Test-First Programming*

Tobias Reinhard

13. Januar 2004

Inhaltsverzeichnis

*Seminararbeit zum Thema Agile vs. klassische Methoden der Software-Entwicklung. Vorgelegt von Tobias Reinhard, 00-713-446, Angefertigt am Institut für Informatik der Universität Zürich, Prof. Dr. Martin Glinz. Betreuer: Christian Seybold

1 Einleitung

Das Testen wird in der Software-Entwicklung häufig sehr stiefmütterlich behandelt. Zum einen steht das Testen in den klassischen Methoden typischerweise am Schluss des Prozesses und wird deshalb als erstes über Bord geworfen, wenn es zeitlich eng wird. Des weiteren ist Testen bei den meisten Entwicklern nicht sonderlich beliebt und wird deshalb selten systematisch durchgeführt.

Test-First Programming ist eines der zentralen Prinzipien des eXtreme Programming [3] und versucht dessen Grundidee, die menschlichen Faktoren vermehrt in den Vordergrund zu stellen, auch beim Testen anzuwenden. Dabei wird vor der Implementierung einer Funktionalität ein automatisch ausführbarer Testfall geschrieben, der diese Funtionalität abdeckt. Danach wird nur gerade soviel Code geschrieben, dass dieser Testfall erfüllt wird.

Die vorliegende Arbeit gibt zuerst einen kurzen Überblick über das eXtreme Programming mit besonderem Augenmerk auf dessen Teststrategie. Der zweite Teil beschreibt die Prinzipien des Test-First Programming und zeigt das Vorgehen anhand eines kleinen Beispiels auf. Im letzten Teil wird auf Erfahrungen mit Test-First Programming sowohl im akademischen Umfeld als auch in der Praxis eingegangen.

2 Extreme Programming

Als Reaktion auf die Beobachtung, dass die Kosten für Änderungen bei der Entwicklung von Software über die Zeit dramatisch ansteigen, entwickelten Royce [8] und Boehm [5] vor mehr als 30 Jahren das Wasserfallmodell, das einen lineraren Ablauf von Entwicklungsphasen vorsieht. Dabei sollen die Entscheidungen mit den grössten Auswirkungen möglichst früh getroffen werden, um teure Änderungen in späteren Phasen zu verhindern. Da die Anforderungen jedoch nicht immer von Beginn an klar ersichlich sind und Fehler, die in einer vorhergehenden Phasen gemacht worden sind, oft erst in einer späteren Phase erkennbar werden, sind beim Wasserfallmodell jedoch Iterationen über mehrere Phasen hinweg unvermeidlich.

Aus dieser Beobachtung heraus enstanden Wachstumsmodelle, bei denen das Gesamtprojekt in autonome Teilprojekt zerlegt wird, in denen die Phasen dann wiederum linear ablaufen. Das von Kent Beck und Ward Cunningham propagierte eXtreme Programming [2][3] (XP) führt diesen Ansatz weiter, indem das Projekt in möglichst kurze Entwicklungszyklen zerlegt wird, in denen alle Aktivitäten (Analyse, Design, Implementierung und Testen) gleichzeitig ausgeführt werden. Diese kurzen Zyklen sollen es ermöglichen, schneller und leichter auf Änderungen zu reagieren, und möglichst früh ein

Feedback vom Kunden zu erhalten. XP eignet sich deshalb besonders für kleinere bis mittlere Projekte, die sich an Änderungen der Anforderungen oder in der Umgebung anpassen müssen. Abbildung 1 veranschaulicht die Entwicklung des eXtreme Programming aus dem Wasserfallmodell.

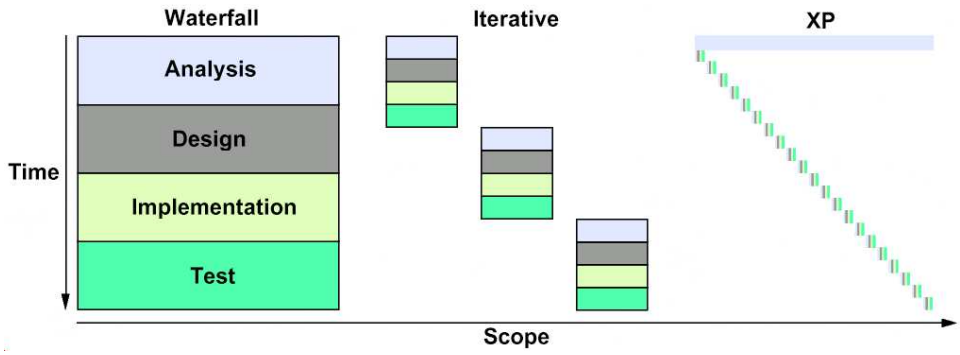

Abbildung 1: Entwicklung von XP aus dem Wasserfall-Modell [2]

2.1 Grundprinzipien

Extreme Programming verspricht, im Widerspruch zur klassischen Sichtweise [5], die Änderungskosten während des ganzen Entwicklungsprozesses konstant tief zu halten, indem es auf folgenden Grundprinzipien aufbaut:

- **Schnelles Feedback**
  Aus der Psychologie ist bekannt, dass die Zeit zwischen einer Handlung und dem daraus folgenden Feedback für das Lernen von grösster Bedeutung ist. Deshalb versucht XP den Entwicklern Feedback innerhalb von Sekunden, Minuten oder Tagen zu geben und nicht innerhalb von Monaten oder Jahren wie bei klassischen Methoden.

- **Einfachheit**
  XP zwingt den Entwickler, für alle Probleme die einfachste Lösung anzuweden, auch wenn diese Lösung für einen vorausschauenden Entwickler lächerlich erscheinen mag. Dabei ist auch darauf zu achten, nur die heutigen Probleme zu lösen und nicht für die Zukunft zu planen.

- **Inkrementelle Änderungen**
  Änderungen sollen nicht in einem grossen Schritt, sondern in möglichst vielen kleinen Teilschritten durchgeführt werden, da dadurch das Risiko eines Fehlschlages sinkt oder, im Falle eines Fehlschlages, die Folgen geringer sind.

- **Änderungen willkommen heissen**
  Änderungen sind jederzeit willkommen, da die Entwickler gelernt haben mit ihnen umzugehen und darauf vorbereitet sind.

- **Qulitätsarbeit**
  Jedermann möchte gute Arbeit leisten und sollte dabei soweit als möglich unterstützt werden.

2.2 Praktiken

Aus diesen Prinzipien leiten sich die folgenden Praktiken bei der Anwendung von XP ab. Diese Praktiken sind auch in vielen klassischen Ansätzen zu finden, XP wendet sie jedoch in einer extremeren Form an.

- **Planungsspiel**
  Der Kunde und die Entwickler entscheiden gemeinsam über den Umfang des nächsten Releases, wobei der Kunde die Funktionalität mit der höchsten Priorität nennt und die Entwickler entscheiden, in welcher Zeit und welchem Umfang diese Funktionalität realisiert werden kann.

- **Kleine Releases**
  Das Ziel ist es, möglichst schnell mit einem einfachen System produktiv zu werden und weitere Funktion in kleinen Zyklen hinzuzufügen. Ein Release sollte dabei so klein wie möglich sein und nur die Funktionalität enthalten, die dem Kunden im Moment den grössten Nutzen bringt.

- **Metapher**
  Jedes XP Projekt wird von einer Metapher geleitet, die für jederman verständlich beschreibt, wie das System im groben funktioniert. Die Metapher ersetzt zu einem Teil die Architektur der klassischen Ansätze.

- **Simples Design**
  Das Design des Systems soll zu jedem Zeitpunkt so einfach wie möglich sein. Dies im krassen Gegensatz zu den klassischen Methoden der Software-Entwicklung, bei denen immer auch ein vorausschauendes Design angestrebt wird.

- **Testen**
  Die Entwickler schreiben fortlaufend automatisch ausführbare Tests, und zwar bevor sie den eigentlichen Code schreiben. Zusätzlich schreiben die Kunden funktionale Tests. Auf die Rolle des Testen in XP geht der nächste Abschnitt vertieft ein.

- **Refaktorisierung**
  Die Struktur des Systems wird kontinuierlich vereinfacht und verbessert, ohne dabei dessen äusseres Verhalten zu verändern.

- **Paarprogrammierung**
  Produktiver Code wird nur zu zweit an einem Computer geschrieben. Die Paarprogrammierung soll die Idee der Codeinspektion direkt in den Entwicklungsprozess einbauen.

- **Kollektive Verantwortlichkeit**
  Der gesamte Code kann von jedermann zu jeder Zeit verändert werden, jeder Entwickler trägt jedoch auch die Verantwortung für die gesamte Codebasis.

- **Kontinuierliche Integration**
  Code wird so oft wie möglich (innerhalb von Stunden) getestet und dann integriert. Integriert werden darf jedoch nur, falls alle Tests zu 100% funktionieren.

- **40 Stunden Woche**
  Überzeit darf nicht zum Normalfall werden, da sie oft auf ein ernstes Problem des Projekts hinweist.

- **Kunde vor Ort**
  Ein Kundenvertreter muss jederzeit für Fragen vor Ort verfügbar sein. Der Kunde muss sich dabei bewusst sein, das die Entwicklung zu einem grossen Teil von dieser Person gesteuert wird.

- **Codierrichtlinien**
  Da sich alle Entwickler mit dem gesamten Code beschäftigen, sind Codierrichtlinien, die gewisse Standards vorgeben, unumgänglich.

2.3 Teststrategie in XP

Testen geht gegen die menschliche Natur, was sich daran zeigt, dass nur testet, wer auf irgendeine Art und Weise dazu gezwungen wird. Niemand testet gerne, obwohl jederman weiss, dass zwingend getestet werden muss und dass normalerweise viel zu wenig getestet wird. Massimo Arnoldi wird dazu in [3] mit folgenden Worten zitiert:

> *"Unfortunately at least for me (and not only) testing goes against human nature. If you release the pig in you, you will see that you program without tests. Then after a while, when your rational part wins, you stop and start writing tests. You mentioned too, pair programming reduces the probability that both partners are releasing their pigs at the same moment."*

Da sich XP dem Grundsatz “Arbeite mit der menschlichen Natur, und nicht gegen sie.” verpflichtet sieht, versucht es das Testen so schmerzlos wie möglich zu machen und den Entwicklern mit dem Testen ein nützliches Instrument zur Verfügung zu stellen.

Dabei schreiben die Entwickler bei XP einen Modultest, im Sinne eines White-Box-Tests, bevor sie den eigentlichen Code schreiben. Die Entwicklung wird dadurch durch die Tests geleitet, da nur das implementiert werden muss, was von einem Test verlangt wird. Daraus hat sich auch der Begriff des “Test-Driven Development” [4] entwickelt. Die Tests übernehmen dabei auch zu einem gewissen Teil die Rolle der Anforderungsspezifikation und der Dokumentation der klassischen Ansätzen und ermöglichen erst die kontinuierliche Vereinfachung des Codes durch das Refactoring.

Tests in XP müssen isoliert und automatisch ausführbar sein. Tests dürfen nicht voneinander abhängig sein, da sonst, falls ein einzelner Test nicht erfüllt wird, eine ganze Reihe von weiteren Tests nicht mehr laufen würden. Die Motivation zu testen hängt entscheidend davon ab, dass durch einen kleinen Fehler nicht ein grosser Teil der Tests nicht mehr läuft. Tests müssen zudem automatisch ausführbar sein, da sie in hektischen Phasen am wertvollsten sind, und in solchen Situationen nicht mehr ausgeführt werden, wenn sie nicht so einfach wie ein Compiler zu bedienen sind.

Neben den von den Entwicklern geschriebenen Modultests, schreiben die Kunden oder spezielle Tester im Auftrag des Kunden funktionale Tests auf Basis der im Planungsspiel entwickelten Stories. Die Modultests müssen immer zu 100% laufen, während die funktionalen Tests nicht in jedem Fall zu 100% erfüllt werden können, da sie von einer anderen Quelle stammen als der Code.

3 Test-First Programming

Damit Testen im Programmieralltag selbst bei zunehmendem Stress nicht vernachlässigt wird, ist es nötig die menschlichen Faktoren stärker in der Vordergrund zu rücken. Effektives Testen sollte dabei folgende Anforderungen erfüllen:

- Testen muss möglichts zeitnah zur Programmierung erfolgen, damit das Feedback möglichst schnell erfolgt und das Testen nicht zu einem nachgelagerten Prozess wird, der unter Zeitdruck leicht über Bord geworfen werden kann.
- Die Tests müssen automatisch ausführbar und wiederholbar sein, da sie sonst unter Stress nicht mehr ausgeführt werden.

- Das Testen darf von den Entwicklern nicht als lästige Pflicht wahrgenommen werden, sondern muss ebensoviel Spass machen wie das Programmieren.
- Die Tests müssen ebenso häufig ausgeführt werden, wie kompiliert wird.
- Das Ausführen der Tests muss so einfach sein wie das Kompilieren.

3.1 Grundsätze

Beim Test-First Programming schreiben die Entwickler automatisierte Modultests für die Anforderungen an den Code. Diese Tests prüfen die Software dann in kleinen unabhängigen Einheiten. Zusätzlich schreiben die Tester (oder die Kunden) automatisierte Akzeptanztest, welche die Anforderungen an die Software als integrierte Einheit testen.

Der Testcode wird dabei vor dem eigentlichen Code geschrieben. Dadurch wird jede Änderung der Funktionalität durch einen Test motiviert. Der auf diese Weise erstellte Test muss dabei zuerst fehlschlagen, da die getestete Funktionalität noch gar nicht implementiert worden ist. Durch das Ausführen des Tests vor der Implementierung der getesteten Funktionalität wird der Test selbst getestet. Danach wird gerade soviel Code geschrieben, dass der Test erfüllt wird. Die Entwicklung wird somit durch das konkrete Feedback der Tests inkrementell angetrieben.

Beim Test-First Programming wird damit jede Änderung der Funktionalität der Software durch einen automatisierten Test motiviert. Das Schreiben der Tests ermöglicht dabei die Anforderungen an den Code anhand der Tests zu beschreiben. Zusätzlich bleibt der Code testbar, da nicht testbarer Code gar nicht entstehen kann, weil es keinen Test gibt, der solchen Code motiviert.

Die Testfälle müssen jedoch auch beim Test-First Programming systematisch erstellt werden (z.B. mit Hilfe von Äquivalenzklassen), da sie sonst eine Sicherheit vortäuschen, die sie gar nicht bieten können. Kent Beck schreibt denn in [4] auch:

> *"One of the ironies of TDD [Test-Driven Development] is that it isn't a testing technique [...]. It's an anlysis technique, a design technique, really a technique for structuring all the activites of development."*

Als weiterer Grundsatz gilt, dass der Code immer in der einfachst möglichen Form sein muss. Um dies zu erreichen, muss es auch möglich sein beim Hinzufügen neuer Funktionalität auch Änderungen im bereits bestehenden Code vorzunehmen, ohne dabei bereits bestehende Funktionalität zu verlieren.

Das Test-First Programming ermöglicht solche Änderungen, da die gesamte bereits bestehende Funktionalität durch die Tests abgedeckt wird und nicht mehr verloren gehen kann, ohne dass mindestens ein Test nicht mehr läuft.

3.2 Gründe

Test-First Programming ist in erster Linie ein Versuch, dem Testen innerhalb der Software- Entwicklung die Stellung einzugestehen, die ihm zusteht. Das Testen soll dabei von den Entwicklern als nützliches Instrument zur Verbesserung der Qualität der Software und nicht als lästige Pflicht wahrgenommen werden.

3.2.1 Softwarequalität und Lebensdauer

Sowohl die funktionale als auch die strukturelle Qualität der Software soll durch das Test-First Programming über einen weiten Zeitraum möglichst hoch gehalten werden. Die funktionale Qualität wird dabei durch die automatisierten Test bewahrt und die strukturelle Qualität durch das fortlaufende Refactoring.

Software, die kontinuierlich weiterentwickelt wird, leidet unter Entropie, da mit jeder Zeile, die neu hinzugefügt wird, die Geschichte der Software neue Bögen schlägt. Durch das Refactoring und den Grundsatz, dass der Code immmer in der einfachsten Form sein muss, soll diesem Verrotten der Software entgegengewirkt werden. Das Refactoring, d.h. das Vereinfachen und Verbessern der inneren Struktur der Software ohne ihr äusseres Verhalten zu verändern, ist dabei nur mit Hilfe der automatisierten Tests möglich, da nur diese einen Schutz gegen ungewollte Seiteneffekte bei Änderungen bieten.

3.2.2 Vertrauen in den Code

Die automatisierten Tests machen die Fortschritte bei der Entwicklung greifbar, da die Software selbst, auf Grund ihrer imatriellen Natur, nicht greifbar ist. Kent Beck [3] vergleicht denn das Entwickeln mit Test-First Programming auch mit einem Kletterer in einer Steilwand, der seine Schritte durch das Setzen von Sicherheitshaken absichert. Dabei reduziert der Kletterer mit jedem Haken die Tiefe, die er im Falle eines Fehltrittes hinunterstürzt. Der bis zum letzten gesetzten Haken zurückgelegte Weg gehört ihm dabei in jedem Fall. Zusätzlich kann er mit dem Abstand zwischen den Haken das Risiko, das er eingehen will, selbst wählen.

3.2.3 Erweiterung und Anpassung

Software ist während ihrer Lebenszeit kontinuierlicher Veränderungen ausgesetzt. Dabei hat Software, die nicht mehr geändert werden kann, in den meisten Fällen keinen Wert mehr. Die bereits bestehenden Tests machen dabei die Änderung der Software einfacher und sicherer, da sie ein dichtes Sicherheitsnetz bilden, das ungewollte Änderungen des Programmverhaltens bei Erweiterungen oder Anpassungen aufdecken kann.

3.2.4 Testbarer Code

Bereits bestehender Code ist häufig schlecht testbar, da er nicht im Hinblick auf das Testen erstellt wurde. Dadurch, dass die gesamte Erstellung des Codes nur durch Tests motiviert wird, kann gar kein nicht testbarer Code entstehen. Es ist jedoch auch anzufügen, dass gewisse Konstrukte wie z.B. Nebenläufigkeit durch automatisierte Modultests kaum oder gar nicht getestet werden können.

3.3 Vorgehen

Der Zyklus beim Test-First Programming setzt sich aus den folgenden vier Schritten zusammen:

1. Füge einen kleinen Test hinzu.
2. Führe alle Tests aus. Dabei muss der neue Test scheitern.
3. Ändere den Code nur soweit, dass der Test erfüllt wird.
4. Führe alle Tests aus und erfülle sie.
5. Refaktorisiere und entferne Duplikate (Code vereinfachen).

Da zum Test-First Programming ein Testframework benötigt wird, lässt sich das Vorgehen auch anhand dieses Frameworks beschreiben. Im folgenden dient dazu JUnit [10] die Java Version des xUnit Frameworks von Kent Beck und Erich Gamma. xUnit ist ein kleines aber mächtiges Testframework, das für über 30 Sprachen existiert. JUnit zeigt in seiner graphischen Version in einem Balken an, ob alle Test erfüllt wurden oder nicht. Ist der Balken rot, wurde mindestens ein Test nicht erfüllt. Erscheint der Balken grün, wurden alle Tests erfolgreich durchlaufen. Das Vorgehen lässt sich damit durch den Wechsel der Farbe des Balkens beschreiben:

1. **Von grün nach rot:** Die neuen Anforderungen an den Code werden in einem Testfall beschrieben. Beim Ausführen aller Tests muss dieser neue Test scheitern.

2. **Von rot nach grün:** Der Test wird in kleinen Schritten und auf möglichst einfache Art und Weise erfüllt.

3. **Von grün nach grün:** Nachdem alle Tests erfolgreich ausgeführt wurden, muss der Code in die einfachst mögliche Form gebracht werden. Dabei sichert der grüne Balken ab, dass beim Refactoring keine neuen Fehler entstehen.

Der nächste Abschnitt beschreibt das Vorgehen beim Test-First Programming anhand eines kurzen Beispiels.

4 Test-First Programming Episode

Der folgende Abschnitt zeigt an einem sehr einfachen Beispiel das Vorgehen beim Test-First Programming. Das Schulbuchbeispiel ist eine vereinfachte Version aus [9].

Die Aufgabe besteht darin, die Miete für das Ausleihen von DVDs zu berechnen. Dabei kostet eine DVD für den ersten Tag 2.- und für die darauffolgenden Tage sFr. 1.50 pro Tag.

Die Lösung des Problems beginnt mit einem Testfall. Als Testframework dient dabei JUnit, das eine Oberklasse (junit.framework.TestCase) für alle Testfälle zur Verfügung stellt. Methoden, deren Name mit test beginnt werden dabei automatisch als Testmethoden erkannt und ausgeführt. Für die eigentlichen Test stehen Zusicherungen wie z.B. assertTrue(...) oder equals(...) zur Verfügung.

Der erste Testfall überprüft die Berechnung des Preises für eine DVD, die für einen Tag ausgeliehen wurde:

```
public class CustomerTest extends junit.framework.TestCase {
    public void testRentingOneMovie() {
        Customer customer = new Customer();
        customer.rentMovie(1);
        assertTrue(customer.getTotalCharge() == 2);
    }
}
```

Dieser erste Testfall lässt sich nicht kompilieren, da die Klasse Customer noch gar nicht existiert. Der Test lässt sich jedoch sehr einfach zum Laufen bringen:

```
public class Customer {
    public void rentMovie(int daysRented) {
    }

    public int getTotalCharge() {
        return 2;
    }
}
```

Diese Lösung mag zwar sehr simpel erscheinen, stellt aber ohne Zweifel die einfachste Lösung für das gestellte Problem dar. Nachdem der erste Testfall erfüllt wird, soll ein zweiter Testfall das Ausleihen eines Filmes für zwei Tage abdecken.

```
public class CustomerTest extends junit.framework.TestCase {
    ...
    public void testRentingTwoMovies() {
        Customer customer = new Customer();
        customer.rentMovie(1);
        customer.rentMovie(2);
        assertEquals(5.5, customer.getTotalCharge(), 0.001);
    }
    ...
}
```

Der Testfall wird wiederum nicht erfüllt, da die getestete Funktionalität noch nicht implementiert ist. Der dritte Parameter der assertEquals(...) Methode gibt dabei die Genauigkeit beim Vergleich zweier Fliesskommazahlen an.

```
public class Customer {
    private double totalCharge = 0;

    public void rentMovie(int daysRented) {
        totalCharge += 2;
        if(daysRented > 1) {
            totalCharge += 1.5;
        }
    }

    public int getTotalCharge() {
        return totalCharge;
    }
}
```

Der letzte Testfall überprüft den Preis für die Ausleihe einer DVD für drei Tage:

```
public class CustomerTest extends junit.framework.TestCase {
    ...
    public void testRentingThreeMovies() {
        Customer customer = new Customer();
        customer.rentMovie(1);
        customer.rentMovie(2);
        customer.rentMovie(3);
        assertEquals(10.5, customer.getTotalCharge(), 0.001);
    }
    ...
}
```

Die Erfüllung dieses Testfall führt schliesslich zu folgendem Code:

```
public class Customer {
    private double totalCharge = 0;

    public void rentMovie(int daysRented) {
        totalCharge += 2;
        if(daysRented > 1) {
            totalCharge += (daysRented - 1) * 1.5;
        }
    }

    public int getTotalCharge() {
        return totalCharge;
    }
}
```

Der Code entsteht beim Test-First Programming also inkrementell durch schrittweises Wachstum und ständige Überarbeitung. Die Struktur des obenstehenden Codes liesse sich durch Refactoring noch verbessern, indem zum Beispiel die magischen Zahlen noch durch symbolische Konstanten ersetzt würden.

5 Erfahrungen mit Test-First Programming

Die folgenden beiden Abschnitte sollen einen kleinen Einblick in die Erfahrungen mit Test-first Programming geben. Der erste Abschnitt geht dabei auf ein Experiment mit Test-First Programming im universitären Umfeld ein. Danach werden noch kurz Erfahrungen mit Test-First Programming aus einem Projekt in der Praxis beschrieben.

5.1 Experiment mit Test-first Programming

Von Juli bis August 2001 wurde an der Universität Karlsruhe ein Experiment [6] über die Auswirkungen von Test-First Programming durchgeführt. Dabei sollte empirisch überprüft werden, ob die Effizienz bei der Programmierung, die Zuverlässigkeit des resultierenden Codes und das Verständnis für das Programm mit Test-First Programming höher ist als beim klassischen Testen nach der Implementierung. Zur Vereinfachung wurde der Test-First Ansatz dabei isoliert von den anderen XP Praktiken untersucht. 19 Informatikstudenten mussten dabei die Hauptklassen einer gegebenen Bibliothek zur Darstellung und Analyse von Graphen implementieren, wobei 10 Studenten gemäss dem Test-First Ansatz zu verfahren hatten und die übrigen 9 als Kontrollgruppe gemäss dem klassischen Ansatz. Nach der Implementierung mussten beide Gruppen an ihrer Lösung arbeiten, bis eine Reihe von automatisierten Akzeptanztests erfüllt wurden.

Bei der zur Lösung des Problems benötigten Zeit zeigten sich zwischen der Test-First Gruppe und der Kontrollgruppe keine signifikanten Unterschiede. Bei der Zuverlässigkeit des resultierenden Codes zeigten sich Unterschiede zwischen den beiden Gruppen beim erstmaligen Durchführen der Akzeptanztests. Die Lösungen der Test-First Gruppe lieferten beim erstmaligen Durchlaufen dieser Tests bedeutend schlechtere Ergebnisse als die der Kontrollgruppe. Ob dieser Unterschied auf Grund eines zu hohen Vertrauens in die Modultests oder fehlender Erfahrung mit dem Test-First Programming zurückzuführen ist, bleibt dabei offen. Beim Verständnis des Programms, das in der Anzahl der wiederverwendeten Methoden und der gescheiterten Methodenaufrufe gemessen wurde, zeigten sich leichte Vorteile für die Test-First Gruppe. Dabei produzierten diese Studenten signifikant weniger Fehler bei der mehr als einmaligen Wiederverwendung von Methoden.

Diese Ergebnisse sind jedoch mit Vorsicht zu geniesen, da einerseits die Anzahl der Probanden mit 19 sehr klein ist und andererseits die Fähigkeiten unter den Studenten normalerweise sehr unterschiedlich sind. Zusätzlich hatten die beteiligten Studenten nur wenig Erfahrungen im allgemeinen und mit XP im speziellen. Des weiteren ist die gestellte Aufgabe nicht unbedingt als solche anzusehen, bei der eine agile Methode ihre Stärken ausspielen könnte.

5.2 Test-First Programming in der Praxis

Als Fallbeispiel der Anwendung des Test-First Programming soll hier das CARUSO [1] Projekt dienen. Customer Care and Relationship Support Office (CARUSO) ist ein Projekt der Europäischen Union zur Entwicklung eines Frameworks für die Implementierung von Customer Relationship Management (CRM) Applikationen für kleinere Unternehmen. Das Projekt dauerte von Januar 2000 bis Juni 2002 und war als Partnerschaft zwischen einem

Energielieferanten aus den Niederlanden, einem deutschen Softwarehaus und der Ludwig-Maximilian-Universität in München angelegt. Die Gründe für den Einsatz von XP Praktiken wird dabei folgendermassen beschrieben [1]:

> *For example, [the customer] was referring to CARUSO as their customer care dream. As is common with dreams, CARUSO was supposed to do everything; but because of the complexity of CRM, no concrete requirements were given, since nobody knew where to start.*

Bei der Implementierung einer Skript Engine zur Unterstützung von Mitarbeitern im Call-Center wurde dabei streng nach den Test-First Prinzipien verfahren. Dabei wurde, jedesmal wenn eine neue Methode benötigt wurde, zuerst ein automatisierter Test für diese Methode geschrieben. Der grosse Vorteil dieses Vorgehens zeigte sich dabei vor allem darin, dass neue Funktionalität relativ einfach hinzugefügt werden konnte ohne unwissentlich bereits bestehende Funktionalität zu beeinträchtigen oder zu verlieren. Dabei konnte trotz der Erweiterungen der Code sauber und einfach gehalten werden, da dank der Tests auch Änderungen am bereits bestehenden Code ohne grosse Probleme durchgeführt werden konnten.

6 Fazit

In Situationen, in denen klassische Ansätzen ungenügend auf Änderungen in den Anforderungen oder der Umwelt reagieren können, scheint das Test-First Programming eine interessante Alternative zu sein. Die Vorstellung, dass der bereits erstellte Code so stark durch automatisierte Tests abgesichert ist, dass durch Änderungen oder Erweiterungen auch im bereits bestehenden Code die Struktur verändert und vereinfacht werden kann, ohne dass es dabei zur Katastrophe kommt, lässt das Herz der meisten Programmierer höher schlagen. Dabei bleibt jedoch immer zu beachten, dass auch der Testcode Fehler enthalten kann, und es dadurch zu einem zu starken Vertrauen in die Testfälle kommen kann. Ob Test-First Programming funktioniert hängt damit zum grössten Teil von der Güte der Tests ab.

Auch die Vereinigung des Programmierers und des Testers in einer Person ist nicht unproblematisch, da die Aufteilung dieser Rollen in den klassischen Ansätzen auch damit zu begründen ist, dass vier Augen mehr und vorallem auch anderes sehen als zwei. Bei der Anwendung des Test-First Programming im Rahmen eines XP Projekts wird dieser Nachteil jedoch zu einem gewissen Teil durch die Paarprogrammierung aufgehoben.

Für einen klassische Informatiker oder Entwickler kann die Umstellung auf das Test-First Programming jedoch relative schwierig [7] sein, da er nor-

malerweise darauf getrimmt ist möglichst weit im Voraus zu planen und alle Eventualitäten in die Planung einzubeziehen. Der Ansatz, den Testcode vor dem eigentlichen Code zu schreiben, ist jedoch bestimmt einer der interessantesten Ansätze aus dem grossen Pool der agilen Methoden der Software-Entwicklung.

Literatur

[1] Baumeister H., Wirsing M., *Applying Test-First Programming and Iterative Development in Building an E-Business Application.* Proceedings International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet, SSGRR 2002, L'Aquila, Italy, January 2002

[2] Beck K., *Embracing Change with Extreme Programming.* IEEE Computer, p. 70-77, Oct. 1999

[3] Beck K., *Extreme Programming Explained.* Addison Wesley, 1999

[4] Beck K., *Test-Driven Development by Example.* Addison Wesley, 2003

[5] Boehm B., *Software Engineering Economics.* Englewood Cliffs, N.J.: Prentice-Hall, 1981

[6] Müller M., Hagner O., *Experiment about Test-first programming.* Conference on Empirical Assessment In Software Engineering (EASE), Keele, UK, April 2002

[7] Müller M., Tichy W., *Case study: Extreme programming in a University Environment.* International Conference on Software Engineering (ICSE), p. 537-544, Toronto, Canada, May 2001

[8] Royce W., *Managing the Development of Large Software Systems.* Proceedings IEEE WESCON. 1-9, 1970

[9] Westphal F., *Testgetriebene Entwicklung mit JUnit.* Dpunkt Verlag, erscheint im März 2004 ausgewählte Kapitel unter http://www.frankwestphal.de/TestgetriebeneEntwicklung.html

[10] http://www.junit.org