Besseren Code Schneller Schreiben

Test-driven Development

Traditionelle Softwareentwicklung legt nahe, zunächst Anforderungen umzusetzen und diese in einem nachgelagerten Schritt zu testen. Funktionalität kommt somit vor Qualität. Agile Methoden wie Extreme Programming (XP) propagieren eine umgekehrte Reihenfolge: Erst wird der Testfall erstellt, dann die zugehörige Funktionalität implementiert. Bessere Software entsteht so schneller: Debugging-Aktivitäten werden quasi überflüssig. Der Artikel gibt einen Überblick über Test-driven Development und bietet Tips aus der Praxis zum erfolgreichen Einsatz an.

Einleitung

Um das Wichtigste vorwegzunehmen: Test-driven Development (TDD) ist keine Testmethode oder von primärem Interesse für Testspezialisten. Im Gegenteil: TDD ist eine Entwicklungsmethode, die Softwareentwicklern hilft, besseren Code schneller zu schreiben. Dabei ist das Prinzip von TDD einfach. Ändern müssen Sie allerdings Ihre Vorgehensweise bei der Softwareentwicklung.

TDD besteht aus folgenden Schritten: Zuerst erstellen wir einen Testfall und legen in diesem die Schnittstelle der zugehörigen Fachklasse fest. Danach implementieren wir die Klasse genau soweit, dass der Testfall erfolgreich ausgeführt wird. Anschließend refaktorieren wir die Klasse. Das Mantra von TDD lässt sich als „*Write a test. Make it run. Then make it right*“ zusammenfassen [4].

Test-driven Development Illustriert

Viel lässt sich über TDD theoretisieren. Um Ihnen aber einen praktischen Eindruck zu vermitteln, möchte ich ein Beispiel bemühen. Der XP-Empfehlung „*Choose the simplest thing that could possibly work*“ [3] folgend, wähle ich dabei bewusst ein sehr einfaches Beispiel aus. Um die Diskussion auf TDD zu fokussieren, verzichte ich auf eine Erläuterung des benutzten Testframeworks JUnit [8]. Für mehr Informationen siehe z.B. [2], [7].

Erster Schritt: Erstellung des Testfalls

Nehmen wir an, wir möchten eine Funktionalität bereitstellen die uns Auskunft über den Geschmack von Äpfeln gibt, beispielsweise als Teil einer Managementanwendung für Lebensmittel oder als Hilfestellung für den unerfahrenen Hobby-Koch. TDD folgend, schreiben wir zunächst folgenden Testfall:

```
public class AppleTest extends TestCase {

   public void testTaste() {
       assertEquals("Golden Delicious should taste sweet",
                    "SWEET",
                    apple.getTaste());
   }
}
```

Das oben stehende Code-Fragement stellt die Testmethode `testTaste()` zur Verfügung. Im Methodenkörper haben wir zunächst festgehalten, was wir testen wollen: Der Geschmack eines Apfelobjekts soll süß sein. Dabei haben wir das TDD-Muster „Assert First“ [4] angewandt. In einem zweiten Schritt vervollständigen wir nun unseren Testfall, indem wir eine Apfelinstanz anlegen:

```
public class AppleTest extends TestCase {

    public void testTaste() {
        Apple apple = new Apple("GOLDEN DELICIOUS");
        assertEquals("Golden Delicious should taste sweet",
                     "SWEET",
                     apple.getTaste());
    }
}
```

Zweiter Schritt: Erstellung der Fachklasse

Beachten Sie, dass zu diesem Zeitpunkt noch keine Apfelklasse verfügbar ist. Versuchen wir, unseren Testfall zu kompilieren, erhalten wir einen Fehler, der uns anzeigt, dass noch mehr Arbeit zu leisten ist: Wir müssen die Apfelklasse implementieren. Dabei implementieren wir zunächst nur, was unser Programm kompilierfähig macht.

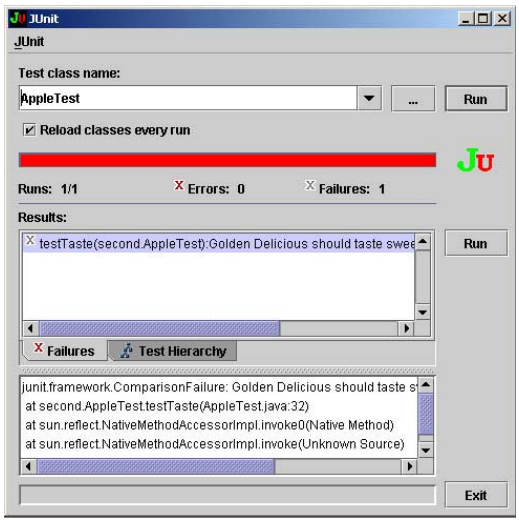
```
public class Apple {

    private String type;

    public Apple(String type) {
        this.type = type;
    }

    public String getTaste() {
        return "";
    }
}
```

Nun kompiliert unser Testfall und lässt sich ausführen. Allerdings erhalten wir folgende Fehlermeldung:



Die nichterfolgreiche Ausführung des Testfalls zeigt uns, dass wir die Methode `getTaste()` weiter verfeinern müssen.

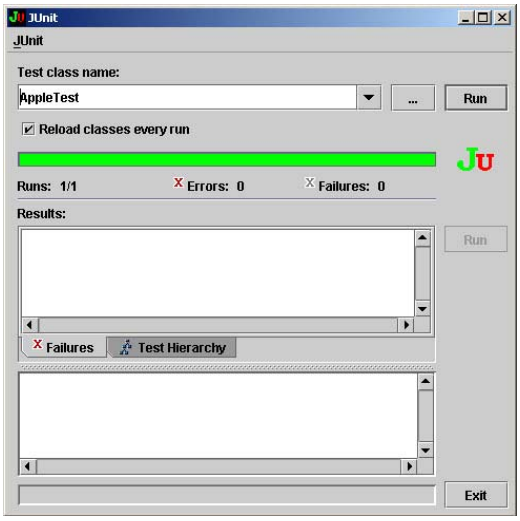
```
public class Apple {

    private String type;

    public Apple(String type) {
        this.type = type;
    }

    public String getTaste() {
        if (type == "GOLDEN DELICIOUS") return "SWEET";
        else return "SOUR";
    }
}
```

Führen wir nun unseren Testfall aus, erhalten wir folgende Meldung:



Unser Testfall wurde erfolgreich ausgeführt! Dabei haben wir die minimal benötigte Funktionalität implementiert. Anders als bei einer konventionellen Vorgehensweise haben wir nicht zuerst die Apfelklasse detailliert spezifiziert, alle Methodensignaturen festgelegt, implementiert und anschließend getestet. Stattdessen haben wir in Minizyklen Testfall und Fachklasse soweit implementiert, bis der Testfall erfolgreich ausgeführt wurde, gemäß dem Motto: „*Let the computer tell you when you are finished.*“ [2]

Betrachten wir allerdings die Apfelklasse, so fällt auf, dass wir die Apfelsorte hartcodiert haben. Clients der Apfelklasse müssen wissen, welche Apfelsorten existieren und den korrekten Namen als String an den Konstruktor der Klasse übergeben. Unser Code riecht nicht besonders gut: Es ist Zeit zum refaktorieren!

Dritter Schritt: Refaktorierung

Unsere Refaktorierung beginnt mit dem Anpassen unseres Testfalls:

```
public class AppleTest extends TestCase {

    public void testTaste() {
        Apple apple = new GoldenDelicious();
        assertEquals("Golden Delicious should taste sweet",
                     "SWEET",
                     apple.getTaste());
    }
}
```

Anstelle einer allgemeinen Apfelinstanz erzeugen wir ein `Golden Delicious`-Objekt. Wir behandeln dabei `Apple` nicht länger als Klasse sondern als Interface und wenden das Refaktorierungsmuster „Extract Interface“ an [5]. Folglich müssen wir `Apple` wie folgt abändern:

```
public interface Apple {

    public String getTaste();
}
```

Nun implementieren wir die Klasse `Golden Delecious`:

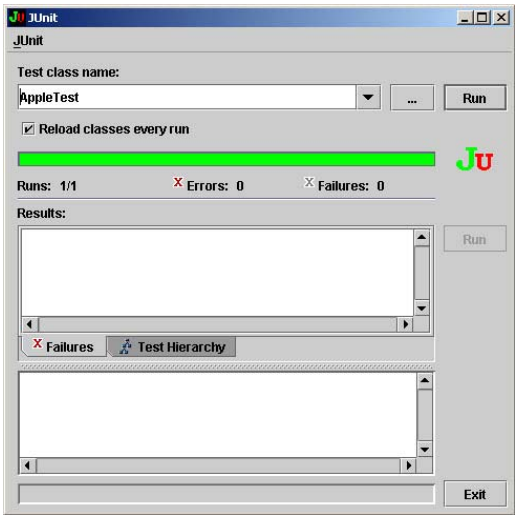
```
public class GoldenDelicious implements Apple {

    private static final String TASTE = "SWEET";

    public String getTaste() {
        return TASTE;
    }

}
```

Spätestens nun ist es an der Zeit, unseren Testfall wieder auszuführen, um zu überprüfen, ob das modifizierte Programm noch korrekt ist:



Unser Testfall lässt sich als auch nach erfolgten Refaktorierungsarbeiten erfolgreich ausführen. Der existierende Testfall gibt uns dabei die Sicherheit, Verbesserungsarbeiten ausführen zu können ohne dabei unbemerkt Fehler in den Code einzuführen. Anders gesagt: Der Testfall gibt uns die Freiheit, nachträgliche Änderungen schnell umzusetzen, ohne danach das Debugger Feature unserer IDE auf sein Leistungsvermögen testen zu müssen.

Test-driven Development in der Praxis

TDD und Unit Tests

Zugegeben: Das obige Beispiel ist trivial. Dennoch zeigt es eine wichtige Auswirkung von TDD: Zu jeder Fachklasse existiert ein automatisierter Testfall, der die Klasse in Isolation testet (Unit Test). Dies stellt sicher, dass Wartungsarbeiten an Klassen schnell und mit geringem Defektrisiko durchgeführt werden können, ohne dass dabei die gesamte Infrastruktur benötigt wird, wie z.B. eine Datenbank.

In der Praxis ist es jedoch nicht immer einfach, für jede Klasse einen Unit Test zu erstellen. Ein Testfall für eine Enterprise JavaBean (EJB) muss beispielsweise eine Remote-Verbindung zu dieser aufbauen. Damit ist der Test kein Unit Test sondern ein Integrationstest. Dies führt uns in folgendes Dilemma: Die Erstellung eines Unit Test ist aufwendig, der Verzicht reduziert jedoch die Wartbarkeit und Erweiterbarkeit unserer Software.

Um diesem Dilemma zu entgehen, empfehle ich Ihnen: Denken Sie um und erhöhen Sie durch ein Redesign bzw. Refaktorieren die Unittestbarkeit Ihrer Software. Durch die Verlagerung der Funktionalität aus einer EJB in eine einfache Java-Klasse können Sie beispielsweise einen Unit Test für die zu entstehende Klasse testgesteuert entwicklen. Hilft ein Redesign nicht, so kann der Einsatz von Mock-Objekten Unittestbarkeit herstellen [9], [10]. Allerdings sollten Sie sich über die Aufwände zur Erstellung und Pflege von Mock-Objekten im Klaren sein. Integrationstests im J2EE-Umfeld lassen sich übrigens durch Testframeworks wie Cactus [5] automatisieren.

TDD und Entwicklungsprozesse

Auch wenn TDD im Rahmen agiler Methoden verbreitet wurde: TDD lässt sich unabhängig vom Entwicklungsprozess anwenden. Agile Methoden wie Extreme Programming (XP) unterstützen jedoch TDD und fordern explizit den konsequenten Einsatz von TDD [3]. In XP harmoniert TDD mit anderen zentralen Praktiken wie Refaktorieren, Schlichtes Design und Programmierrichtlinien. TDD lässt sich aber auch in konventionellen, wasserfall-basierten Projekten gewinnbringend einsetzten: Schließlich handelt es sich bei TDD um eine Entwicklungsmethode, deren Einführung keine Veränderung des Prozessmodels impliziert. Wichtig ist jedoch, bei der Anwendung von TDD auf den einheitlichen Einsatz eines Testframeworks wie z.B. JUnit zu achten.

Auswirken hat TDD auf die Vorgehensweise beim Design: Auch bei testgesteuertem Vorgehen empfiehlt es sich, zuerst eine Architekturvision der zu implementierenden Funktionalität zu entwerfen. Dieses wird dann zügig mit Hilfe von TDD umgesetzt und getestet. Die detaillierte Designaktivitäten werden somit weitestgehend im Rahmen der testgesteuerten Programmierung vorgenommen [1]. Vorteilhaft ist hierbei, dass

Lösungsideen schnell auf Ihre Tragfähigkeit überprüft werden. Denn: Papier ist geduldig und erst der ausgeführte Testfall zeigt, ob eine Lösung adäquat ist.

Zusammenfassung

TDD hilft besseren Code schneller zu entwickeln: Code wird extensiv getestet, aufwendige Debugging-Aktivitäten werden überflüssig. Darüber hinaus führt TDD zu einer benutzerorientierten Programmierung: Wir denken erst darüber nach, wie wir eine bestimmte Funktionalität benützen möchten, bevor wir diese implementieren.

TDD lässt sich einfach erlernen und mit geringem Investitionsaufwand einführen. Um die TDD-Vorteile auch zu nutzen, empfehle ich Ihnen einen konsequenten Einsatz der Entwicklungsmethode. Dieser führt letztlich zu einer verbesserten Entwicklungskultur, in der Entwickler Code schreiben, den Kollegen leicht verstehen und ändern können. Denn wie sagt Martin Fowler so treffend: „*Any fool can write code that a computer can understand. Good programmers write code that humans can understand.*“ [5]

Literatur und Links

[1] Scott W. Ambler, Ron Jeffries. *Agile Modeling: Effective Practices for Extreme Programming and the Unified Process*. Wiley, 2002
[2] David Astels. *Test Driven Development: A Practical Guide*. Prentice Hall, 2003
[3] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.
[4] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2002
[5] Cactus. jakarta.apache.org/cactus
[6] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
[7] Andy Hunt, Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003
[8] JUnit Homepage. www.junit.org
[9] Tim Mackinnon, Steve Freeman, Philip Craig. *Endo-Testing: Unit Testing with Mock Objects*. Paper presented at XP2000. www.connextra.com/aboutUs/mockobjects.pdf
[10] MockObjects. www.mockobjects.com

Biografie

Roman Pichler arbeitet als Berater der Siemens AG, Corporate Technology. Er unterstützt Siemensbereiche bei der Optimierung ihrer Softwareentwicklung und ist auf agile Methoden spezialisiert. Roman besitzt langjährige Erfahrung in der Softwareentwicklung und hat verschiedene Artikel zum Thema Agilität veröffentlicht, u.a. über Schlanke Softwareentwicklung und den Einsatz agiler Methoden bei verteilter Entwicklung. E-Mail: roman.pichler@siemens.com