Clean Code Development - Analyse von Opensource Frameworks auf die CCD Kriterien

Stefan Kert

BACHELORARBEIT

Nr. S1310307019

eingereicht am

Fachhochschul-Bachelorstudiengang

Software Engneering

in Hagenberg

im Juli 2016

Diese Arbeit entstand im Rahmen des Gegenstands

Gegenstand??

im

Semester??

Betreuer:

Josef Pichler, Dr.

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

Stefan Kert

Inhaltsverzeichnis

Vorwort

Dies ist **Version 2015/09/19** der LaTeX-Dokumentenvorlage für verschiedene Abschlussarbeiten an der Fakultät für Informatik, Kommunikation und Medien der FH Oberösterreich in Hagenberg, die mittlerweile auch an anderen Hochschulen im In- und Ausland gerne verwendet wird.

Das Dokument entstand ursprünglich auf Anfragen von Studierenden, nachdem im Studienjahr 2000/01 erstmals ein offizieller LaTeX-Grundkurs im Studiengang Medientechnik und -design an der FH Hagenberg angeboten wurde. Eigentlich war die Idee, die bereits bestehende *Word*-Vorlage für Diplomarbeiten „einfach“ in LaTeX zu übersetzen und dazu eventuell einige spezielle Ergänzungen einzubauen. Das erwies sich rasch als wenig zielführend, da LaTeX, vor allem was den Umgang mit Literatur und Grafiken anbelangt, doch eine wesentlich andere Arbeitsweise verlangt. Das Ergebnis ist – von Grund auf neu geschrieben und wesentlich umfangreicher als das vorherige Dokument – letztendlich eine Anleitung für das Schreiben mit LaTeX, ergänzt mit einigen speziellen (mittlerweile entfernten) Hinweisen für *Word*-Benutzer. Technische Details zur aktuellen Version finden sich in Anhang **??**.

Während dieses Dokument anfangs ausschließlich für die Erstellung von Diplomarbeiten gedacht war, sind nunmehr auch *Masterarbeiten*, *Bachelorarbeiten* und *Praktikumsberichte* abgedeckt, wobei die Unterschiede bewusst gering gehalten wurden.

Bei der Zusammenstellung dieser Vorlage wurde versucht, mit der Basisfunktionalität von LaTeX das Auslangen zu finden und – soweit möglich – auf zusätzliche Pakete zu verzichten. Das ist nur zum Teil gelungen; tatsächlich ist eine Reihe von ergänzenden „Paketen“ notwendig, wobei jedoch nur auf gängige Erweiterungen zurückgegriffen wurde. Selbstverständlich gibt es darüber hinaus eine Vielzahl weiterer Pakete, die für weitere Verbesserungen und Finessen nützlich sein können. Damit kann sich aber jeder selbst beschäftigen, sobald das notwendige Selbstvertrauen und genügend Zeit zum Experimentieren vorhanden sind. Eine Vielzahl von Details und Tricks sind zwar in diesem Dokument nicht explizit angeführt, können aber im zugehörigen Quelltext jederzeit ausgeforscht werden.

Zahlreiche KollegInnen haben durch sorgfältiges Korrekturlesen und konstruktive Verbesserungsvorschläge wertvolle Unterstützung geliefert. Speziell

bedanken möchte ich mich bei Heinz Dobler für die konsequente Verbesserung meines „Computer Slangs“, bei Elisabeth Mitterbauer für das bewährte orthographische Auge und bei Wolfgang Hochleitner für die Tests unter Mac OS.

Die Verwendung dieser Vorlage ist jedermann freigestellt und an keinerlei Erwähnung gebunden. Allerdings – wer sie als Grundlage seiner eigenen Arbeit verwenden möchte, sollte nicht einfach („ung'schaut“) darauf los werken, sondern zumindest die wichtigsten Teile des Dokuments *lesen* und nach Möglichkeit auch beherzigen. Die Erfahrung zeigt, dass dies die Qualität der Ergebnisse deutlich zu steigern vermag.

Der Quelltext zu diesem Dokument sowie das zugehörige LaTeX-Paket sind in der jeweils aktuellen Version online verfügbar unter

https://sourceforge.net/projects/hgbthesis/.

Trotz großer Mühe enthält dieses Dokument zweifellos Fehler und Unzulänglichkeiten – Kommentare, Verbesserungsvorschläge und passende Ergänzungen sind daher stets willkommen, am einfachsten per E-Mail direkt an mich:

Dr. Wilhelm Burger, Department für Digitale Medien,
Fachhochschule Oberösterreich, Campus Hagenberg (Österreich)
wilhelm.burger@fh-hagenberg.at

Übrigens, hier im Vorwort (das bei Diplom- und Masterarbeiten üblich, bei Bachelorarbeiten aber entbehrlich ist) kann kurz auf die Entstehung des Dokuments eingegangen werden. Hier ist auch der Platz für allfällige Danksagungen (z. B. an den Betreuer, den Begutachter, die Familie, den Hund, ...), Widmungen und philosophische Anmerkungen. Das sollte allerdings auch nicht übertrieben werden und sich auf einen Umfang von maximal zwei Seiten beschränken.

Kurzfassung

An dieser Stelle steht eine Zusammenfassung der Arbeit, Umfang max. 1 Seite. Im Unterschied zu anderen Kapiteln ist die Kurzfassung (und das Abstract) üblicherweise nicht in Abschnitte und Unterabschnitte gegliedert. Auch Fußnoten sind hier falsch am Platz.

Kurzfassungen werden übrigens häufig – zusammen mit Autor und Titel der Arbeit – in Literaturdatenbanken aufgenommen. Es ist daher darauf zu achten, dass die Information in der Kurzfassung für sich *allein* (d. h. ohne weitere Teile der Arbeit) zusammenhängend und abgeschlossen ist. Insbesondere werden an dieser Stelle (wie u. a. auch im *Titel* der Arbeit und im *Abstract*) normalerweise *keine Literaturverweise* verwendet! Falls unbedingt solche benötigt werden – etwa weil die Arbeit eine Weiterentwicklung einer bestimmten, früheren Arbeit darstellt –, dann sind *vollständige* Quellenangaben in der Kurzfassung selbst notwendig, z. B. [Zobel J.: *Writing for Computer Science – The Art of Effective Communication*. Springer-Verlag, Singapur, 1997].

Weiters sollte daran gedacht werden, dass bei der Aufnahme in Datenbanken Sonderzeichen oder etwa Aufzählungen mit „Knödellisten“ in der Regel verloren gehen. Dasselbe gilt natürlich auch für das *Abstract*.

Inhaltlich sollte die Kurzfassung *keine* Auflistung der einzelnen Kapitel sein (dafür ist das Einleitungskapitel vorgesehen), sondern dem Leser einen kompakten, inhaltlichen Überblick über die gesamte Arbeit verschaffen. Der hier verwendete Aufbau ist daher zwangsläufig anders als der in der Einleitung.

Abstract

This should be a 1-page (maximum) summary of your work in English.

Im englischen Abstract sollte inhaltlich das Gleiche stehen wie in der deutschen Kurzfassung. Versuchen Sie daher, die Kurzfassung präzise umzusetzen, ohne aber dabei Wort für Wort zu übersetzen. Beachten Sie bei der Übersetzung, dass gewisse Redewendungen aus dem Deutschen im Englischen kein Pendant haben oder völlig anders formuliert werden müssen und dass die Satzstellung im Englischen sich (bekanntlich) vom Deutschen stark unterscheidet (mehr dazu in Abschn. **??**). Es empfiehlt sich übrigens – auch bei höchstem Vertrauen in die persönlichen Englischkenntnisse – eine kundige Person für das „proof reading“ zu engagieren.

Die richtige Übersetzung für „Diplomarbeit“ ist übrigens schlicht *thesis*, allenfalls „diploma thesis“ oder „Master's thesis“, auf keinen Fall aber „diploma work“ oder gar „dissertation“. Für „Bachelorarbeit“ ist wohl „Bachelor thesis“ die passende Übersetzung.

Übrigens sollte für diesen Abschnitt die *Spracheinstellung* in LaTeX von Deutsch auf Englisch umgeschaltet werden, um die richtige Form der Silbentrennung zu erhalten, die richtigen Anführungszeichen müssen allerdings selbst gesetzt werden (s. dazu die Abschnitte **??** und **??**).

Kapitel 1

Einleitung

1.1 Überblick

In der modernen Softwareentwicklung geht es in erster Linie um das Umsetzen funktionaler sowie nicht funktionaler Anforderungen und um die Behebung von Fehlern welche bei der Umsetzung dieser Anforderungen häufig auftreten. Ein wichtiger Punkt dabei ist, den Code so zu gestalten, dass er nicht nur vom Programmierer, der ihn geschrieben hat, gelesen werden kann, sondern auch von anderen Programmieren und dies auch möglichst noch nach mehreren Monaten. Um dies zu erreichen, müssen gewisse Grundsätze angewendet werden. In den letzten Jahren hat sich zu diesem Thema eine Strömung ergeben welche sich als Clean Code Development (im folgenden nur noch als CCD abgekürzt) bezeichnet. Geprägt wurde diese Bewegung in erster Linie von Robert. C. Martin und seinem Bestseller Clean Coder.

1.2 Gliederung

Die Arbeit ist in drei größere Teile gegliedert. Der erste Teil beschäftigt sich in erster Linie mit der Erklärung des CCD Begriffes und des .....

Im zweiten Teil werden die einzelnen Regeln und Grundsätze anhand von Beispielen erläutert wie sie den Code verbessern bzw. welche Änderungen an Codestücken vorgenommen werden müssen, um diese Regeln und Grundsätze zu entsprechen.

Der dritte Teil beschäftigt sich mit der Analyse einiger Opensource Frameworks hinsichtlich der CCD Kriterien und deren Erfüllung. Es sollten dabei Codestücke gezeigt werden, welche die Kriterien erfüllen, sowie Codestücke die diese nicht erfüllen und dann abgeändert werden.

Kapitel 2

Einführung in CCD

2.1 Allgemeines zum CCD

Wie bereits in im Überblick erläutert, beschäftigt sich CCD in erster Linie mit Praktiken und Strategien, wie man Code so gestalten kann, dass er möglichst leicht zu Lesen, zu Warten und Anzupassen ist. Dabei sind diese Praktiken unabhängig von der Programmiersprache. Auf Grund der Tatsache, dass vor allem ältere Programmiersprachen gewisse Restriktionen wie zum Beispiel eine maximal Länge von Variablen aufweisen sind jedoch nicht alle Praktiken zur Gänze umsetzbar, jedoch kann sich trotzdem daran orientiert werden. Es ist weiterhin nicht relevant welches Programmierparadigma der Programmiersprache zu Grunde liegt. Die meisten Regeln gelten gleichermaßen für objektorientierte, funktionale und imperative Programmiersprachen. Einige Regeln sind auf Grund der unterschiedlichen Eigenschaften der einzelnen Paradigmen nicht zur Gänze umsetzbar, jedoch gibt es auch Abwandlungen der Regeln welche in den unterschiedlichen Paradigmen gelten.

2.2 Warum CCD?

Die Frage die sich im Zusammenhang mit CCD immer wieder ergibt ist: Warum CCD? Erst in den letzten Jahren hat sich CCD zu einem wirklich wichtigen Teil der Softwareentwicklung entwickelt. Bei einer Suche nach C̈lean Code Developmentërgeben sich 39.600.000 Ergebnisse (Stand November 2015). Auch bei Inseraten für Jobangebote ist Clean Code Development ein sehr wichtiger Punkt: Bei einer Suche nach dem selben Begriff auf stack overflow careers (http://careers.stackoverflow.com) ergeben sich hier 973 Treffer (Stand November 2015).

Diese Zahlen geben Aufschluss darüber, dass ein sehr großes Interesse an CCD besteht und dieses auch in sehr vielen Bereichen Anwendung findet. Dies liegt in erster Linie an einer der

Kapitel 3

Grundlagen von CCD

In diesem Abschnitt sollten die Grundlagen für die unterschiedlichen Praktiken des CCD erläutert werden. Um den Rahmen dieser Arbeit nicht zu sprengen, werden hier nur die zentralsten Praktiken erläutert. Auf Grund der Tatsache, dass sich der dritte Teil dieser Arbeit in erster Linie mit dem Refactoring von Open Source Frameworks hinsichtlich der CCD Praktiken beschäftigt, werden die Beispiele in diesem Abschnitt kurz gehalten und sollten nur als kurze Einführung dienen.

3.1 Namensgebung

Eine der häufigsten Aufgaben eines Programmierers ist das vergeben von Namen. Variablen, Methoden, Klassen, Module, Bibliotheken, Programme. Alle diese Teile brauchen Namen und diese Namen sollten möglichst aussagekräftig sein. Eine Bibliothek die zum Loggen von Informationen wird, sollte im besten Fall das Wort Logging beinhalten, damit sich der Verwender dieser Bibliothek beim ersten Blick darauf schon eine Ahnung hat was sich darin befinden wird und er sich schon einen ersten Eindruck machen kann, was ihn in dieser Bibliothek erwartet.

Im CCD gibt es für die Namensgebung eine grundlegende Regel: Der Name sollte der Größe des Gültigkeitsbereich entsprechen. Wenn man z.B. eine Methode, die in einem globalen Kontext gültig ist, sollte einen dementsprechend deskriptiven Namen aufweisen. Zu beachten ist natürlich auch der Kontext in dem sich diese befinden, wofür die Methode verwendet werden sollte. Hier nun ein kleines Beispiel. In einem fiktiven Projekt gibt es folgende zentrale Klasse mit einer globalen Property.

```
1 public static class Globals {
2     public static string Information {get; set;}
```

```
3 }
```

Listing 3.1: Beispiele für die Verwendung von *GetByPredicate*

Trotz der Kürze dieser Klasse ergibt sich bereits ein Problem. Aus dem Namen der Property kann man die Aufgabe dieser in keinster Weise ablesen. Es wird irgendeine Information gespeichert, doch welche Information genau wird gespeichert und wozu wird diese verwendet? Ein Blick auf die Verwendungen dieser Variable, führt zu der Erkenntnis, dass die Information die Version des Programmes darstellt. Der Programmierer hätte diese Verwirrung vermeiden können indem er die Namen anders gewählt hätte. Hier ein Beispiel zu einer besseren Variante:

```
1 public static class ApplicationInformation {
2     public static string Version {get; set;}
3 }
```

Listing 3.2: Beispiele für die Verwendung von *GetByPredicate*

Durch eine Umbenennung der Klasse wird es auch um einiges klarer, auf welchen Teil des Programmes sich die Version bezieht. Man könnte jetzt noch den Datentyp Version implementieren, welcher die z.B. die Major-, Minor- und Patchnummer beinhaltet:

```
1 public static class ApplicationInformation {
2     public static Version Version {get; set;}
3 }
4
5 public class Version {
6   public int Major { get; set;}
7   public int Minor { get; set;}
8   public int Patch { get; set;}
9 }
```

Listing 3.3: Beispiele für die Verwendung von *GetByPredicate*

Bei diesem letzten Beispiel kann man ohne viele Schritte erkennen, welche Aufgabe dieser Variable zu Teil wird. Durch die Erweiterung um einen eigenen Datentyp ist es jetzt auch leichter weitere Dinge zur Version hinzuzufügen. Angenommen eine zukünftige Anforderung für das Programm verlangt es, bei der Versionsnummer immer die Buildnummer mit anzugeben. Mit der letzten Variante ist dies kein Problem und es kann auch explizit darauf zugegriffen werden, ohne dass man wie bei der ersten Variante die Zeichenkette aufteilen muss.

3.2 Methoden

Eine weitere wichtige grundsätzliche Regel im CCD ist es Funktionen und Methoden möglichst klein zu halten. Es sollte darauf geachtet werden, dass

diese dabei das sogenannte Single responsibility principle (kurz SRP) befolgen. Dieses Prinzip besagt, dass alle Aspekte des Softwareentwurfs so geplant werden sollten, sodass sie nur eine Aufgabe erfüllen. Wenn dieses Prinzip für Funktionen und Methoden befolgt wird, sollten diese automatisch sehr klein werden. Wenn eine Methode oder eine Funktion sehr lange ist, deutet dies meist auf eine Verletzung dieses Prinzips hin. Ein weiterer guter Indikator ist die Schwierigkeit mit der man eine Methode oder Funktion testen kann.

3.3 Klassen

Kapitel 4

Analyse von Open-Source Frameworks

In diesem Abschnitt erfolgt die Analyse der unterschiedlichen Opensource Frameworks hinsichtlich der CCD Kriterien. Wie bereits erläutert, wird es bei dieser Analyse in erster Linie um beispielhafte Verbesserungen gehen, die an diesen Frameworks vorgenommen werden können. Dabei wird jedem der analysierten Frameworks ein kleiner Abschnitt gewidmet in dem mehrere Aspekte analysiert werden. Es wird dabei auch die Struktur des Projektes im Allgemeinen betrachtet werden und am Schluss des Abschnitts wird ein kurzes Fazit Aufschluss darüber geben, welche Teile des Frameworks besonders positiv, bzw. besonders negativ herausgestochen sind.

Es sollten folgende Frameworks näher betrachtet werden:

- Log4net (.NET)
- Hibernate (Java)
- Roslyn (.NET)

Nach der ersten Analyse und einer Beschreibung der Problematik des aktuellen Codestückes wird eine verbesserte Variante des Codes erläutert.

4.1 Log4net

4.1.1 Allgemeines

Log4net ist wohl eines der bekanntesten Logging Frameworks für .NET und wird als solches von zahlreichen Programmierern verwendet. Es ist eine Portierung des im Java Bereich bekannten Log4j und ist mittlerweile über 10 Jahre alt. Das Framwork an sich steht unter der Apache License 2.0 bereit und ist dabei ein Teil der Apache Logging Services und somit auch ein Teil der Apache Software Foundation. Das Repository auf welches sich im Folgenden bezogen wird lässt sich auf GitHub unter https://github.com/apache/log4net finden.

4.1.2 Kommentare

- Betreffende Klasse: *ConfigurationChangedEventArgs*
- Betreffender Namespace: *log4net.Repository*

Eines der wohl am meisten diskutierten Themen hinsichtlich CCD ist das entfernen von nicht benötigten Kommentaren. Dabei gehen die Meinungen sehr stark außeinander, was jetzt als, nicht mehr benötigter, oder als wichtiger Kommentar durchgeht. Robert C. Martin schreibt ist in dieser Hinsicht sehr pragmatisch. Alle Kommentare die nicht unbedingt nötig sind werden gelöscht. Zu den unbedingt notwendigen gehören für ihn dabei rechtliche Kommentare die auf eine Lizenze verweisen, oder auch Kommentare die erläutern, warum etwas genau so implementiert wurde. Zu überflüssigen Kommentaren zählt er dabei vor allem redundanten Kommentare. Auch in Log4net lassen sich solche Kommentare finden. Wenn man sich den Konstruktor aus der Klasse ConfigurationChangedEventArgs ansehen kann man feststellen, dass die Kommentare im Prinzip redundant sind.

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

Listing 4.1: Beispiele für überflüssige Kommentare

Er bringt für den Programmierer keinen wirklichen Mehrwert. Was genau in der übergebenen *Collection* übertragen wird, wird auch durch den redundanten Kommentar nicht näher erläutert. Da der <summary> Abschnitt sowieso leer ist, könnte man diesen getrost entfernen, da er nur den Quelltext aufbläht.

4.1.3 Namensgebung - Klassen

- Betreffende Klasse: *LogLog*
- Betreffender Namespace: *log4net.Util*

Bei der Namensgebung kommt es immer wieder zu Unklarheiten und Problemen. Oft werden Kommentare geschrieben, um dem Leser zu vermitteln, wofür genau diese Methode, Klasse oder auch Variable verwendet wird. Dabei sind diese Probleme umso größer, je größer der Kontext ist in dem diese Namen gültig sind. Bei Klassen zum Beispiel , kann dies zu einem sehr großen Problem werden. In Log4net gibt es die Klasse *LogLog*. Dieser Name ist nicht nur unglücklich gewählt, er lässt auch eine gewisse Redundanz vermuten. Die Aufgabe dieser Klasse lässt sich nach einem näheren Betrachten dieser ergründen. Sie besteht darin das interne Logging im Log4net Framework zu realisieren. Im Kommentar wird dem Leser erklärt, dass es für Log4net Komponenten nicht möglich ist Log4net Aufrufe zu tätigen. Daher auch der Name. Die Klasse stellt einen Log für den Log dar. Ein Blick in den Quelltext von Log4j lässt darauf schließen, dass diese Klasse einfach übernommen wurde und für .NET angepasst wurde. Die Kommentare sind sehr ähnlich und auch der Aufbau ist sehr ähnlich. Wahrscheinlich wäre durch ein Refactoring des Frameworks und der Verwendungen dieser Klasse eine bessere Variante möglich, jedoch sollte sich hier nur auf kleine Änderungen konzentriert werden. Daher sollte an dieser Stelle nur ein besserer Name für dies Klasse gefunden werden, welcher auch Aufschluss darüber gibt, dass diese Klasse nur zum internen Logging in Log4net verwendet wird. *AssemblyInternalLogging* wäre ein besser Name für diese Klasse. Ein weiterer Punkt der hier anzumerken ist: Die Klasse ist als *public* gekennzeichnet und so ist sie für alle Verwender der Bibliothek zugänglich. Aus dem Kommentar lässt sich darauf schließen, dass dies nicht gewünscht ist, wodurch eine Änderung der Sichtbarkeit in *internal* vernünftig wäre, da es dadurch nur im aktuellen Assembly zugänglich gemacht wird.

4.1.4 Gute Projektstruktur

Beim ersten Blick auf das GitHub Repository finden man eine sehr übersichtliche, gut strukturierte Projektanordnung. Die Gliederung des Projektes in die einzelnen Abschnitte ist dabei sehr gelungen und es ermöglicht ein leichtes Auffinden der gesuchten Aspekte der Bibliothek. Besonders die sehr gering gehaltene Anzahl an Elementen im Root Namespace sollte hier hervorgehoben werden. Man findet dort nur die wichtigsten Klassen die auch für den Verwender dieser Bibliothek von enormer Wichtigkeit sind. Dies sind vor allem die Klasse *LogManager* und das Interface *ILog*.

4.1.5 Fazit

Im Großen und Ganzen lässt sich sagen, dass bei der Entwicklung von Log4net sehr darauf geachtet wurde, die Codebasis so zu gestalten, dass man sich sehr leicht einarbeiten kann. Die Bibliothek ist sehr logisch aufgebaut und die Klassen bzw. Namespacenamen sind so gewählt, dass man sich gut zurecht findet. Es sollte also für keinen Programmierer ein gröberes Problem sein, sich in diese Bibliothek einzuarbeiten. Einige Dinge sind auf Grund des Alters von Log4net mit den modernen IDEs nicht mehr nötig, wie z.B. die Präfixe für Variablen, jedoch wird dies meist über Coding Conventions festgelegt und sollte daher auch so im ganzen Projekt durchgezogen werden. Bei zahlreichen Klassen in Log4net merkt man außerdem die Verwandtschaft zu Log4j. Einige Implementierungen oder Strukturierungen würde man in .net anders vornehmen, jedoch kann dies auch dem wie bereits angemerkten Alter der Bibliothek geschuldet sein.

4.2 Hibernate

4.2.1 Allgemeines

Die Bibliothek Hibernate ist der wohl bekannteste OR-Mapper für Java. Mit Hibernate ist es möglich ein objektorientiertes Datenmodell auf ein relationales Datenbankmodell zu mappen. Die Bibliothek ist mittlerweile über 10 Jahre alt und wurde unter anderem nach .NET portiert und heißt dort NHibernate. Entwickelt wird sie von Red Hat und steht unter der GNU LGPL zur verfügung. Das Repository lässt sich auf GitHub unter https://github.com/hibernate/hibernate-orm finden.

4.2.2 Auskommentierter Code

- Betreffende Klasse: *BindHelper*
- Betreffendes Paket: *org.hibernate.cfg*

Ein sehr häufig gesehenes Missachten der CCD Kriterien ist das Auskommentieren von Code der nicht mehr benötigt wird. Dies war vor vielen Jahren noch nötig, da es nicht die Versionsverwaltungssysteme hatte die es heute gibt. Sei es SVN, Git oder irgendein beliebiges Versionsverwaltungssystem. All diese Programme bieten zahlreiche Features zum wiederherstellen, oder zum Auffinden alter Versionen, es ist daher nicht notwendig die Codedateien als History zu verwenden. Code der nicht mehr benötigt wird sollte daher nicht auskommentiert werden, sondern einfach gelöscht werden und mit einer vernünftigen Commit Message versehen werden. Im Hibernate Framework ist eine Klasse welche dieses Kriterium missachtet die BindHelper Klasse. Hier gibt es zahlreiche Codeabschnitte die auskommentiert sind und eigentlich gelöscht werden könnten oder zumindest geändert gehören. Eine sehr problematische Stelle befindet sich in dieser Klasse in Zeile 421. Dort gibt es folgenden Codeausschnitt:

```
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 4.2: Beispiele für die Verwendung von *GetByPredicate*

Der Kommentar deutet darauf hin, dass es in diesem Codeabschnitt einen Fehler gibt der behoben werden muss. Anstatt diesen Fehler zu beheben wurde der Code einfach auskommentiert und nach ein einigen Wochen weiß niemand mehr, dass es diesen Fehler gibt, da er auch nicht mehr auftreten wird. Hier sollte entweder in einem Issue Tracking System genau mitdokumentiert werden, wo der Fehler auftritt und welche Möglichkeiten es gibt diesen zu beheben.

4.2.3 Unleserliche boolsche Ausdrücke

- Betreffende Klasse: *AbstractPropertyHolder*
- Betreffendes Paket: *org.hibernate.cfg*

Ein weiteres sehr oft auftretendes Problem sind sehr komplexe boolsche Ausdrücke. Oft wird auf Grund ihrer Komplexität ein Kommentar geschrieben, welcher Aufschluss darüber geben sollte, was genau gewünscht ist. Jedoch ist die Komplexität meist nicht das einzige Problem. Hierzu ist die *AbstractPropertyHolder* Klasse ein sehr gutes Beispiel.

```
1  @Override
2  public JoinColumn[] getOverriddenJoinColumn(String
     propertyName) {
3    JoinColumn[] result = getExactOverriddenJoinColumn(
     propertyName );
4    if ( result == null && propertyName.contains(
     ".collection&&element." ) ) {
5      //support for non map collections where no prefix is
     needed
6      //TODO cache the underlying regexp
7      result = getExactOverriddenJoinColumn(
     propertyName.replace( ".collection&&element.", "."  ) );
8    }
9    return result;
10 }
```

Listing 4.3: Komplexe boolsche Ausdrücke 1 Zeile 255 - 264

Ein paar Zeilen weiter in derselben Klasse findet sich folgende Methode:

```
1 public JoinTable getOverriddenJoinTable(String
    propertyName) {
2  JoinTable result =
    getExactOverriddenJoinTable(propertyName);
3  if(result == null
4    && propertyName.contains(".collection&&element.")){
5    //support for non map collections where no prefix is
    needed
```

```
6     //TODO cache the underlying regexp
7     result = getExactOverriddenJoinTable(
      propertyName.replace( ".collection&&element.", "."  ) );
8   }
9   return result;
10 }
```

Listing 4.4: Komplexe boolsche Ausdrücke 2 Zeile 305 - 313

Konkret geht es in diesem Beispiel um folgenden boolschen Ausdruck, welcher in dieser Klasse insgesamt drei mal verwendet wird.

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

Listing 4.5: Boolscher Audruck

Wie man anhand dieser beiden Methoden leicht erkennen kann, wird dieser Ausdruck in beiden Methoden verwendet. Das heißt, es handelt sich hier im Grunde genommen um Code Duplizierung, eines der problematischten Dinge in der Programmierung. Ein weiteres Problem ergibt sich hier, dies sollte aber in einem der folgenden Abschnitte über Magic String geklärt werden. Ein weiters Problem, welches sich hier wie bereits oben beschrieben ergibt, ist die Tatsache, dass dieser boolsche Kommentar ohne einen Kommentar oder ein genaues Lesen nicht sehr leicht zu verstehen ist. Man könnte hier beide Probleme beseitigen, in dem man aus dem boolschen Audruck eine eigene Methode extrahiert. Dabei wird hier nur der zweite Teil des boolschen Ausdrucks extrahiert, da der erste Teil mit dem Nullcheck spezifisch für die einzelnen Methoden verwendet werden muss. Ein Beispiel für diese Methode könnte folgendermaßen aussehen:

```
1 private bool
      isPlaceholderForCollectionAndElementInPropertyName(String
      propertyName) {
2   return propertyName.contains(".collection&&element.");
3 }
```

Listing 4.6: Boolscher Ausdruck neu

Durch ein Extrahieren der Überprüfung des Parameters *propertyName* kann für diese Überprüfung ein eigener Name gewählt werden, der klar machen sollte, welchen Zweck diese Überprüfung hat. Durch diese eigene Methode werden außerdem die Codeduplizierungen eliminiert und es bleibt nur noch die Sicherheitsüberprüfung auf null.

4.2.4 Fazit

Auch beim Hibernate Framework merkt man, dass es bereits seit einigen Jahren entwickelt wird. Im Grunde genommen wirkt es sehr ausgereift und auch sehr

durchdacht. Einige Dinge, vor allem hinsichtlich überflüssiger Kommentare und auskommentierter Codeabschnitte, aber auch Codeduplizierungen könnten vermieden werden, in dem man kleinere Refactorings vornimmt. Wie in erläutert könnten einige dieser Duplizierungen und Unklarheiten durch solche Refactorings behoben werden. Was besonders positiv auffällt ist, dass selbst die Testsuite sehr sauber gehalten wurde und das auch dort auf Clean Code Richtlinien geachtet wurde. Die Struktur ist für einen Programmierer der aus der .NET Welt kommt nicht sehr übersichtlich, dies liegt aber eher in den Unterschieden in der Projektstruktur zwischen .NET Projekten und Java Projekten.

4.3 Roslyn

4.3.1 Allgemeines

Der von Microsoft als Opensource veröffentlichte Compiler mit dem Codenamen Roslyn ist wie viele Teile des .NET Frameworks seit dem Jahr 2014 auf GitHub verfügbar. In diesem Abschnitt sollte der Quellcode näher betrachtet werden.

4.3.2 Kapseln von Errorhandling in eigene Methoden

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

Das behandeln von Fehlern und Ausnahmen ist ein sehr zentraler Aspekt eines jeden Frameworks. Eine richige und gut implementierte Strategie zur Fehlerbehandlung kann sehr viel Einfluss auf die Wartbarkeit eines Programmes haben. Oft gibt es dabei aber ein Problem. Die Fehlerbehandlung vermischt sich mit der Logik und trägt daher nicht sehr zur besseren Lesbarkeit dieser bei. Da es bei der Fehlerbehandlung um einen eigenen Aspekt geht könnte man sagen, man lagert die Fehlerbehandlung in eine eigene Methode aus, welche als Wrapper für die zu behandelte Methode fungiert. Hier ein Beispiel dazu:

```
1 private FileKey? GetUniqueFileKey(string filePath)
2 {
3   try
4   {
5     return FileKey.Create(filePath);
6   }
7   catch (Exception)
8   {
9     // There are several exceptions that can occur
10    // here: NotSupportedException or
11    // PathTooLongException
12    // for a bad path, UnauthorizedAccessException
13    // for access denied, etc. Rather than listing
14    // them all, just catch all exceptions.
15    return null;
16  }
17 }
```

Listing 4.7: Beispiele für getrennten Aspket der Fehlerbehandlung

Um sich auf den wesentlichen Punkt in diesem Abschnitt, die Fehlerbehandlung, zu konzentrieren, wird der redundante Kommentar und die Rückgabe eiens null-Wertes ignoriert. Dies sollte in einem anderen Abschnitt näher

behandelt werden. Was man aber an diesem Beispiel schön sehen kann, ist die Trennung der Aspekte. Es wird eine Methode zum Erstellen eines *FileKey* Objektes aufgerufen. Anstatt die Fehlerbehandlung in der *Create* Methode zu erledigen, wird sie außerhalb dieser Methode implementiert und führt daher zu keiner Vermischung der Aspekte. Natürlich ergibt sich hier ein weiteres Problem. Beim Aufruf der Methode *Create* kann auf die Fehlerbehandlung vergessen werden, was im schlimmsten Fall zu einer unbehandelten Ausnahme führen kann. Es wäre daher vernünftiger 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 folgender Codeänderung in der Klasse *FileKey* resultieren.

```
1 public static FileKey Create(string fullPath)
2 {
3   return new FileKey(fullPath,
      FileUtilities.GetFileTimeStamp(fullPath));
4 }
```

Listing 4.8: 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 Create(filePath);
11  }
12  catch (Exception)
13  {
14    return null;
15  }
16 }
```

Listing 4.9: 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, welcher den Wert null annehmen kann.

4.3.3 Rückgabe von Nullwerten

4.3.4 Fazit

Kapitel 5

Schlussbemerkungen[1]

An dieser Stelle sollte eine Zusammenfassung der Abschlussarbeit stehen, in der auch auf den Entstehungsprozess, persönliche Erfahrungen, Probleme bei der Durchführung, Verbesserungsmöglichkeiten, mögliche Erweiterungen usw. eingegangen werden kann. War das Thema richtig gewählt, was wurde konkret erreicht, welche Punkte blieben offen und wie könnte von hier aus weitergearbeitet werden?

5.1 Lesen und lesen lassen

Wenn die Arbeit fertig ist, sollten Sie diese zunächst selbst nochmals vollständig und sorgfältig durchlesen, auch wenn man vielleicht das mühsam entstandene Produkt längst nicht mehr sehen möchte. Zusätzlich ist sehr zu empfehlen, auch einer weiteren Person diese Arbeit anzutun – man wird erstaunt sein, wie viele Fehler man selbst überlesen hat.

5.2 Checkliste

Abschließend noch eine kurze Liste der wichtigsten Punkte, an denen erfahrungsgemäß die häufigsten Fehler auftreten (Tab. 5.1).

[1] Diese Anmerkung dient nur dazu, die (in seltenen Fällen sinnvolle) Verwendung von Fußnoten bei Überschriften zu demonstrieren.

Tabelle 5.1: Checkliste. Diese Punkte bilden auch die Grundlage der routinemäßigen Formbegutachtung in Hagenberg.

- ☐ **Titelseite:** Länge des Titels (Zeilenumbrüche), Name, Studiengang, Datum.
- ☐ **Erklärung:** vollständig Unterschrift.
- ☐ **Inhaltsverzeichnis:** balancierte Struktur, Tiefe, Länge der Überschriften.
- ☐ **Kurzfassung/Abstract:** präzise Zusammenfassung, passende Länge, gleiche Inhalte und Struktur.
- ☐ **Überschriften:** Länge, Stil, Aussagekraft.
- ☐ **Typographie:** sauberes Schriftbild, keine „manuellen" Abstände zwischen Absätzen oder Einrückungen, keine überlangen Zeilen, Hervorhebungen, Schriftgröße, Platzierung von Fußnoten.
- ☐ **Interpunktion:** Binde- und Gedankenstriche richtig gesetzt, Abstände nach Punkten (vor allem nach Abkürzungen).
- ☐ **Abbildungen:** Qualität der Grafiken und Bilder, Schriftgröße und -typ in Abbildungen, Platzierung von Abbildungen und Tabellen, Captions. Sind *alle* Abbildungen (und Tabellen) im Text referenziert?
- ☐ **Gleichungen/Formeln:** mathem. Elemente auch im Fließtext richtig gesetzt, explizite Gleichungen richtig verwendet, Verwendung von mathem. Symbolen.
- ☐ **Quellenangaben:** Zitate richtig referenziert, Seiten- oder Kapitelangaben.
- ☐ **Literaturverzeichnis:** mehrfach zitierte Quellen nur einmal angeführt, Art der Publikation muss in jedem Fall klar sein, konsistente Einträge, Online-Quellen (URLs) sauber angeführt.
- ☐ **Sonstiges:** ungültige Querverweise (**??**), Anhang, Papiergröße der PDF-Datei (A4 = 8.27×11.69 Zoll), Druckgröße und -qualität.

Quellenverzeichnis