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 Problemstellung

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 Motivation

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 woltle.

1.3 Lösungsvorschlag

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

2.3 Die Pfadfinderregel

Oft

2.4 Überprüfung der CCD Kriterien

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 Kommentare

In der Softwareentwicklung sind Kommentare ein Mittel um dem Leser verschiedene Dinge zu vermitteln. Einerseits können Gründe vermittelt werden, wieso etwas so implementiert wurde, oder was sich der Autor dabei gedacht hat. Auch rechtliche Kommentare, meist gibt es diese bei Opensource Projekten, sind ein sehr häufige Form der Kommentare. Für Bibliotheken gibt es weiters die Möglichkeit, Beschreibungen für Parameter, Rückgabewerte und allgemein den Zweck von Methoden oder Klassen zu erläutern.

Ein sehr problematisches Thema beim Kommentieren von Quellcode ist die Tatsache, dass oft darauf vergessen wird bei einem Refactoring diese auch zu ändern.

4.1.1 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 4.2 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 4.1: Beispiele für überflüssige Kommentare

4.1.2 Auskommentierter Code

- 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 4.4 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 4.2: Beispiele für die Verwendung von *GetByPredicate*

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.

4.2 Namensgebung

4.3 Fehlerbehandlung

4.3.1 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 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 4.9 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 4.3: 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 4.10 und in Listing 4.11 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 4.4: 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 4.5: 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.

4.4 Grenzen

4.5 Testen

4.6 Funktionen und Methoden

4.7 Klassen

4.7.1 Namensgebung - Klassen

- 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 4.3 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 4.6: 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 4.3 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.

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