16 Bad Smells

In diesem Kapitel werfen wir einen Blick auf verbreitete Fallstricke und Probleme, die einem Entwickler immer mal wieder begegnen. Man spricht in diesem Zusammenhang auch von »Bad Smells«. Das bezeichnet Abschnitte des Sourcecodes, die im übertragenen Sinne einen schlechten Geruch verbreiten – an denen potenziell etwas »faul« ist. Dieser eingängige Begriff wurde von Kent Beck und Martin Fowler im Buch »Refactoring« [20] zur Beschreibung derartiger Programmabschnitte eingeführt.

In diesem Kapitel werden Sie erfahren, welche Tücken und Fehler sich leicht in den Sourcecode einschleichen. Wenn wir diese Bad Smells erkennen, lokalisieren und verstehen können, sind wir auch in der Lage, Gegenmaßnahmen zu ergreifen. In den folgenden Abschnitten präsentiere ich dazu einen Katalog von Bad Smells, der sich in die Abschnitte 16.1 »Programmdesign«, 16.2 »Klassendesign« und 16.3 »Fehlerbehandlung und Exception Handling« gliedert und diverse Sourcecode-Auszüge unterschiedlicher Qualität zeigt. Wir analysieren jeweils die darin enthaltenen Probleme und schulen damit Ihr Auge. Das Erkennen eines Problems ist gut, allerdings sollten wir auch in der Lage sein, es zu beheben. Daher werden zu jedem Bad Smell passende Tipps zur Vermeidung gegeben und kleinere Abhilfemaßnahmen sofort vorgestellt. Allgemeingültige Umbaumaßnahmen stelle ich separat in Kapitel 17 »Refactorings« vor. Den Abschluss dieses Kapitels bilden in Abschnitt 16.4 »Häufige Fallstricke« einige Programmiertücken, die nicht die Schwere eines Bad Smells besitzen, aber dennoch hier erwähnt werden sollen, um diese in eigenen Programmen zu vermeiden.

Bad Smells am Beispiel

Bekanntermaßen beschreiben Bad Smells Sourcecode-Abschnitte, an denen möglicherweise etwas nicht in Ordnung ist. Diese Programmteile vergrößern die Gefahr für Fehlfunktionen oder enthalten bereits Fehler – manchmal sind solche Sourcecode-Stellen aber auch akzeptabel. In letzterem Fall sollte man dies durch einen Kommentar beschreiben.

Mit ein wenig Übung springen beim Analysieren von Sourcecode potenziell gefährliche Stellen schnell ins Auge. Außerdem ist es in der Regel so, dass Bad Smells gehäuft auftreten. Zum besseren Verständnis betrachten wir ein Beispiel mit der folgenden Methode `setDataState(int, OperationState)`:

```
public void setDataState(final int department, final OperationState state)
{
    if (this.dataState == null)
    {
        this.dataState = new HashMap<Integer, OperationState>();
    }
    this.dataState.put(new Integer(department), state);

    if (log.isInfoEnabled())
    {
        String msg = "device data state for department " + department + ": ";
        switch (state)
        {
            case RUNNING:
                msg += "running";
                break;
            case GOING_DOWN:
                msg += "going down";
                break;
            case DOWN:
                msg += "down";
                break;
            case GOING_UP:
                msg += "going up";
                break;
            default:
                msg += "unknown";
        }
        log.info(msg);
    }
}
```

Diese Methode scheint zunächst durchaus in Ordnung. Allerdings offenbaren sich dem geübten Auge hier unter anderem folgende Schwachstellen:

1. **Fehlende Plausibilitätsprüfungen der Parameter** – Es lauern Probleme durch unerwartete Parameterwerte: Nur ein kleiner Teil des `int`-Wertebereichs des Parameters `department` entspricht gültigen Werten. Außerdem ist für den Parameter `state` der Wert `null` ungültig und sollte zurückgewiesen werden.
2. **Unpassender Einsatz von Lazy Initialization** – Die Gefahr für mögliche Multithreading-Probleme wird durch den unsynchronisierten Einsatz von Lazy Initialization[1] für das Attribut `dataState` erhöht. Tatsächlich traten diese Probleme in der Praxis auf und führten zu Inkonsistenzen in den gespeicherten Werten.
3. **Unausgewogenheit beim Logging** – Der Logging-Code dominiert die Funktionalität: Nur die ersten Programmzeilen tragen zur Anwendungsfunktionalität bei. Die restlichen Zeilen bereiten Log-Informationen auf und sollten in eine Methode herausfaktoriert werden, um die Struktur klarer erkennbar zu machen. Ergänzend bietet es sich an, das `switch-case` durch eine Lookup-Map (vgl. Abschnitt 6.1.10) zu ersetzen.

[1] Lazy Initialization beschreibt die Technik, ein Attribut nicht direkt zu initialisieren, sondern zunächst mit `null` zu belegen, um die Konstruktion komplexer Objekte zu vermeiden. Erst beim ersten Zugriff erfolgt dann eine Initialisierung. Kapitel 22 beschreibt diese Technik im Detail.

16.1 Programmdesign

Nach diesem einführenden Beispiel stelle ich nun einen Katalog von Bad Smells vor und beginne dabei mit möglichen Problemen im Programmdesign.

16.1.1 Bad Smell: Verwenden von Magic Numbers

Oftmals kommen an beliebigen Stellen im Sourcecode verschiedene Zahlenwerte vor, die explizit als Zahlenliteral – als sogenannte ***Magic Number*** – angegeben werden. Zum Teil existieren funktionale und semantische Abhängigkeiten zwischen diesen Werten, die jedoch nicht im Sourcecode, sondern bestenfalls durch Kommentare ausgedrückt werden können.

Wir betrachten hier eine Klasse `CommandExecutor`, die im Konstruktor einen `int`-Wert erhält, der steuert, wie Kommandos verwaltet und einer internen Datenstruktur hinzugefügt werden. Für diesen Zweck gibt es mehrere Konstanten, unter anderem die Konstante `ADD_AS_LAST` mit dem Wert `2`. Im folgenden Beispiel werden zwei Instanzen der Klasse `CommandExecutor` erzeugt, sowohl unter Verwendung der Konstantendefinition als auch durch Angabe des korrespondierenden Zahlenwerts:

```
final CommandExecutor executor1 = new CommandExecutor(ADD_AS_LAST);

final CommandExecutor executor2 = new CommandExecutor(2);  // Magic Number 2
```

Warum ist das ein Bad Smell? Bereits anhand dieses einfachen Beispiels sieht man, dass die Lesbarkeit unter dem Einsatz von Magic Numbers leidet. Das liegt daran, dass ein Zahlenwert wenig semantische Aussagekraft besitzt. Erschwerend kommt hinzu, dass Änderungen an den Werten – etwa wenn obige Konstante z. B. später den Wert 7 erhält – überall im nutzenden Sourcecode nachgezogen werden müssen. Das kann sich aufwendig und fehleranfällig gestalten, alle betroffenen Vorkommen einer Magic Number lediglich anhand ihres Werts aufzuspüren: Logischerweise repräsentiert nicht jedes Vorkommen der Zahl `2` die Konstante `ADD_AS_LAST`. Wird bei einer solchen Korrektur eine Stelle übersehen, so kann dies zu unerwarteten Resultaten und schwer zu findenden Fehlern führen: Wird irgendwann einmal der Wert der Konstante auf `4711` verändert und eine andere Konstante erhält den Wert `2`, so besagt diese `2` nun beispielsweise das Einfügen an erster Stelle und verändert den Programmablauf damit komplett. Wird dagegen eine andere Programmstelle versehentlich geändert, so kommt es dort zu Fehlern im Programmverhalten, weil der neue Wert (die `4711`) dort keinen Sinn ergibt und möglicherweise sogar außerhalb des zulässigen Wertebereichs liegt.

Tipps und Refactorings Mit sprechenden Konstantennamen kann man leicht ausdrücken, was bewirkt werden soll, hier das Einfügen an letzter Position. Zudem stellen nachträgliche Änderungen der Werte kein Problem im nutzenden Sourcecode dar. Ein weiterer Vorteil der Verwendung von Konstanten ist, dass diese im Nachhinein, wie

in diesem Beispiel sicher sinnvoll, leicht und ohne größere Änderungen in eine `enum`-Aufzählung umgewandelt werden können. Wenn jedoch Magic Numbers verwendet wurden, dann sollte man jedes Vorkommen überprüfen und gegebenenfalls anpassen.

16.1.2 Bad Smell: Konstanten in Interfaces definieren

Java erlaubt es, Konstanten in einem Interface zu definieren. Dabei lassen sich zwei Varianten unterscheiden. In der ersten wird ein Interface ausschließlich zur Definition von Konstanten verwendet, hier durch das Interface `FigureConstants` gezeigt:

```
//ACHTUNG: Interfaces (möglichst) so nicht verwenden
public interface FigureConstants
{
    // Randbreite und -höhe
    int BORDER_WIDTH  = 5;
    int BORDER_HEIGHT = 5;

    // Abstand der Punkte im Raster in X- und Y-Richtung
    int GRID_SIZE_X   = 20;
    int GRID_SIZE_Y   = 20;
}
```

In der zweiten Variante – hier am Beispiel des Interface `FigureIF` – werden sowohl Konstanten als auch Methoden definiert:

```
//ACHTUNG: Interfaces (möglichst) so nicht verwenden
public interface FigureIF
{
    // Rückgabewerte für hitTest
    int BORDER_HIT   = 0;
    int TITLE_HIT    = 1;
    int SIZE_BOX_HIT = 2;

    int hitTest(final int x, final int y);
    void draw(final Graphics g);
}
```

Warum ist das ein Bad Smell? Die Definition von Konstanten in Interfaces ist in vielerlei Hinsicht ungünstig. Zunächst ist durch die JLS jede dort definierte Konstante implizit `public static final` und jede Methode `public abstract`. Daher kann die Sichtbarkeit nicht eingeschränkt werden. Die Situation hat sich mit Java 8 und 9 leider nicht zum Guten geändert: Seit Java 8 sind neben Defaultmethoden auch statische Methoden in Interfaces erlaubt (vgl. Abschnitt 4.2). Mit Java 9 lassen sich sogar private Methoden in Interfaces definieren, was allerdings kein schönes Design ist.

Zudem ist es möglich, ein Konstanten-Interface durch eine Klasse zu implementieren. Dadurch werden die Konstantennamen in den Namensraum der jeweiligen Klasse aufgenommen, und somit geht der Verweis auf den tatsächlichen Definitionsort der Konstante verloren. Für Konstanten als Bestandteile eines Methoden deklarierenden Interface besteht das Problem gleichermaßen.

In der folgenden Klasse `BadFigure`, die zu Demonstrationszwecken das Interface `FigureConstants` implementiert, ist dies in der Methode `isInside(int, int)` mit der Verwendung der Konstanten `BORDER_WIDTH` und `BORDER_HEIGHT` gezeigt, wobei zum einen eine Referenzierung über `FigureConstants` und zum anderen über `BadFigure` erfolgt:

```
// ACHTUNG: Interfaces (möglichst) so nicht verwenden
public class BadFigure implements FigureConstants
{
    final Rectangle boundingRect;

    BadFigure(final Rectangle boundingRect)
    {
        this.boundingRect = boundingRect;
    }

    public boolean isInside(final int x, final int y)
    {
        final Rectangle innerRect = new Rectangle(boundingRect);

        // Definitionsort der Konstanten BORDER_WIDTH und BORDER_HEIGHT
        // ist bei Referenzierung über BadFigure unklar
        innerRect.grow(-FigureConstants.BORDER_WIDTH,
                       -BadFigure.BORDER_HEIGHT);

        return innerRect.contains(x, y);
    }
}
```

Die beschriebene und im Listing gezeigte »Namensraumverschmutzung« scheint zunächst ein etwas hergeholtes Argument zu sein. Doch in der Praxis erschwert eine solche Vorgehensweise mögliche Erweiterungen, Refactorings und die Nachvollziehbarkeit. Betrachten wir dazu die Problematik einmal genauer: Nehmen wir an, eine Klasse `BaseFigure` implementiert versehentlich ein Konstanten-Interface `Constants` mit der Konstante `OK`. Eine Erweiterung der Klassenhierarchie um eine Subklasse `ProcessingFigure` führt zu Kompilierfehlern, wenn diese Subklasse ein Interface `ProcessingResults` implementiert, in dem ebenfalls eine Konstante `OK` definiert ist: Die Konstante `OK` ist dann mehrdeutig. Das zeigt Abbildung 16-1.

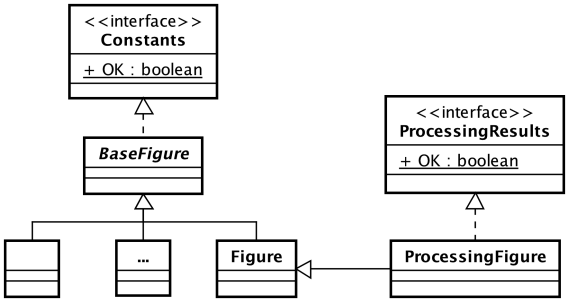

Abbildung 16-1 *Doppeldeutigkeit der Konstante* `OK`

Tipps und Refactorings Benutzen Sie Interfaces nur dazu, wozu sie aus OO-Sicht dienen sollen, nämlich, um eine Schnittstelle mit angebotenen Methoden zu definieren.

Reine Konstanten-Interfaces können in der Regel in finale Konstantensammlungsklassen mit privaten Konstruktoren oder `enum`-Aufzählungen umgewandelt werden. Wie man dabei vorgeht, beschreibt das Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM in Abschnitt 17.4.12 als Schritt-für-Schritt-Anleitung.

Wurden solche Interfaces allerdings bereits (versehentlich) implementiert, dann muss mühselig jedes Vorkommen einer Konstante mit einer Referenz auf die neue Konstantenklasse versehen werden. Dabei kann es zu Problemen kommen, wenn man nicht alle Klassen im Zugriff hat oder verändern kann, die das Interface implementiert haben. Daher ist es sinnvoll, alle bekannten Nutzer vor einer solchen Maßnahme über den Umbau zu informieren. Ist die Nutzerbasis groß (z. B. für das Java-API), so ist ein solches Vorgehen nicht ohne Schwierigkeiten möglich und eine solche Designsünde lässt sich kaum mehr korrigieren, sondern oftmals nur als veraltet markieren. Dazu nutzt man das Javadoc-Tag `@deprecated` oder die Annotation `@Deprecated`. ***An diesem Beispiel erkennt man, wie wichtig es ist, gleich von Anfang an mit großer Sorgfalt zu arbeiten***. Dies gilt insbesondere bei der Implementierung aller nach außen sichtbaren Klassen und Methoden.

Info: Ähnliche Probleme durch statischen Import (`import static`)

Statische Imports ermöglichen es, Attribute oder Methoden anderer Klassen ohne Angabe des qualifizierenden Klassennamens zu nutzen. Es werden dadurch weitere Namen in den Namensraum der eigenen Klasse aufgenommen.

Beim Import statischer Attribute kommt es bei gleichnamigen, eigenen Klassenattributen zu Namenskonflikten und Kompilierproblemen, wie dies z. B. beim Implementieren verschiedener Interfaces mit gleichnamigen Konstanten der Fall ist.

Bei Mehrdeutigkeiten in Bezug auf Methoden treten keine Fehler beim Kompilieren auf, stattdessen überdeckt eine Objektmethode aus der eigenen Klasse eine statisch importierte Methode. Dadurch kommt es gegebenenfalls allerdings zu folgendem Problem: ***Wird bei einer Erweiterung eine Objektmethode eingeführt, so wird das Verhalten unerwartet verändert, da als Folge nicht mehr die statisch importierte Methode aufgerufen wird, sondern die neu eingeführte Methode der eigenen Klasse.*** Dies gilt übrigens auch dann, wenn die aufrufende Methode selbst statisch ist!

16.1.3 Bad Smell: Zusammengehörende Konstanten nicht als Typ definiert

Müssen einige Werte als Konstanten definiert werden, wie etwa die Einfügestrategien für Kommandos aus dem Beispiel der Klasse `CommandExecutor`, sieht man häufiger Definitionen zusammengehörender Konstanten als `int`-Werte:

```
/** altes Kommando durch neues ersetzen */
public static final int REPLACE_OLD_OR_ADD_AS_LAST = 0;

/** Kommando als erstes Kommando neu erzeugen */
public static final int ADD_AS_FIRST               = 1;

/** Kommando als letztes Kommando neu erzeugen */
public static final int ADD_AS_LAST                = 2;
```

Warum ist das ein Bad Smell? Werden Konstanten als `int` definiert, so sind als Werte grundsätzlich alle aus dem Wertebereich des Typs erlaubt. Dadurch können auch unsinnige Werte Anwendung finden, wie folgendes Beispiel zeigt:

```
// Was mag die 4711 bedeuten?
final CommandExecutor executor = new CommandExecutor(4711);
```

Wünschenswert ist es, sicherzustellen, dass die übergebenen Werte im Bereich der definierten Konstanten liegen. Das erfordert eine Bereichsprüfung und wird schnell unübersichtlich. Dieser Effekt verstärkt sich, wenn die Prüfungen an verschiedenen Stellen im Sourcecode notwendig sind, die Werte keinen zusammenhängenden Bereich abbilden oder sich Werte nachträglich ändern bzw. neue Konstanten hinzukommen. Machen wir es konkret für den folgenden Konstruktor:

```
public CommandExecutor(final int strategy)
{
    if (strategy < REPLACE_OLD_OR_ADD_AS_LAST || strategy > ADD_AS_LAST)
    {
        throw new IllegalArgumentException("parameter 'strategy' is invalid: " +
                    "value='" + strategy + "' not in range [" +
                    REPLACE_OLD_OR_ADD_AS_LAST + " -- " + ADD_AS_LAST + "]");
    }
    this.registrationStrategy = strategy;
}
```

Prinzipiell gut an der Parameterprüfung ist, dass sie überhaupt vorhanden ist und der Benutzer bei einem Fehler im Aufruf detailliert sowohl über die Wertebelegung als auch über gültige Parameterwerte informiert wird.

Allerdings stehen diesen Vorteilen einige Nachteile gegenüber. Die obige Wertebereichsprüfung und Fehlerbehandlung ist aus folgenden Gründen ziemlich fragil:

1. Es werden Annahmen über die konkreten Werte der Konstanten `REPLACE_OLD_OR_ADD_AS_LAST` und `ADD_AS_LAST` gemacht, insbesondere, dass der Wert von `ADD_AS_LAST` den größeren der beiden Werte repräsentiert. Wird der Wert der Konstanten `REPLACE_OLD_OR_ADD_AS_LAST` bzw. `ADD_AS_LAST` derart verändert, dass diese Annahme nicht mehr gilt, scheitert die Wertebereichsprüfung.
2. Der Vergleich setzt voraus, dass durch die Konstanten ein zusammenhängender Wertebereich abgedeckt wird. Diese Forderung erschwert unter Umständen eine sinnvolle Vergabe der Parameterwerte.

3. Wird im Nachhinein eine zusätzliche Kommandoart hinzugefügt, erfordert dies nicht nur die Definition einer neuen Konstanten, sondern es müssen überall im verwendenden Sourcecode Anpassungen in den Wertebereichsprüfungen erfolgen.

Tipps und Refactorings In der Regel möchte man Aufrufern lediglich erlauben, symbolische Werte, d. h. Konstantennamen, zu verwenden. Das lässt sich elegant und einfach durch das Sprachmittel `enum` erreichen, mit dem man Konstanten zu einem neuen Typ zusammenfassen kann:

```
enum RegistrationStrategyType
{
    REMOVE_OLD_AND_ADD_AS_LAST(0, "altes Kommando durch neues ersetzen"),
    ADD_AS_FIRST(1, "Kommando als erstes Kommando neu erzeugen"),
    ADD_AS_LAST(2, "Kommando als letztes Kommando neu erzeugen");

    final int    value;
    final String description;

    RegistrationStrategyType(final int value, final String description)
    {
        this.value = value;
        this.description = description;
    }
}
```

Die durch Einsatz eines `enum` gewonnene Typsicherheit löst sowohl die Probleme der Bereichsprüfung – diese ist nun überflüssig, da sie implizit durch den Compiler basierend auf der Typangabe sichergestellt wird – als auch die der nicht zusammenhängenden Wertebereiche oder sich ändernder Konstantenwerte. Eine Anleitung liefert das Refactoring WANDLE KONSTANTENSAMMLUNG IN ENUM UM in Abschnitt 17.4.12.

16.1.4 Bad Smell: Casts auf unbekannte Subtypen

Casts werden in Programmen verwendet, wenn eine Typumwandlung gewünscht wird, die vom Compiler nicht garantiert werden kann. In wenigen Fällen ist ein Einsatz unumgänglich, beispielsweise wenn man ein Zahlenliteral vom Typ `int` an eine Methode mit `short`-Parametern übergeben möchte. In der Regel weist der Einsatz eines Casts allerdings auf eine potenzielle Fehlerquelle hin, und man sollte einen genaueren Blick auf die entsprechende Sourcecode-Stelle werfen.

Betrachten wir die Problematik am Beispiel einer Methode `getPersons()`, die in einem Interface `IDataAccess` definiert ist:

```
interface IDataAccess
{
    List<Person> getPersons();
}
```

Der folgende Sourcecode-Ausschnitt nutzt eine Realisierung dieses Interface zum Aufruf von `getPersons()`. Vor der Zuweisung an die Variable `persons` findet ein Cast auf eine `LinkedList<Person>` statt: