Clean Code Development - Grundlagen und Anwendung

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

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.

Kapitel 1

Einleitung

1.1 Problemstellung

In der Softwareentwicklung gibt es einen sehr zentralen Aspekt der sich nahezu über alle Elemente des Prozesses der Softwareentwicklung erstreckt oder Einfluss auf diese hat. Es geht in erster Linie immer darum, ein Produkt weiterzuentwickeln, bestehende Fehler zu beheben, oder neue Funktionalität hinzuzufügen. Dieser Änderungsprozess bezieht sich jedoch nicht nur auf die Software selbst, sondern auch auf die gesamte Infrastruktur, sowie die Menschen die mit der Software arbeiten, diese entwickeln, oder planen. Die Änderungen, welche in dem Bereich Infrastruktur stattfinden sind sehr tiefgreifend. Ein neues Betriebssystem für das die Software angepasst werden muss wäre ein Beispiel für solch eine Änderung. Häufig kommt es auch zu einer Änderung der Tools mit denen der Programmierer oder die Programmiererin arbeitet. Neue Updates kommen heraus, oder es wird schlichtweg eine neue Entwicklungsumgebung eingesetzt, mit denen die alten Tools nicht mehr verwendet werden können. Aber auch auf personeller Ebene wird sich im Laufe der Zeit eines Softwareprojektes einiges ändern. Neue MitarbeiterInnen kommen hinzu, andere MitarbeiterInnen wechseln in eine andere Abteilung und sind daher nicht mehr für das Projekt verfügbar. Vor allem diese personellen Änderungen führen oft zu großen Problemen, da viele Programmteile von der Person abhängen, die diese implementiert hat. Wenn diese Person jetzt wechselt, muss sich jemand anderes in diesen Programmteil einarbeiten. Dies kann je nach Komplexität sehr stark variieren. Wenn der Ersteller oder die Erstellerin dieses Programmteiles nicht auf lesbaren Code geachtet hat, wird die Zeit für die Einarbeitung noch verlängert. Dies führt dazu, dass viele alte Programmteile nicht mehr geändert werden und bei Fehlern zahlreiche Workarounds implementiert werden. Wenn dieses Problem in einem größeren Kontext betrachtet wird, kann es im schlimmsten Fall zum Scheitern eines Projektes führen, da es unmöglich wird Änderungen vorzunehmen.

1.1.1 Theorie der zerbrochenen Fenster

Robert C. Martin bringt in seinem Buch **??** die *Theorie der zerbrochenen Fenster* in Verbindung mit der Softwareentwicklung. Zuerst beschrieben wurde diese Theorie in einem Artikel von James Q. Wilson und George L.Kelling. Die Autoren haben in Ihrem Artikel **??** unter anderem erwähnt, dass ein zerbrochenes Fenster in eime Gebäude, welches nicht repariert wird, die Zerstörung weiterer Fenster nach sich zieht, da es so aussieht, als ob sich niemand darum kümmern würde. Robert C. Martin verwendet diese Theorie und versucht sie für die Softwareentwicklung anzupassen. Dabei beschreibt er einen sehr ähnlichen Fall wie der eines zerbrochenen Fensters. Bei einem Programmabschnitt, sei es eine Klasse, eine Methode oder ähnliches, welche bereits schlecht gestaltet und unlesbar 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 meist auch nicht 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 wird, sondern von vielen verschiedenen ProgrammiererInnen. 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 nur noch CCD) 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, desto leichter wird es diesen auch noch nach einiger Zeit wieder zu Lesen. In dieser Arbeit sollten häufig auftretende Probleme in der Softwareentwicklung betrachtet werden und an Hand der von Robert C. Martin definierten Regelen gezeigt werden, wie diese gelöst werden könnten.

Kapitel 2

Clean Code Development Grundlagen

2.1 Womit beschäftigt sich CCD?

Wie in der Einleitung bereits erläutert beschäftigt sich CCD in erster Linie mit dem Schreiben von lesbaren Code. Es sollte dabei bei der Implementierung geachtet werden, dass In der Abbildung 2.1 ist ein Kreislauf dargestellt, der die drei Hauptpunkte darstellt und auch zeigt, dass diese sich gegenseitig beeinflussen. 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 sind die Einarbeitungszeiten für Mitarbeiter außerdem viel kürzer. 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.

2.2 Woher kommt CCD?

Wie zahlreiche Ideen in der Softwareentwicklung gibt es auch die Idee des CCD seit vielen Jahren.

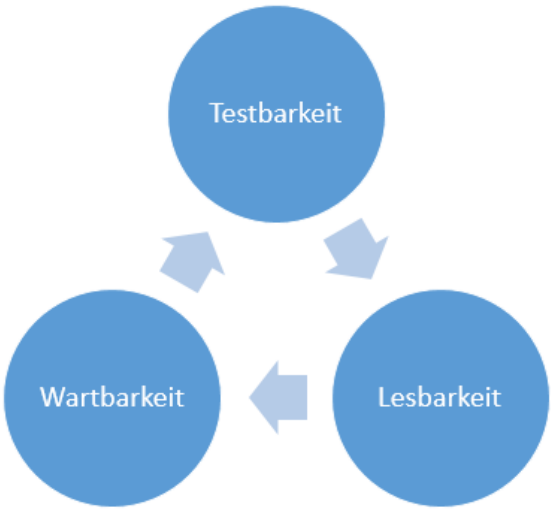

Abbildung 2.1: CCD Zyklus

2.3 Ziele von CCD

Die wichtigsten Ziele von CCD sind es, den Quellcode

2.4 Die Pfadfinder-Regel

Eine der grundlegendsten Regeln im CCD ist die sogenannte Pfadfinder-Regel. Robert C. Martin definiert diese in [**CleanCode**] wie folgt:

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

Auf die Arbeit eines Programmierers oder einer Programmiererin umgemünzt bedeutet dieser Leitsatz, dass jede Klasse, welche man auscheckt oder betrachtet verbessert werden sollte. Dies muss nicht immer ein großes Refactoring oder ein verbessern der Gesamtstruktur sein. Es reicht meist schon, für eine Variable einen besseren Namen zu vergeben, oder einen unnötigen Kommentar zu entfernen. Robert C. Martin schreibt weiters, dass sich durch dieses vorgehen ein für die Softwareentwicklung ungewöhnlicher Trend ergibt. Der Quelltext wird über die Zeit besser. Er verbindet dieses Verhalten außerdem mit professionelem Verhalten. Ein professioneller, verantwortungsbewusster Programmierer versucht ständig, die Codebasis zu verbessern.

2.5 Smells und Heuristiken

Robert C. Martin beschreibt in [**CleanCode**] eine Liste von Smells und Heuristken, welche Probleme beschreiben, die sehr häufige Ursachen für schlechten Code darstellen. In den folgenden Abschnitt werden diese in Tabellen gruppiert aufgelistet.

2.5.1 Kommentare

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

Tabelle 2.1: Smells und Heuristiken für Kommentare

2.5.2 Umgebung

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

Tabelle 2.2: Smells und Heuristiken für die Umgebung

2.5.3 Funktionen

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

Tabelle 2.3: Smells und Heuristiken für Funktionen

2.5.4 Allgemein

2.5.5 Namen

2.5.6 Tests

2.6 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 Opensource 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# [**CSHARPCoding**]. Meist werden diese grundlegenden Konventionen von Unternehmen verwendet und nur teilweise angepasst, da es durch einen durchgängigen Programmierstil für eine Plattform leichter ist, dass sich neue Programmierer oder 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 2.7 einige Möglichkeiten erläutert, wie die Überprüfung dieser Kriterien erfolgen kann.

2.7 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 2.6 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.

2.7.1 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 2.6 beschriebenen Coding Conventions festegelegt 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 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 (http://www.ndepend.com/). 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 programmiert wurde. Im Java Bereich gibt es das sehr hilfreiche Tool JDepend

(http://clarkware.com/software/JDepend.html) welches ähnlich aufgebaut ist wie NDepend.

2.7.2 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 Entwicklern vorgenommen werden. Dabei ist einer der wichtigsten Punkte, dass das Review nicht durch die Person erfolgt, die den Quellcode produziert hat, sondern durch jemand anderen. Weiters gibt es die Möglichkeit ein sogenanntes Peer Review durchzuführen. Bei diesem führen die Person, die den Quellcode produziert hat und eine zweite Person das Review durch und besprechen den vorliegenden Code. An dieser Stelle sollte das Prinzip des Pair Programmings, das aus dem Extreme Programming kommt erwähnt werden. [**BeckExtreme**]

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 Abhängikeiten in phyische 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

Tabelle 2.4: Smells und Heuristiken für Kommentare

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

Tabelle 2.5: Smells und Heuristiken für die Umgebung

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 druch Coverage-Patterns
Testen sollten schnell sein

Tabelle 2.6: Smells und Heuristiken für die Umgebung

Kapitel 3

Anwendung der Clean Code Development Kriterien

Im folgenden Abschnitt sollten einige der wichtigsten CCD Kriterien verwendet werden, um ausgewählte Codeabschnitte aus verschiedenen Opensourceframeworks zu verbessern.

Folgende Frameworks dienen der Veranschaulichung:

- Log4net (.NET)
- Hibernate (Java)
- Roslyn (.NET)
- Asp.net Websockets (C#)

3.1 Duplizierungen beim Entwickeln von Software

Eines der größten Probleme bei der Entwicklung von Software stellt die Duplizierung dar. Dabei lässt sich Duplizierung nicht nur im Code finden, sondern auch in anderen Vorgängen die für die Softwareentwicklung von Bedeutung sind. Für jeden Entwickler gibt es zahlreiche Aufgaben, welche er immer wieder erledigen muss. Diese sollten möglichst automatisiert und mit wenig Aufwand möglich sein, da dies ansonsten eine Duplizierung der Arbeit bedeuten würde.

Auf Code bezogen verbirgt sich Duplizierung nicht immer nur auf Bibliothek-, oder Klassenebene, sondern auch in Methoden. Dabei muss man zwischen einfacher, und logischer Duplizierung unterscheiden. Bei der einfachen Duplizierung handelt es sich um schlichtweg gleichen Code. Meist wurde dieser kopiert und nur ein wenig angepasst. Die logische Duplizierung ist meist schwieriger zu erkennen, denn in diesem Fall sind die duplizierten Stellen vom Aufbau her unterschiedlich, sie gleichen sich aber in der Aufgabe welche diese erledigen. Oft treten solche logischen Duplizierungen, wenn bei der Benennung der vorhandenen Methoden beziehungsweise Klassen nicht auf eine ausreichend offensichtliche Namensgebung geachtet wurde. Ein sehr wichtiges Prinzip hinsichtlich Codeduplizierung ist das von Dave Thomas und Andy Hunt beschrieben DRY-Prinzip (Don´t Repeat Yourself [**PRAG1999**]).

Wie bereits erwähnt, ergeben sich durch die Duplizierung zahlreiche Probleme. Einerseits müssen Arbeiten doppelt erledigt werden, 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. Bei boolschen Ausdrücken ist dies sehr oft der Fall. Eine logische Verzweigung, welche auf zum Beispiel die Validität einer Eingabe prüft, könnte 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 Legacy Code

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 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.

3.2.3 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 3.8: 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 3.9 dargestellte Interface implementiert.

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

Listing 3.9: 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 3.10: 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.

3.2.4 Rückgabe von Null

- 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 3.11.

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

```
8   }
9 }
```

Listing 3.11: 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.

3.3 Komplexe Strukturen und schwer verständliche Konstrukte

Sehr häufig kommt es in der Softwareentwicklung zu Unklarheiten. Der Leser des Codes schließt aus dem betrachteten Quelltext auf etwas anderes, wie de 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 Schlecht gewählte Namen

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

Der Ausschnit in Listing 3.12, 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 3.12: Beispiele für schlechte Namensgebung

Das Problem, welches sich bei dieser Klasse ergibt, ist die Nötigkeit eines Kommentares um klarzumachen, welche Aufgabe die Klasse erfüllt. Durch den gewählten Namen *LogLog* wird dieses keineswegs

Grundsätzlich kann man anhand des Namens *LogLog* keine genauen Aussagen machen, welche Aufgabe diese Klasse erfüllt. Ein Blick in den im Listing 3.12 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.

```
1   public sealed class Log4NetInteralLogging
```

Listing 3.13: Beispiele für bessere Namensgebung

3.3.2 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 3.14 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 3.14: 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 3.15 und in Listing 3.16 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 3.15: 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 3.16: 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 Falsche 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 Quelltext, welchen diese dokumentieren überdauern. Dadurch ergibt sich meist, dass der Kommentar entweder ungültig ist und im besten Fall nur etwas nicht mehr vorhandenes beschreibt. Im schlimmsten Fall kann dieses überdauern des eigentlichen Codes dazu führen, dass der Kommentar schlichtweg falsch ist und dieser für den Programmierer, welcher diesen Kommentar liest, keine Unterstützung bietet und diesen 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. Ein Beispiel für einen solchen falschen Kommentar wird in Listing 3.17 gezeigt.

```
1         TTODOOOOO ADD SOURCE
```

Listing 3.17: Falscher Kommentar

3.3.4 Verwendung von Magic Strings/Numbers

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

Ein Problem welches sich sehr häufig ergibt sind sogenannte Magic Strings oder Magic Numbers. Es handelt sich dabei um Zeichenketten oder Zahlen die direkt in den Quelltext geschrieben werden. In Listing 3.18 ist ein Beispiel für mehrere solcher Magic Numbers.

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

Listing 3.18: MagicNumbers

Durch diese direkt geschriebenen Zahlen 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 die Extrahierung der Zahlen in eine eigene Konstante. Für das Beispiel in Listing 3.18 würde sich der in Listing 3.19 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 3.19: Magic number

Mit dem in Listing 3.19 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 dieser Extrahierung 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[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 Fazit zu CCD

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 CCD im alltäglichen Gebrauch

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