Probleme in der Softwareentwicklung und wie Clean Code Development dabei unterstützen kann

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 Problemstellung

Eine sehr zentrale Eigenschaft von Software ist die Tatsache, dass immer wieder Änderungen stattfinden. Durch die Entwicklung, welche meist über lange Zeit von verschiedenen Programmierern oder Programmiererinnen vorgenommen wird, ergibt sich meist ein sehr unlesbarer, schwer zu wartender und schwer zu testender Code. Diese Probleme führen in weitere Folge zu Qualitätseinbußen, da durch die schlechte Testbarkeit schlichtweg auf wichtige Tests verzichtet wird, da die Zeit für die Implementierung dieser zu lange dauern würde. Eine weitere Folge der schlechten Lesbarkeit, ist die sich daraus ergebenden Schwierigkeiten beim Ändern des vorhandenen Codes. Dieser lässt sich nämlich nur noch unter sehr großem Aufwand und sehr großem Risiko, da er meist auch nicht durch Tests abgedeckt ist ändern, was wiederum dazu führt, dass weniger Änderungen vorgenommen werden. Im schlimmsten Fall führen diese Probleme schlussendlich zum Scheitern des Projektes und es muss eingestellt werden, da der Aufwand der nötig wäre neue Änderungen oder Fehlerbehebungen vorzunehmen zu groß wäre.

1.2 Motivation

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. Umso genauer man darauf achtet, den Code beim Schreiben lesbar zu gestalten, umso leichter wird es diesen auch noch nach einiger Zeit wieder zu Lesen. Bei schlecht geschrieben Code kann es sein, dass man bereits nach einigen Tagen nicht mehr genau weiß, was man damit bezwecken wollte.

1.3 Lösungsvorschlag

CCD bietet für diese Probleme eine Lösung, welche aber über mehrere Jahre hinweg in den Entwicklungsteams etabliert werden muss. Der wichtigsten Punkte in CCD sind die folgenden:

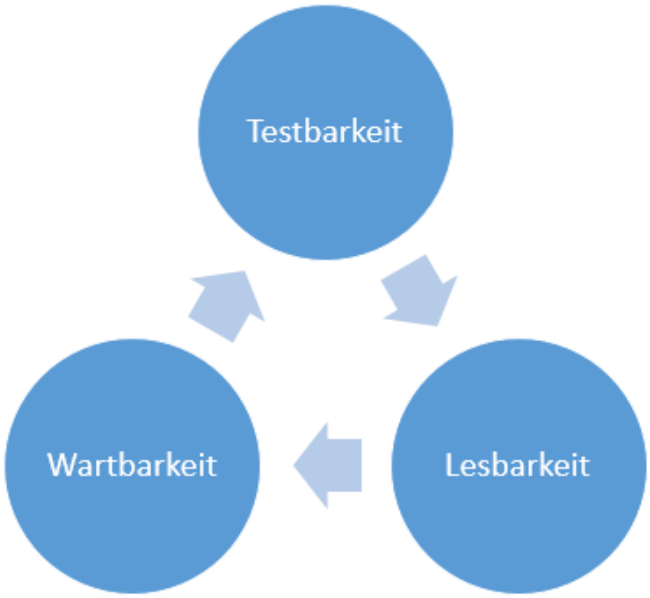

Abbildung 1.1: CCD Zyklus

In der Abbildung 1.1 kann man gut erkennen, dass diese drei Hauptpunkte voneinander abhängen. Durch die gute Lesbarkeit, wird die Wartbarkeit des Codes erhöht da sich selbst ein neuer Mitarbeiter schnell einlesen kann und gut erkennen kann welche Aufgabe der aktuell betrachtete Code hat. Durch die bessere Wartbarkeit ergibt sich im weiteren auch die besseren Testbarkeit, welche für die Qualität der Software von hoher Bedeutung ist, da nur für getesteten Code sichergestellt werden kann, dass dieser auch wirklich die gewünschte Aufgabe richtig erledigt. Diese verbesserte Testbarkeit führt schließlich zu

einer besseren Lesbarkeit, da die Test mit einer Dokumentation des Codes verglichen werden können. Sie zeigen welche Ausgabe erwartet werden kann und meist zeigen diese Tests auch Grenzfälle.

Kapitel 2

Clean Code Development Grundlagen

2.1 Was ist CCD?

Wie bereits 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.

2.2 Ziele von CCD

Die wichtigsten Ziele von CCD sind es, den Quellcode

2.3 Die Pfadfinderregel

Oft

2.4 Überprüfung der CCD Kriterien

Kapitel 3

Analyse der CCD Kriterien

In diesem Abschnitt erfolgt die Analyse einiger wichtiger CCD Kriterien, welche im Buch Clean Code von Robert C. Martin beschriebenen sind. Dabei dienen verschiedene Opensource Frameworks zur Veranschaulichung dieser. Die einzelnen Kriterien, welche zur Analyse ausgewählt wurden, sind einige der am häufigsten missachteten Regeln und diese werden daher genauer betrachtet.

Folgende Frameworks dienen der Veranschaulichung:

- Log4net (.NET)
- Hibernate (Java)
- Roslyn (.NET)
- Swift (C++)

3.1 Duplizierungen beim Entwickeln von Software

Eines der größten Probleme bei der Entwicklung von Software stellt die Duplizierung von Code dar. Dabei ist die Duplizierung oft nicht so offensichtlich wie es bei kopierten Methoden der Fall ist. Duplizierung kann genau so in einfachen boolschen Statements auftreten, oder auch durch logische Duplizierungen, wo der Quellcode zwar verschieden ist, jedoch die gleiche Aufgabe erledigt. Die Probleme die bei der Duplizierung von Code entstehen sind sehr vielschichtig. Falls Fehler auftreten müssen diese an allen Stellen behoben werden, wo dieser Codeabschnitt dupliziert ist. Oft wird jedoch darauf vergessen den Fehler an allen Stellen zu Verbessern, wodurch der Fehler später wieder auftreten wird. Ein weiteres Problem ergibt sich durch die Tests, welche mehrfach geschrieben werden müssen und dadurch auch mehrfach gewartet werden müssen. Im Falle, dass ein weiters Feature zu einer Codestelle hinzugefügt werden muss, welche eine Duplizierung darstellt, kann wieder wie im Fehlerfall darauf vergessen werden, alle Codestellen zu ändern.

Probleme mit Duplizierungen entstehen jedoch nicht nur bei der direkten Arbeit mit Code sondern auch beim Testen, Erzeugen oder Verteilen von Software. Auch diese Aspekte der Duplizierung sollten im folgenden Abschnitt analysiert werden.

3.1.1 Duplizierung boolscher Ausdrücke

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

Wie bereits beschrieben treten Codeduplizierungen sehr oft in boolschen Ausdrücken auf. Dies ist oft ein sehr subtiles Problem und kann nicht so leicht festgestellt werden. Meist ist dies bei komplexen boolschen Ausdrücken der Fall. Diese werden oft von mehreren Funktionen benötigt und daher einfach in den einzelnen Funktionen verwendet. Ein Beispiel für so einen boolschen Ausdruck befindet sich in Listing **??**.

```
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 3.1: Komplexe boolsche Ausdrücke 1 Zeile 255 - 264

Ein paar Zeilen weiter in derselben Klasse findet sich die in Listing 3.2 dargestellte 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 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 **??** dargestellt erfolgen müsste, würde dies eine Änderung in allen Funktionen welche dieses boolsche Statement verwenden nach sich ziehen.

```
1  propertyName.contains("</collectionType><element>")
```

Listing 3.4: Boolscher Ausdruck neu

Falls beim Ändern der Überprüfung auf eine der Funktionen, in denen diese verwendet wird, vergessen wird, würde dies zu einem Fehler führen. Ein weiteres Problem, das sich hier wie bereits oben beschrieben ergibt, ist die Tatsache, dass dieser boolsche Kommentar ohne einen Kommentar oder ein genaues

Lesen sehr schwer 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. Eine solche Methode könnte wie in Listing **??** aussehen.

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

Listing 3.5: Boolscher Ausdruck neu

Durch ein Extrahieren der Überprüfung des Parameters *propertyName* kann für diese Methode ein eigener Name gewählt werden, der klar den Zweck dieser Überprüfung vermittelt und durch diese Extrahierung wird auch die sehr problematische Codeduplizierung verhindert. Codeduplizierungen eliminiert und es bleibt nur noch die Sicherheitsüberprüfung auf null.

3.1.2 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 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.

3.1.3 Redundante Kommentare

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

Redundante Kommentare findet man sehr häufig in den verschiedensten Projekten. Oft werden für Methoden Kommentare geschrieben, welche nicht mehr als den Namen der Methode beinhalten. Dies führt natürlich zu keiner Verbesserung der Lesbarkeit und führt im Weiteren auch zu einer Aufblähung des Codes. Sehr häufig tritt dieses Problem auch bei Vorgeschriebenen Kommentaren auf. bei denen es in Coding Conventions vorgeschrieben ist, dass jede Methode einen Header zur Dokumentation bekommen sollte. Dabei kommt es häufig zu redundanten Kommentaren, welche keinen wirklichen Mehrwert bringen. 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 Regeln verzichten, da diese eben genau zu den genannten Probleme führen.

```
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

3.2 Komplexe Strukturen und schwer zu verstehende Konstrukte

Sehr oft kommt es in der Softwareentwicklung zu einer sehr komplexen Struktur der einzelnen Komponenten. Klassen mit sehr vielen Funktionen. Funktionen, welche mehr als eine Aufgabe erfüllen und daher oft sehr lange werden. Nicht mehr benötigte Codeabschnitte, welche aus Angst vor einem Verlust dieser nicht gelöscht werden. Dies sind nur einige der Probleme, welche in der Softwareentwicklung auftreten können. Viele dieser Probleme sind leicht zu lösen. Durch moderne Versionsverwaltungssysteme ist es nicht mehr nötig Codeabschnitte, die nicht mehr benötigt werden auszukommentieren. Diese können einfach gelöscht werden. Einige Probleme, welche sich zum Beispiel bei Klassen mit sehr vielen Methoden ergeben, können nur durch Erfahrung und Übung erkannt werden. Es sollte bei Klassen und Funktionen vor allem darauf geachtet werden, dass diese nur genau einen Zweck erfüllen. Ein weiterer wichtiger Indikator, ob eine Komponente gut programmiert ist, ist die Testbarkeit dieser. Bei schlecht programmierten beziehungsweise geplante Klassen ist es nur sehr schwer - oder gar nicht - möglich diese zu Testen. Im folgenden Abschnitt werden die häufigsten Probleme für komplexe Strukturen beziehungsweise schwer zu verstehende Konstrukte aufgearbeitet.

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. Im Hibernate Framework ist eine Klasse die einen solchen auskommentierten Codeabschnitt enthält die *BindHelper* Klasse. Eine sehr problematische Stelle befindet sich in dieser Klasse in Zeile 421. Dort gibt es den in Listing 3.7 beschriebenen 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 3.7: 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 in einem Issue Tracking System genau mitdokumentiert werden, wo der Fehler auftritt und Möglichkeiten diesen zu beheben, oder 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, wodurch sich vermutlich weitere Probleme ergeben. 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 auch den Quellcode unnötig aufbläht.

3.2.2 Kapseln von Errorhandling in eigene Methoden

- 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 richige 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 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 Quelltext von Roslyn ist die in Listing 3.8 gezeigte Methode.

```
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 3.8: 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 eines null-Wertes ignoriert. Dies sollte in einem anderen Abschnitt näher behandelt werden. Was man an diesem Beispiel gut erkennen 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. Dabei ergibt sich ein weiters Problem. Beim Aufruf der Methode *Create* kann auf die Fehlerbehandlung vergessen werden, was im schlimmsten Fall zu einer unbehandelten Ausnahme führt. 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 den in Listing 3.9 und in Listing 3.10 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 3.9: 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 3.10: 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.

3.2.3 Rückgabe von Fehlercodes

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 Fehlercodes zurückgegeben, was dazu führte, dies führte zu mehreren Problemen:

- Beim Aufruf dieser Methode muss darauf geachtet werden, dass alle möglichen Fehlercodes abgedeckt werden.
- Da diese Fehlercodes 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 Quelltext, oder der Dokumentation beschrieben werden.
- Durch die Rückgabe eines Fehlercodes ist es nicht mehr möglich einen Wert zurückzugegeben, wodurch meist Parameter als Outputparameter verwendet werden, welche den gewünschten Wert zurückliefern.

In den modernen Programmiersprachen gibt es wie bereits beschrieben die Möglichkeit von Ausnahmen. Diese Ausnahmen werden dabei im Fehlerfall ausgelöst und schließlich im aufrufenden 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 weiters 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 Fehlercodes und daher sollte bei Verwendung von moderneren Programmiersprachen auf Fehlercodes unbedingt verzichtet werden.

3.2.4 Switch Statements

- 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*. Über diese wird meist geregelt, welche Aktionen abhängig von einer Eingabe durchgeführt werden. Dies führt zu einer Vermischung der Aspekte. Die Funktion, welche diese Überprüfungen vornimmt hat dadurch mindestens zwei Aufgaben. Einerseits wird die Eingabe überprüft und abhängig davon eine Aktion vorgenommen. Dies führt im Weiteren auch zu einer schlechteren Testbarkeit. In objektorientierten Programmiersprachen können die meisten *Switch-Statements* durch einfache Abstrahierungen ersetzt werden. Ein gutes Beispiel, wo diese Verbesserung angewandt werden könnte wird in Listing **??** gezeigt.

```
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 3.11: Beispiele für Switch Statement; label

Hier wird überpüft, welches *LogLevel* übergeben wird und anschließend wird die jeweilige Methode aufgerufen. Hier wäre es jedoch einfach möglich, statt dem *LogLevel* ein Objekt zu übergeben, welches das in Listing 3.12 dargestellte Interface implementiert.

```
1 public interface ILogger
2 {
3   void WriteLog(string message);
4 }
```

Listing 3.12: Beispiele für ein Log Interface

Dadurch, dass es jetzt möglich ist, die Methode WriteLog des übergebene Objektes aufzurufen ergibt sich der in Listing **??** dargestellte Quellcode.

```
1 public virtual void Log(
2   ILogger logger,
3   int eventId,
4   object state,
5   Exception exception,
6   Func<object, Exception, string> formatter)
7 {
8   //Building message
9   ...
10  //
11
12  logger.WriteLog(message.ToString());
13 }
```

Listing 3.13: Beispiele für Switch Statement; label

Durch diese Änderung wird die Funktion um einiges kürzer und dadurch übersichtlicher. Es wird außerdem dadurch die Aufgabe der Überprüfung ausgelagert. Beim Aufrufen dieser Methode kann jetzt zusätzlich zu den bekannten

LogLevels auch ein spezieller Logger übergeben werden, welcher zum Beispiel in eine Datenbank schreibt. Die wichtigste Verbesserung ist hier, wie bereits erwähnt, die stark verbesserte Lesbarkeit.

3.3 Namensgebung

Eine weiterer sehr zentraler Punkt in CCD ist die Namensgebung. Robert C. Martin vergleicht die Namensgebung in der Programmierung dabei mit der Namensgebung für Kinder. Damit versucht er dem Leser klar zu machen, welche Rolle die Namensgebung in der Programmierung spielt. Im folgenden Abschnitt werden einige wichtigen Punkte der Namensgebung aufgearbeitet.

3.3.1 Namensgebung - Klassen

- Projekt: *Log4Net*
- Programmiersprache: *C#*
- Betreffende Klasse: *LogLog*
- Betreffender Namespace: *log4net.Util*

Bei der Namensgebung kommt es immer wieder zu Unklarheiten und Problemen wodurch es oft nötig wird, mit Kommentaren zu beschreiben, wofür diese Komponente verwendet wird. Dabei werden diese Probleme umso größer, je größer der Gültigkeitsbereich dieses Namens ist. Ein Beispiel für eine solche schlechte Namensgebung in einem großen Gültigkeitsbereich in Log4net, ist die in Listing 3.14 dargestellte Klasse *LogLog*.

```
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 3.14: Beispiele für schlechte Namensgebung

Grundsätzlich kann man anhand des Namens *LogLog* keine genauen Aussagen machen, welche Aufgabe diese Klasse erfüllt. Ein Blick in den im Listing

3.14 stehenden Kommentar gibt Aufschluss darüber, dass das Logging über Log4Net für Log4Net Komponenten nicht möglich ist, wodurch es notwendig ist, eine eigene Klasse für das interne Logging zu implementieren. Der Kommentar könnte durch eine bessere Namensgebung für die Klasse überflüssig gemacht werden. Ein Beispiel für einen bessern Namen wäre *Log4NetInternalLogging*, wodurch gleich klar wird, dass diese Klasse nur für das interne Logging zuständig ist.

3.3.2 Aussprechbare Namen

- Projekt: *Swift*
- Programmiersprache: *C++*
- Betreffende Klasse: *LogLog*
- Betreffender Namespace: *log4net.Util*

Die Softwareentwicklung ist ein sehr kollaborativer Prozess. Meist wird in einem Team von mehreren Leuten zusammengearbeitet, wobei die einzelnen Mitgliedern oft an ähnlichen Teilen des Systems arbeiten. Oft kommen dabei fragen auf, wo mit Kollegen diskutiert wird, welchen Nutzen dieses oder jenes hat. Dabei sind Namen, welche aussprechbar sind, sehr hilfreich, da die Kommunikation dadurch sehr stark vereinfacht wird. Ein Beispiel für einen unaussprechbaren Namen findet man im Quellcode der neuen Programmiersprache Swift von Apple. Dort gibt es den in Listing 3.15 dargestellten Namen.

Listing 3.15: Unaussprechlicher Name

3.4 Fehlerbehandlung

3.4.1 Kein 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 Enumerationstypen wie Listen oder ähnlichem sollte eine leere Liste 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. Bei Objekten gibt es die Möglichkeit das sogenannte *Nullobjectpattern* zu verwenden. Bei diesem Pattern wird statt des Wertes *null* eine leere Implementierung des Objektes zurückgegeben. Ein Beispiel, wo ein solches *Nullobjectpattern* Anwendung finden könnte wäre der Codeabschnitt in Listing 3.16.

```
1 protected List<ProcessModule> InnerList
2 {
3   get
4   {
5       if (_list == null)
6         _list = new List<ProcessModule>();
7     return _list;
8   }
```

```
9 }
```

Listing 3.16: Beispiele für Rückgabe 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.

Kapitel 4

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?

4.1 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.

4.2 Checkliste

Abschließend noch eine kurze Liste der wichtigsten Punkte, an denen erfahrungsgemäß die häufigsten Fehler auftreten (Tab. 4.1).

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

Tabelle 4.1: 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