

Anti-Pattern

Seminararbeit zu wissenschaftliches Arbeiten in der Informatik 1
Referentin: Prof. Dr. Inge Schestag

Christopher Hohberg
Hochschule Darmstadt – FB Informatik
christopher.hohberg@gmx.de

Kevin Rojczyk
Hochschule Darmstadt – FB Informatik
kevin.roj@p8h.de

15.06.2016

Zusammenfassung

In dieser Arbeit zum Thema Anti-Pattern, die in erster Linie eine Literatur Recherche ist, werden wir einen kurzen Überblick über die geschichtliche Entstehung des Begriffs *Anti-Pattern* geben sowie die Abgrenzung zu ähnlichen Begriffen erklären. Der Hauptteil besteht aus einer Übersicht zu einigen von uns ausgewählten Anti-Pattern. Zum Schluss folgt ein kurzer Abschnitt der die softwareunterstützte Erkennung von Anti-Pattern behandelt.

Inhaltsverzeichnis

1	Einleitung in Anti-Pattern	3
1.1	Geschichtliche Entstehung	3
1.2	Abgrenzung zu Design-Pattern	3
1.3	Abgrenzung zu “technische Schuld”	3
2	Anti-Pattern	4
2.1	... im Projekt-Management	4
2.1.1	Requirement Creep	4
2.1.2	Hidden Requirements	5
2.1.3	Train the Trainer	5
2.2	... in der Software-Architektur	6
2.2.1	The Blob	6
2.2.2	Swiss Army Knife	6
2.2.3	Spaghetti Code	7
2.3	... in der Softwareentwicklung	7
2.3.1	Lava Flow	7
2.3.2	Golden Hammer	8
2.4	... in der testgetriebenen Entwicklung	8
2.4.1	Second Class Citizens	9
2.4.2	The Free Ride / Piggyback	9
2.4.3	The Inspector	9
2.4.4	The Enumerator	9
2.5	... in spezifischen Entwicklungsumgebungen	9
2.5.1	Übertragbarkeit	9
2.5.2	Java	10
3	Hilfswerkzeuge zum Erkennen von Anti-Pattern	11
	Literatur	12

1 Einleitung in Anti-Pattern

“Ich kann freilich nicht sagen, ob es besser werden wird, wenn es anders wird; aber so viel kann ich sagen: es muss anders werden, wenn es gut werden soll.”

— Georg Christoph Lichtenberg, 1853 [1]

1.1 Geschichtliche Entstehung

Der grundlegende Gedanke hinter den Anti-Pattern, ein fehlerhaftes Designkonzept in der Softwareentwicklung zu identifizieren und zu bereinigen, wurde erstmals bereits 1975 von Frederick Brooks in seinem Buch *The Mythical Man-Month* [2] genannt, auch wenn dies nicht zur Etablierung des Begriffs führte.

Nach der Veröffentlichung des Buches *Patterns. Elements of Reusable Object-Oriented Software* [3], welches den *Design-Pattern* erstmals zu großer Bekanntheit in der Welt der Softwareentwicklung verhalf, griffen verschiedene Autoren die Idee von Brooks wieder auf, jedoch nannte keiner den Begriff *Anti-Pattern*. Das Wort *Anti-Pattern* wird zum ersten mal in der Arbeit *AntiPatterns: Vaccinations against Object Misuse* [4] von Michael Akroyd förmlich benutzt, welche er erstmals 1996 auf der *Object World West Conference* vorstellte. Danach wurden viele verschiedene Anti-Pattern im Bereich der Softwareentwicklung entdeckt und bekannt.

1.2 Abgrenzung zu Design-Pattern

Der Unterschied zwischen Design-Pattern und Anti-Pattern entsteht in erster Linie durch deren Zweck. So stellt ein Design-Pattern ein Konzept dar, welches bei der Lösung einer gängigen Problematik in der Softwareentwicklung das Grundgerüst bietet, und somit helfen soll den richtigen Ansatz zur Lösung zu finden.

Ein Anti-Pattern hingegen soll davor schützen einen groben Fehler bei der Lösung eines Problems zu begehen, denn es gibt ein Konzept an, das zwar zur Lösung eines bestimmten Problems führt, bei der Umsetzung jedoch neue Probleme generiert, die erst wesentlich später zur Geltung kommen und dann nur noch unter meist enormem Mehraufwand behoben werden können.

1.3 Abgrenzung zu “technische Schuld”

Nicht wortverwandt aber von der Bedeutung her sehr ähnlich ist der Begriff “technische Schuld”. Er beschreibt das bewusste Einplanen von schlecht geschriebenem Code und dessen Folgen in ein Softwareentwicklungsprojekt. Dagegen kann man *Anti-Pattern* als schlechte Code in Folge von Faulheit und Unprofessionalität bezeichnen. Somit ist die Abgrenzung zwischen Anti-Pattern und technischer Schuld gegeben durch den Zeitpunkt an dem der schlecht geschriebene Code auffällt. Ein Anti-Pattern fällt immer erst durch die aus ihm resultierenden Probleme auf. Der “technischen Schuld” ist man sich jedoch schon bewusst bevor der schlechte Code überhaupt geschrieben wurde. Technische Schulden werden also gemacht, wenn ein Entwicklerteam sich dazu entscheidet schnell unsauberen Code zu schreiben, anstatt ein Designkonzept zu verwenden was viel Zeit in Anspruch nehmen würde, um zum Beispiel einen Release-Termin einhalten zu können.

2 Anti-Pattern

Die Benennung und die genaue Beschreibung der Anti-Pattern unterscheidet sich in der Literatur häufig geringfügig. In dieser Arbeit wird versucht eine möglichst allgemeine und geläufige Beschreibung der Anti-Pattern zu verwenden. Um nicht noch mehr Verwirrung durch Übersetzungen zu stiften und die Vergleichbarkeit zur englischen Literatur zu erhalten, werden ausschließlich die etablierten, englischen Namen der Anti-Pattern verwendet.

Es wird hier nur eine Auswahl an Anti-Pattern erwähnt, weil es sonst den Rahmen dieser Arbeit sprengen würde. Unterteilt sind diese in fünf Kategorien. Die ersten drei Kategorien Projekt-Management, Software-Architektur und Softwareentwicklung sind die typischen Kategorien, die auch im Standardwerk Anti-Patterns [5] Erwähnung finden und die die größte Verbreitung haben. Die letzten zwei Kategorien testgetriebene Entwicklung und spezifische Entwicklungsumgebungen sind weniger weit verbreitet, werden aber mit aufgenommen da diese unseres Erachtens nicht minder wichtiger für einen Softwareentwickler sind.

Um eine höhere Flexibilität bei der Beschreibung der Anti-Pattern zu haben, wird auf eine feste Strukturierung der einzelnen Beschreibungen verzichtet. Allerdings ist diese sehr wohl vorhanden und verbreitet in der Literatur. Schon das erste Anti-Pattern Buch [5] führt eine feste Strukturierung ein. Weitere Arbeiten übernehmen diese Struktur oder lehnen sich daran an z.B. [6].

2.1 Anti-Pattern im Projekt-Management

2.1.1 Requirement Creep¹

Die Anforderungen oder der Anwendungsbereich werden ständig erweitert, ohne dass sie zuvor bei der Projektplanung beachtet wurden. Dies tritt auf, wenn der Kunde während der Entwicklung ständig neue Funktionen in das Projekt integriert haben möchte. Der Kunde hat in der Phase der Projektplanung noch keine klare Vorstellung über die Anforderungen, sie haben sich kurzfristig geändert, oder er versucht sich weitere Leistungen zu erschleichen.

Dieser Typ von Anti-Pattern kommt auch häufig bei der Übernahme von *legacy code* oder bei *code refactoring* vor, wobei in der Entwicklungsphase die Anforderungen für die nachfolgenden Projekte noch nicht genau bekannt waren oder das Wissen über die Funktionalität der Software zum Zeitpunkt der Wiederverwendung nicht mehr im Detail vorhanden ist.

Als Beispiel dient das amerikanische Weltraumprogramm der NASA und andere Weltraumprojekte während der 1990er Jahre, die erhebliche Rückschläge durch den überraschend hohen Verlust an Weltraumfahrzeugen und Experimenten hinnehmen mussten. Während das Wiederverwenden von mechanischen und elektrischen Konstruktionen durch Simulationen und Redundanz qualitätsgesichert werden kann, ist dies bei Software nur bedingt möglich [7]. Dies führte zum Beispiel zu teuren Fehlfunktionen beim *Mars Polar Lander* (MPL) der NASA, der mit zu hoher Geschwindigkeit auf dem Mars landete und zerschellte [8] oder der neuen Ariane 5, Flug 501². Die Rakete von der europäischen Weltraumbehörde

¹Auch bekannt unter dem Namen: *scoop creep* oder *feature creep*

²Flug 501 war der erste Start einer Ariane 5 Rakete

ESA, die kurz nach dem Start zerbrach, verwendete die Software der Vorgängerin Ariane 4, die nicht für die Anforderung der Ariane 5 entwickelt wurde [9].

2.1.2 Hidden Requirements

Erzeugt wird dieses Anti-Pattern, wenn die Anforderungen des Projekts ungenau oder überhaupt nicht dokumentiert sind. Dies kann aus Ignoranz, Unfähigkeit oder Böswilligkeit geschehen [10].

Eine Studie hat ergeben, dass dieses Pattern eines der häufigsten in der finnischen Software-Industrie ist [6]. Dieses Ergebnis lässt sich vermutlich auch auf andere europäische Länder übertragen.

Hidden Requirements kann im schlimmsten Fall zum Scheitern des Projekts führen, aber mindestens zum nachträglichen Mehraufwand, um die fehlenden Anforderungen zu evaluieren oder sie nachträglich hinzuzufügen. Bei einem böswilligen Motiv lässt sich damit ein Projekt sabotieren oder absichtlich einem Projektpartner schaden.

Begegnet werden kann diesem Problem mit einer genauen Spezifizierung der Anforderungen gemeinsam mit dem Kunden (*Design by Contract*³). Gegebenenfalls hilft es bei komplexen Themen, einen externen Berater mit dem nötigen Spezialwissen einzubeziehen.

Der Cartoon Abb. 1 auf der nächsten Seite verdeutlicht auf einfache Art, wie wichtig es ist, Anforderungen gründlich mit dem Kunden zu erfassen.

2.1.3 Train the Trainer

Dieses Anti-Pattern beschreibt die Situation, in der eine ganze Projektgruppe eine Fortbildung bekommt, auch wenn es ausreichen würde, einen Teil der Gruppe fortzubilden. Dieser Teil der Gruppe könnte somit sein Wissen an die anderen Gruppenmitglieder weitergeben.

Dies ist ein weiteres Anti-Pattern, das weite Verbreitung in der Software-Industrie [6] genießt. Der Grund ist oft, dass es einfacher ist die gesamte Gruppe zu einer Fortbildung zu schicken. Die Annahme ist, dass nach der Fortbildung alle gleich gut in die neue Thematik eingearbeitet sind. Des Weiteren gibt es dem Projektmanager die Sicherheit, dass der Verlust eines Projektmitglieds, zum Beispiel durch Krankheit oder Kündigung, nicht auch zum Verlust von projektwichtigem Spezialwissen führt.

Fast vollständig ignoriert wird dabei der allgemein bekannte Grundsatz aus dem Schulwesen, dass kleinere Klassen besser lernen können [13].

Oft würde es sich lohnen wenige, auserwählte Gruppenmitglieder intensiver fortzubilden, welche bei Bedarf ihr Wissen weiter geben können.

³*Design/Programming by Contract* bezeichnet die vertragsbasierende Softwareentwicklung [11].

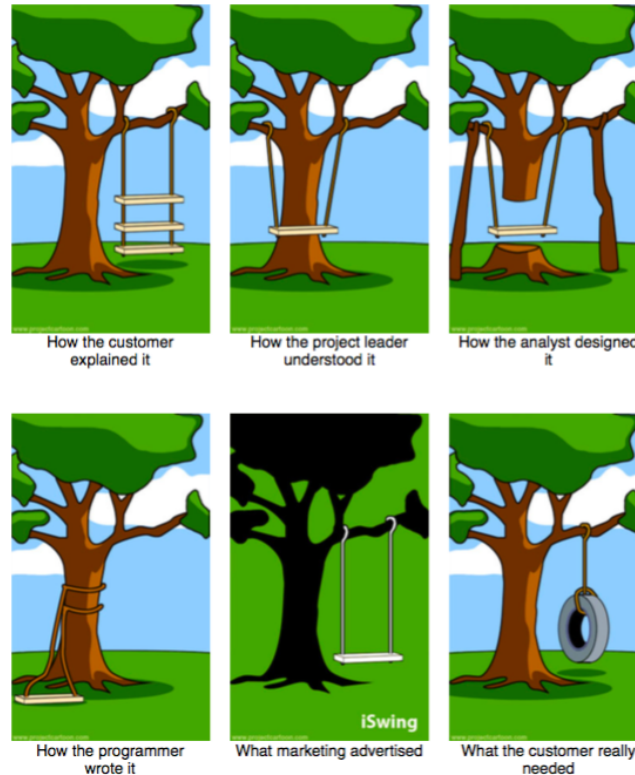


Abbildung 1: Unterschiedliches Verständnis der Anforderung [12]

2.2 Anti-Pattern in der Software-Architektur

2.2.1 The Blob

Auch unter dem Namen *God class* bzw. *God object* bekannt stellt der *Blob* in der objektorientierten Programmierung ein Objekt dar, das zu viel weiß bzw. tut. Damit verstößt es gegen die grundlegende Idee in der objektorientierten Programmierung, dass ein großes Problem in viele kleine zerlegt wird, und durch die Lösung der kleinen Probleme auch das Große gelöst wird.

Der *Blob* ist also ein Objekt das fast sämtliche Funktionalität eines Programs selbst beinhaltet, anstatt sie auf andere Objekte zu verteilen. Durch die starke Abhängigkeit des Programms vom *Blob* wird die Wartung sehr schwer und aufwändig, und damit auch sehr teuer.

Einhalten von gängigen Entwurfsmustern sowie Kapselung und Aufteilen von Verantwortlichkeiten innerhalb des Programms sind die besten Möglichkeiten einen *Blob* zu vermeiden bzw. zu entfernen.

2.2.2 Swiss Army Knife

Das *Swiss Army Knife* könnte man als nahen Verwandten des Blobs bezeichnen, denn auch hier geht es um eine einzelne Klasse, die mit Funktionalität überladen ist. Aber es gibt einen gravierenden Unterschied zwischen den beiden. Während der *Blob* eine monopolistische Stellung innerhalb der Architektur hat und alle Funktionalität und Daten

eigennützig verwendet, versucht das *Swiss Army Knife* seine Funktionalität für möglichst viele andere Klassen bereitzustellen.

Das *Swiss Army Knife Anti-Pattern* beschreibt ein über alle Maße komplexes Interface, bei dem der verantwortliche Designer versucht hat, für alle nur erdenklichen Anforderungen an die entsprechende Klasse, eine geeignete Schnittstelle einzubauen. Dies geschieht oft wenn ein Designer keinen klaren und eindeutigen Zweck für eine Klasse vor Augen hat. Zu Problemen kommt es, wenn andere Entwickler das Klasseninterface nutzen sollen, durch die enorme Komplexität aber nichteinmal verstehen, wozu die Klasse überhaupt zu gebrauchen ist. Auch bei Dokumentation und Debugging stellt die Komplexität ein großes Problem dar.

Eine Lösung für dieses Problem ist ein sogenanntes *profile*. Hierbei handelt es sich um dokumentierte Konventionen, die den Einsatz einer Technologie bzw. Klasse erklären und zum Beispiel eindeutige Werte angeben, die in den Parametern übergeben werden können. Des weiteren sollte das *profile* Ausführungsreihenfolge, Methoden Aufrufe und "exception handling" dokumentieren [14].

2.2.3 Spaghetti Code

Spaghetti Code bezeichnet ein Stück Quellcode, das durch viele, und vor allem überflüssige Sprunganweisungen geprägt ist. Meist ist der Codeblock sehr kompakt, und da kein Anfang und Ende auf den ersten Blick zu erkennen ist, liegt der Vergleich mit einem Topf Spaghetti sehr nahe, daher auch der Name.

Jedoch muss Software, die *Spaghetti Code* aufweist nicht zwangsläufig schlecht funktionieren oder besonders unperformant sein. Der größte Makel liegt hier in der Wartung und Wiederverwertung des Codes. Die Wartung ist aufwändig, unübersichtlich und nimmt viel Zeit in Anspruch, dadurch wird sie unnötig teuer. Die Wiederverwertung von *Spaghetti Code* ist fast gänzlich ausgeschlossen da viel zu viel Zeit damit verbracht werden muss, den Code überhaupt erst einmal zu verstehen, da macht es mehr Sinn die Funktionalität neu zu implementieren.

Spaghetti Code entsteht oftmals bei unerfahrenen Programmierern, die einfach drauf los programmieren ohne sich vorher Gedanken über eine Struktur zu machen. Wenn man sich jedoch an grundlegende Verhaltensregeln der objektorientierten Programmierung hält und seinen Code in überschaubaren Funktionen bzw. Methoden fasst kann man dem Entstehen von *Spaghetti Code* gut vorbeugen. Eine weitere Maßnahme ist es, sich vor dem Programmieren eines Abschnitts klare Gedanken über dessen Funktionalität zu machen und vielleicht sogar eine kleine Skizze anzufertigen.

2.3 Anti-Pattern in der Softwareentwicklung

2.3.1 Lava Flow

Lava Flow Code wird am häufigsten in Systemen gefunden die als Prototyp oder Forschungszweck gedacht waren, am Ende aber doch zum fertigen Produkt weiterentwickelt wurden. Er zeichnet sich durch Codeblöcke aus, die noch von alten alpha-Versionen des Produkts stammen.

Wegen mangelnder Dokumentation, kann bei diesen Codeblöcken meistens Niemand mehr so richtig sagen wozu sie eigentlich gut sind, und ob sie überhaupt noch gebraucht werden. Da sich auch Niemand traut den Code einfach zu entfernen, da er ja vielleicht

doch noch irgendwo Verwendung hat, wird er bei jedem Test mit compiliert und geladen und führt so zu unnötig langen Wartezeiten. Außerdem muss der *Lava Flow* Code auch im Endprodukt geladen werden und kann so die Performance des Systems beeinträchtigen.

Eine Anekdote, die dieses Problem sehr gut verdeutlicht, stammt von einem Blogger, der unter dem Namen “sourcemaking“ bekannt ist. Es ist eine Unterhaltung die er so, oder so ähnlich mit einem Kollegen geführt haben soll.

“Oh that! Well Ray and Emil (they’re no longer with the company) wrote that routine back when Jim (who left last month) was trying a workaround for Irene’s input processing code (she’s in another department now, too). I don’t think it’s used anywhere now, but I’m not really sure. Irene didn’t really document it very clearly, so we figured we would just leave well enough alone for now. After all, the bloomin’ thing works doesn’t it?!“ [15]

2.3.2 Golden Hammer

Ein Team von Softwareentwicklern, das eine hohe Kompetenz mit einer bestimmten Herangehensweise für Probleme oder einem Produkt erreicht hat, neigt oft dazu dies als den sogenannten *Golden Hammer* zu sehen.

Das Team verwendet diesen *Golden Hammer* als Allzweckwaffe, und versucht ihn für jedes neue Projekt einzusetzen, übersieht dabei jedoch dass sich nicht jedes Projekt auf die gleiche Weise lösen lässt. Anstatt sich jetzt neue, besser passende Lösungsansätze zu erarbeiten wird lieber versucht neue Kunden vom *Golden Hammer* zu überzeugen und das zu entwickelnde Produkt anzupassen.

Verglichen mit anderen, ähnlichen Lösungen aus der Softwareindustrie leiden solche Produkte oft unter Performance- und Anpassungsproblemen. Desweiteren isolieren sich die verantwortlichen Entwickler vom Rest der Industrie, da sie große Lücken im Wissen über den Umgang mit alternativen Lösungsansätzen haben, und sich nicht weiterentwickeln [16].

2.4 Anti-Pattern in der testgetriebenen Entwicklung

Die *Agile Softwareentwicklung* gewinnt zurzeit immer mehr an Bedeutung [17]. Im Zuge dessen rückt auch die Softwareentwicklung mit intensiver Nutzung von Tests, die *testgetriebene Entwicklung* oder auch *test-driven development (TDD)*, stärker in den Fokus [18].

Die großen Werke über Anti-Pattern besitzen keine eigenen Kategorien für Anti-Pattern in der testgetriebenen Entwicklung. Um der Wichtigkeit von TDD in heutiger Zeit gerecht zu werden, ist deshalb diese Kategorie ein Bestandteil dieser Arbeit.

Wir zitieren hier eine Sammlung von Anti-Pattern aus zwei verschiedenen Sekundärquellen. Die erste Quelle ist die Webseite von James Carr [19], auf der eine Auflistung von Anti-Pattern veröffentlicht ist, die zum Teil mit einer Mailingliste auf *yahoogroup* erarbeitet sind.

Die zweite Quelle ist ein Beitrag auf *Stack Overflow* [20], in der der Benutzer aufgefordert wurde, bekannte Anti-Pattern zu veröffentlichen. Des Weiteren sollten sie Abstimmen, welche Anti-Pattern am häufigsten vorkommen. Es wurden 31 Anti-Pattern gesammelt und das Anti-Pattern mit den meisten Stimmen *Second Class Citizens* hat 70 Stimmen, wobei angemerkt werden muss, dass Benutzer auch negative Stimmen verteilen können.

2.4.1 Second Class Citizens

Der Testcode ist schlecht gewartet und daher von geringerer Qualität als der zu testende Code. Der Testcode enthält unnötige Duplikate und ist unleserlich.

Dies ist ein typisches Phänomen, wenn der Testcode als weniger wichtig wahrgenommen wird, als der zu testende Code. Die Weiterentwicklung des Testcodes und der eigentlichen Software wird dadurch aber unnötig erschwert.

2.4.2 The Free Ride / Piggyback

Um eine neue, unabhängige Funktionalität zu testen wird eine neue *Assertion* in eine bestehende Testmethode hinzugefügt, statt eine neue Testmethode zu erstellen.

Schlägt nun eine Testmethode fehl, kann womöglich keine direkte Aussage mehr getroffen werden, welche Funktionalität nicht mehr gegeben ist.

2.4.3 The Inspector

Es wird ein Unittest beschrieben, welcher die Regel der Kapselung ignoriert, um 100% Codeabdeckung zu erreichen und dabei eine große Menge an internen Informationen des zu testenden Codes beinhaltet.

Die Wartung und Weiterentwicklung des bestehenden Codes führt schnell zum Scheitern des Unittests. Dabei muss sich die zu testende Funktionalität „von außen“ nicht geändert haben und kann immer noch richtig sein.

2.4.4 The Enumerator

Die Namen der Testmethoden sind alle gleich und werden einfach nur durchnummeriert. Zum Beispiel test1, test2, test3,

Die Namen der Testmethoden haben dadurch keine Aussagekraft. Erst das mühselige Studieren der Testmethoden verrät ihren Zweck.

2.5 Anti-Pattern in spezifischen Entwicklungsumgebungen

In dieser Kategorie geht es um Anti-Pattern, die nur bei bestimmten Programmiersprachen eine Wirkung haben.

2.5.1 Übertragbarkeit

Platformspezifische Anti-Pattern können, müssen aber nicht übertragbar sein. Als Beispiel nehmen wir hier ein JavaScript Anti-Pattern [21], welches für ein C++ Programm keine allgemeine Geltung hat.

Listing 1: for-Schleifen in JavaScript

```
// sub-optimal loop
for (var i = 0; i < myarray.length; i++) {
    // do something with myarray[i]
}

// optimization 1
// cache the length of the array with the use of 'max'
```

```
for (var i = 0, max = myarray.length; i < max; i++) {
    // do something with myarray[i]
}
```

Diese Optimierung speichert den Wert von *myarray.length* zwischen und verschiebt zusätzlich die Variable in den schneller erreichbaren lokalen for-loop Kontext [22]. Ein äquivalentes C-Programm braucht nicht nach dem selben Schema optimiert zu werden. Ein moderner Compiler führt diese und weitere Optimierungen, wie *loop-unrolling*, beim Kompilieren automatisch aus [23].

Hier zeigt sich, wie wichtig es für einen Softwareentwickler ist, die technische Umsetzung der verwendeten Programmiersprache zu verstehen. Erst damit kann man Anti-Pattern, die zum Beispiel die Performanz betreffen, auch wirklich verstehen.

2.5.2 Java

Im folgenden beschreiben wir einige der, unserer Meinung nach, interessantesten Anti-Pattern für Java. Titel und teilweise auch der Beispiel-Code sind getreu vom Artikel "Java Anti-Patterns" von der Webseite odi.ch [24] übernommen.

Testing for string equality

Der Vergleich von einem String birgt insbesondere für Java-Neulinge einen Stolperstein. Ein Vergleichen über den „==“-Operator vergleicht lediglich die Referenzen der String-Objekte aber nicht den Inhalt.

Listing 2: Anti-Pattern: String-Vergleich

```
if (name == "John") ...
if (name.compareTo("John") == 0) ...
if (name.equals("John")) ...
```

Die Methode *compareTo* ist zwar richtig aber im Gegensatz zu der Methode *equals* langsamer. Allerdings wird im Anti-Pattern Beispiel das *equals* 'falsch herum' angewendet. Falls die Variable *name* den Wert *NULL* hat, führt dieser Aufruf zur einer *NullPointerException*.

Listing 3: String-Vergleich richtig

```
if ("John".equals(name)) ...
```

String concatenation

Listing 4: Anti-Pattern: Strings verknüpfen

```
String s = "";
for (Person p : persons) {
    s += "," + p.getName();
}
```

Ein häufig vorkommendes Anti-Pattern ist das typische iterative Verknüpfen von Strings. Es ist funktional richtig, aber skaliert schlecht bei einer großen Anzahl von Konkatenierungen, da der zugewiesene Speicher für die Variable ständig erweitert werden muss. Besser ist die Nutzung der Klasse *StringBuilder* mit einer passenden Buffer-Größe.

Listing 5: Strings verknüpfen richtig

```
// well estimated buffer
StringBuilder sb = new StringBuilder( persons.size() * 16);

for (Person p : persons) {
    sb.append(p.getName());
}
```

Platform dependent filenames

Listing 6: Anti-Pattern: Betriebssystemabhängige Pfade

```
File tmp = new File("C:\\Temp\\1.tmp");
```

Während dieses Anti-Pattern auf Rechnern mit Windows funktioniert, wird die Ausführung des Java-Programms auf einem Unix-System scheitern. Java selbst ist zwar „plattformunabhängig“, dies bedeutet aber trotzdem, dass die Pfade für die jeweiligen Systeme angepasst werden müssen.

Eine Lösung könnte sein, den von Java zur Verfügung gestellten Separator zu nutzen oder statt der Angabe des Pfades als String den Pfad mit den Methoden von *File* zusammen zu bauen.

Listing 7: Betriebssystemunabhängige Pfade

```
File f = new File(path + File.separatorChar + filename);
// or even better
File dir = new File(path);
File f = new File(dir, filename);
```

Unbuffered streams

Listing 8: Anti-Pattern: Nicht gepufferter Datenstrom

```
InputStream in = new FileInputStream( file );
int b;
while ((b = in.read()) != -1) {
    ...
}
```

Ein ungepufferter Datenstrom, zum Beispiel ein *FileInputStream*, ist bei häufigen, sukzessiven Zugriffen extrem ineffizient. Jeder einzelne *read*- oder *write*-Aufruf führt direkt zu einer blockierenden I/O-Aktion auf dem *langsamen* Dateisystem.

Abhilfe bringt die Nutzung eines Puffers, zum Beispiel das schon ab *Java Development Kit 1* enthaltene *BufferedInputStream*.

Listing 9: Gepufferter Datenstrom

```
InputStream in
    = new BufferedInputStream(new FileInputStream( file ));
```

3 Hilfswerkzeuge zum Erkennen von Anti-Pattern

Anti-Pattern lassen sich auch softwareunterstützt erkennen. Insbesondere im Quelltext lassen sich Anti-Pattern gut detektieren, wie bereits bestehende Arbeiten zeigen [25].

Es gibt mehrere Anbieter, die online den eigenen Quellcode auf Qualität überprüfen. Auch wenn sie nicht immer explizit darauf hinweisen, erkennen sie unter anderem auch Anti-Pattern. Dienste, wie *codeclimate.com* [26], benutzen dafür schon bestehende Tools die im Fachjargon auch *Linters*⁴ genannt werden. Der Dienst *quantifiedcode.com* [27] benutzt zusätzlich noch eine Sammlung eigener Anti-Patterns, die der User ergänzen und anpassen kann.

Diese Tools sind für Open-Source Projekte kostenlos und die Verwendung führt zu einem einheitlichen Code-Style und früherem Erkennen einer großen Anzahl von Anti-Pattern. Ersetzen kann dies aber den Code-Review eines professionellen Programmierers nicht. Dennoch lohnt sich der Einsatz meistens, denn noch wichtiger als das Verhindern von Anti-Pattern ist es, zu wissen wo und welche Anti-Pattern sich im Projekt befinden. Siehe auch dazu Abschnitt 1.3 zu *technische Schuld*.

Literatur

- [1] Georg Christoph Lichtenberg's *vermischte Schriften*. Bd. 1. Verlag Dieterich (siehe S. 3).
- [2] Frederick Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975 (siehe S. 3).
- [3] Erich Gamma, Richard Helm, Ralph E. Johnson u. a. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, Okt. 1994 (siehe S. 3).
- [4] Michael Akroyd. „AntiPatterns: Vaccinations against Object Misuse“. In: San-Jose, Object World West Conference, Aug. 1996 (siehe S. 3).
- [5] William J. Brown, Raphael C. Malveau, Hays W. McCormick III u. a. *AntiPatterns - Refactoring Software, Architectures, and Projects in Crisis*. Wiley, New York, 1998 (siehe S. 4).
- [6] Raptopoulou Charikleia, Berki Eleni, Poranen Timo u. a. *Management Anti-patterns in Finnish Software Industry*. Department of Informatics, Aristotle University of Thessaloniki und School of Information Sciences, University of Tampere (siehe S. 4 f.).
- [7] Nancy G. Leveson. „Software challenges in achieving space safety“. In: (2009). URL: <http://dspace.mit.edu/handle/1721.1/58930> (besucht am 18.04.2016) (siehe S. 4).
- [8] JPL Special Review Board. *Report on the Loss of the Mars Polar Lander and Deep Space 2 Missions*. NASA Jet Propulsion Laboratory, 22. März 2000. URL: http://spaceflight1.nasa.gov/spacenews/releases/2000/mpl/mpl_report_1.pdf (besucht am 22.05.2016) (siehe S. 4).
- [9] Report by the Inquiry Board. *ARIANE 5 Failure - Full Report*. ESA, 19. Juli 1996. URL: <http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html> (besucht am 22.05.2016) (siehe S. 5).
- [10] *Hidden Requirements*. ManagementAntiPatterns. URL: <http://www.c2.com/cgi/wiki?HiddenRequirements> (besucht am 22.05.2016) (siehe S. 5).

⁴Lint/Linters dienen zur statischen Code-Analyse um z.B. das Quelltext-Layout und die Syntax zu prüfen. Sie sind in der Regel Programmiersprachen abhängig.

- [11] *Design by contract*. In: *Wikipedia*. Page Version ID: 148851256. 7. Dez. 2015. URL: https://de.wikipedia.org/w/index.php?title=Design_by_contract&oldid=148851256 (besucht am 23.05.2016) (siehe S. 5).
- [12] *Project Cartoon: Hidden Requirement*. URL: <http://projectcartoon.com/cartoon/1519210> (besucht am 22.05.2016) (siehe S. 6).
- [13] Dr. Christoph Paulus. „Der Einfluss der Klassengröße auf Schülerleistungen an Grundschulen“. In: (2009). URL: <http://psydok.sulb.uni-saarland.de/volltexte/2009/2369/> (besucht am 23.05.2016) (siehe S. 5).
- [14] SourceMaking. *Swiss Army Knife*. Englisch. URL: <https://sourcemaking.com/antipatterns/swiss-army-knife> (besucht am 11.06.2016) (siehe S. 7).
- [15] SourceMaking. *Lava Flow*. Englisch. URL: <https://sourcemaking.com/antipatterns/lava-flow> (besucht am 13.06.2016) (siehe S. 8).
- [16] SourceMaking. *Golden Hammer*. Englisch. URL: <https://sourcemaking.com/antipatterns/golden-hammer> (besucht am 13.06.2016) (siehe S. 8).
- [17] *Agile Softwareentwicklung*. In: *Wikipedia*. Page Version ID: 154268147. 10. Mai 2016. URL: https://de.wikipedia.org/w/index.php?title=Agile_Softwareentwicklung&oldid=154268147 (besucht am 11.06.2016) (siehe S. 8).
- [18] *Testgetriebene Entwicklung*. In: *Wikipedia*. Page Version ID: 153622107. 19. Apr. 2016. URL: https://de.wikipedia.org/w/index.php?title=Testgetriebene_Entwicklung&oldid=153622107 (besucht am 11.06.2016) (siehe S. 8).
- [19] James Carr. *TDD Anti-Patterns — James Carr*. URL: <http://blog.james-carr.org/2006/11/03/tdd-anti-patterns/> (besucht am 19.04.2016) (siehe S. 8).
- [20] *Unit testing Anti-patterns catalogue - Stack Overflow*. URL: <http://stackoverflow.com/questions/333682/unit-testing-anti-patterns-catalogue> (besucht am 19.04.2016) (siehe S. 8).
- [21] *JavaScript Patterns*. URL: <https://shichuan.github.io/javascript-patterns/> (besucht am 09.06.2016) (siehe S. 9).
- [22] Nicholas C. Zakas. *High performance JavaScript*. 1st ed. OCLC: ocn460060155. Sebastopol, CA: O'Reilly, 2010. 209 S. ISBN: 978-0-596-80279-0 (siehe S. 10).
- [23] *Optimize Options - Using the GNU Compiler Collection (GCC)*. URL: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html> (besucht am 09.06.2016) (siehe S. 10).
- [24] Ortwin Glück. *Java Anti-Patterns - odi.ch*. URL: <http://odi.ch/prog/design/newbies.php> (besucht am 22.04.2016) (siehe S. 10).
- [25] Tse-Hsun Chen, Weiyi Shang, Zhen Ming Jiang u. a. „Detecting performance anti-patterns for applications developed using object-relational mapping“. In: *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, S. 1001–1012. URL: <http://dl.acm.org/citation.cfm?id=2568259> (besucht am 10.06.2016) (siehe S. 11).
- [26] *Available Analysis Engines · Code Climate*. Code Climate. URL: <https://docs.codeclimate.com/docs/list-of-engines> (besucht am 10.06.2016) (siehe S. 12).

- [27] *The Little Book of Python Anti-Patterns — Python Anti-Patterns documentation.*
URL: <http://docs.quantifiedcode.com/python-anti-patterns/> (besucht am 10.06.2016) (siehe S. 12).

Erklärung

Wir versichern hiermit, dass wir die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht.

Die Zeichnungen oder Abbildungen in dieser Arbeit sind von uns selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Christopher Hohberg

Datum, Ort

Kevin Rojczyk

Datum, Ort