

DIE TRÜGERISCHE SICHERHEIT DES GRÜNEN BALKENS

„Test-Driven Development“ wird derzeit zwar in mancher Hinsicht kontrovers diskutiert, dessen ungeachtet jedoch immer häufiger in der Praxis eingesetzt. Ein wichtiger Aspekt dieses Vorgehens ist der weitgehende Verzicht auf eine umfassende Dokumentation, an deren Stelle die Erstellung der Tests vor der Programmierung tritt. In dem Artikel wird an einem Beispiel demonstriert, welche Vorteile gerade hier eine systematische Erstellung der Testfälle mit sich bringt.

Agile Methoden erfreuen sich nicht nur bei der Entwicklung objektorientierter Software zunehmender Popularität. Eines ihrer zentralen Prinzipien ist, dass die Entwicklung funktionierender Software höhere Priorität hat als die Erstellung einer umfassenden Dokumentation (siehe [Fow01]). Folgerichtig kommt dem Testen im Sinne eines Nachweises des Funktionierens der Software ein hoher Stellenwert zu.

Beim *Test-Driven Development (TDD)* (siehe [Bec02], [Bec-WWW]) werden die Testfälle zu Beginn der Entwicklung eines jeden Bausteins oder Inkrements (d.h. einer funktionalen Erweiterung der Software) in Verbindung mit dem fachlichen Konzept erstellt. Als wichtiges Hilfsmittel kommen in diesem Zusammenhang Frameworks zur Erstellung und automatisierten Ausführung von Unit-Tests wie z. B. *JUnit* zum Einsatz, für die häufig der Sammelbegriff *xUnit* verwendet wird, wobei *x* für den Anfangsbuchstaben der unterstützten Programmiersprache steht. Der Nutzen dieser Frameworks zum Test der Softwarebausteine besteht darin, dass die Tests schnell geschrieben bzw. – präziser gesagt – programmiert und aufgrund ihrer automatisierten Ausführbarkeit leicht wiederholbar sind. Außerdem wird der Dokumentationsaufwand reduziert, da die Testklassen gleichzeitig Testimplementierung und -dokumentation der zu implementierenden Klassen darstellen. Oft dienen die Testklassen als Ersatz für eine nicht vorhandene Spezifikation.

Bei der Auswahl und Spezifikation der Testfälle bieten diese Frameworks allerdings keine Unterstützung, sodass diese im Allgemeinen intuitiv erfolgen. Ausgewählt und ausgeführt werden die so codierten Testfälle oft über eine grafische Benutzungsoberfläche (GUI), die bei *JUnit* beispielsweise mittels eines farbigen Balkens anzeigt, ob alle Testergebnisse den Erwartungen entsprechen (grün) oder

nicht (rot). Ist der Balken grün gefärbt, ist das Ziel erreicht: Die Software funktioniert im Sinne der Testfälle, sie kann weiter entwickelt oder ausgeliefert werden.

Dieses Vorgehen bringt bedeutende Vorteile mit sich:

- Die Entwicklung erfolgt zwangsläufig unter dem Gesichtspunkt der Testbarkeit.
- Die Implementierung wird für jedes Inkrement getestet (Regressionstests).
- Da die Testfälle schon früh erstellt werden und überdies automatisch ausführbar sind, verringert sich die Gefahr, dass aus Zeitgründen gegen Ende des Projekts nur eingeschränkt getestet wird. Trotz dieser Vorteile betrachten Tester die Testpraxis der agilen Methoden mit Misstrauen. Zu oft wurden sie mit fehlerhafter Software konfrontiert, obwohl der Balken grün war. Von einer Software, die läuft, zu einer Software, die korrekt läuft, ist der Weg dann häufig noch weit. Eine große Herausforderung besteht darin Testfälle zu spezifizieren, die möglichst alle Eingabekonstellationen der Anwendung prüfen. Denn es sind häufig die unerwarteten Eingaben oder Randbedingungen, die bei der Entwicklung nicht berücksichtigt wurden und bei der Anwendung der Software zu Fehlern führen.

Die Qualität der Software hängt somit wesentlich von der Qualität der Tests und demzufolge von der Qualität der Testfälle ab. Es gibt eine Vielzahl verschiedener Techniken zur Testfallspezifikation, die aus der Spezifikation der Funktionen des Programms so genannte funktionale Testfälle oder aus dem Programmcode so genannte strukturelle Testfälle ableiten und mit weitaus höherer Wahrscheinlichkeit Fehler aufdecken, als dies mit zufällig ausgewählten Testfällen erreicht werden kann (siehe [Mye79], [Spi04], [Lig02]). Auch zum Testen von objektorientierter entwickelter Software gibt es entsprechende Techniken (siehe [Bin00], [Sne02]). Obwohl umfang- ▶

die autoren

Falk Fraikin (E-Mail: fraikin@informatik.tu-darmstadt.de) ist wissenschaftlicher Mitarbeiter an der TU Darmstadt, wo er sich hauptsächlich mit entwicklungsbegleitender Qualitätssicherung beschäftigt.

Dr. Matthias Hamburg (E-Mail: mhamburg@psi.de) ist leitender Berater der PSI AG im Geschäftsbereich Public Management. Seine fachlichen Schwerpunkte sind Software-Qualitätssicherung und -Qualitätsmanagement.

Stefan Jungmayr (E-Mail: stefan.jungmayr@fernuni-hagen.de) ist wissenschaftlicher Mitarbeiter an der FernUniversität in Hagen mit den Schwerpunkten Software Engineering, Testen und Testbarkeit.

Thomas Leonhardt (E-Mail: leonhardt@informatik.tu-darmstadt.de) ist wissenschaftlicher Mitarbeiter an der TU Darmstadt und beschäftigt sich hauptsächlich mit der Frage, wie aus Prototypen graphischer Benutzungsschnittstellen frühzeitige funktionale Testfälle abgeleitet werden können.

Andreas Schönknecht (E-Mail: andreas.schoenknecht@tui.de) ist Software-Engineer bei der TUI InfoTec. Er unterstützt Projekte bei der Anwendung von Entwicklungsmethoden und Werkzeugen.

Prof. Dr.-Ing. Andreas Spillner (E-Mail: spillner@informatik.hs-bremen.de) ist Hochschullehrer an der Hochschule Bremen. Seine Fachgebiete sind Softwaretechnik und Qualitätssicherung.

Prof. Dr. Mario Winter (E-Mail: winter@gm.fh-koeln.de) ist Hochschulprofessor an der Fakultät für Informatik und Ingenieurwissenschaften der FH Köln mit den Fachgebieten Softwareentwicklung und Projektmanagement sowie Software-Qualitätssicherung.

Der Beitrag ist im Arbeitskreis „Testen objektorientierter Programme“ der GI-Fachgruppe „Test, Analyse und Verifikation von Software“ entstanden.

```
public class Money {
    private int amount;
    public Money(int amount)
        { this.amount = amount; }
    public int getAmount() { return amount; }
    public Money add(Money m) {
        return new Money(getAmount() +
            m.getAmount());
    }
    public Money subtract(Money m) {
        return new Money(getAmount() - m.getAmount());
    }
    public Money percentOf(int percentage) {
        return new Money(Math.round((float) getAmount() *
            ((float) percentage) / 100));
    }
    public boolean equals(Object obj) {
        if (obj instanceof Money) {
            return (getAmount() ==
                ((Money)obj).getAmount());
        }
        return false;
    }
    public String toString() { return "" + getAmount(); }
}
```

Listing 1: Die Klasse „Money“

reiches Testwissen zur Verfügung steht, werden die in der Praxis bewährten Techniken zur Testfallspezifikation bei der agilen Entwicklung zu wenig beachtet. Intuition und fachliche Entwicklungskompetenz der Beteiligten stehen bei der Erstellung der Testfälle im Vordergrund.

TDD alleine ist noch keine Garantie für Qualität. Erst im Zusammenspiel mit systematischem Vorgehen und dem Einsatz bewährter Techniken bei der Testfallspezifikation ergeben sich ein umfassender Mehrwert und die Sicherheit ausreichend getestet zu haben.

Um den Nutzen eines systematischen Vorgehens bei der Testfallspezifikation zu verdeutlichen, konzentrieren wir uns auf funktionale Testfälle und präsentieren zunächst ein kleines Fallbeispiel. Für dieses werden dann zwei Episoden der Testfallspezifikation dokumentiert. In Episode I entwickeln unsere TDD-Spezialisten Kent und Glen die Testfälle und die entsprechenden Klassen. Danach überlegen sie, ob sie wirklich an ausreichend viele unterschiedliche Testfälle gedacht haben (Episode II). Hier bringt Glen sein Testwissen ein. Was aus dem Beispiel zu lernen ist, fassen wir am Schluss zusammen.

Fallbeispiel „Kassenbon“

Um den Wert einer systematischen Testfallspezifikation zu demonstrieren, betrachten wir das folgende Fallbeispiel: Eine Firma bekommt den Auftrag für ein kleines Geschäft ein einfaches Programm zur Ausstellung der Kassenbons zu schreiben. Als erste Grundlage für die Entwicklung dienen folgende drei User-Storys:

```
public class Article {
    private String name;
    private Money price;
    private int vatRate;
    public Article(String name, Money price, int vatRate) {
        this.name = name;
        this.price = price;
        this.vatRate = vatRate;
    }
    public String getName() { return name; }
    public Money getPrice() { return price; }
    public int getVATRate() { return vatRate; }
    public Money getVAT() {
        return getPrice().percentOf(vatRate); }
    public Money getPriceIncludingVAT() {
        return getPrice().add(getVAT());
    }
}
```

Listing 2: Die Klasse „Article“

Story 1: Man kann einen Kassenbon für mehrere Artikel erstellen. Jeder Artikel besitzt einen Namen und einen Preis. Der Kassenbon weist den Gesamtpreis aus.

Story 2: Zu jedem Artikel kann man die Mehrwertsteuer individuell angeben.

Story 3: Vor 10 Uhr soll ein Frühkaufabbrabatt von 5% gewährt werden.

Ausgehend von diesen Storys haben Kent und Glen bisher einige (hier nicht behandelte) Testfälle und die Klassen Money (**Listing 1**), Article (**Listing 2**) und Receipt (**Listing 3**) in Java entwickelt. Die Klasse Money repräsentiert ganzzahlige Geldbeträge z. B. in Eurocent (und basiert auf einer Klasse aus [Bec99]). Die Klasse Article repräsentiert einen Artikel, für den der Name, der Preis und die Mehrwertsteuer angegeben werden können. Die Klasse Receipt stellt den Kassenbon dar, auf dem mehrere Artikel eingetragen werden können.

Episode I

Im Folgenden entwickeln Kent und Glen das Kassensystem aufgrund zweier neuer Storys weiter.

Kent: Die nächste Story lautet:

Story 4: Ein Artikel soll storniert werden können.

Glen: Lass uns also wie immer zunächst einen einfachen Test erstellen, der diese Funktionalität fordert: Der Test erzeugt zunächst einen leeren Bon, bucht dann zum Beispiel einen Ball, prüft, ob die Rechnung stimmt, storniert den Ball dann wieder und prüft, ob die Rechnung wieder zu Null gesetzt wurde.

```
public void testRemoveOneArticle() {
    Receipt receipt = new Receipt(14);
    Article ball = new Article("Ball",
        new Money(532), 16);
    receipt.add(ball);
}
```

```
import java.util.LinkedList;
import java.util.List;
public class Receipt {
    private List articles;
    private Money balance = new Money(0);
    private Money vat = new Money(0);
    private Money balanceIncludingVAT = new Money(0);
    private int earlyHourDiscountRate;
    public Receipt(int hour) {
        articles = new LinkedList();
        if (hour < 10) {
            earlyHourDiscountRate = 5;
        } else {
            earlyHourDiscountRate = 0;
        }
    }
    public int size() { return articles.size(); }
    public void add(Article article) {
        articles.add(article);
        balance = balance.add(article.getPrice());
        vat = vat.add(article.getVAT());
        balanceIncludingVAT = balanceIncludingVAT.
            add(article.getPriceIncludingVAT());
    }
    public Money getBalance() {
        return applyDiscount(balance); }
    public Money getVAT() { return applyDiscount(vat); }
    public Money getBalanceIncludingVAT() {
        return applyDiscount(balanceIncludingVAT);
    }
    public int getEarlyHourDiscountRate() {
        return earlyHourDiscountRate;
    }
    private Money applyDiscount(Money money) {
        return money.
            percentOf(100 - getEarlyHourDiscountRate());
    }
    public Money getEarlyHourDiscount() {
        Money undiscountedBalance = balance;
        Money discountedBalance = getBalance();
        return
            undiscountedBalance.subtract(discountedBalance);
    }
}
```

Listing 3: Die Klasse „Receipt“

```
assertEquals(new Money(532),
    receipt.getBalance());
assertEquals(new Money(85),
    receipt.getVAT());
assertEquals(new Money(617),
    receipt.getBalanceIncludingVAT());
receipt.remove(ball);
assertEquals(new Money(0),
    receipt.getBalance());
assertEquals(new Money(0),
    receipt.getVAT());
assertEquals(new Money(0),
    receipt.getBalanceIncludingVAT());
}
```

O.k., hier ist er. Wie erwartet funktioniert es nicht, da die remove()-Methode in der Klasse Receipt noch nicht existiert.

Kent: Das kriegen wir schnell in den Griff, ist ja im Prinzip wie add() (*nimmt seinerseits die Tastatur und tippt*):

```
public void remove(Article article) {
    articles.remove(article);
    balance = balance.subtract(article.getPrice());
    vat = vat.subtract(article.getVAT());
    balanceIncludingVAT = balanceIncludingVAT.
        subtract(article.getPriceIncludingVAT());
}
```

... und schon ist der Balken wieder grün. Was liegt als nächstes an?



Story 5: Einige Artikel sollen als Sonderangebote einen speziellen Rabatt bekommen. Für diese Artikel soll kein zusätzlicher Frühkaufabbrabatt gewährt werden.

Glen: Dazu müssen wir wohl die Klasse Article erweitern. Hier zunächst der Testfall:

```
public void testSpecialOfferArticle() {
    Article article = new Article("Coat",
        new Money(12995), 16);
    article.setSpecialOfferDiscountRate(20);
    assertEquals(new Money(10396),
        article.getPrice());
    assertEquals(new Money(1663),
        article.getVAT());
    assertEquals(new Money(12059),
        article.getPriceIncludingVAT());
    assertTrue(article.isSpecialOffer());
}
```

Die Methoden setSpecialOfferDiscountRate() und isSpecialOffer() fehlen.

Kent: Kein Problem:

```
class Article {
    private int specialOfferDiscountRate = 0;
    public void setSpecialOfferDiscountRate(
        int specialOfferDiscountRate) {
        this.specialOfferDiscountRate
            = specialOfferDiscountRate;
    }
    public boolean isSpecialOffer() {
        return (specialOfferDiscountRate > 0);
    }
}
```

Oh, der Balken ist immer noch rot (*denkt nach*). Na klar, beim Preis muss der Rabatt noch berücksichtigt werden:

```
public Money getPrice() {
    return price.
        percentOf(100 - specialOfferDiscountRate);
}
```

Voilà.

Glen: Jetzt müssen wir aber noch sicherstellen, dass die Sonderangebote keinen Frühkaufabbrabatt bekommen.

```
public void testSpecialOfferEarlyHour() {
    Receipt receipt = new Receipt(9);
    Article article = new Article("Coat",
        new Money(12995), 16);
    article.setSpecialOfferDiscountRate(20);
    receipt.add(article);
    assertEquals(new Money(10396),
        receipt.getBalance());
    assertEquals(new Money(1663),
        receipt.getVAT());
    assertEquals(new Money(12059),
        receipt.getBalanceIncludingVAT());
}
```

Und schon ist der Balken wieder rot.

Kent: Aber nicht mehr lange. Wir müssen beim Hinzufügen in der add()-Methode eben

prüfen, ob der neue Artikel ein Sonderangebot ist oder nicht. Dann berechnen wir in den Instanzvariablen den echten Betrag inklusive sämtlicher Discounts. Außerdem benötigen wir eine weitere Instanzvariable für den gewährten Rabatt. Nennen wir sie earlyHourDiscount. Damit wird dann auch getEarlyHourDiscount() vereinfacht.

```
private Money earlyHourDiscount =
    new Money(0);
public void add(Article article) {
    articles.add(article);
    if (article.isSpecialOffer()) {
        balance = balance.add(applyDiscount
            (article.getPrice()));
        vat = vat.add(applyDiscount
            (article.getVAT()));
        balanceIncludingVAT =
            balanceIncludingVAT.
                add(applyDiscount
                    (article.getPriceIncludingVAT()));
        earlyHourDiscount =
            earlyHourDiscount.add(article.getPrice().
                subtract(applyDiscount
                    (article.getPrice())));
    } else {
        balance = balance.add(article.getPrice());
        vat = vat.add(article.getVAT());
        balanceIncludingVAT =
            balanceIncludingVAT.add
                (article.getPriceIncludingVAT());
    }
}
public Money getEarlyHourDiscount()
    { return earlyHourDiscount; }
```

Der Balken ist immer noch rot. Ach ja, wir machen die Aufrufe von applyDiscount() ja nun in add(). Die Getter dürfen nun nur noch die reinen, unrabattierten Werte zurückgeben:

```
public Money getBalance()
    { return balance; }
public Money getVAT()
    { return vat; }
public Money getBalanceIncludingVAT()
    { return balanceIncludingVAT; }
```

Nach erneuter Testausführung ist der Balken grün!

Kent: Super, da haben wir wieder zwei Storys abgeschlossen. Lass uns einen Kaffee trinken.

Die beiden gehen Kaffee trinken – Kent hochzufrieden, Glen etwas nachdenklich.

Episode II

Nach einer knappen Viertelstunde kommen die beiden wieder an den Rechner zurück.

Kent: Möchtest du nun das Keyboard für die nächste Story übernehmen?

Glen: Moment. Wir haben nun ja einige Erweiterungen am Code gemacht und sollten erst einmal testen, ob noch Fehler drin sind.

Kent: Wieso? Der Balken ist grün und durch unsere Tests haben wir doch nach-

gewiesen, dass die gestellten Anforderungen erfüllt sind.

Glen: Aber wir haben nicht systematisch getestet, sondern bisher nur für einige einfache Fälle die Funktionalität nachgewiesen.

Kent: Hm, na gut, aber es funktioniert doch! Warum denn weiter testen, wenn es schon läuft?

Glen: Wir müssen weitere Testfälle schreiben, die prüfen, ob sich der Code auch bei anderen Werten korrekt verhält. Dabei helfen uns methodische Techniken wie die Äquivalenzklassenanalyse.

Kent: Was für Klassen?

Glen: Äquivalenzklassen. Damit werden z.B. die Eingabebereiche der Parameter einer Operation in Unterbereiche zerlegt, bei denen für alle Werte eines Unterbereiches das gleiche Verhalten der Operation erwartet wird.

Kent: Aha – damit soll bestimmt sichergestellt werden, dass alle Möglichkeiten der Operation ausgeleuchtet werden.

Glen: Genau! Ich zeige es dir mal für die Operation „storniere Artikel“, die wir in der Methode remove() der Klasse Receipt realisiert haben. In Story 5 sehen wir, dass Artikel einen eigenen Rabatt haben können, aus Story 3 wissen wir, dass es einen Frühkaufabbrabatt gibt. Unsere Methode remove() muss dies alles berücksichtigen und die Berechnung des Preises neu durchführen (*nimmt ein Blatt Papier und erstellt unter anderem die folgenden Tabellen*):

Operation: Storniere Artikel	
Eingabedaten: Artikel, Preis, Sonderangebotsrabatt, Uhrzeit	
Äquivalenzklassen zum Eingabedatum Artikel:	
Bezeichnung	Beschreibung
GÄk1_1	Artikel ist enthalten
UÄk1_1	Artikel ist nicht enthalten
Äquivalenzklassen zum Eingabedatum Preis:	
Bezeichnung	Beschreibung
GÄk2_1	Preis > 0
GÄk2_2	Preis = 0
UÄk2_1	Preis < 0
Äquivalenzklassen zum Eingabedatum Sonderangebotsrabatt:	
Bezeichnung	Beschreibung
GÄk3_1	Sonderangebotsrabatt = 0
GÄk3_2	0 < Sonderangebotsrabatt < 100
GÄk3_3	Sonderangebotsrabatt = 100
UÄk3_1	Sonderangebotsrabatt < 0
UÄk3_2	Sonderangebotsrabatt > 100



(Vorläufige) Äquivalenzklassen zum Eingabedatum Uhrzeit:

Bezeichnung	Beschreibung
GÄk4_1	00:00 <= Uhrzeit < 10:00
GÄk4_2	10:00 <= Uhrzeit <= 23:59
UÄk4_1	Uhrzeit < 00:00
UÄk4_2	Uhrzeit > 23:59

Kent: Was bedeutet denn „GÄk“ und „UÄk“ in den Tabellen?

Glen: Damit bezeichnet man gültige bzw. ungültige Äquivalenzklassen. Werte aus ersteren sind erlaubt und sollten erfolgreich verarbeitet werden können, Werte aus letzteren sollten vom Programm nicht bearbeitet werden, sondern müssen erkannt und z.B. durch Werfen einer Ausnahme zurückgewiesen werden. Sieh dir zum Beispiel die Äquivalenzklasse „UÄk1_1“ für das Eingabedatum „Artikel“ der Operation „Storniere Artikel“ an. Hier wird getestet, ob die Operation behandeln kann, dass der übergebene Artikel gar nicht in der Liste der berechneten Artikel enthalten ist; oder die Äquivalenzklasse „UÄk4_1“ für das Eingabedatum „Uhrzeit“, mit dem eine „negative“ Zeitangabe getestet wird.

Kent: Sollten wir für die Uhrzeit nicht auch noch die gesetzlich vorgeschriebenen Öffnungszeiten betrachten?

Glen: Gute Idee! Aber wir müssen das dann heute Nachmittag bei unserer Besprechung mit dem Auftraggeber noch absegnen lassen. Hier die erweiterte Tabelle:

(Endgültige) Äquivalenzklassen zum Eingabedatum Uhrzeit:

Bezeichnung	Beschreibung
GÄk4_1	08:00 <= Uhrzeit < 10:00
GÄk4_2	10:00 <= Uhrzeit <= 20:00
UÄk4_1	Uhrzeit < 00:00
UÄk4_2	00:00 <= Uhrzeit < 08:00
UÄk4_3	20:00 < Uhrzeit <= 23:59
UÄk4_4	Uhrzeit > 23:59

Für einen Testfall müssen wir nun für jedes Eingabedatum eine Äquivalenzklasse auswählen. Im Allgemeinen muss man die Kombinationen der gültigen Testfälle betrachten und jeweils einen Testfall pro ungültige Äquivalenzklasse. Fangen wir also mit dem Testfall bestehend aus den gültigen Äquivalenzklassen GÄk1_1, GÄk2_1, GÄk3_1 und GÄk4_1 an. Dieser Fall besagt, dass der zu stornierende Artikel enthalten ist, einen positiven Preis hat, dass dieser Artikel kein Sonderangebot ist und zudem ein Frühkaufabbratt besteht. Hier der zugehörige Testfall (natürlich müssen wir konkrete Werte aus den jeweiligen

Äquivalenzklassen auswählen, dies können aber beliebige Werte sein):

```
public void
testRemoveExistingArticleAtEarlyHour() {
    Receipt receipt = new Receipt(9); // GÄk4_1
    Article article = new Article("Ball",
        new Money(532), 16); // GÄk2_1
    article.setSpecialOfferDiscountRate(0); // GÄk3_1
    receipt.add(article); // GÄk1_1
    receipt.remove(article);
    assertEquals(5,
        receipt.getEarlyHourDiscountRate());
    assertEquals(0,
        receipt.getEarlyHourDiscount().getAmount());
    assertEquals(0,
        receipt.getBalance().getAmount());
    assertEquals(0,
        receipt.getVAT().getAmount());
    assertEquals(0,
        receipt.getBalanceIncludingVAT().getAmount());
}
```

Der Balken ist rot, dann sind wir wohl doch noch nicht fertig! Siehe da, nach dem Entfernen des Artikels liefert getEarlyDiscount() immer noch „27“ (5% von 532).

Kent: Oh, wir haben vergessen den earlyHourDiscount beim remove neu zu berechnen. Wie ist es damit?

```
public void remove(Article article) {
    articles.remove(article);
    balance = balance.subtract(article.getPrice());
    vat = vat.subtract(article.getVAT());
    balanceIncludingVAT =
        balanceIncludingVAT.subtract
            (article.getPriceIncludingVAT());
    earlyHourDiscount =
        earlyHourDiscount.subtract(
            article.getPrice().subtract
                (applyDiscount(article.getPrice())));
}
```

Glen: Immer noch rot; getBalance() liefert „-27“. Beim remove() werden offenbar immer die Vollpreise zurückberechnet. Der Frühkaufabbratt wird nicht beachtet.

Kent: Unglaublich, was da noch an Fehlern drinsteckt. Ich programmiere mal den Rabatt rein:

```
public void remove(Article article) {
    articles.remove(article);
    balance = balance.subtract
        (applyDiscount(article.getPrice()));
    vat = vat.subtract
        (applyDiscount(article.getVAT()));
    balanceIncludingVAT
        = balanceIncludingVAT.subtract(
            applyDiscount(article.
                getPriceIncludingVAT()));
    earlyHourDiscount
        = earlyHourDiscount.subtract(
            article.getPrice().subtract
                (applyDiscount(article.getPrice())));
}
```

Glen: Ok. Der Test läuft nun. Als nächstes könnten wir uns den Testfall bestehend aus den Äquivalenzklassen GÄk1_1, GÄk2_1, GÄk3_2 und GÄk4_1 vorneh-

men. Dieser unterscheidet sich vom vorherigen dadurch, dass der Artikel diesmal ein Sonderangebot ist:

```
public void
testRemoveExistingSpecialOfferAtEarlyHour() {
    Receipt receipt = new Receipt(9); // GÄk4_1
    Article article = new Article("Ball",
        new Money(532), 16); // GÄk2_1
    article.setSpecialOfferDiscountRate(30); // GÄk3_2
    receipt.add(article); // GÄk1_1
    receipt.remove(article);
    assertEquals(5,
        receipt.getEarlyHourDiscountRate());
    assertEquals(0,
        receipt.getEarlyHourDiscount().getAmount());
    assertEquals(0,
        receipt.getBalance().getAmount());
    assertEquals(0,
        receipt.getVAT().getAmount());
    assertEquals(0,
        receipt.getBalanceIncludingVAT().getAmount());
}
```

Wieder Fehler. Der earlyHourDiscount steht auf „-19“. Unsere remove()-Methode berechnet für den Artikel einen Frühkaufabbratt, der aber, weil es sich um ein Sonderangebot handelt, gar nicht gewährt wurde.

Kent: Stimmt. Wie ist es damit?

```
public void remove(Article article) {
    articles.remove(article);
    if (!article.isSpecialOffer()) {
        balance = balance.subtract
            (applyDiscount(article.getPrice()));
        vat = vat.subtract(applyDiscount
            (article.getVAT()));
        balanceIncludingVAT =
            balanceIncludingVAT.subtract(
                applyDiscount(article.getPriceIncludingVAT()));
        earlyHourDiscount =
            earlyHourDiscount.subtract(
                article.getPrice().subtract
                    (applyDiscount(article.getPrice())));
    } else {
        balance = balance.subtract(article.getPrice());
        vat = vat.subtract(article.getVAT());
        balanceIncludingVAT =
            balanceIncludingVAT.subtract(
                article.getPriceIncludingVAT());
    }
}
```

Glen: So sieht's gut aus.

Kent: Mir fällt gerade auf, dass remove() und add() sich inzwischen sehr ähnlich sehen. Wir sollten die Berechnung der verschiedenen Variablen in eine eigene Methode extrahieren und die Berechnung jeweils vor der Abfrage der einzelnen Werte durchführen.

Der Programmcode wird refaktoriert. Anschließend werden weitere Tests implementiert und der Code noch um die Behandlung der ungültigen Werte erweitert. Schließlich laufen alle Tests und der Balken ist grün.

Kent: Mensch, hätte ich nicht gedacht, dass wir da noch Fehler finden, geschwei-



ge denn so schwere. Die hätten uns später echt Mühe machen können, wenn wir die nicht jetzt schon entdeckt hätten. Da zeigen sich doch mal wieder die Vorteile des Pair-Programmings. Jetzt haben wir es aber geschafft, oder?

Glen sagt nichts. Er denkt an die Worte von E. W. Dijkstra: „Testing can only show the presence of errors, not their absence.“

Weitere Überlegungen

Glen antwortet nicht mit „ja“, denn auch wenn durch die Bildung der gültigen und ungültigen Äquivalenzklassen weitere sinnvolle Testfälle identifiziert werden konnten, sind noch andere Testmethoden zur Erstellung zusätzlicher Testfälle anzuwenden. So lassen sich an den Grenzen der Äquivalenzklassen (z. B. genau 10:00 Uhr als Testdatum) oft Fehler nachweisen. Als ein weiteres Beispiel sei die Berücksichtigung von Objektzuständen genannt. Das Verhalten vieler Operationen hängt direkt oder indirekt von der Belegung der Attribute eines Objekts ab, also seinem Zustand. Dies wird schon bei der Betrachtung der Äquivalenzklassen für die Methode `remove()` der Klasse `Receipt` deutlich. Denn diese hängen nicht nur von dem zu entfernenden Artikeln ab, sondern auch vom Zustand des `Receipt`-Objekts, d.h. davon, ob das betreffende Objekt bereits hinzugefügt wurde oder nicht. Dies ist noch ein einfacher Fall, komplizierter wird es bei der Berücksichtigung der verschiedenen Rabatte. Hier ist eine Identifikation von notwendigen Testfällen nur möglich, wenn die Attribute `specialOfferDiscountRate` und `earlyHourDiscountRate` berücksichtigt werden.

Neben dem Erstellen der Testfälle sind auch eine Strukturierung und eine Organisation der Testfälle entscheidend. Viele Testfälle prüfen im Prinzip mehrere Dinge gleichzeitig. Der Testfall `testRemoveOneAr-`

`ticle` prüft neben dem Entfernen des Artikels auch die `get`-Methoden und dass um 14:00 Uhr kein Frühkaufabbratt gewährt wird. Solche Abhängigkeiten müssen bei der Organisation der Testfälle berücksichtigt und dokumentiert werden. Neue Anforderungen (die nächste Story) wirken sich oft auf bereits bestehende Testfälle aus. Nur wenn klar ist, welche Beziehungen zwischen Testfällen und Code bestehen, ist es möglich, schnell und angemessen zu reagieren und auch die bisherigen Testfälle gegebenenfalls anzupassen.

Fazit

Episode II des Fallbeispiels belegt, dass der grüne Balken nur dann eine Aussagekraft hat, wenn die Testfälle systematisch erstellt werden und ein breites Spektrum der möglichen Anwendungen der Klassen und Operationen abdecken. Zusammenfassend lässt sich feststellen:

- Den Test vor der Realisierung zu betrachten und von Anfang an zu automatisieren ist ein großer Vorteil beim TDD.
- Eine enge Zusammenarbeit von Entwicklern und Testern bringt Vorteile für beide Seiten. Die Bausteine werden testbarer und alle möglichen Nutzungskonstellationen werden berücksichtigt.
- Die systematische Testfallerstellung führt zur Vollständigkeit der Implementierung der Bausteine. Ausnahmefälle werden nicht so leicht übersehen.
- Aber: Selbst der ausgiebigste Test der einzelnen Bausteine garantiert kein fehlerfreies Gesamtsystem. Integrations- und Systemtest dürfen nicht vernachlässigt werden.

Wir sagen damit nichts Neues, auch Kent Beck schreibt: „One of the ironies of TDD

is that it isn't a testing technique ... It's an analysis technique, a design technique, really a technique for structuring all the activities of development" ([Bec03], S. 204). Um Grundlagen des methodischen Testens in agilen Methoden zu integrieren, ist ein aufeinander Zugehen von Entwicklern und Testern wünschenswert. Als Basis der Zusammenarbeit ist grundlegendes Wissen über Sichtweisen, Methoden und Techniken der jeweils anderen Gruppe erforderlich – dann wird die Zusammenarbeit auch Früchte tragen. ■

Literatur & Links

- [Bec99] K. Beck, *Extreme Programming Explained*, Addison-Wesley, 1999
- [Bec02] K. Beck, *Test-Driven Development*, Addison-Wesley, 2002
- [Bec03] K. Beck, *Test-Driven Development by Example*, Addison-Wesley, 2003
- [Bec-WWW] K. Beck, E. Gamma, *Test Infected: Programmers Love Writing Tests*, siehe: members.pingnet.ch/gamma/junit.htm
- [Bin00] R.V. Binder, *Testing Object-Oriented Systems*, Addison-Wesley, 2000
- [Fow01] M. Fowler, J. Highsmith, *The Agile Manifesto*, in: *Software Development*, Aug. 2001 (siehe: www.agilemanifesto.org)
- [Lig02] P. Liggesmeyer, *Software-Qualität – Testen, Analysieren und Verifizieren von Software*, Spektrum Akademischer Verlag, 2002
- [Myc79] G. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979
- [Sne02] H.M. Sneed, M. Winter, *Testen objektorientierter Software. Das Praxisbuch für den Test objektorientierter Client/Server-Systeme*, Hanser, 2002
- [Spi04] A. Spillner, T. Linz, *Basiswissen Softwaretest*, 2. Auflage, dpunkt.verlag, 2004