Universität Stuttgart

Institut für Parallele und Verteilte Systeme
Anwendersoftware

Code-Qualität: Must Reads

Seminararbeit

Studien Projekt (WS 2015/16)

Betreuer: Christoph Stach

Tobias Boceck

Stuttgart, 03.03.2016

Code-Qualität: Must Reads

Tobias Boceck

boceckts@gmail.com

Zusammenfassung. Code-Qualität ist wichtig da durch sie die Lesbarkeit, Wartbarkeit und Performance des Programmcodes verbessert werden kann. In dieser Arbeit wird auf die wichtigsten Techniken des guten Programmierens eingegangen und gezeigt, auf welche Art diese die Code-Qualität verbessern. Es werden Muster zum effektiven objektorientierten Programmieren vorgestellt und Methoden zur Optimierung bestehenden Codes beschrieben. Dabei spielen besonders der Code-Style und die Code-Dokumentation eine wichtige Rolle. Es werden zudem einige Methoden des Code-Refactorings beschrieben.

1 Einleitung

Der Computer ist heutzutage allgegenwärtig und damit auch die Software, die auf diesem läuft. Die Kosten für diese Software und Programme werden im Gegensatz zu den Hardwarekosten immer größer. Daher ist es wirtschaftlich lohnenswert, die Kosten im Bereich der Software zu reduzieren. Nach Kostenanalysen im Bereich der Software-Entwicklung wurde gezeigt, dass die meisten Kosten durch die Wartung der Software entstehen. Dies ist ein Zeichen mangelnder Softwarequalität. Ein erster Schritt, die Wartungskosten zu reduzieren, ist die Code-Qualität zu verbessern. Durch erhöhen der Code-Qualität kann sichergestellt werden, dass einfache Fehler vermieden werden, die Wiederverwendbarkeit von Code-Fragmenten erhöht und die Einarbeitungszeit verkürzt wird [1].

Da nicht jeder Software Entwickler ein System von Grund auf neu erstellt, sondern oft bestehende Systeme warten und erweitern muss, ist es wichtig, wie man Code-Qualität auch bei schon bestehenden Systemen verbessern kann. Daher liefert diese Arbeit nicht nur Methoden zum Erhöhen der Code-Qualität während des Programmierens, sondern auch die wichtigsten Techniken zur Code-Optimierung und Verbesserung der Code-Qualität bei bestehenden Programmen.

Zum Lesen dieser Arbeit wird ein gewisses Grundverständnis von (objektorientierten) Programmiersprachen und der Programmentwicklung vorausgesetzt.

Diese Arbeit soll nur einen kurzen Überblick gegeben, welche Bereiche der Code-Qualität es gibt und wie man diese verbessern kann. Zum weiteren Nachschlagen über gutes Programmieren in Java empfiehlt sich das Buch *Effective Java* von Joshua Bloch. In diesem werden unter anderem die Entwurfsmuster und das generische Programmieren ausführlicher erklärt, die auch hier vorgestellt werden.

Wer mehr über das Thema Code-Qualität lernen möchte, ohne sich dabei auf eine Programmiersprache festzulegen, dem empfehle ich das Buch *Code Complete* von

Steve McConnell zu lesen. In diesem wird detailliert auf alle Bereiche der Code-Qualität eingegangen. Außerdem werden in diesem Buch auch Code-Qualität Merkmale, wie Dokumentation und Formatierung sehr umfangreich behandelt.

In Abschnitt 2 werden die wichtigsten Entwurfsmuster für objektorientierte Programmiersprachen vorgestellt und ihre Vorteile hinsichtlich der Code-Qualität erläutert. In Abschnitt 3 folgen Techniken zum Optimieren von bestehendem Code und in Abschnitt 4 geht es um Maßnahmen zur Verbesserung der Code-Qualität bei bereits bestehendem Code. Dabei spielt die Dokumentation, der Code-Style und das Code-Refactoring eine große Rolle. Zum Abschluss der Arbeit wird in Abschnitt 5 das Wichtigste zusammengefasst.

2 Richtlinien zum Erstellen guten Codes

Die Richtlinien zum Erstellen guten Codes sollen Softwareentwicklern helfen, ihren Code so zu gestalten, dass er eine größtmögliche Erweiterbarkeit, Lesbarkeit und Wartbarkeit aufweist. Die Unterabschnitte beschreiben, welche Gedanken man sich bereits vor dem Beginn des Programmierens machen sollte, um die zuvor beschriebenen Qualitätsmerkmale halten zu können.

2.1 Entwurfsmuster beim objektorientierten Programmieren

Dieser Abschnitt dreht sich um die grundlegenden Entwurfsfragen bei objektorientierten Programmiersprachen. Bevor man eine Klasse erstellt, muss man sich fragen welche Strategie man damit verfolgen will. Soll die Klasse eine Datenklasse mit vielen Variablen sein, die hauptsächlich zur Speicherung von Daten dient, ist sie nur eine Generalisierung anderer Klassen oder darf von ihr möglicherweise nur eine einzige Instanz existieren? Im folgenden Abschnitt werden einige Entwurfsmuster vorgestellt, welche eine Hilfestellung bei der Beantwortung dieser Fragen bieten.

Statische Fabrik-Methoden. *Statische Fabrik-Methoden* können als gute Alternative bzw. Erweiterung zu einem herkömmlichen Konstruktor angesehen werden. Anstatt ein Objekt einer Klasse direkt durch den Aufruf ihres Konstruktors zu instanziieren wird in diesem Entwurfsmuster zunächst eine statische Fabrik-Methode aufgerufen. Diese kann flexibel eine neue Instanz oder eine schon bestehende Instanz der Klasse zurückliefern. Außerdem können in der statischen Fabrik-Methode zunächst Parameter auf ihre Gültigkeit geprüft werden oder in Abhängigkeit von Übergabeparametern spezielle Vorbereitungen getroffen werden.

Im Folgenden ist ein Beispiel einer Klasse mit statischer Fabrik-Methode zu sehen, welche für jeden Thread eine unterschiedliche Instanz zurückgibt. Aufrufe der statischen Fabrik-Methode aus dem gleichen Thread geben jedoch die gleiche Instanz zurück.

```
public class MyClass {
    private static HashMap<Long, MyClass> instancesPerThreads = new
            HashMap<Long, MyClass>();
    public static MyClass getInstanceForCurrentThread() {
        MyClass instance = instancesPerThreads.get(
            Thread.currentThread().getId());
        if (instance == null) {
            instance = new MyClass();
            instancesPerThreads.put(
              Thread.currentThread().getId(), instance);
        }
        return instance;
    }
    private MyClass() {
    }
}
```

Auflistung 1. Statische Fabrik-Methode die für jeden Thread eine andere Instanz liefert

Die *HashMap* hält zu jedem laufenden Thread eine Instanz. Die statische Fabrik-Methode garantiert dabei, dass immer nur eine Instanz erzeugt wird, falls noch keine für den aktuellen Thread existiert. Der private Konstruktor verhindert, dass die Klasse versehentlich direkt instanziiert wird. Würde man anstelle dieser Methode einen Konstruktor verwenden, wäre dieses Szenario nur sehr umständlich möglich. Jedes Objekt das eine Instanz dieser Klasse erzeugen will, müsste selbst überprüfen, ob es schon eine Instanz dieser Klasse in diesem Thread gibt, oder ob eine neue Instanz erstellt werden muss. Bei objektorientierten Programmiersprachen sollte jede Klasse selbst kontrollieren, welche und wie viele Instanzen es geben darf.

Ein Vorteil, der aus dem Entwurfsmuster aus dem Beispiel direkt sichtbar wird, ist, dass statische Fabrik-Methoden im Gegensatz zu klassischen Konstruktoren einen aussagekräftigen Namen haben. Damit der Programmierer weiß, welche Funktion eine statische Fabrik-Methode hat, sollte neben einem aussagekräftigen Namen auch eine Bemerkung in der Dokumentation stehen, dass diese Klasse durch eine statische Fabrik-Methode instanziiert wird. Dabei sollte man dennoch darauf achten, dass einige Namensgebungen unter Softwareentwicklern bereits bekannt sind. Zu diesen gehören: *valueOf*, *getInstance*, *newInstance*, *getType* und *newType*.

Ein weiterer Vorteil solcher Methoden kann der Performance Gewinn gegenüber herkömmlichen Konstruktoren sein. Je nachdem, wie die statische Fabrik-Methode aufgebaut ist, kann es nützlich sein, immer das gleiche Objekt bei gleichen Übergabe-

parametern zurück zu liefern. Dadurch kann die Anzahl an existierenden Objekten stark reduziert werden.

Klassen die zu jedem Zeitpunkt wissen welche bzw. wie viele Instanzen existieren, werden auch *instance-controlled* genannt. Zudem muss eine statische Fabrik-Methode nicht ein Objekt des Typs der umschließenden Klasse zurückliefern, sondern kann auch ein Objekt einer Unterklasse erstellen und zurückgeben. Dadurch ist man bei Programmen, die Superklassen mit vielen spezialisierten Klassen haben, besonders flexibel in der Erstellung der spezialisierten Objekte. Das Collection Framework von Java nutzt diesen Vorteil stark aus und stellt fast nur statische Fabrik-Methoden in der Klasse *Collections* bereit. Diese Klasse kann nicht instanziiert werden und liefert nur Objekte von Klassen, die die *Collection Schnittstelle* implementieren.

Allerdings gibt es auch Nachteile der statischen Fabrik-Methoden. Um von einer Klasse zu erben, muss der Konstruktor dieser Klasse zwangsweise die Sichtbarkeitsstufe *protected* oder *public* haben. Hat er diese und zusätzlich noch eine statische Fabrik-Methode, so ist jedoch nicht gewährleistet, dass jeder Entwickler, der diese Klasse benutzt, auch die statische Fabrik-Methode aufruft, um eine neue Instanz der Klasse zu bekommen.

Dies führt auch schon zu dem zweiten Nachteil der statischen Fabrik-Methoden, dass sich diese Methoden bis zu dem heutigen Zeitpunkt in keiner IDE von anderen statischen Methoden abheben. Das erschwert es dem Anwender zu sehen, wie man solch eine Klasse instanziieren kann, oder noch schlimmer, er denkt die Klasse kann nicht instanziiert werden.

Alles in allem sollte jeder Softwareentwickler von dem Entwurfsmuster der statischen Fabrik-Methode gehört haben. Ob man dieses dann auch verwendet, muss situationsabhängig entschieden werden. Besonders bei großen Projekten, in denen sich die Anforderungen ständig ändern können, gibt dieses Entwurfsmuster einen sehr großen Grad an Flexibilität [2].

Singleton Klassen. Das Entwurfsmuster der *Singleton Klassen* baut zum Teil auf die soeben vorgestellten statischen Fabrik-Methoden auf. Zunächst ist eine Singleton Klasse eine Klasse, von der genau eine Instanz existieren darf. Dies ist sinnvoll, falls Objekte aus der realen Welt nur einmal existieren und in der Welt der objektorientierten Programmierung modelliert werden sollen. Eine Anwendung des Singleton Entwurfsmusters ist z. B. das Betriebssystem auf dem eine Anwendung läuft.

Die einfachste Implementierung einer Singleton Klasse wird realisiert durch eine konstante statische Variable, der direkt ein Objekt zugewiesen wird. Zudem muss man die Sichtbarkeit des Konstruktors auf *private* einschränken, sodass die Klasse nicht zufällig von einem Programmierer instanziiert werden kann.

```
public class MyClass {

    public static final MyClass INSTANCE = new MyClass();

    private MyClass() {

    }

}
```

Auflistung 2. Singleton Entwurfsmuster implementiert mit konstanter Klassenvariable

Implementiert man eine Singleton Klasse auf diese Weise, so wissen auch andere Programmierer, die an der derselben Klasse arbeiten sofort, dass es sich um eine Singleton Klasse handelt. Denn konstante Variablen sowie die Instanz Variable, sollten immer an oberster Stelle einer Klasse stehen.

In der nächsten Implementierung wird das Entwurfsmuster der statischen Fabrik-Methode benutzt. Das heißt, man benutzt eine statische Methode, um immer auf dieselbe Instanz der Klasse zuzugreifen.

```
public class MyClass {

    private static final MyClass INSTANCE = new MyClass();

    public static MyClass getInstance() {

        return INSTANCE;

    }

    private MyClass() {

    }

}
```

Auflistung 3. Singleton Entwurfsmuster implementiert mit statischer Fabrik-Methode

Der größte Vorteil an dieser Implementierung gegenüber der Vorigen ist die erweiterte Flexibilität. Ändern sich nach der Implementierung die Anforderungen an die Klasse und man möchte nun mehrere Instanzen zulassen, so müsste man in der ersten Implementierung die Sichtbarkeit des Konstruktors ändern und die Singleton Instanz Variable entfernen. Damit müssen alle Klassen geändert werden, die direkt auf die Instanz Variable zugegriffen haben, denn diese existiert nicht mehr. In der gerade vorgestellten Implementierung muss jedoch nur der Inhalt der statischen Methode geändert werden, die Methodensignatur bleibt allerdings gleich. Somit muss keine andere Klasse ihre Zugriffsmethode ändern.

Beide Implementierungen haben aber einen großen Nachteil, nämlich die Serialisierung. Bei beiden Implementierungen ist es nicht ausreichend die Schnittstelle *Serializable* zu implementieren, denn werden mehrere Objekte wieder eingelesen, so existieren mehr als ein einziges Objekt dieser Singleton Klasse. Um dies zu verhindern, muss man die Methode *readResolve* zusätzlich implementieren und dort die Eine,

echte Klasseninstanz zurückgeben. Diese Methode ersetzt nämlich das importierte Objekt durch die Singleton Instanz Variable.

Die dritte Art der Singleton Implementierung ist über ein Enum Konstrukt. Dabei wird ein Enum mit nur einem Element, wie im folgenden Programmcode, erstellt.

```
public enum MyClass {

    INSTANCE;

}
```

Auflistung 4. Singleton Entwurfsmuster implementiert mit einem Enum

Um nun die Singleton Instanz der Klasse zu bekommen greift man auf das *INSTANCE* Element des Enums, wie bei einem Objekt einer Klasse zu.

```
MyClass singleton = MyClass.INSTANCE;
```

Auflistung 5. Enum Singleton Objektzugriff

Die Nachteile der reflektierten Instanziierung und der Serialisierung sind hier nicht gegeben. Des Weiteren ist diese Art der Implementierung die wohl einfachste, denn es wird nur ein Enum mit einem Element erstellt. Diese Art der Instanziierung einer Singleton Klasse ist die am wenigsten verbreitete, jedoch sollte man eine Singleton Klasse immer auf diese Weise implementieren [2].

Builder Klassen. Man hat eine Klasse mit vielen Attributen, denen aber nicht allen, bei der Objektinstanziierung, zwangsweise ein Wert zugewiesen werden muss. Deshalb möchte man eine möglichst große Flexibilität für die Instanziierung dieser Klasse. Dafür benötigt man für jede Kombination der Übergabeparameter einen Konstruktor, um so die Attribute zu initialisieren. Dadurch wird die Klasse nicht nur unnötig groß, sondern auch unübersichtlich und man kann die Reihenfolge der Parameter schnell verwechseln. Hat man zudem Attribute desselben Datentyps, so kann man den Konstruktor nicht auf die Weise überladen, dass man unterscheiden kann, welchem Attribut der übergebene Parameter zugeordnet werden soll.

Die bereits vorgestellten statischen Fabrik-Methoden sind bei vielen Übergabeparametern auch keine große Hilfe, denn man bräuchte genau so viele statische Fabrik-Methoden, wie Konstruktoren.

Der Oracle Standard des *JavaBeans*[1] Entwurfsmusters löst dieses Problem indem es einen parameterlosen Konstruktor zur Verfügung stellt und eine Zugriffsmethode für jedes Attribut bereitstellt. Das heißt, es wird erst das Objekt erstellt und dann die Attribute initialisiert, die man gerne möchte [3]. Der größte Nachteil dieses Entwurfsmusters ist, dass das erzeugte Objekt in einem unvollständigen Zustand sein kann und es keine Konstanten haben darf. Diese können nämlich nicht durch Zugriffsmethoden verändert werden. Weiß der Programmierer außerdem nicht welche

[1] http://www.oracle.com/technetwork/java/javase/documentation/spec-136004.html

Parameter zwingend notwendig sind, könnte er Methoden aufrufen, die zu einem Fehlverhalten aufgrund der nicht initialisierten Attribute führen [2].

Ein Entwurfsmuster, das keine dieser Nachteile hat, ist das Entwurfsmuster der *Builder Klasse*. Anstatt direkt ein Objekt durch den Konstruktor zu erzeugen wird in diesem Entwurfsmuster zunächst ein Builder Objekt erzeugt, das auch als eine Art Prototyp der eigentlich zu erstellenden Klasse angesehen werden kann. Die Builder Klasse wird dabei als statische innere Klasse mit der Sichtbarkeit *public* implementiert und besitzt meist genau die gleichen Attribute wie die äußere Klasse.

Im Folgenden wird ein Kontakt als Klasse mittels dem Builder Klassen Entwurfsmuster modelliert. Dabei sind die Attribute *name* und *adress* notwendig, alle anderen sind optional.

```
public class Contact {

            private final String name;

            private final Adress adress;

            private final String secondName;

            private final Adress adress2;

            private final String notes;

            private final String title;

            private final int phoneNumber;


            protected Contact(Builder builder) {

              this.name = builder.name;

              this.adress = builder.adress;

              this.secondName = builder.secondName;

              this.adress2 = builder.adress2;

              this.notes = builder.notes;

              this.title = builder.title;

              this.phoneNumber = builder.phoneNumber;

            }


            public static class Builder {

              private final String name;

              private final Adress adress;

              private String secondName = "";
```

```
private Adress adress2 = new Adress(“”);

private String notes = “”;

private String title = “”;

private int phoneNumber = -1;


public Builder(String name, Adress adress) {

  this.name = name;

  this.adress = adress;

}


public Builder secondName(String secondName) {

  this.secondName = secondName;

  return this;

}


public Builder adress2(Adress adress2) {

  this.adress2 = adress2;

  return this;

}


public Builder notes(String notes) {

  this.notes = notes;

  return this;

}


public Builder setTitle(String title) {

  this.title = title;

  return this;

}
```

```
        public Builder phoneNumber(int phoneNumber) {
          this.phoneNumber = phoneNumber;
          return this;
        }

        public Contact build() {
          return new Contact(this);
        }
      }
  }
```

Auflistung 6. Kontakt Klasse implementiert mit einer Builder Klasse

Wie schon erwähnt, wird zunächst ein Builder Objekt mit einem Konstruktor erzeugt, der nur die wirklich notwendigen Parameter verlangt. In diesem Beispiel heißen die Parameter *name* und *adress*. Nun hat das Builder Objekt Zugriffsmethoden für jede der optionalen Attribute, ähnlich dem JavaBeans Entwurfsmuster.

Der wesentliche Unterschied besteht jedoch darin, dass jede der Zugriffsmethoden das Objekt selbst wieder zurückliefert. Somit kann auf jeder Zugriffsmethode eine weitere Zugriffsmethode und am Ende die *build* Methode aufgerufen werden, welche dann das Objekt der äußeren Klasse erzeugt. Im Konstruktor der äußeren Klasse werden dann alle Werte der Attribute des Builder Objektes kopiert. Alle optionalen Parameter, die man nicht setzt, werden dann mit einem Standardwert initialisiert. Auf diese Weise erzeugte Objekte sind in einem gültigen Zustand.

Eine mögliche Objekterzeugung der gerade gezeigten Klasse kann dann folgendermaßen aussehen.

```
    Contact me = new Contact.Builder("Tobias", new
Adress("Stuttgart")).notes("Student").build();
```

Auflistung 7. Objekterzeugung mit Builder Klasse

Anhand dieses Entwurfsmusters sieht man sofort, welche optionalen Parameter bei der Objekterzeugung übergeben wurde. Durch das Builder Klassen Entwurfsmuster hat man außerdem dieselbe Flexibilität wie bei Klassen mit herkömmlichen Settern, mit dem Vorteil, dass konstante Variablen in solch einer Klasse existieren können.

Der wohl größte Vorteil dieses Entwurfsmusters ist allerdings, dass man, ähnlich wie bei den statischen Fabrik-Methoden, auch Unterklassen der zu instanziierenden Klasse zurückgeben kann. Eine Builder Klasse kann somit auch genutzt werden, um Objekte von verschiedenen Klassen, die voneinander erben, dynamisch zu instanziieren.

Einen Nachteil muss man vor allem bei performancekritischen Anwendungen bedenken, nämlich, dass vor jeder Instanziierung einer solchen Klasse zunächst auch ein

Builder Objekt erzeugt wird. In alltäglichen Programmierszenarien sollte dies jedoch kein Grund sein, um das Entwurfsmuster nicht benutzten zu können. Als Faustregel kann man sagen, dass dieses Entwurfsmuster für Klassen, deren Konstruktoren oder statische Fabrik-Methoden mehr als fünf Übergabeparameter haben, eine gute Entwurfsentscheidung ist [2].

Sichtbarkeit von Variablen und Methoden. Das Prinzip des *Information Hiding* sagt, dass die Implementierung einer Klasse von der Außenwelt nicht sichtbar sein soll. Kommunizieren sollte man nur über das Klassen- bzw. Objektinterface. Je mehr man sich diesem Prinzip unterordnet, desto einfacher ist es für verschiedene Teams an einem Projekt zu arbeiten. Man sichert einer Methode eine gewisse Funktion zu und kann durch das Wissen ihrer Funktionsweise, die Methode über ihre Schnittstelle bereits in neuen Code einbinden. Wie die Methode ihre Funktion erfüllt ist für andere Entwickler nicht wichtig [4].

Selbiges gilt auch für Klassen und ihre Implementierungen. Das Abkapseln der Daten erleichtert außerdem eine Umänderung der internen Struktur, ohne dass andere Objekte ihre Methodenaufrufe anpassen müssen. Bei Klassen sollten außerdem, so weit möglich, auf Klassenvariablen verzichtet werden. Diese statischen Variablen werden in einem globalen Speicherbereich angelegt und existieren daher über die gesamte Dauer der Programmausführung [5]. Damit nehmen sie unnötig lang Speicherplatz in Anspruch. Dasselbe Prinzip sollte auch bei Methoden angewandt werden. Statische Methoden sollten durch lokale Methoden ersetzt werden, falls sie nicht unbedingt nötig sind. Sind sie außerdem nur zum Berechnen interner Prozesse, sollte ihre Sichtbarkeit auf *private* gesetzt werden.

Allgemein gilt bei Variablen und Methoden die Sichtbarkeit und den Gültigkeitsraum so gering wie möglich zu halten. Dadurch wird nicht nur die Lesbarkeit des Codes und damit auch dessen Wartbarkeit verbessert, sondern es erschwert es auch anderen Benutzern falsche Methoden und Variablen aufzurufen. Objekte und Klassen, bei denen nur die Methoden Schnittstellen sichtbar sind, sind zwar gut designt, allerdings muss dies nicht heißen, dass sie eine gute Performance haben. Durch das Einschränken der Sichtbarkeit der Variablen wird nur die Lesbarkeit, Wiederverwendbarkeit und Wartbarkeit einer Programmklasse verbessert [2].

2.2 Generisches Programmieren

Unter dem Begriff des generischen Programmierens versteht man die Fähigkeit einer Klasse oder einer Methode, mit Parameter beliebigen Typs zu parametrisieren und die Klasse oder die Methode somit mit mehr Flexibilität auszustatten. Für generische Klassen eignen sich Datenstrukturen, die mit möglichst vielen verschiedenen Datentypen funktionieren sollen.

Dies zeigt sich auch an der von Java gegebenen Klasse *ArrayList*, die generische Typ Parameter verwendet. Auf diese Weise müssen wir schon beim Erstellen der Liste den gewünschten Typ angeben, wie das folgende Beispiel verdeutlicht.

```
ArrayList<String> names = new ArrayList<String>();
```

Auflistung 8. Objekterzeugung mit Typ Parameter

In diesem Beispiel wurde eine Array Liste für String Objekte erstellt. Der Compiler weiß, dass diese Liste nur auf String Objekten arbeitet und gibt eine Fehlermeldung falls man dennoch versucht ein anderes Objekt hinzufügt. Dies macht diese Art der Implementierung t*ypsicher*.

Wäre diese Klasse nicht mit Java *Generics* umgesetzt worden, so hätte man die Methoden der Listen Klasse mit dem Übergabe- bzw. Rückgabetyp *Object* versehen müssen. Es wäre somit auch möglich Parameter jeden Types zu übergeben. Beim Zurückgeben der Objekte müsste nun jedoch gecastet werden. Dabei kann der Compiler keine Überprüfung hinsichtlich der Typsicherheit ausführen und mögliche Fehler werden deshalb erst zur Laufzeit entdeckt.

Allgemein sind generische Typen und Methoden einfacher und sicherer zu benutzen als der *Object* Typ mit anschließendem Cast und sollten deshalb immer bevorzugt werden [2].

3 Optimierung bestehenden Codes

Bei der Optimierung bestehenden Codes muss man sich immer im Klaren sein, dass eine Optimierung eines Aspektes zum Teil negative Auswirkungen auf andere Teile haben kann. So kann zum Beispiel eine Optimierung hinsichtlich der Ausführungsgeschwindigkeit Nachteile gegenüber dem zu verwendenden Speicherverbrauch haben. Das heißt, dass man sich gut überlegen muss, ob eine Optimierung seitens der Software überhaupt notwendig ist, oder man diese Zeit nicht für etwas Anderes investieren kann. Man sollte jedoch immer die Kosten gegen den eigentlichen Nutzen abgleichen und darauf basierend eine Entscheidung treffen.

Im Rahmen der Code-Qualität zeigen die folgenden Unterabschnitt, wie man vermeidet, dass unnötig viele Objekte und damit unnötig viel Speicher verbraucht wird. Auch zu erwähnen ist, dass bei Sprachen, die zu kompilieren sind, der Compiler einen großen Teil selbst effizient optimieren kann, ohne dass der Entwickler seinen eigenen Code anpassen muss. Oft ist es daher ratsam einen effizienten Compiler zu erwerben und zu benutzen anstelle von Hand zu optimieren. Wird das Endprodukt keine Anwendersoftware die an viele Privatpersonen geht, kann es außerdem sinnvoller sein, die ausführende Hardware aufzurüsten und somit auch bei zukünftigen Programmen keine Performanzprobleme zu haben. Man sollte aber in jedem Fall erst mit dem Optimieren von bestehendem Code anfangen, wenn die Funktionalität bereits gegeben ist. Denn eine Software die nicht funktioniert, aber optimiert ist, bringt keinem etwas [6].

3.1 Datentypoptimierung

Die Datentypoptimierung ist wohl die wichtigste Optimierung bei bestehendem Code und sollte bei jedem objektorientierten Programm angewandt werden. Die wichtigste

Optimierung besteht darin, dass man primitive Datentypen den Objekten von Wrapper Klassen vorzieht. Diese einfache Optimierung verhindert das Erstellen von zahlreichen unnötigen Objekten, die durch primitive Datentypen dargestellt werden könnten. Der andere große Vorteil ist, dass sich dadurch einfache Fehler, wie ein Vergleich der Objektreferenzen durch == anstelle des Wertes des Objektes durch den Methodenaufruf *equals()*, vermeiden lassen. Das nachfolgende Beispiel verdeutlicht diese Fehler nochmals.

```
Integer a = new Integer(500);

Integer b = new Integer(500);

System.out.println(a == b);          // Ausgabe: false

System.out.println (a.equals(b));    // Ausgabe: true
```

Auflistung 9. Vergleich von Objekten von Wrapperklassen

Wie im Beispiel zu sehen ist, erstellen wir zwei Integer Objekte und initialisieren diese mit dem Wert 500. Nun geben wir aus, ob der Vergleichsoperator == die beiden Objekte als gleich ansieht. Die Ausgabe ist *false* und sagt damit aus, dass die Objekte *a* und *b* nicht gleich sind obwohl sie beide den gleichen Wert haben. Die Ausgabe des Vergleiches basierend auf der *equals()* Methode eines der Objekte, ist *true*. Anhand dieses Beispiels sieht man, dass man beim Arbeiten mit Objekten von Wrapper Klassen immer mit der *equals()* Methode vergleichen sollte. Dies kann jedoch schnell vergessen werden.

Benutzt man stattdessen primitive Datentypen kann dieser Fehler nicht passieren. Im folgenden Beispiel nehmen wir zwei Variablen vom primitiven Typ *int* und initialisieren diese mit denselben Werten wie oben, nämlich 500.

```
int c = 500;

int d = 500;

System.out.println(c == d);          // Ausgabe: true
```

Auflistung 10. Vergleich von primitiven Datentypen

Benutzt man den primitiven Typ *int*, kann man die Methode *equals()*nicht ausführen, da primitive Typen keine Objekte sind. Deshalb hat man so die Gewissheit dass == die richtige Wahl ist. Die Ausgabe des Vergleichs der zwei Variablen ist natürlich *true*, das heißt die Werte sind gleich.

Ein weiterer Nachteil von Objekten von Wrapper Klassen ist, dass eine *null* Initialisierung und damit auch entsprechende Fehler möglich sind.

```
Integer a = null;

Integer b = null;

System.out.println(a.equals(b));     // java.lang.NullPointerException
```

Auflistung 11. Vergleich von Nullobjekten von Wrapperklassen

Der obige Programmcode ist korrekt und kann mit Java ohne Fehlermeldung kompiliert werden. Bei der Ausführung jedoch wird eine *NullPointerException* geworfen. Dieser Fehler kann bei primitiven Datentypen nicht passieren, denn diese sind nicht mit *null* initialisierbar [2].

3.2 Ausführungsgeschwindigkeit optimieren

Das Programm ist fertig geschrieben und funktioniert, jedoch nur sehr träge. Dies kann ein Hinweis auf schlechte Programmierung sein, muss es aber nicht. Grundsätzlich sollte man bei Bedenken hinsichtlich der Ausführungsgeschwindigkeit immer erst ein klares Bild davon verschaffen, was die langsamsten Operationen sind und an welchen Stellen das Programm viel Zeit benötigt. Dabei reicht es nicht ungefähr zu schätzen, an welchen Operationen es liegt, sondern man muss dies genau messen, denn Optimierung an der falschen Stelle kann auch kontraproduktiv sein. Außerdem muss man wissen, ob das träge Verhalten der Software tatsächlich durch umschreiben des Codes geändert werden kann, oder ob die langsamen Prozesse möglicherweise externe Dienste, wie z. B. des Betriebssystems sind, auf die man als Softwareentwickler gar keinen Einfluss nehmen kann.

Allgemein gilt für Programmiersprachen, je weniger oft ein Code-Fragment ausgeführt wird, desto schneller ist das Programm. Deshalb sollten if-Abfragen, wenn möglich, immer außerhalb von Schleifen stehen. Außerdem sollten Schleifen mit einem *break* unterbrochen werden falls nur eine Ausführung bis zu einer bestimmten Abbruchbedingung vorgesehen ist. Allerdings sollte man mit dieser Art der Optimierung vorsichtig sein, denn dadurch kann die Lesbarkeit des Programmcodes leiden.

Die zeitaufwändigsten Operationen bei den meisten Programmen sind Ein- und Ausgabeoperationen. Lesen und Schreiben von Daten auf eine Festplatte dauert im Gegensatz zu einfachen Rechenoperationen oder Programmspeicherzugriffen bis zu 1000-mal länger. Möchte man also ein möglichst schnelles Programm und hat die Wahl zwischen externen Daten auf der Festplatte oder die Daten direkt im Arbeitsspeicher zu lagern, sollte man sich immer für letzteres entscheiden. Natürlich wird bei dieser Art der Optimierung der Speicherverbrauch in den Hintergrund gestellt [6].

4 Code-Qualität

Programmcode mit guter Code-Qualität hat bestimmte Merkmale wie aussagekräftige Methoden- und Variablennamen, gute Dokumentation und gute Formatierung. Berücksichtigt man dies in seinem geschriebenen Programmcode sieht man, dass er leichter zu lesen, zu portieren und zu warten ist. All dies sind Merkmale die einen Programmcode mit guter Code-Qualität ausmacht.

In den folgenden Unterabschnitten wird gezeigt, was man unter guter Dokumentation und Formatierung versteht. Außerdem wird gezeigt, wie man bestehenden Code durch Refactoring so umändern kann, dass die Code-Qualität verbessert wird.

4.1 (Selbst-) Dokumentation

Dokumentation ist ein Begriff der im Prozess der Softwareentwicklung sehr facettenreich sein kann. Zunächst muss man unterscheiden zwischen externer und interner Dokumentation. Externe Dokumentation umfasst alle Dokumente, die während der Softwareentwicklung angefertigt wurden, während interne Dokumentation Kommentare im Code wiederspiegelt. Man kann allerdings auch den Code-Style als eine Form der Dokumentation ansehen. Je konsistenter man Code-Zeilen einrückt, desto leichter ist es später für andere Programmierer den Zusammenhang der Code-Zeilen zu verstehen.

Es gibt einzeilige und mehrzeilige Kommentare, für kurze bzw. lange und ausführliche Erklärungen. Mehrzeilige Kommentare sollten über Methoden oder Klassen stehen. Innerhalb von Methoden sollten nur kurze einzeilige Kommentare stehen, da sonst die Methode zu unübersichtlich wird. Außerdem sollte man Kommentare über die zu dokumentierenden Code-Zeile schreiben und nicht dahinter, da sonst der Code-Style darunter leiden kann.

Jeder Softwareentwickler der guten Code schreiben möchte sollte außerdem wissen, wie man effektive Kommentare schreibt. Kommentare sollten auf keinen Fall den Code einfach nur in einer anderen Form wiedergeben. Die einzigen Ausnahmen sind JavaDoc Kommentare, welche vor allem bei öffentlichen Bibliotheken hilfreich sind um die Verwendung einer Methode bzw. Klasse zu erläutern. Der Kommentar im folgenden Programmcode ist zwar ein JavaDoc, dennoch ist er nutzlos, da die Methode ihre Funktionalität durch ihren Namen verrät und des weiteren keine Seiteneffekte bei ihrer Verwendung auftreten. Er sollte daher weggelassen werden.

```
/**
 * Return the name of this import format.
 */
@Override
public String getFormatName() {
    return "Ovid";
}
```

Auflistung 12. Redundantes Kommentar

Meist werden Kommentare verwendet, um die Funktionalität von Methoden und ihren Seiteneffekten anderen Entwicklern zu offenbaren. Jedoch sollten Namen für Variablen, Methoden und Klassen immer so gewählt werden, dass sich deren Sinn auch ohne Kommentare erschließt. All zu oft jedoch werden Methoden mit mehreren Funktionalitäten implementiert, entweder weil der Programmierer sich Schreibarbeit ersparen möchte oder weil zusätzliche Bedingungen im Nachhinein hinzugekommen sind und diese entfernt zur Kernfunktionalität gehören.

Natürlich ist es dann sinnvoll Kommentare für die Methode zu schreiben, noch sinnvoller wäre es natürlich die komplette Methode zu überarbeiten und im Endeffekt

eine Methode mit einer Funktionalität, ohne Seiteneffekte und Dokumentation durch aussagekräftige Bezeichner und Methodennamen zu haben. Schlecht hingegen ist, wenn die Bezeichner bereits die Funktionalität verraten und es dennoch Kommentare gibt, die dies einfach wiederholen. Das folgende Code-Beispiel verdeutlicht die Redundanz.

```
/**
 * If an autocompleter exists for the "journal" field, add all
 * journal names in the journal abbreviation list to this
 * autocompleter.
 */
public void addJournalListToAutoCompleter() {
    AutoCompleter<String> autoCompleter = get("journal");
    if(autoCompleter != null) {
        for(Abbreviation abbreviation :
          Abbreviations.journalAbbrev.getAbbreviations()) {
            autoCompleter.addItemToIndex(abbreviation.getName());
        }
    }
}
```

Auflistung 13. Kommentar wiederholt Methodennamen in einem Satz

Der Kommentar in diesem Code-Beispiel wiederholt nur den Namen der Methode und macht daraus einen Satz, dieser Kommentar ist somit überflüssig [6].

4.2 Code-Style

Neben der Dokumentation gehört auch der Style des Codes zu einem Qualitätsmerkmal beim Programmieren. Der Code-Style hilft Programmcode einfacher zu verstehen und die Zusammenhänge deutlich zu machen. Jeder Programmierer hat seinen eigenen Style und seine eigenen Vorstellungen, wie Programmcode formatiert werden soll. Arbeitet man aber in einem Team sollte man sich immer auf einen gemeinsamen Style festlegen, dadurch findet sich jeder im Code zurecht. Das folgende Beispiel zeigt welche Verwirrung bei nicht formatiertem Code entstehen kann.

```
public boolean isRecognizedFormat(InputStream stream) throws IOException
{BufferedReader in = new BufferedReadr(
ImportFormatReader.getReaderDefaultEncoding(stream)); String str; int i =
0; while (((str = in.readLine()) != null) && (i < 50)) { if
(str.toLowerCase().contains("<pubmedarticle>")) {return true; } i++; }
return false; }
```

Auflistung 14. Code komplett ohne Formatierung

Im obigen Beispiel kann man zunächst nur durch die Bezeichner erraten was die Methode machen soll. Um genau zu verstehen was passiert muss man jedoch einiges an Zeit investieren. Diese Zeit kann man sich sparen, wenn man die Methode besser formatieren würde, wie im folgenden Code-Beispiel.

```
public boolean isRecognizedFormat(InputStream stream) throws IOException
{

        BufferedReader in = new BufferedReader(

             ImportFormatReader.getReaderDefaultEncoding(stream));

        String str;

        int i = 0;

        while (((str = in.readLine()) != null) && (i < 50)) {

            if (str.toLowerCase().contains("<pubmedarticle>")) {

                return true;

            }

            i++;

        }

        return false;

    }
```

Auflistung 15. Code mit logisch formatierten Abschnitten

Im Gegensatz zu dem Beispiel davor kann man nun mit Leichtigkeit erkennen, wie die Methode arbeitet und spart sich eine Menge Zeit.

Doch nicht nur Zeit lässt sich einsparen, man kann durch Code-Style auch Leichtsinnsfehlern vorbeugen. Menschen tendieren dazu Objekte, die näher zusammen sind, als Einheit anzusehen [7]. Der auszuführende Computer hat dieses Problem nicht und führt deshalb Programmzeilen anders aus als wir sie interpretieren würden. Das folgende Beispiel zeigt eine Implementierung einer einfachen Rechnung, allerdings mit schlechter Formatierung.

```
double ergebnis = 3+9 / 3;
```

Auflistung 16. Schlecht formatierter Code

Auf den ersten Blick könnte man meinen es wird (3+9) / 3 berechnet und das Ergebnis wäre somit 4. Doch der Computer wird beim Ausführen dieser Code-Zeile 6 als Ergebnis berechnen. Hier ist der gleiche Code nochmals, allerdings mit einer etwas anderen Formatierung.

```
double ergebnis = 3 + 9/3;
```

Auflistung 17. Besser formatierter Code

Durch eine einfache Modifizierung des Code-Styles kann man nun mit einem Blick erkennen, dass durch die Punkt-vor-Strich Regel zunächst geteilt und dann erst addiert wird. Hierbei muss man allerdings anmerken, dass diese Formatierung nicht dem allgemeinen *Java Style Guide*[2] entspricht. Dieser sieht nämlich vor, jede Operation gleich, nämlich mit einem Leerzeichen zu trennen.

Das Code-Beispiel zeigt, wie die Formatierung uns denken lässt, dass der Code die Zahlen von 1 bis 10 in der Konsole ausgibt.

```
int i = 1;

while (i <= 10)

    System.out.println(i);

    i--;
```

Auflistung 18. Formatierung des Codes, die zu Fehlinterpretation neigt

In Wirklichkeit wird nur die Zahl 1 auf der Konsole ausgegeben und das Programm wird sich nicht beenden, denn die Anweisung *i--* wird nie ausgeführt werden. Durch die gleichen Einzüge der Anweisungen *System.out.println(i)* und *i--* werden wir allerdings dazu verleitet, dass wir annehmen beide Anweisungen gehören zu der While-Schleife. Mit der richtigen Formatierung jedoch sieht man direkt was wohin gehört und wie oft ausgeführt wird.

```
int i = 1;

while (i <= 10)

    System.out.println(i);

i--;
```

Auflistung 19. Formatierung des Codes, die den Programmablauf wiederspiegelt

[2] http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

Man sieht also, dass gute Formatierung des Programmcodes nicht nur die Lesbarkeit erhöht, sondern auch Fehler verhindert werden können. Man sollte seinen Programmcode deshalb immer schon während des Programmierens auch gleich formatieren [6].

4.3 Code-Refactoring

Unter dem Begriff des Refactorings versteht man das Überarbeiten bestehenden Codes ohne dabei dessen Funktionsweise zu verändern [8]. Es ist ein wichtiges Konzept beim Programmieren, das von jedem Softwareentwickler während bzw. nach dem Schreiben von Code angewandt werden sollte.

Das Code-Refactoring soll unter anderem die Lesbarkeit, Wartbarkeit, Erweiterbarkeit und Wiederverwendbarkeit des geschriebenen Codes erhöhen [9]. Wie bereits erwähnt verändert sich die Funktionsweise des Codes allerdings nicht und somit kann Code-Refactoring keine Fehler beheben [10].

Allerdings kann durch gutes Refactoring der Aufwand den man benötigt, um etwaige Fehler zu finden, beschleunigt werden. Außerdem kann der Programmcode eines zu Programmes so umgeschrieben werden, dass neue Funktionen integriert werden können, welche davor, aufgrund des Designs, nicht umsetzbar waren [8].

Im Folgenden werden die wichtigsten Techniken des Code-Refactorings vorgestellt.

Umbenennen. Umbenannt werden kann so ziemlich alles von Variablen-, über Methoden- bis hin zu Klassennamen [6]. Oft bekommen Variablen Bezeichner, die möglichst kurz sind um Schreibarbeit zu sparen und daher meist schwer zu lesen sind. Auch deshalb ist das Umbenennen wohl die am häufigsten angewandte Art des Refactorings und ist bei vielen Entwicklungsumgebungen über einen Schnellzugriff aufrufbar. Das Umbenennen der Variablen-, Methoden und Klassennamen wird dann auf Wunsch, auf alle Vorkommnisse dieses Namens angewandt und diese durch den neu eingegebenen Namen ersetzt [8].

Als Beispiel verwenden Schleifen oft einfache Buchstaben wie *i, j* oder *n* als Zählvariable, besser ist jedoch, wenn man der Zählvariablen einen treffenderen Namen gibt, der angibt, was gezählt wird. Im folgenden Beispiel wird ein zweidimensionales Integer-Array erstellt und in jedes Feld 0, oder falls *i* gleich *j* ist, eine 1 geschrieben.

```
int matrix[][] = new int[10][10];

for (int i = 0; i < matrix.length; i++) {

    for (int j = 0; j < matrix[i].length; j++) {
```

```
            matrix[i][j] = (i == j) ? 1 : 0;

        }

    }
```

Auflistung 20. Generalisierte Namen verwendet

Doch was stellt das Array *matrix* dar, wie sind *i* und *j* zu interpretieren? Nach einem „Umbenennen“ Refactoring der Variablen, sieht der gleiche Code schon aussagekräftiger aus.

```
int einheitsMatrix[][] = new int[10][10];

for (int reihe = 0; reihe < einheitsMatrix.length; reihe++) {

    for (int spalte = 0; spalte < einheitsMatrix[reihe].length; spalte++)
{

        einheitsMatrix[reihe][spalte] = (reihe == spalte) ? 1 : 0;

    }

}
```

Auflistung 21. Spezialisierte Namen verwendet

Nach dem Refactoring wird klar, dass hier die Einheitsmatrix erstellt und initialisiert wird, denn diese hat genau in ihrer Diagonalen Einsen (also genau dann, wenn Reihennummer gleich der Spaltennummer ist) und sonst nur Nullen. Verwendet man diese Matrix nun in einem anderen Teil des Programms muss man nicht extra Kommentare lesen oder im Programmcode nachschauen, welche Daten die Variable *matrix* denn genau hat, sondern kann dies auch direkt am Namen *einheitsMatrix* erkennen.

Zwar sollte man bereits während des Programmierens alle Namen sorgfältig wählen (Abschnitt 4.1), allzu oft jedoch will man erst ausprobieren, ob eine neue Idee auch wirklich funktioniert und ist darauf fixiert möglichst schnell, viel Code zu schreiben. Dabei wird oft wenig Rücksicht auf die Namen von z. B. temporären Variablen oder Methoden genommen. Dies ist sehr schlecht, denn aussagekräftige Namen helfen nicht nur beim Einarbeiten funktionierenden Codes, gerade auch beim Debuggen von defektem Code helfen treffende Namen um den Prozess logisch nachvollziehen zu können.

Extrahiere Methode. Diese Art des Refactorings sollte immer dann angewandt werden, wenn eines dieser drei Probleme auftritt:

1. Eine Methode ist zu lang, in objektorientierten Sprachen sollten Methoden nicht länger als ein Bildschirm hoch ist, sein.
2. Eine Methode erfüllt mehr als eine Aufgabe.
3. Es gibt Code-Duplikate

Oft gehen mehrere der oben aufgeführten Probleme einher. So sind Methoden oft sehr lang, die mehr als eine Aufgabe erfüllen oder sogar Code-Duplikate enthalten.

Durch das „Extrahiere Methode“ Refactoring wird ein Teil des Programmcodes in eine neue Methode ausgelagert. Diese neue Methode sollte dann auch nur eine Funktionalität haben. Hat sie dies nicht, so wird das gleiche Refactoring so lange wieder angewandt, bis die ursprüngliche und jede neue Methode, jeweils nur eine Funktionalität haben [6][8].

Das folgende Code-Beispiel zeigt eine Methode für das Abheben von Geld von einem Konto.

```
public boolean geldAbheben(double betrag) {

    if (betrag > maxGeldAbhebeBetrag) {

        return false;

    } else if (betrag < 0) {

        return false;

    } else {

        kontostand -= betrag;

        return true;

    }

}
```

Auflistung 22. Methode mit mehreren Funktionalitäten

Auch wenn die eigentliche Aufgabe der Methode das Abheben des Geldes ist, so muss sie auch verifizieren, dass der Geldbetrag überhaupt abgehoben werden kann. Dies ist eine andere Zuständigkeit und sollte daher mit dem „Extrahiere Methode“ Refactoring zu dem Folgenden umgeschriebenen werden.

```
public boolean geldAbheben(double betrag) {

    if (verifiziereBetrag(betrag)) {

        kontostand -= betrag;

        return true;

    } else {

        return false;

    }

}

public boolean verifiziereBetrag(double betrag) {

    if (betrag > maxGeldAbhebeBetrag) {

        return false;
```

```
        } else if (betrag < 0) {
            return false;
        } else {
            return true;
        }
    }
```

Auflistung 23. Methoden mit jeweils einer Funktionalität

Durch das Refactoring werden alle Code-Zeilen, die nicht zu der Funktionalität der eigentlichen Methode gehören in eine neue Methode ausgelagert, die in unserem Beispiel den abzuhebenden Betrag zunächst verifiziert. Diese neue Methode wird nun in der *geldAbheben* Methode aufgerufen. Dadurch hat sich nichts an der Ausführungsreihenfolge der Code-Zeilen geändert, aber jede Methode erfüllt nun nur eine Aufgabe.

Man sieht, dass man durch ein einfaches Refactoring die Lesbarkeit und Wiederverwendbarkeit von Methoden erhöhen kann, ohne ihre eigentliche Arbeitsweise zu verändern.

Extrahiere lokale Variable. Bei diesem Refactoring werden Zahlen oder Zeichenketten, die direkt im Programmcode verwendet werden, durch eine lokale, nur in der Methode sichtbare Variable ersetzt. Die Variable wird mit demselben Wert wie die zuvor verwendete Zahl bzw. Zeichenkette hatte, initialisiert.

Der Vorteil hierbei ist, dass man anstelle der reinen Zahlen oder Zeichenketten einen aussagekräftigen Namen für die Variable wählen kann. Falls sich die Anforderungen ändern kann man außerdem direkt am Anfang der Methode die gewünschten Variablen ändern [6][8].

Das folgende Beispiel berechnet den Umfang eines Kreises.

```
float umfang = 2 * Math.PI * 5f;
```

Auflistung 24. Radius direkt als Zahl geschrieben

Die Implementierung ist korrekt, jedoch könnte man die Zahl *5f* durch den Bezeichner *r* ersetzen und somit eine „Extrahiere lokale Variable “ Refactoring durchführt. Dadurch würde die implementierte Formel so aussehen, wie sie auch formal geschrieben wird, nämlich als 2 * Pi * r.

```
float r = 5f;

float umfang = 2 * Math.PI * r;
```

Auflistung 25. Radius in lokaler Variable gespeichert

Nach dem Refactoring steht die Formel in dem Programm, wie man sie auch sagt. Dies macht die Aussage der Formel verständlicher und gibt zusätzlich die Flexibilität, den Wert der Variable zentral anpassen zu können.

Extrahiere Feld. Dieses Refactoring kann bei objektorientierten Sprachen angewandt werden, falls Zahlen oder Zeichensequenzen direkt im Programmcode verwendet werden. Beim Refactoring ersetzt man das Vorkommen der Zahlen oder der Zeichenkette durch eine in der kompletten Klasse sichtbaren Variable, einem Feld, mit demselben Wert. Diese Variable, auch Feld genannt, wird an den selben Stellen wie vorher, in den Programmcode eingebunden.

Benötigt man den Wert nur innerhalb einer Methode, also als lokale Variable, so sollte man eher das „Extrahiere lokale Variable“ Refactoring anwenden.

Der Vorteil hierbei ist, dass man konstante Zahlen oder mehrmals verwendete Zeichenketten an einem Ort hat und diese im Zweifelsfall auch nur einmal ändern muss. Außerdem hat so jedes Feld und damit jede genutzte Zahl bzw. Zeichenkette einen wählbaren Namen, der den Sinn widerspiegelt [6][8].

Im folgenden Beispiel wird eine DateiZuGrossException geworfen, falls die Größe eines Byte-Arrays größer als die konstante Zahl 1024 ist. Doch weshalb ausgerechnet dieser Wert?

```
public class MyServer extends Server {

    public void checkSize(byte[] daten) throws DateiZuGrossException {

        if (byte.length > 1024) {

            throw new DateiZuGrossException();

        }

    }

    public void upload(byte[] daten) {...};

}
```

Auflistung 26. Zahl ohne Erläuterung verwendet

Offensichtlich stellt der Wert 1024 eine Größenbeschränkung für Daten dar. Doch der Sinn ist nicht direkt ersichtlich.

Das „Extrahiere Feld“ Refactoring zeigt den Sinn des Wertes und bringt gleichzeitig die Flexibilität, den Wert des Feldes dynamisch bei der Objekterzeugung einmal einzulesen und zu behalten oder auf Grund von anderen Code-Fragmenten weiter anzupassen.

```
public class MyServer extends Server {

    private int maxUploadGroesseInBytes = 1024;

    public void checkSize(byte[] daten) throws DateiZuGrossException {

        if (byte.length > maxUploadGroesseInBytes) {
```

```
            throw new DateiZuGrossException();

        }

    }

    public void upload(byte[] daten) {...};

}
```

Auflistung 27. Zahl in Feld gespeichert

Nach dem Refactoring sieht man nun, dass die Dateibegrenzung eine Upload Begrenzung ist. Benutzt man nun immer diese Variable für Programmfragmente, die mit Dateiupload zu tun haben, so kann der Wert im späteren Verlauf der Entwicklung einfach an einer zentralen Stelle in der Klasse angepasst werden. Außerdem wird der Sinn der Zahl durch den Namen der Variable verdeutlicht.

Extrahiere Superklasse. Ein weiteres wichtiges Refactoring bei objektorientierten Programmiersprachen ist, Felder und Methoden in eine Superklasse zu extrahieren, falls diese in mehreren Klassen vorhanden sind und auch gebraucht werden. Meist ist dies bei spezialisierten Klassen der gleichen Gruppierung der Fall. Dann erstellt man eine neue generalisierte Superklasse und zieht gleiche Felder und Methoden von den spezialisierten Klassen in die Superklasse [6].

Falls die Implementierungen der Methoden der spezialisierten Klassen sich untereinander unterscheiden, kann man entweder eine Standardimplementierung der Methoden in der Superklasse implementieren oder aber die Methoden als abstrakt deklarieren. Bei abstrakten Methoden wird nur die Methodensignatur festgelegt und die Implementierung muss spezifisch für jede abgeleitete Klasse implementiert werden.

Im folgenden Beispiel sehen wir die zwei Klassen Student und Dozent mit den gleichen Methoden, *schlafen* und *lernen*.

```
public class Student {

    public void lernen() {

        readBooks();

    }

    public void schlafen(int millis) throws InterruptedException {

        Thread.sleep(millis * 2);

    }

}

public class Dozent {

    public void lernen() {

        readBooks();
```

```
    }

    public void schlafen(int millis) throws InterruptedException {

        Thread.sleep(millis);

    }

}
```

Auflistung 28. Zwei nicht verwandte Klassen mit gleichen Methoden

Die Gruppierung der beiden Klassen ist unserem Fall der Mensch. Beim Refactoring wird nun zunächst die Klasse *Mensch* erstellt und die Methode *lernen* von den Klassen Lehrer und Schüler mitsamt ihrer Implementierung in die Superklasse verschoben. Nun wird noch die Methode *schlafen* in die Superklasse verschoben, da aber die Implementierung dieser Methode unterschiedlich in den beiden spezialisierten Klassen ist, wird dies eine abstrakte Methode. Damit muss nun auch die Superklasse selbst abstrakt sein.

Nach dem Refactoring sieht der Code mit derselben Funktionalität wie oben, folgendermaßen aus.

```
public abstract class Mensch {

    public void lernen() {

        readBooks();

    }

    public abstract void schlafen(int millis)

            throws InterruptedException();

}

public class Student extends Mensch {

    @Override

    public void schlafen(int millis) throws InterruptedException {

        Thread.sleep(millis * 2);

    }

}

public class Dozent extends Mensch {

    @Override

    public void schlafen(int millis) throws InterruptedException {

        Thread.sleep(millis);

    }
```

```
}
```

Auflistung 29. Zwei verwandte Klassen mit gemeinsamer Superklasse

Nach dem Refactoring werden die Struktur und die Beziehung der spezialisierten Klassen deutlich.

Übersicht. Die nachfolgende Tabelle zeigt eine Übersicht der vorgestellten Refactoring Methoden. In der ersten Zeile steht das jeweilige Refactoring. In der ersten Spalte stehen Problemfälle in denen es sich empfiehlt, ein Code-Refactoring durchzuführen. Die Haken zeigen, welches Refactoring bei welcher Probleminstanz angewandt werden sollte.

Tabelle 1. Übersicht der Refactoring Methoden

Probleminstanz	Umbenennen	Extrahiere Methode	Extrahiere lokale Variable	Extrahiere Feld	Extrahiere Superklasse
Unverständlicher Code	✔	✔	✔	✔	(✔)
Code-Duplikate		✔			✔
Werte und Zeichen-ketten hart codiert			✔	✔	
Gleiche Methoden-signatur					✔

5 Zusammenfassung

Es wird deutlich, dass sich Code-Qualität aus verschiedenen Faktoren zusammensetzt und sich anhand dieser auch verbessern lässt. Wer Code mit guter Qualität schreiben möchte, der sollte sich bereits vor dem eigentlichen Programmieren überlegen, wie er sein Programm logisch aufbaut und dennoch Platz für mögliche spätere Änderungen hat. Während des Programmierens sollte immer auf eine sinnvolle Formatierung, sowie aussagekräftige Bezeichner geachtet werden. Dies erleichtert es nicht nur für andere Entwickler den Code einfacher verstehen zu können, sondern es hilft auch einem selbst, einfache Fehler zu vermeiden.

Möchte man stattdessen schnell einen funktionierenden Code schreiben, so kann man durch Refactoring die Qualität des bereits geschriebenen Codes auch noch nachträglich verbessern. Auch wenn man besonders performante Programme benötigt, sollte man immer zunächst die Funktionalität bereitstellen und erst anschließend durch konkrete Messungen die Performance Probleme analysieren. Auf diese Weise kann man am effektivsten die Probleme aufdecken und im Anschluss beheben.

Literatur

1. Liggesmeyer P (2009) Software-Qualität: Testen, Analysieren und Verifizieren von Software, 2nd ed. Springer Spektrum
2. Joshua B (2008) Effective Java, 2nd ed. Prentice Hall PTR, Upper Saddle River, NJ, USA
3. Englander R (1997) Developing Java Beans. O'Reilly Media, Inc.
4. Information Hiding. http://www.itwissen.info/definition/lexikon/Information-Hiding.html. Accessed 3 Mar 2016
5. Küchlin W, Weber A (2013) Einführung in die Informatik: Objektorientiert mit Java. Springer-Verlag
6. McConnell S (2004) Code Complete, 2nd ed. Microsoft Press, Redmond, WA, USA
7. The Gestalt Principles. http://graphicdesign.spokanefalls.edu/tutorials/process/gestaltprinciples/gestaltprinc.htm. Accessed 3 Mar 2016
8. Enns R (2004) Refactoring in eclipse. Dep. Comput. Sci. Univ. Manitoba, Winnipeg, Manitoba, Canada, Tech. Rep
9. Mens T, Tourwé T (2004) A survey of software refactoring. Softw Eng IEEE Trans 30:126–139.
10. Opdyke WF (1992) Refactoring object-oriented frameworks. University of Illinois at Urbana-Champaign

Alle Links wurden zuletzt am 3. März 2016 geprüft.