl, durch einen einzigen Klick, oder wenn möglich sogar automatisiert erledigt werden. Es sollten dafür keine komplizierten Operationen nötig sein, da dies zu einer Duplizierung der Arbeit führt und immer wieder Zeit benötigt. Mit modernen Entwicklungsumgebungen wie Visual Studio oder Eclipse sind meist schon Werkzeuge integriert, welche das Erstellen und das Testen von Software durch eine einfache Tastenkombination oder einen einzelnen Klick ermöglichen. Diese Features sollten daher auch verwendet werden.

2.4.2 Werkzeuge für das Refactoring

Nahezu alle modernen Entwicklungsumgebung bieten Möglichkeiten für ein effizientes und einfaches Ändern von Namen. Dabei übernimmt die Entwicklungsumgebung alle Aufgaben, die für eine korrekte Umbenennung nötig sind. Alle Referenzen werden umbenannt, teilweise werden sogar in Kommentaren Namen geändert die auf diesen Namen verweisen. Durch dieses Feature ist es nicht mehr nötig, eine Umbenennung auf manuelle Art durchzuführen, da dies meist zu zahlreichen Fehlern führt.

2.4.3 Coding Conventions

Coding Conventions sind ein sehr effizientes Mittel projektweite, oder auch unternehmensweite Regeln zu definieren, wie einzelne Aspekte der Programmierung gestaltet werden sollten. Vor allem für Open Source Projekte ist dies ein sehr wichtiges Mittel wie ein durchgängiger Codierungsstil gewählt werden kann. Meist werden auch für Frameworks Coding Conventions definiert wie zum Beispiel für C# [20]. Meist werden diese grundlegenden Konventionen von Unternehmen verwendet und nur teilweise angepasst, da es durch einen durchgängigen Programmierstil leichter ist, dass sich neue ProgrammiererInnen in die Codebasis einarbeiten. Ein wichtiger Aspekt der bei Coding Conventions beachtet werden muss, ist die Möglichkeit der automatischen Überprüfung dieser. Dazu werden in Abschnitt 2.4.4 einige Möglichkeiten erläutert, wie die Überprüfung dieser Kriterien erfolgen kann.

2.4.4 Überprüfung der CCD Kriterien

Häufig stellt sich beim CCD die Frage, wie es möglich ist die einzelnen Kriterien zu überprüfen. Hierzu gibt es verschiedene Varianten. Es gibt einerseits die statische Codeanalyse, welche dafür Sorgen kann, dass die in Abschnitt 2.4.3 beschriebenen Coding Conventions eingehalten werden. Ein sehr wirksames Mittel für die Überprüfung des Quellcodes allgemein sind Coding Reviews. Es sollte in den folgenden Abschnitten näher auf diese beiden Werkzeuge eingegangen werden.

Statische Codeanalyse

Die statische Codeanalyse bietet eine Möglichkeit zur Analyse des Quellcodes nach fix vorgegebenen Regeln. Dabei werden für ein Unternehmen, oder für ein Projekt die Abschnitt 2.4.3 beschriebenen Coding Conventions festgelegt und an Hand dieser Regeln definiert. Ein Beispiel für solch eine Regel wäre das in C# übliche *I* vor einem Interfacenamen. Das Tool für die statische Codeanalyse würde im Falle einer Missachtung dieser Regel eine Warnung ausgeben und der Programmierer/die Programmiererin würde direkt darauf aufmerksam gemacht werden, dass er sich nicht an die Coding Conventions hält. Im .NET

Bereich ist das wohl bekannteste statische Codeanalyse Tool NDepend ([21]). Mit diesem ist es neben den überprüfen der Coding Conventions auch möglich, zirkuläre Abhängigkeiten zwischen Klassen zu erkennen, zyklische Komplexitäten von Methoden zu analysieren und viele weitere Metriken zu erzeugen, die Aufschluss darüber geben, wie sauber der analysierte Code ist. Im Java Bereich gibt es das sehr hilfreiche Tool JDepend ([16]) welches ähnlich aufgebaut ist wie NDepend. Vor allem im Webbereich haben sich in den letzten Jahren statische Codeanalysetools etabliert. Im JavaScript bereich ist vor allem JSHint ([17]) und JSLint ([18]) sehr populär.

Code Reviews

Code Reviews sind ein sehr wirksames Mittel um geschriebenen Code zu Überprüfen. Es handelt sich bei diesen Reviews um manuelle Überprüfungen, die meist von erfahreneren Programmierern/Programmiererinnen vorgenommen werden. Dabei ist einer der wichtigsten Punkte, dass das Review nicht durch die Person erfolgt, die den Code entwickelt hat, sondern durch jemand anderen. Weiters gibt es die Möglichkeit ein sogenanntes Peer Review durchzuführen. Bei diesem führen die Person, die den Code entwickelt hat und eine zweite Person das Review durch und besprechen den vorliegenden Code. An dieser Stelle sollte das Prinzip des *Pair Programming* [22], das aus dem *Extreme Programming* kommt erwähnt werden [1]. Beim *Pair Programming* arbeiten bei der Erstellung des Quelltextes jeweils zwei ProgrammiererInnen an einem Rechner. Dabei schreibt ein Programmierer/eine Programmiererin den Code und der/die zweite überlegt sich die Problemstellung, kontrolliert den geschriebenen Quelltext und spricht Probleme, welche dabei auffallen sofort an. Bei *Extreme Programming* geht es darum, dass formalisiertes Vorgehen in den Hintergrund gerückt wird und in erster Linie das Lösen einer Programmieraufgabe betrachtet wird.

2.5 Woher kommt CCD?

Wie zahlreiche Ideen in der Softwareentwicklung gibt es auch die Idee des CCD seit vielen Jahren. Alle der im Abschnitt 2.1 erwähnten Programmierer sind schon seit sehr vielen Jahren mit dem Programmieren von Code beschäftigt und haben sich dabei Regeln zu Grunde gelegt, wie sie Code möglichst wartbar, gut lesbar und wiederverwendbar gestalten können. CCD ist dabei wie erwähnt keine neue Idee, es ist eher eine Zusammenfassung vieler Regeln, welche sich in den letzten Jahren etabliert haben um gute Software zu gestalten.

Eine Vorgehensweise, welche sich sehr stark damit beschäftigt Code so zu gestalten, dass er besser wiederverwendbar ist, ist die Verwendung von Design Patterns. Dazu gibt es das von der *Gang of Four* geschrieben Buch Design Patterns- Elements of Reusable Object-Oriented Software [8]. Dieses Buch

beschreibt zahlreiche Möglichkeiten, wie verschiedene Design Patterns in objektorientierten Programmiersprachen angewandt werden können um den Code besser zu gestalten. Dabei trägt die korrekte Verwendung dieser auch sehr stark zur Lesbarkeit des Codes bei. Wenn eines dieser Design Patterns im Code verwendet wird und die Klasse, welche dieses Design Pattern implementiert korrekt benannt ist, kann man bereits durch den Namen dieser Klasse darauf schließen wie sie funktionieren wird. Ein Beispiel hierfür wäre das Singleton. Ein Singleton hat die Aufgabe der Erzeugung und Verwaltung der einzigen Instanz einer Klasse zu übernehmen. Es kapselt dabei den Zugriff in einem globalen Gültigkeitsbereich über eine Methode *getInstance()*. Der Leser der Klasse, welche dieses Singleton implementiert kann durch eine korrekte Benennung und das Vorhandensein der *getInstance()* - Methode sofort darauf schließen, dass es sich dabei um ein Singleton handelt. Diese Eigenschaft von Design Patterns führt wie erwähnt dazu, dass der Code leichter zu lesen und einfacher zu verstehen ist. Voraussetzung dafür ist natürlich, dass die Design Patterns korrekt implementiert sind, ansonsten wäre die Grundvoraussetzung - der Code sollte genau so funktionieren, wie man es erwartet - nicht erfüllt und somit würde eine falsche Implementierung eines Design Patterns, dieses auch ad absurdum führen. Wichtig bei der Unterscheidung zwischen Clean Code und Design Patterns ist die Tatsache, dass sich Design Patterns auf der Entwurfsebene befinden, Clean Code sich jedoch auf der Implementierungsebene befindet.

Ein weiteres Buch, welches man als Vorgängen von CCD bezeichnen könnte ist *Agile Software Development. Principles, Patterns, and Practices* [11] von Robert C. Martin. In diesem Buch beschreibt er sehr zentrale Prinzipien in der modernen Softwareentwicklung. Im folgenden werden die wichtigsten dieser Prinzipien, welche heutzutage auch kurz als SOLID - Prinzipien bezeichnet werden, kurz beschrieben und ihr Vorteil hinsichtlich CCD erwähnt. Für die einzelnen Prinzipien gibt es meist verschiedene Definitionen, im folgenden werden jedoch die originalen Definitionen von Robert C. Martin verwendet, welche er unter [19] publiziert hat.

Single-Responsibility-Prinzip (SRP)

> A class should have one, and only one, reason to change.

Mit dem SRP werden sehr viele grundlegende Aspekte von CCD abgedeckt. Klassen, welche nur einen Zweck erfüllen und daher nur einen Grund haben sich zu ändern sind leichter zu Warten und zu Ändern, da sich der ProgrammiererInnen auf die eine Aufgabe konzentrieren kann, welche diese Klasse erfüllt. Weiters wird das Testen dieser Klasse aus den selben Gründen vereinfacht.

The Open Closed Principle (OCP)

> You should be able to extend a classes behavior, without modifying it.

Beim OCP geht es in erster Linie darum, Klassen so zu gestalten, dass sie geschlossen für Änderungen und offen für Erweiterungen sind. Daraus ergibt sich, dass das Hinzufügen von neuen Funktionalitäten leichter wird, da bei Einhaltung dieses Prinzips keine Änderung der Grundlogik nötig ist, diese kann ohne weiteres gleich wie vorher verwendet werden und die neue Funktionalität wird nur als Erweiterung angeboten. Das Einhalten dieses Prinzips führt automatisch zu besser strukturierten und gekapselten Klassen und schlussendlich auch zu einer leichteren Erfüllung des SRP.

The Liskov Substitution Principle (LSP)

> Derived classes must be substitutable for their base classes.

Bei einer korrekten Anwendung des LSP sollte Klassen so gestaltet sein, dass bei jeder Verwendung des Objektes einer Oberklasse, dieses Objekt durch ein Objekt einer abgeleiteten Klasse ersetzt werden kann und dadurch kein fehlerhaftes Verhalten hervorgerufen wird. Der Vorteil der sich bei Einhaltung dieses Prinzips ergibt ist, dass bei Aufruf einer Methode einer abgeleiteten Klasse kein unerwartetes Verhalten hervorgerufen wird. Grundlegend lässt sich sagen, dass Verhalten, welches definiert ist, auch implementiert sein sollte.

The Interface Segregation Principle (ISP)

> Make fine grained interfaces that are client specific.

Durch feingranular definierte Interfaces wird der von Dave Thomas erwähnte Punkt von klaren und minimalen APIs erfüllt. Sehr minimal definierte Interfaces erfüllen außerdem das SRP und sind durch diese Granularität leichter zu verwenden. Ein weiterer positiver Aspekt von Interfaces welche das ISP erfüllen ist die leichtere Implementierung dieser.

The Dependency Inversion Principle(DIP)

> Depend on abstractions, not on concretions.

Zwei Dinge, welche durch DIP ermöglicht werden, sind die bessere Testbarkeit, da man die konkrete Implementierung durch eine Testimplementierung mit Dummydaten ersetzen kann und die Tatsache, dass die Grenzen zwischen den einzelnen Softwaresystemen besser abgetrennt sind. Weiters ergibt sich dadurch der Vorteil, dass bei Komponenten, welche noch nicht zur Verfügung stehen eine Dummyimplementierung verwendet werden kann.

Kapitel 3

Anwendungsbeispiele für Clean Code Development

Im folgenden Abschnitt sollten einige der wichtigsten CCD Kriterien verwendet werden, um ausgewählte Codeabschnitte aus verschiedenen Open Source Projekten zu verbessern. Dabei wird ein Codebeispiel geliefert, welches eines oder mehrere der CCD Kriterien missachtet, beschrieben welche Probleme bei den gezeigten Codestücken auftreten können und wie diese Probleme mit Hilfe von CCD behoben werden können.

Die Beispiele aus den folgenden Frameworks dienen der Veranschaulichung:

- Log4net
  - Hersteller: Apache Software Foundation
  - Programmiersprache: C#
  - Referenz: https://logging.apache.org/log4net/
  - Zugriffsdatum: 12.11.2015
  - Repository: https://github.com/apache/log4net
- Roslyn
  - Hersteller: Microsoft
  - Programmiersprache: C#
  - Referenz: https://github.com/dotnet/roslyn
  - Zugriffsdatum: 18.11.2015
  - Repository: https://github.com/dotnet/roslyn
- Asp.net Websockets
  - Hersteller: .NET Foundation
  - Programmiersprache: C#
  - Referenz: https://github.com/aspnet/WebSockets

  - Zugriffsdatum: 20.12.2015
  - Repository: https://github.com/aspnet/WebSockets
- Entity Framework
  - Hersteller: .NET Foundation
  - Programmiersprache: C#
  - Referenz: https://github.com/aspnet/EntityFramework
  - Zugriffsdatum: 20.12.2015
  - Repository: https://github.com/aspnet/EntityFramework
- CoreFx
  - Hersteller: .NET Foundation
  - Programmiersprache: C#
  - Referenz: http://dotnet.github.io/
  - Zugriffsdatum: 15.1.2016
  - Repository: https://github.com/dotnet/corefx
- Hibernate
  - Hersteller: JBoss (Red Hat)
  - Programmiersprache: Java
  - Referenz: http://hibernate.org/orm/
  - Zugriffsdatum: 15.10.2015
  - Repository: https://github.com/hibernate/hibernate-orm
- HBase
  - Hersteller: Apache Software Foundation
  - Programmiersprache: Java
  - Referenz: https://hbase.apache.org/
  - Zugriffsdatum: 15.1.2016
  - Repository: https://github.com/apache/hbase

3.1 Vermeiden von Duplizierungen

Eines der größten Probleme bei der Entwicklung von Software stellt die Duplizierung von Code dar. Diese verbirgt sich dabei nicht immer nur auf Bibliothek-, oder Klassenebene, sondern auch in Methoden. Bei Duplizierungen wird grundsätzlich zwischen vier Typen unterschieden. Diese erstrecken sich von der lexikalischen bis zur semantischen Duplizierung. Diese unterschiedlichen Formen von Duplizierungen sind hinlänglich beschrieben [2] und es wird daher hier nicht näher auf die einzelnen Formen eingegangen.

Oft treten Duplizierungen auf, wenn bei der Benennung der vorhandenen Methoden beziehungsweise Klassen nicht auf eine ausreichend offensichtliche Namensgebung geachtet wurde. Ein sehr wichtiges Prinzip hinsichtlich der Vermeidung von Codeduplizierung, ist das von Dave Thomas und Andy Hunt beschrieben DRY-Prinzip (Don´t Repeat Yourself [5]). Die Probleme, welche sich dadurch ergeben sind einerseits, dass Arbeiten doppelt erledigt werden müssen, andererseits kann im Falle eines Fehlers auf das Beheben dieses in einem duplizierten Abschnitt vergessen werden. Robert C. Martin bietet in seinem Buch Clean Code einige Möglichkeiten wodurch diese Probleme vermieden werden können. Diese sollten im folgenden Abschnitt näher betrachtet werden.

3.1.1 Duplizierung boolscher Ausdrücke

- Projekt: *Hibernate*
- Programmiersprache: *Java*
- Betreffende Klasse: *AbstractPropertyHolder*
- Betreffendes Paket: *org.hibernate.cfg*

Wie bereits erwähnt, treten Codeduplizierungen nicht nur auf Bibliothek-, bzw. Klassenebene auf, sonder meist in einem viel kleineren Kontext. Dies ist oft ein sehr subtiles Problem und kann nicht so leicht festgestellt werden. Oft lassen sich Duplizierungen in bei komplexen boolschen Ausdrücken finde. Diese werden oft von mehreren Funktionen benötigt und daher einfach in den einzelnen Funktionen verwendet. Ein Beispiel für eine solche Duplizierung befindet sich in den Listings 3.1 und 3.2.

```
1  @Override
2  public JoinColumn[] getOverriddenJoinColumn(String
     propertyName) {
3    JoinColumn[] result = getExactOverriddenJoinColumn(
     propertyName );
4    if (result == null &&
     propertyName.contains(".collection&&element.")) {
5      ...
6    }
7    return result;
```

```
8   }
```

Listing 3.1: Komplexe boolsche Ausdrücke 1 Zeile 255 - 264

```
1 public JoinTable getOverriddenJoinTable(String propertyName) {
2   JoinTable result = getExactOverriddenJoinTable(propertyName);
3   if(result == null &&
      propertyName.contains(".collection&&element.")){
4     ...
5   }
6   return result;
7 }
```

Listing 3.2: Komplexe boolsche Ausdrücke 2 Zeile 305 - 313

Konkret geht es in diesem Beispiel um den in Listing 3.3 beschriebenen boolschen Ausdruck der in dieser Klasse insgesamt drei mal vorkommt.

```
1 result == null && propertyName.contains(".collection&&element.")
```

Listing 3.3: Boolscher Audruck

Wie man anhand dieser Listings leicht erkennen kann, wird dieser Ausdruck in beiden Methoden verwendet, wodurch sich eine Code Duplizierung ergibt. Dieses Problem kommt vor allem bei zukünftigen Änderungen zu Tragen. Wenn zum Beispiel für die Variable *propertyName* eine *XML-Zeichenkette* übergeben wird und die Überprüfung wie in Listing 3.4 dargestellt erfolgen müsste, würde dies eine Änderung in allen Funktionen welche dieses boolsche Statement verwenden nach sich ziehen.

```
1  result == null &&
     propertyName.contains("</collectionType><element>")
```

Listing 3.4: Boolscher Ausdruck neu

Falls beim Ändern der Überprüfung auf eine der Verwendungen vergessen wird, würde dies zu einem nur sehr schwer zu findenden Fehler führen. Ein weiteres Problem, das sich hier wie bereits oben beschrieben ergibt, ist die Tatsache, dass dieser boolsche Ausdruck ohne einen Kommentar oder ein mehrmaliges Lesen sehr schwer zu verstehen ist. CCD empfiehlt hier den boolschen Ausdruck in eine eigene Methode zu extrahieren. Eine solche Methode könnte wie in Listing 3.5 aussehen.

```
1 private bool
     isPlaceholderForCollectionAndElementInPropertyName(object
     result, String propertyName) {
2   return result == null &&
     propertyName.contains(".collection&&element.");
```

```
3 }
```

Listing 3.5: Boolscher Ausdruck neu

Durch ein Extrahieren der Überprüfung des Parameters *propertyName* und des Null-Checks kann für diese Methode ein eigener Name gewählt werden, der klar den Zweck dieser Überprüfung vermittelt wodurch die Codeduplizierung eliminiert wird.

3.1.2 Redundante Kommentare

- Projekt: *Log4Net*
- Programmiersprache: *C#*
- Betreffende Klasse: *ConfigurationChangedEventArgs*
- Betreffender Namespace: *log4net.Repository*

Redundante Kommentare findet man sehr häufig in den verschiedensten Projekten. Ein Beispiel für einen solchen Kommentar lässt sich in Listing 3.6 finden.

```
1 /// <summary>
2 ///
3 /// </summary>
4 /// <param name="configurationMessages»</param>
5 public ConfigurationChangedEventArgs(ICollection
      configurationMessages)
6 {
7     this.configurationMessages = configurationMessages;
8 }
```

Listing 3.6: Beispiele für überflüssige Kommentare

Oft werden für Methoden Kommentare geschrieben, welche nicht mehr als den Namen der Methode beinhalten. Dies verschlechtert die Lesbarkeit, führt zu einer unnötigen Aufblähung des Codes und stellt im Grunde genommen eine Codeduplizierung dar. Sehr häufig tritt dieses Problem bei Projekten auf, in denen es Coding Conventions gibt die vorschreiben, dass jede Methode einen sogenannten Methodenheader besitzen muss. Robert C. Martin schreibt hierzu (Clean Code, Seite 93 - 96), dass solche Regeln meist zu Verwirrung, Lügen und einer allgemeinen Unordnung führen. Wenn man nun das in Listing 3.6 stehende Beispiel betrachtet, fällt einen sofort der Kommentar auf, welcher keine richtigen Mehrwert für den Leser bringt. Wie Robert C. Martin erwähnt, führt dieser nur zu einer Unordnung und kann im schlimmsten Fall sogar zu einer fälschlichen Information führen, falls der Parameter umbenannt wird und der Kommentar dafür nicht angepasst wird. Auf Grund dieser Tatsache, sollte man laut Robert C. Martin auch auf solche Kommentare, sowie Regeln die

einen solchen Methodenheader vorschreiben verzichten, da diese eben genau zu den genannten Probleme führen.

3.1.3 Duplizierung von Testcode

- Projekt: *Log4Net*
- Programmiersprache: *C#*
- Betreffende Klasse: *FixingTest*
- Betreffender Namespace: *log4net.Tests.Core*

Laut Robert C. Martin sollte auch Testcode den CCD Kriterien entsprechen. Dies ist umso wichtiger, da Testcode oft als Dokumentation für den getesteten Code dienst. Ein Programmierer/eine Programmiererin der einen Code betrachtet der gut getestet ist, kann sich sofort ein Bild von den einzelnen Grenzfällen machen indem er die Tests betrachtet. Dort sollten die wichtigsten Grenzfälle abgedeckt sein und sie geben außerdem Aufschluss darüber wie die Methode verwendet werden kann. Ein sehr wichtiger Punkt im Zusammenhang mit Duplizierung ist, dass vor allem bei Tests darauf geachtet werden, dass diese keine Duplizierungen enthalten. Ein Beispiel für eine solche Duplizierung lässt sich in Listing 3.7 finden.

```
1     [Test]
2     public void TestUnfixedValues()
3     {
4       ...
5
6       LoggingEvent loggingEvent = new LoggingEvent(
7         loggingEventData.LocationInfo.GetType(),
8         LogManager.GetRepository(TEST_REPOSITORY),
9         loggingEventData.LoggerName,
10        loggingEventData.Level,
11        loggingEventData.Message,
12        new Exception("This is the exception"));
13
14      ...
15    }
16
17      [Test]
18    public void TestAllFixedValues()
19    {
20      ...
21
22      LoggingEvent loggingEvent = new LoggingEvent(
23        loggingEventData.LocationInfo.GetType(),
24        LogManager.GetRepository(TEST_REPOSITORY),
25        loggingEventData.LoggerName,
```

```
26          loggingEventData.Level,
27          loggingEventData.Message,
28          new Exception("This is the exception"));
29
30        ...
31      }
```

Listing 3.7: Beispiele für Duplizierung in Tests

Das Problem welches sich hier ergibt ist die Instanziierung der Klasse *LoggingEvent*. Über den Konstruktor werden die verschiedenen Parameter definiert und anschließend in der Klasse verwendet. Wenn sich die Anforderungen für die Klasse *LoggingEvent* ändern, müssen an allen Stellen, an denen der Konstruktor verwendet wird Änderungen vorgenommen werden. Roy Osherove zeigt in seinem Buch [13] eine Möglichkeit auf, wie mit solchen Initialisierungscode vorgegangen werden kann. Er empfiehlt, dass der Code welcher die zu testende Klasse instanziiert bzw. initialisiert in eine eigene *Factory* - Methode ausgelagert wird. Dies hat den Vorteil, dass die Duplizierung eliminiert wird und der Code wiederverwendet werden kann. Bei einer Änderung der Parameter des Konstruktors muss man in den Tests nur noch an einer Stelle Anpassungen vornehmen. In Listing 3.8 wird diese Variante dargestellt.

```
1     private LoggingEvent GetDefaultLoggingEvent(LoggingEventData
      loggingEventData){
2       LoggingEvent loggingEvent = new LoggingEvent(
3         loggingEventData.LocationInfo.GetType(),
4         LogManager.GetRepository(TEST_REPOSITORY),
5         loggingEventData.LoggerName,
6         loggingEventData.Level,
7         loggingEventData.Message,
8         new Exception("This is the exception"));
9     }
10
11    [Test]
12    public void TestUnfixedValues()
13    {
14      ...
15      LoggingEvent loggingEvent =
      GetDefaultLoggingEvent(loggingEventData);
16      ...
17    }
18
19      [Test]
20    public void TestAllFixedValues()
21    {
22      ...
```

```
23            LoggingEvent loggingEvent =
      GetDefaultLoggingEvent(loggingEventData);
24          ...
25      }
```

Listing 3.8: Eliminerung der Duplizierung durch Factorymethode

Wie man anhand dieses Beispiels leicht erkennen kann, lassen sich neben dem Eliminieren der Duplizierung auch noch einige Zeilen Code sparen und der Code wird um einiges übersichtlicher.

3.2 Legacy Code vermeiden und refaktorieren

Bei Elementen die sehr lange in der Codebasis einer Software verbleiben kommt es oft dazu, dass der Quellcode immer schlechter zu lesen wird. Oft wird diese Art von Code dann als Legacy Code [3] bezeichnet. Features werden hinzugefügt, für z.B. Performanceschwierigkeiten wird ein Workaround eingebaut, der die Struktur weiter verschlechtert. Falls es Tests gibt, was bei Legacy Code meist nicht der Fall ist, sind diese durch die zahlreichen Änderungen die vorgenommen werden müssen immer schwerer zu warten und werden meist einfach gelöscht oder nicht mehr ausgeführt. Da die meisten problematischen Elemente in einer Codebasis Legacy Code sind sollte in diesem Abschnitt darauf eingegangen werden, wie CCD dabei unterstützen kann, Legacy Code zu vermeiden und Legacy Code zu refaktorieren.

3.2.1 Auskommentierter Code

- Projekt: *Hibernate*
- Programmiersprache: *Java*
- Betreffende Klasse: *BindHelper*
- Betreffendes Paket: *org.hibernate.cfg*

Code der nicht mehr benötigt wird, wird häufig einfach auskommentiert und bleibt danach über lange Zeit im Quellcode bestehen. Dies wäre jedoch bei den modernen Versionsverwaltungssystemen gar nicht mehr notwendig, da diese eine genaue Auflistung der gelöschten, geänderten oder hinzugefügten Abschnitte anbieten. Es ist mit diesen auch leicht möglich Abschnitte, die man gelöscht hat, wieder aufzufinden, sowie diese wiederherzustellen. Code der nicht mehr benötigt wird sollte daher einfach gelöscht werden und mit einer vernünftigen Commit Message versehen werden. Eine sehr problematische Beispiel dafür befindet sich in dem in Listing 3.9 gezeigten Codeabschnitt.

```
1 /*FIXME cannot use subproperties becasue the caller needs top
      level properties
```

```
2 //if (property.isComposite()) {
3 //  Iterator subProperties =
      ((Component)property.getValue()).getPropertyIterator();
4 //  while (subProperties.hasNext()) {
5 //    matchColumnsByProperty(((Property)subProperties.next()),
      columnsToProperty);
6 //  }
7 }*/
```

Listing 3.9: Beispiele für auskommentierten Code

Der Kommentar deutet darauf hin, dass es in diesem Codeabschnitt einen Fehler gibt der behoben werden müsste. Anstatt diesen Fehler zu beheben wurde der Code einfach auskommentiert und nach einigen Wochen weiß niemand mehr, dass es diesen Fehler gibt. Hier sollte entweder eine genaue Dokumentation dieses Fehlers in einem Issue Tracking System erfolgen, oder versucht werden den Fehler direkt zu beheben. Diesen einfach stehen zu lassen und die fehlerhafte Codestelle auszukommentieren ist dabei wohl der schlechteste Weg, da so der Fehler nicht mehr auftreten wird und er somit vergessen wird. Auch Robert C. Martin schlägt in seinem Buch eine sehr pragmatische Lösung vor: Auskommentierter Code sollte immer gelöscht werden, da er zusätzlich zu den genannten Gründen den Quellcode unnötig aufbläht.

3.2.2 Ausnahmen sind besser als Fehler-Codes

In den modernen Programmiersprachen wie C++, C# oder Java, gibt es die Möglichkeit, für einen fehlgeschlagenen Vorgang eine Ausnahme zu werfen. In den etwas älteren Programmiersprachen, wie zum Beispiel C, gab es diese Möglichkeit noch nicht. Daher wurden in solchen Situationen sogenannte Fehler-Codes zurückgegeben, was zu folgenden Problemen führen konnte:

- Beim Aufruf dieser Methode muss darauf geachtet werden, dass alle möglichen Fehler-Codes abgedeckt werden.
- Da diese Fehler-Codes meist ganzzahlige Werte darstellen, ist es auch sehr schwierig, diesen eine gewisse Semantik zuordnen. Meist werden für diese ganzzahligen Werte dann Konstanten definiert, welche dann im Quellcode, oder der Dokumentation beschrieben werden.
- Durch die Rückgabe eines Fehler-Codes ist es nicht mehr möglich einen Wert zurückzugegeben, wodurch meist Parameter als Outputparameter verwendet werden, welche den gewünschten Wert zurückliefern.

Die sogenannten Ausnahmen stellen für diese Problemen einen sehr guten Lösungsansatz dar. Im Fehlerfall wird eine Ausnahme ausgelöst und schließlich im aufrufenden, oder in einem hierarchisch höheren Bereich behandelt. Diesen Ausnahmen kann eine Fehlermeldung zugeordnet werden und es gibt die

Möglichkeit über den sogenannten Stacktrace nachzuvollziehen, an welcher Stelle im Code der Fehler aufgetreten. Weiters ist es durch dieses System ohne weiteres möglich bei der Funktion einen normalen Rückgabewert zu definieren, da die Ausnahme nicht als Rückgabewert definiert werden muss. Durch diese Möglichkeiten ergeben sich zahlreiche Vorteile gegenüber Fehler-Codes und daher sollte bei Verwendung von moderneren Programmiersprachen auf Fehler-Codes unbedingt verzichtet werden. Robert C. Martin liefert in Clean Code [12, Seite 78] das in Listing 3.10 gezeigte Beispiel.

```
1 if(deletPage(page) == E_OK){
2     if(registry.deleteReference(page.name) == E_OK){
3         if(configKeys.deleteKey(page.name.makeKey()) == E_OK){
4             logger.log("page deleted");
5         } else {
6           logger.log("configKey not deleted");
7         }
8     }
9     else {
10        logger.log("deleteReference from registry failed");
11    }
12 } else{
13   logger.log("deleted failed");
14   return E_ERROR;
15 }
```

Listing 3.10: Beispiele für die Verwendung von Fehler-Codes

Hier sind einige der beschriebenen Probleme bereits sehr gut zu erkennen. Das größte Problem an dieser Form der Implementierung besteht in der tiefen Verschachtelung der einzelnen Verzweigungen, wodurch die Lesbarkeit sehr stark verschlechtert wird. Bei dieser Art der Implementierung ist es zudem sehr schwierig die Reihenfolge zu ändern. Wenn bei einem späteren betrachten der Methode erkannt wird, dass die Aufrufe möglicherweise in einer umgekehrten Reihenfolge stattfinden müssen, da zu erst die *Keys* und die *References* gelöscht werden müssen, würde eine Änderung der Struktur der Funktion nötig machen. Was in der in Listing 3.11 gezeigten Funktion resultieren würde.

```
1 if(configKeys.deleteKey(page.name.makeKey()) == E_OK){
2     if(registry.deleteReference(page.name) == E_OK){
3         if(deletPage(page) == E_OK){
4             logger.log("page deleted");
5         } else {
6             logger.log("deleted failed");
7         }
8     }
9     else {
10        logger.log("configKey not deleted");
```

```
11     }
12 } else{
13   logger.log("deleteReference from registry failed");
14   return E_ERROR;
15 }
```

Listing 3.11: Geänderte Reihenfolge der Funktion

Für diese grundsätzlich kleine Änderung sind sehr viele Schritte notwendig, welche wiederum sehr schnell zu Fehlern führen können. Wenn statt den Fehler-Codes Ausnahmen verwendet werden ergibt sich für das Beispiel in Listing 3.10 der in Listing 3.12 gezeigte Quellcode.

```
1 try {
2     deletePage(page);
3     registry.deleteReference(page.name);
4     configKeys.deleteKey(page.name.makeKey());
5 } catch(Exception e){
6     logger.log(e.getMessage());
7 }
```

Listing 3.12: Beispiele für die Verwendung von Ausnahmen statt Fehler-Codes

Durch das ersetzen der Fehler-Codes durch Ausnahmen, können die Überprüfungen auf die Fehler-Codes entfernt werden, wodurch sich die Lesbarkeit der Funktion sehr stark verbessert. Es ist nun einfach möglich zu erkennen, welche Schritte durchgeführt werden und eine Änderung dieser Funktion ist sehr leicht zu bewerkstelligen.

3.2.3 Polymorphismus statt If/Else oder Switch/Case verwenden

- Projekt: *Entity Framework*
- Programmiersprache: *C#*
- Betreffende Klasse: *CommandLogger*
- Betreffender Namespace: *Microsoft.Data.Entity.Design.Internal*

Ein Konstrukt das sich in sehr vielen Bibliotheken finden lässt, sind *Switch-Statements*. Ein Beispiel für ein solches Konstrukt lässt sich in Listing 3.13 finden.

```
1 public virtual void Log(
2   LogLevel logLevel,
3   int eventId,
4   object state,
```

```
5   Exception exception,
6   Func<object, Exception, string> formatter)
7 {
8   //Building message
9   ...
10  //
11
12  switch (logLevel)
13  {
14    case LogLevel.Error:
15      WriteError(message.ToString());
16      break;
17    case LogLevel.Warning:
18      WriteWarning(message.ToString());
19      break;
20    case LogLevel.Information:
21      WriteInformation(message.ToString());
22      break;
23    case LogLevel.Debug:
24      WriteDebug(message.ToString());
25      break;
26    case LogLevel.Trace:
27      WriteTrace(message.ToString());
28      break;
29    default:
30      Debug.Fail("Unexpected event type: " + logLevel);
31      WriteDebug(message.ToString());
32      break;
33  }
34 }
```

Listing 3.13: Beispiele für Switch Statement

In dem gezeigten Fall wird abhängig von der Eingabe geregelt, welche Ausgabe erfolgen sollte. Dies ist eine Vermischung der Aspekte und widerspricht dem SRP, da im ersten Schritt die Eingabe überprüft und abhängig davon eine Aktion vorgenommen wird. Dieses Missachten des SRP resultiert in einer schlechteren Testbarkeit. Ein weiteres Prinzip, welches in diesem Fall verletzt wird ist das OCP, da beim Hinzufügen eines weiteren *LogLevel* der vorhandene Code geändert werden muss, was bedeutet, dass die Klasse nicht geschlossen für Änderungen ist. Robert C. Martin empfiehlt hier, dass diese Verzweigung durch Abstrahierungen aufgelöst werden könnte. In dem konkreten Fall wäre es möglich statt dem *LogLevel* ein Delegaten zu übergeben, welcher die gewünschte Logfunktionalität aufruft, wodurch sich der in Listing 3.14 dargestellte Quellcode ergibt.

```
1 public virtual void Log(
```

```
2   Action<string> writeLog,
3   int eventId,
4   object state,
5   Exception exception,
6   Func<object, Exception, string> formatter)
7 {
8   //Building message
9   ...
10  //
11
12  writeLog(message.ToString());
13 }
```

Listing 3.14: Beispiele für Switch Statement

Durch diese Änderung wird die Funktion um einiges kürzer und dadurch übersichtlicher. Es wird weiterhin die Aufgabe der Überprüfung ausgelagert und die Klasse ist geschlossen für Änderungen aber offen für Erweiterungen, was dem SRP und dem OCP entspricht. Beim Aufrufen dieser Methode kann jetzt zusätzlich zu Funktionen, welche sich auf die einzelnen *LogLevels* beziehen, auch eine spezielle Log-Funktion übergeben werden, welche zum Beispiel in eine Datenbank schreibt. Die wichtigste Verbesserung ist hier, wie bereits erwähnt, die stark verbesserte Lesbarkeit.

3.2.4 Keine Null zurückgeben

- Projekt: *CoreFx*
- Programmiersprache: *C#*
- Betreffende Klasse: *ProcessModuleCollection*
- Betreffender Namespace: *System.Diagnostics*

In modernen Programmierumgebungen wie Java und C# stellen die sogenannten *Nullreferenzausnahmen* eine der häufigsten Ausnahmefälle dar. Diese führen dazu, dass beim Zugriff auf ein Objekt, welches den Wert *null* besitzt, ein Laufzeitfehler auftritt, welcher behandelt werden muss. Noch problematischer sind diese Ausnahmen im C++ - Bereich, da dort keine Ausnahme ausgelöst wird, falls das Objekt den Wert null besitzt, da jedes Objekt auf einen gewissen Speicherbereich verweist und dieser Speicherbereich aber vorhanden ist. In C++ führt der Zugriff auf ein *Null-Objekt* zu einem Zugriff auf einen ungültigen Speicherbereich. Solche Fehler sind sehr schwer nachzuvollziehen und führen häufig zu großen Problemen wenn die Software bereits im Betrieb ist. Meist wird diesen Problem durch zahlreiche Überprüfungen ob der zurückgegebene Wert *null* ist, vorgebeugt. Dies führt aber zu stark überladenen Methoden und kann es kann auch sehr schnell darauf vergessen werden diese Überprüfungen einzubauen.

Diese Probleme und die Überladung des Codes mit Überprüfungen können durch das Verzichten auf die Rückgabe von Null Werten verhindert werden. Für Listen beispielsweise sollte eine leere List zurückgegeben werden. Meist wird für Listen nach dem Aufruf eine Form der Iteration durchgeführt. Entweder wird in einer *for-Schleife* über die einzelnen Elemente der Liste iteriert, was bei einer leeren Liste einfach dazu führt, dass die Schleife nicht durchlaufen wird. Der Codeabschnitt in Listing 3.15 zeigt ein Beispiel für diese Vorgehensweise.

```
protected List<ProcessModule> InnerList
{
  get
  {
      if (_list == null)
        _list = new List<ProcessModule>();
    return _list;
  }
}
```

Listing 3.15: Beispiele für Rückgabe einer leeren Liste statt eines Null Wertes

Hier wird zu erst intern überprüft, ob die Liste, welche in einer Membervariable gespeichert ist *null* ist, wenn ja wird der Wert dieser auf eine leere Liste gesetzt und somit können *Nullreferenzausnahmen* verhindert werden. Im Zusammenhang mit Objekten gibt es die Möglichkeit das sogenannte *Null Object pattern* [4] zu verwenden. Bei diesem Pattern wird statt des Wertes *null* eine leere Implementierung des Objektes zurückgegeben. Ein Beispiel, für ein Implementierung des *Null Object pattern* ist in Listing 3.16 dargestellt.

```
public interface IVehicle {
    void Drive();
}

public class FastVehicle : IVehicle {
    public void Drive(){
        Console.WriteLine("Drive Fast");
    }
}

public class NullVehicle : IVehicle {
    public void Drive(){
        //Do nothing here. Null Implementation
    }
}
```

Listing 3.16: Null Object pattern

Die Klasse *NullVehicle* implementiert das Interface *IVehicle*, welches nur eine Methode enthält. In Listing 3.17 wird nun dargestellt, wie eine Variante mit Rückgabe des *Null Object pattern* und eine mit Rückgabe von *null* dargestellt:

```
1 public IVehicle GetVehicle(VehicleType vehicleType){
2     if(vehicleType == VehicleType.Fast){
3         return new FastVehicle();
4     }
5     else {
6         return null;
7     }
8 }
9
10 public IVehicle GetVehicle(VehicleType vehicleType){
11     if(vehicleType == VehicleType.Fast){
12         return new FastVehicle();
13     }
14     else {
15         // Here we return our Null Object pattern.
16         return NullVehicle();
17     }
18 }
```

Listing 3.17: Anwendung des Null Object pattern

Bei der ersten Variante, ist der Aufrufer gezwungen, die Rückgabe auf *null* zu überprüfen. Wenn dies nicht getan wird und ein *null* Wert zurückgegeben wird kommt es beim Aufruf der *Drive()* Methode zu einer *NullReference-Exception*. In der zweite Variante ist diese Überprüfung nicht notwendig. In der aufrufenden Methode kann der Rückgabewert direkt verwendet werden, ohne dass die Gefahr besteht, dass eine *NullReference-Exception* auftritt.

3.2.5 Zu viele Parameter / Flag Argumente in Funktionen

- Projekt: *HBase*
- Programmiersprache: *Jave*
- Betreffende Klasse: *HFileLink*
- Betreffendes Package: *org.apache.hadoop.hbase.io;*

Häufig lassen sich in Legacy Systemen Funktionen finden die mehr als ein vernünftiges Maß an Parametern haben. Robert C. Martin grenzt die maximale Anzahl an Parametern die verwendet werden sollten mit drei ab. Dabei erwähnt er explizit, dass niladische Methoden (kein Parameter) die besten Methoden sind, gefolgt von den monadischen (ein Parameter), den dyadischen (zwei Parameter) und den triadischen (drei Parameter), welche jedoch auch möglichst vermieden werden sollten. Polyadische (mehr als drei Parameter)

sollten seiner Meinung nach unbedingt geändert werden. Ein Beispiel für eine solche polyadische Funktion lässt sich in Listing 3.18 finden.

```
1  public static boolean create(final Configuration conf, final
     FileSystem fs,
2      final Path dstFamilyPath, final TableName linkedTable,
     final String linkedRegion,
3      final String hfileName, final boolean createBackRef)
     throws IOException {
4    ...
5  }
```

Listing 3.18: Beispiele für Rückgabe eines Null Wertes

Zusätzlich zu dem Problem, dass diese Funktion viel zu viele Parameter besitzt, kann davon ausgegangen werden, dass diese Funktion auch das SRP verletzt. Vor allem der boolsche Parameter *createBackRef* deutet drauf hin, da sogenannte Flag-Argumente ein Indiz für die Verletzung des SRP auf Funktionsebene sind. Robert C. Martin geht dabei sogar soweit, dass er Flag-Argumente als schreckliche Technik bezeichnet. Im ersten Schritt sollte also das Flag-Argument entfernt werden. Eine Möglichkeit dieses Flag-Argument aufzulösen besteht darin, die Funktion in zwei Funktionen aufzuspalten, was in dem im Listing 3.19 resultieren.

```
1   public static boolean create(final Configuration conf, final
     FileSystem fs,
2      final Path dstFamilyPath, final TableName linkedTable,
     final String linkedRegion,
3      final String hfileName) throws IOException {
4    ...
5  }
6
7  public static boolean createBackReference(final Configuration
     conf, final FileSystem fs,
8      final Path dstFamilyPath, final TableName linkedTable,
     final String linkedRegion,
9      final String hfileName) throws IOException {
10   ...
11 }
```

Listing 3.19: Beispiele für Eliminierung des Flag Arguments

Durch diese kleine Änderung könnte bereits ein Parameter entfernt werden und durch die Auftrennung in zwei Funktionen können diese beiden Funktion jeweils einzelne Aufgaben erledigen. Zur Vermeidung der restlichen Parameter bietet Robert C. Martin eine sehr einfache, aber effiziente Lösung an, den meist deuten Funktionen, die viele Parameter besitzen daraufhin, dass die Parameter

eigentlich zusammengefasst werden sollten. Die Parameter *dstFamilyPath*, *linkedTable*, *linkedRegion* und *hfileName* sollten in ein eigenes Objekt ausgelagert werden. Der Zugriff auf die einzelnen Parameter würde nun über diese Objekt erfolgen und die Anzahl der Parameter könnte auf drei reduziert werden. Das Ergebnis dieses Refactorings könnte wie in Listing 3.20 aussehen:

```
1   public static boolean create(final Configuration conf, final
     FileSystem fs, HFileLinkPreferences preferences) throws
     IOException {
2     ...
3   }
4
5   public static boolean createBackReference(final Configuration
     conf, final FileSystem fs,
6       final Path dstFamilyPath, final TableName linkedTable,
     final String linkedRegion, HFileLinkPreferences preferences)
     throws IOException {
7     ...
8   }
```

Listing 3.20: Beispiele für Rückgabe eines Null Wertes

Der Vorteil, welcher sich aus dieser Änderung ergibt, ist in erster Linie die bessere Lesbarkeit. Ein weiterer Vorteil ergibt sich dadurch, dass die Klasse *HFileLinkPreferences* um weitere Einstellungen erweitert werden kann.

3.3 Ausdrucksstarker Code

Sehr häufig kommt es in der Softwareentwicklung zu Unklarheiten. Der Programmierer/die Programmiererin hat es beim Schreiben des Codes verabsäumt, diesen so zu gestalten, dass der Leser ohne gröbere Probleme den Zweck dieses erkennen kann. Dies führt oft zu Missverständnissen wie der betrachtete Code eingesetzt werden sollte, oder bei Änderungen zieht dies meist ein längeres Studium des zu ändernden Codes nach sich. Zahlreiche Probleme mit Unklarheiten könnten durch eine vernünftige Namensgebung behoben werden. Daher ist die Namensgebung ein sehr zentraler und wichtiger Aspekt. Robert C. Martin geht dabei sogar soweit, dass er die Namensgebung in der Programmierung mit der Namensgebung für Kinder vergleicht. Damit versucht er dem Leser klar zu machen, welche Rolle die Namensgebung in der Programmierung spielt. Im folgenden Abschnitt werden einige wichtigen Punkte hinsichtlich der Gestaltung von ausdrucksvollen Code gezeigt.

3.3.1 Deskriptive Namen wählen

- Projekt: *Log4Net*

- Programmiersprache: *C#*
- Betreffende Klasse: *LogLog*
- Betreffender Namespace: *log4net.Util*

Der Ausschnit in Listing 3.21, der die Klasse *LogLog* zeigt, ist ein Beispiel für eine schlecht gewählten Namen.

```
1 /// <summary>
2   /// Outputs log statements from within the log4net assembly.
3   /// </summary>
4   /// <remarks>
5   /// <para>
6   /// Log4net components cannot make log4net logging calls.
7   /// However, it is sometimes useful for the user to learn
8   /// about what log4net is doing.
9   /// </para>
10  /// <para>
11  /// All log4net internal debug calls go to the standard output
12  /// stream whereas internal error messages are sent to the
13  /// standard error output stream.
14  /// </para>
15  /// </remarks>
16  /// <author>Nicko Cadell</author>
17  /// <author>Gert Driesen</author>
18  public sealed class LogLog
```

Listing 3.21: Beispiele für schlechte Namensgebung

Das Problem, welches sich im Zusammenhang mit dem gewählten Namen für *LogLog* ergibt, ist die Notwendigkeit eines Kommentars um klarzumachen, welche Aufgabe die Klasse erfüllt. Durch einen Blick in den im Listing 3.21 stehenden Kommentar wird klar, dass diese Klasse das interne Logging für Log4Net realisiert.

Es wäre ein leichtes, die Ausdruckskraft dieses Codes, durch eine Änderung des Namens zu erhöhen. Dadurch könnte auch der Kommentar gelöscht werden, da dieser schlussendlich nicht mehr nötig ist. In Listing 3.22 ist ein Beispiel für eine bessere Namensgebung dargestellt.

```
1   public sealed class Log4NetInteralLogging
```

Listing 3.22: Beispiele für bessere Namensgebung

3.3.2 Try/Catch - Blöcke extrahieren

- Projekt: *Roslyn*
- Programmiersprache: *C#*

- Betreffende Klasse: *MetadataAndSymbolCache*, *FileKey*
- Betreffender Namespace: *Microsoft.CodeAnalysis.CompilerServer*, *Roslyn.Utilities*

Das behandeln von Fehlern und Ausnahmen ist ein sehr zentraler Aspekt eines jeden Programmes. Eine richtige und gut implementierte Strategie zur Fehlerbehandlung kann sehr viel Einfluss auf die Wartbarkeit eines Programmes haben. Oft wird dabei Logik und Fehlerbehandlung vermischt, was meist zu einer Verschlechterung der Lesbarkeit und damit zu einer verminderten Ausdruckskraft führt. Da es bei der Fehlerbehandlung um einen eigenen Aspekt geht, sollte diese in eine eigene Methode ausgelagert werden. Diese Methode ist dabei ein Wrapper für die zu behandelte Methode. Ein Beispiel aus dem Quellcode von Roslyn ist die in Listing 3.23 gezeigte Methode.

```
1 private FileKey? GetUniqueFileKey(string filePath)
2 {
3   try
4   {
5     return FileKey.Create(filePath);
6   }
7   catch (Exception)
8   {
9     ...
10  }
11 }
```

Listing 3.23: Beispiele für getrennten Aspket der Fehlerbehandlung

Was man an diesem Beispiel gut erkennen kann, ist die Trennung der Aspekte. Es wird eine Methode zum Erstellen eines *FileKey* Objektes aufgerufen und anstatt die Fehlerbehandlung in der *Create* Methode zu erledigen, wird sie außerhalb dieser Methode implementiert und führt daher zu keiner Vermischung der Aspekte. Dabei ergibt sich jedoch das Problem, dass beim Aufruf der Methode *Create* auf die Fehlerbehandlung vergessen werden kann, was im schlimmsten Fall zu einer unbehandelten Ausnahme führt. Es wäre daher besser die Fehlerbehandlung direkt in einen Wrapper in der Klasse *FileKey* zu implementieren, der die Fehlerbehandlung vornimmt und danach die Methode *Create* aufruft.

Dies würde in den in Listing 3.24 und in Listing 3.25 gezeigten Änderung in der Klasse *FileKey* resultieren.

```
1 public static FileKey Create(string fullPath)
2 {
3   return new FileKey(fullPath,
      FileUtilities.GetFileTimeStamp(fullPath));
```

```
4 }
```

Listing 3.24: Fehlerbehandlung in der Klasse FileKey vorher

```
1 private static FileKey CreateKey(string fullPath)
2 {
3   return new FileKey(fullPath,
      FileUtilities.GetFileTimeStamp(fullPath));
4 }
5
6 public static FileKey? Create(string fullPath)
7 {
8   try
9   {
10    return CreateKey(filePath);
11  }
12  catch (Exception)
13  {
14    return null;
15  }
16 }
```

Listing 3.25: Fehlerbehandlung in der Klasse FileKey nachher

Der Aufruf würde der gleiche bleiben, da nur die *Create* Methode im öffentlichen Gültigkeitsbereich zugänglich ist, jedoch könnte sichergestellt werden, dass eine Fehlerbehandlung stattfindet. Auf Grund der Tatsache, dass im Fehlerfall *null* zurückgegeben werden sollte, gibt es eine kleine Änderungen an der Signatur, sodass mit dieser Variante ein *FileKey?* zurückgegeben wird, was einem Strukturdatentyp entspricht der den Wert null annehmen kann.

3.3.3 Überholte Kommentare

Wie bereits in vorherigen Abschnitten beschrieben sind Kommentare ein sehr wirksames Mittel um Details zu klären, wieso etwas auf eine spezielle Art und Weise implementiert wurde. Oft gibt es Probleme mit Kommentaren, da diese über sehr lange Zeit im Code verbleiben und meist den Quellcode, welchen diese dokumentieren überdauern, was dazu führt, dass der Kommentar entweder ungültig ist und im besten Fall nur etwas nicht mehr vorhandenes beschreibt, oder im schlimmsten Fall dazu führt, dass der Kommentar schlichtweg falsch ist und dieser für den Programmierer/die Programmiererin, welcher diesen Kommentar liest, keine Unterstützung bietet und diesem sogar falsche Informationen liefert. Aus diesem Grund gibt es seitens des CCD die Empfehlung nur die Kommentare zu schreiben, die unbedingt notwendig sind und wenn ein Kommentar geschrieben wird sollte dieser sehr gut sein. Sobald der Kommentar seine Gültigkeit verliert, sollte mit ihm vorgegangen werden wie mit allen sogenannten

toten Codeabschnitten. Er sollte einfach gelöscht werden. Zu diesem Thema gibt es auch zahlreiche empirische Untersuchen, die unter anderem zeigen, dass in zahlreichen großen Projekten Inkonsistenzen zwischen dem kommentierten Codeabschnitt und dem Kommentar bestehen. Eine dieser Studien wurde unter dem Titel *iComment: Bugs or Bad Comments?* [14] veröffentlicht. Dort werden anhand einiger großer Projekte gezeigt, wie viele Inkonsistenzen durch überholte Kommentare auftreten.

3.3.4 Magische Zahlen durch benannte Konstanten ersetzen

- Projekt: *ASP.Net Websockets*
- Programmiersprache: *C#*
- Betreffende Klasse: *FrameHeader*
- Betreffender Namespace: *Microsoft.AspNetCore.WebSockets.Protocol*

Ein Problem welches in vielen Projekten zu finden ist sind Magische Zahlen. Es handelt sich dabei um Zahlliterale die direkt in den Quellcode geschrieben werden. In Listing 3.26 ist ein Beispiel für eine Magische Zahl.

```
1         public FrameHeader(bool final, int opCode, bool masked,
      int maskKey, long dataLength){
2             ...
3             if (masked){
4                 headerLength += 4;
5             }
6             ....
7         }
```

Listing 3.26: MagicNumbers

Durch die Verwendung eines Zahlliterals in Programmanweisungen ergeben sich mehrere Probleme. Ein Problem besteht darin, dass es sehr schwierig wird nach den Zahlen zu suchen, denn angenommen man würde in der Klasse nach der Länge der maskierten Headerelemente suchen würde, müsste man nach der Zahl 4 suchen, wodurch sich aber sehr viele Ergebnisse ergeben würden. Weiters werden sich beim Ändern der Länge der maskierten Headerelemente Probleme geben, da diese Zahl eventuell auch noch an einer anderen Stelle verwendet wird und darauf vergessen werden kann diese auch zu Ändern. Der Lösungsvorschlag, welchen CCD hier bietet ist das Extrahieren der Zahlen in eine eigene Konstante. Für das Beispiel in Listing 3.26 würde sich der in Listing 3.27 dargestellte Code ergeben.

```
1         private const int _MASKED_HEADER_LENGTH = 4;
2
3         public FrameHeader(bool final, int opCode, bool masked,
      int maskKey, long dataLength){
```

```
4             ...
5             if (masked){
6                 headerLength += _MASKED_HEADER_LENGTH;
7             }
8             ....
9         }
```

Listing 3.27: Magic number

Mit dem in Listing 3.27 dargestellten Code ist es jetzt einerseits leicht möglich, nach der maskierten Header Länge zu suchen und den Wert für diese auch zu ändern. Ein weiterer Vorteil der sich aus dem Extrahieren ergibt ist die Tatsache, dass es für Leser des Codes sehr leicht wird zu verstehen, was zu der eigentlichen Header Länge addiert wird.

Kapitel 4

Schlussbemerkungen

In diesem Abschnitt möchte ich dem Leser noch einen persönliche Meinung, sowie einen Einblick in den Prozess CCD in einem Unternehmen einzuführen geben.

4.1 Fazit zu CCD

Grundsätzlich bin ich der Meinung, dass CCD einen wirklichen Mehrwert für alle Stakeholder einer Software bzw. des Unternehmens welches die Software entwickelt bietet. Durch das Achtgeben auf die Regeln, welche Robert C. Martin in seinem Buch sehr schön definiert ist es auch für Anfänger in der Programmierung sehr leicht möglich besseren Code zu schreiben. Wie auch Robert C. Martin bin ich der Meinung, dass man nicht von heute auf morgen Clean Code programmieren kann und dass selbst erfahrene ProgrammiererInnen es nicht schaffen gleich auf Anhieb die richtige Lösung, welche gut verständlich, sauber und leicht zu warten ist zu finden. Daher ist es meiner Meinung nach umso wichtiger, dass man die *Pfadfinderregel* beachtet und Code denn man liest immer ein bisschen verbessert. Wie bereits erwähnt bringt CCD, wenn es bewusst betrieben wird, einen Mehrwert für alle Stakeholder.

- Kunde: Wenn der Kunde neue Features benötigt können diese schneller umgesetzt werden
- Unternehmen: Durch die saubere Gestaltung des Codes Kosten Änderungen weniger
- ProjektleiterIn: Die Zufriedenheit der Mitglieder eines Entwicklungsteams ist sehr viel höher, wenn der Code leicht lesbar ist
- ProgrammiererIn: Durch bessern Code können Änderungen leichter vorgenommen werden und es ist meist nicht nötig stundenlang zu Debuggen bis herausgefunden werden kann welchen Zweck die betrachtete Funktion erfüllt

4.2 Persönliche Erfahrungen mit CCD

Ich persönlich praktiziere seit mehreren Jahren CCD und wir haben auch in der Firma einige Schulungen absolviert um unseren Codierungsstil zu verbessern. Dadurch haben sich für uns die von Robert C. Martin prognostizierten Vorteile ergeben. Wir können schneller und leichter Fehler beheben, neue Features hinzufügen und der Code ist auch noch nach Wochen und Monaten gut lesbar. Dabei greifen wir intern vor allem auf das abhalten von Code Reviews zurück, bei denen wir gegenseitig die geschriebenen Codeabschnitte betrachten und verbessern. Dabei hat sich für uns eine Kultur eingestellt, dass wir in allen Projekten einen durchgängig sauberen Codierungsstil haben. Probleme gab es beim Einführen von CCD, da das Management nicht wirklich überzeugt davon war, dass wir Vorteile daraus haben den Code sauberer zu gestalten, da davon ausgegangen wurde, dass die Implementierung dadurch viel länger benötigen wird. Es wurde dann schrittweise vorgegangen und durch den Erfolg, welchen wir mit den ergriffenen Maßnahmen hatten wurde es uns ermöglicht noch weitere Maßnahmen einzuführen. Da die von uns entwickelte Software bereits mehrere Jahre alt ist, müssen auch zahlreiche Komponenten neu geschrieben werden. Vor allem bei solchen Refakturierungen fallen die Schwierigkeiten auf, die auftreten, wenn nicht darauf geachtet wurde Clean Code zu schreiben. Variablennamen sind teilweise sehr schlecht gewählt und es kann daher nicht genau gesagt werden wofür diese steht, wodurch wiederum ein längeres Einlesen in den Code und im schlimmsten Fall sogar ein Debuggen notwendig wird. Durch eine sukzessive Neuimplementierung dieser Komponenten unter Beachtung der CCD Kriterien können wir auch einen Benefit für den Kunden liefern, da wir leichter Erweiterungen implementieren können, ohne dass wir das System beeinflussen. Ein sehr wichtiger Punkt in dieser Hinsicht, denn wir auch bei der Entwicklung verbessern müssen ist die Testabdeckung.

Quellenverzeichnis

Literatur

[1] Kent Beck und Cynthia Andres. *Extreme programming explained: Embrace change*. Addison-Wesley, 2005 (siehe S. 13).

[2] Koschke R. ; Antoniol G. ; Krinke J. ; Merlo E. „Comparison and Evaluation of Clone Detection Tools“. In: 2007 (siehe S. 18).

[3] Michael Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2013 (siehe S. 2, 5, 23).

[4] Martin Robert C.; Riehle Dirk; Buschmann Frank. *Pattern Languages of Program Design 3*. Addison-Wesley Professional, 1997 (siehe S. 29).

[5] Andrew Hunt und David Thomas. *The Pragmatic Programmer*. Addison Wesley, 1999 (siehe S. 18).

[6] Ron Jeffries. *Extreme Programming Adventures in C#*. Microsoft Press, 2004 (siehe S. 6).

[7] Ron Jeffries. *Extreme Programming Installed (XP)*. Addison Wesley, 2000 (siehe S. 6).

[8] Gamma Erich; Helm Richard; Johnson Ralph; Vlissides John. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995 (siehe S. 13).

[9] Booch Grady; Maksimchuk Robert; Engle Michael; Young Bobbi; Conallen Jim; Houston Kelli. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional; 2007 (siehe S. 5).

[10] Meir M Lehman. „Programs, Life Cycles, and Laws of Software Evolution“. In: 1980 (siehe S. 1).

[11] Robert C. Martin. *Agile Software Development. Principles, Patterns, and Practices*. Prentice Hall Computer, 2002 (siehe S. 14).

[12] Robert C. Martin. *Clean code: A handbook of agile software craftsmanship*. Prentice Hall, 2008 (siehe S. vi, vii, 2, 4–8, 10, 25).

[13] Roy Osherove. *The Art of Unit Testing*. Manning, 2013 (siehe S. 22).

[14] Lin Tan; Ding Yuan; Gopal Krishna; Yuanyuan Zhou. „iComment: Bugs or Bad Comments?“ In: 2007 (siehe S. 36).

Online-Quellen

[15] *Framework for integrated test*. url: http://fit.c2.com/ (siehe S. 7).

[16] *JDepend*. url: http://clarkware.com/software/JDepend.html (siehe S. 13).

[17] *JSHint*. url: http://jshint.com/ (siehe S. 13).

[18] *JSLint*. url: http://www.jslint.com/ (siehe S. 13).

[19] Robert C. Martin. *The Principles of OOD*. url: http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod (siehe S. 14).

[20] Microsoft. *C# Coding Conventions*. url: https://msdn.microsoft.com/en-us/library/ff926074.aspx (siehe S. 12).

[21] *NDepend*. url: http://c2.com/cgi/wiki (siehe S. 13).

[22] *Pair Programming*. url: http://www.extremeprogramming.org/rules/pair.html (siehe S. 13).

[23] *Wiki*. url: http://c2.com/cgi/wiki (siehe S. 7).

[24] George Wilson James; Kelling. *Broken Windows: The police and neighborhood safety*. url: http://manhattan-institute.org/pdf/_atlantic_monthly-broken_windows.pdf (siehe S. 2).