eXtreme Programming

Konzepte, Ziele, Methoden
Praktische Erfahrungen aus XP-Projekten

Eine Arbeit im Rahmen des Seminars
„Refactoring in eXtreme Programming“
von Björn Zeiger

Universität Paderborn
AG Prof. Kastens

Januar 2003

Inhalt

1 Einleitung, Motivation 3
2 Warum XP? 3
2.1 Was ist eXtreme Programming? 3
2.2 Wann sollte XP eingesetzt werden? 6
3 Konzepte und Methoden 7
3.1 Prozess 7
3.2 Teamwork 7
3.3 Programmieren 9
3.3.1 Programmier-Standards 9
3.3.2 Testen 10
3.3.3 Refactoring 14
3.3.4 Einfaches Design 14
3.4 Abhängigkeiten der Methoden untereinander 15
4 Beispiele aus der Praxis 16
4.1 Projekt „Kermit“ (aus: [LRW02]) 16
4.1.1 Kontext 16
4.1.2 XP-Einsatz 17
4.1.3 Bewertung 17
4.2 Erfahrungen mit XP im universitären Umfeld 17
4.2.1 Erfahrungen: Programmieren in Paaren 18
4.2.2 Erfahrungen: Iterationsplanung 18
4.2.3 Erfahrungen: Testen 18
4.2.4 Erfahrungen: Refactoring 18
4.2.5 Erfahrungen: Skalierbarkeit 18
4.2.6 Zusammenfassung 19
5 Schlussbemerkungen 19
6 Empfehlungen zur Vertiefung des Themas 19
7 Quellenangaben 20

1 Einleitung, Motivation

Diese Ausarbeitung ist entstanden im Rahmen des Seminars *Refactoring in eXtreme Programming* im Januar 2003 an der Universität Paderborn und hat zum Ziel, die Zusammenhänge zwischen Refactoring und dem eXtreme Programming als Modell der Softwareentwicklung herzustellen.

Dazu werden im Wesentlichen die in XP verwendeten Konzepte und Methoden vorgestellt. Darüber hinaus werden auch Beispiele aus der Praxis vorgestellt, um die Auswirkungen des Einsatzes von XP in der Softwareentwicklung aufzuzeigen.

2 Warum XP?

2.1 Was ist eXtreme Programming?

eXtreme Programming - oder kurz XP – ist ein Softwareentwicklungsprozess und im Wesentlichen zurückzuführen auf die Arbeit von Kent Beck. Erste Publikationen zum Thema gab es um 1995, die Diskussion auf breiter Ebene fand ihre Anfänge auf der OOPSLA 1998. Grundsätzlich ist XP den so genannten *leichten* oder auch *agilen* Prozessen zuzuordnen. Agile Prozesse sind geprägt von ihrer Dynamik, von der Fähigkeit, sich an sich ändernde Bedingungen anzupassen. Im Gegensatz dazu verfolgen die *schweren* Prozesse, wie zum Beispiel der Unified Process oder das V-Modell, eher statische und wenig flexible Abläufe.

Insgesamt hat sich XP im laufe der letzten Jahre als *der* leichte Prozess schlechthin etabliert. Eines der auffälligsten Charakteristiken dieses Modells im Vergleich zu Vertretern der schweren Prozesse ist der Wert des Programmierens, der hier eindeutig in den Vordergrund gestellt wird. Weitere bezeichnende Merkmale sind die Anwenderorientierung und der Pragmatismus: es wir nur das entwickelt, was wirklich gefordert ist.

Das eXtreme Progamming besteht im Wesentlichen aus zwölf Konzepten und Methoden, was XP zu einem einfachen Modell macht. Die Firma Hype Softwaretechnik GmbH beschreibt auf ihrer Website XP im Vergleich zum Unified Process wie folgt:

XP	Unified Process
Als Leitbild: Software-Entwicklung als Kommunikationsprozess	Leitbild: Software-Entwicklung als Ingenieurs-Wissenschaft
Der Kunde formuliert seine Anforderungen in Interaktion mit der Software, die der Entwickler in mehreren Stufen entsprechend darauf abstimmt.	Der Kunde legt seine Anforderungen fest, der Entwickler setzt sie in eine Software um.
Die Anforderungsanalyse erfolgt auf Story-Cards[1] und am "lebenden Programm"	Anforderungsanalyse in einem formal aufgebauten Pflichtenheft
frühe Produktivschaltung, kurze Zyklen	späte Produktivschaltung, lange Zyklen

[1] Siehe *Prozess: Planungsspiel*

Vertrauensbildung durch schnelle Umsetzung von Kundenwünschen am laufenden Programm	Vertrauensbildung durch Zertifizierungen und formale Qualitätssicherungsprozesse
Das Entwicklungs-Risiko wird als unvermeidlich akzeptiert und aktiv gemanagt	Das Entwicklungs-Risiko soll von vornherein so weit wie möglich ausgeschlossen werden
Metapher: Autofahrt (Der Kurs kann während der Fahrt jederzeit den Begebenheiten angepasst werden.)	Metapher: Mondrakete (Der Kurs steht bereits vor dem Start fest und kann während der Reise nicht mehr geändert werden.)

Eine wesentlicher Unterschied zwischen leichten und schweren Prozessen im Allgemeinen und dem Unified Process und eXtreme Programming im Speziellen wird deutlich, vergleicht man die Ablauffolgen einzelner Phasen der Prozesse.

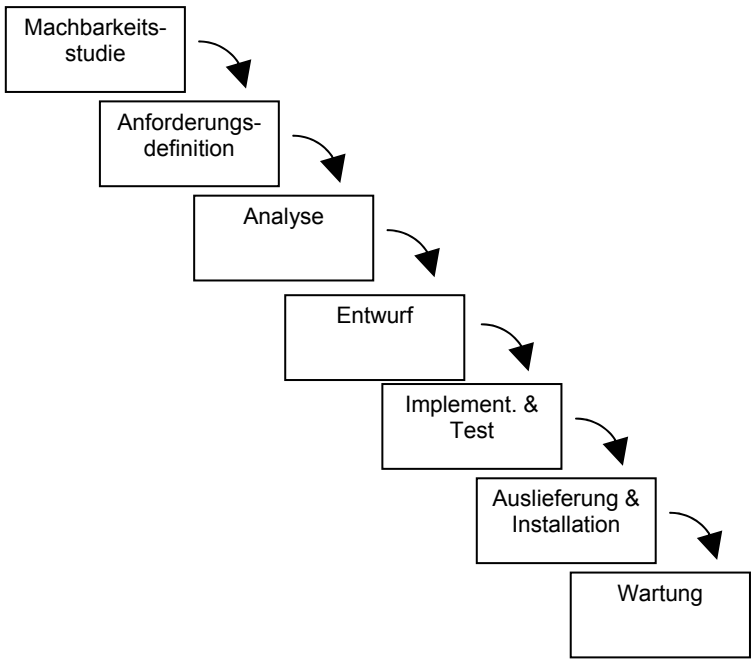

Abbildung 1: Phasenablauf im Unified Process

Wie aus Abbildung 1 hervorgeht, handelt es sich beim Unified Process um einen relativen starren, unflexiblen Phasenablauf. Phasen werden der Reihe nach abgearbeitet, Rückschritte, zum Beispiel bei sich ändernden Anforderungen, sind nicht vorgesehen. Dies ist ein wesentlicher Faktor, zumal sich dieses Vorgehen entscheidend auf die Kosten von Änderungen im Verlauf des Projektes auswirkt.

Diverse wissenschaftliche Studien bestätigen, dass durch diesen strikten Phasenablauf die Kosten in Relation zur Zeit exponentiell ansteigen, das heißt, eine Änderung, die gegen Ende des Projektes vorgenommen wird, ist um ein vielfaches teurer als die gleiche Änderung zu Anfang des Projektes. Dieser Kostenverlauf ist in Abbildung 2 schematisch dargestellt.

Ein direkter Vergleich mit den Abläufen in XP erweist sich als schwierig, da relativ wenige Überschneidungen in beiden Modellen existieren. Zum Verständnis kann a-

ber ein ungefährer Ablauf wie in Abbildung 3 zu sehen ist herangezogen werden. Daraus wird vor allem eines deutlich: Phasenabläufe sind in diesem Modell wesentlich flexibler gehalten, Rücksprünge zu an sich bereits abgeschlossenen Phasen sind nicht nur möglich, sondern explizit gefordert.

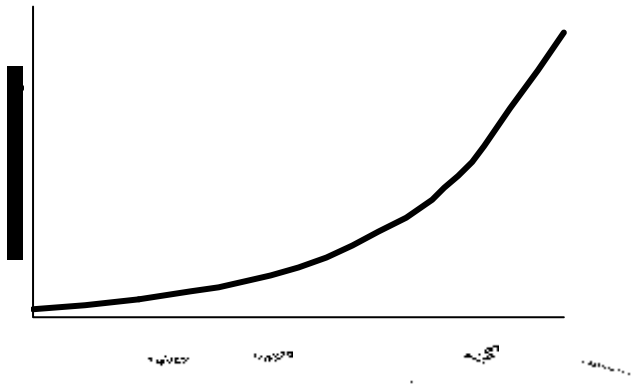

Abbildung 2:Kosten von Änderungen im UP

Diese Tatsache bringt Kent Beck in [B00] zu der These, dass sich der Zeitpunkt, an dem eine Änderung vorgenommen wird, kaum auf die damit verbundenen Kosten auswirkt. Leider wird diese These bisher nicht durch entsprechende Untersuchungen belegt, Beck beschreibt diesen Umstand jedoch als *die* technische Prämisse von XP; eine eventuelle Entscheidung für XP als zu verwendendes Modell geschieht also nicht auf Grundlage dessen, *weil* der in Abbildung 4 skizzierte Kostenverlauf erreicht wird, sondern *damit* dieser realisiert werden kann.

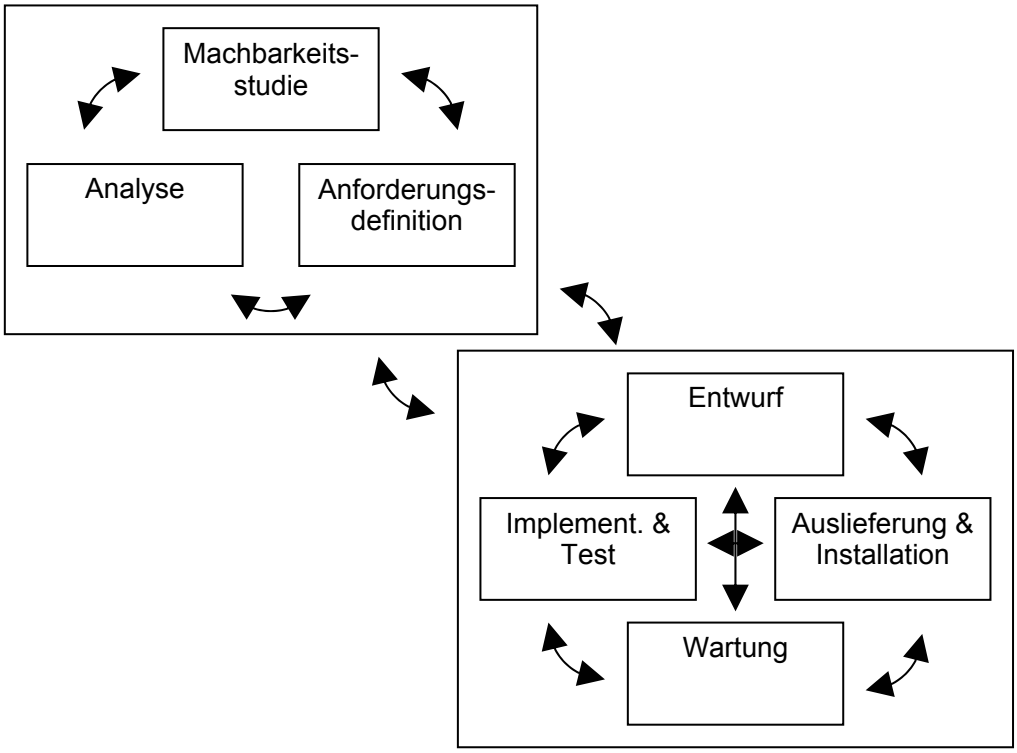

Abbildung 3: Phasenablauf in XP

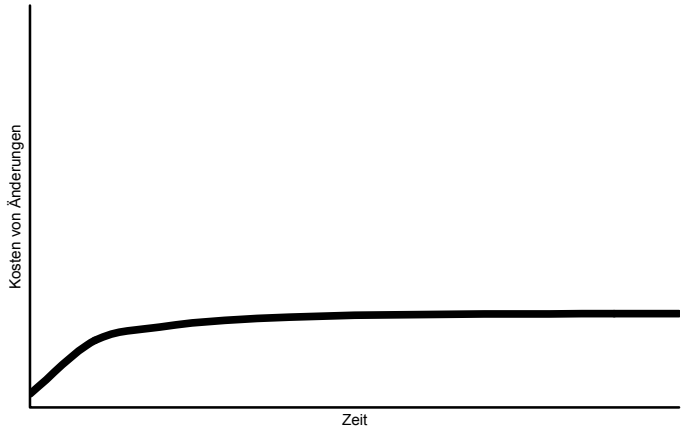

Abbildung 4: Kosten von Änderungen in eXtreme Programming

2.2 Wann sollte XP eingesetzt werden?

Als Indikatoren für die Entscheidung, ob XP das „richtige“ Modell ist, können unter anderem die folgenden vier Punkte benannt werden:

1. **Keine klaren Vorstellungen über Funktionalität**

   Die Problembeschreibung ist recht einfach: Ein potentieller Kunde weiß, dass er ein System braucht um ein Problem zu lösen, hat jedoch keine genauen Vorstellungen darüber, was dieses System im Einzelnen für Funktionalität zur Verfügung stellen muss. Daraus ergeben sich unmittelbar

2. **Sich ständig ändernde Anforderungen**

   Im Verlauf des Projekts stellt sich heraus, dass eine vorab spezifizierte Funktion nicht das erfüllt, was der Kunde eigentlich will oder braucht. Es müssen entsprechende Änderungen am System vorgenommen werden.

3. **Hohes Projektrisiko**

   Von hohem Projektrisiko muss gesprochen werden, wenn das Entwickler-Team entweder mit der Entwicklung eines Systems betraut wird, das von der Art her (Art beinhaltet sowohl Architektur bzw. Technik als auch Inhalt) neu für das Team ist oder wenn der Erfolg des Projekts ausschließlich von Fristen abhängig ist (z.B. Fertigstellung bis zu Beginn einer Messe).

4. **Einbeziehung des Kunden**

   Gerade bei hohem Projektrisiko kann für den Erfolg entscheidend sein, dass der Kunde von Anfang bis Ende des Projekts involviert ist.

Darüber hinaus gibt es selbstverständlich weitere Kriterien, unter anderem spielt die Größe des Teams eine wesentliche Rolle bei der Entscheidung für oder gegen XP.[2]

[2] vgl. *Beispiele aus der Praxis/Erfahrungen mit XP im universitären Umfeld. Dort ist die Skalierbarkeit der Teamgröße bei XP-Projekten ein wesntlich Bestandteil der Untersuchungen.*

3 Konzepte und Methoden

Wie im vorangegangene Abschnitt bereits erwähnt, definiert sich XP im Wesentlichen durch die Kombination von zwölf Konzepten und Methoden, die, für sich betrachtet, weder neu noch besonders revolutionär sind, in Kombination aber sowohl Stärken optimal zur Geltung bringen als auch Schwächen untereinander kompensieren können:

1. Kunde vor Ort
2. Planungsspiel
3. Testen
4. Metapher
5. 40-Stunden-Woche
6. Kurze Releasezyklen
7. Programmieren in Paaren
8. Gemeinsame Verantworlichkeit
9. Fortlaufende Integration
10. Refactoring
11. Einfaches Design
12. Programmier-Standards

Zur besseren Übersicht werden diese zwölf Methoden im Folgenden in drei Bereiche untergliedert: Programmieren, Teamwork und Prozess. Der Schwerpunkt der weiteren Betrachtung liegt auf dem Bereich Programmieren, der die Methoden *Testen*, *Programmierstandards*, *Refactoring* und *Einfaches Design* umfasst.

3.1 Prozess

Der Bereich Prozess umfasst Methoden, bei denen der Kunde direkt mit einbezogen ist, das *Planungsspiel* und *Kunde vor Ort*.

Das **Planungsspiel** beschreibt Treffen zwischen Kunden und Entwicklern. Hier wird die Funktionalität des zu entwickelnden Systems definiert. Der Kunde beschreibt seine Anforderungen informell auf sog. Story-Cards (i.d.R. handelt es sich hierbei um nichts anderes als gewöhnliche Karteikarten), diese werden anschließend durch die Entwickler mit einer Aufwandsschätzung versehen. Im letzten Schritt des Planungsspiels werden die Stories vom Kunden priorisiert und es erfolgt eine Festlegung der Reihenfolge. (siehe auch *Kurze Releasezyklen*).

Die Besonderheit beim Planungsspiel gegenüber der Anforderungsanalyse des Unified Process besteht darin, dass in XP das Planungsspiel nicht nur einmalig am Anfang des Projekts durchgeführt, sondern in regelmäßigen Abständen wiederholt wird.

Beim **Kunden vor Ort** handelt es sich im Idealfall zum einen um einen späteren Anwender des zu entwickelnden Systems, zum anderen ist er tatsächlich physisch vor Ort. Neben der Teilnahme am Planungsspiel kommen ihm zwei weitere Aufgaben zu: Er steht dem Entwicklerteam als Ansprechpartner für inhaltliche Fachfragen zur Verfügung und dient gleichzeitig als Pilotanwender, heißt, er hat ständig Zugriff auf das gerade aktuelle System.

3.2 Teamwork

Die Methoden des Bereichs Teamwork beschreiben die Art und Weise, in der Arbeit in einem XP-Team verrichtet wird.

Die **Metapher** dient zur Beschreibung einer gemeinsamen Zielvorstellung und ist handlungsleitend für die Beschäftigung mit Architekturen und Basis für die Kommunikation zwischen Entwickler und Kunde, um einen gemeinsamen Namensraum zu schaffen.

Eine weitere XP-Methode, die die Zusammenarbeit zwischen Entwicklern und Kunde unterstützen soll, sind die **Kurzen Releasezyklen**. In der unter *Planungsspiel* bereits erwähnten Festlegung der Reihenfolge für die Implementierung der User-Stories gibt es zwei Unterteilungen: die Stories werden zum einen in Releases, diese wiederum in Iterationen unterteilt. Diese Unterteilung in zeitliche Abschnitte hat den Grund, dass auf diesem Weg dem Kunden zeitnah konkrete Statusmeldungen über den Stand der Entwicklung gegeben werden können.

Nach jeder Iterationsphase folgt eine Präsentation über deren Ergebnisse, nach jedem Releasezyklus erfolgt zusätzlich eine Installation des Systems beim Kunden. Die Aufteilung der Releases in Iterationen beruht auf der Tatsache, dass die Erstellung eines Relaeses immer mit Mehraufwand verbunden ist, wie z.B. der Erstellung von Installationsroutinen oder der Installation beim Kunden.

Als Grundlage für die Aufteilung der Stories in Releases kann z.B. eine Aufteilung nach Benutzergruppen dienen.

Die nun folgenden Methoden zielen weniger auf die Zusammenarbeit zwischen Kunden und Entwicklern ab, sie beschreiben vielmehr die Arbeitsorganisation innerhalb des Entwicklerteams.

Die beiden hierbei die Arbeit am direktesten beeinflussenden Methoden sind die *Gemeinsame Verantwortlichkeit* und das *Programmieren in Paaren*.

Die Methode **Gemeinsame Verantwortlichkeit** sagt aus, dass jeder Entwickler für den kompletten Code verantwortlich ist. Das heißt: Sieht jemand eine Möglichkeit, Verbesserungen am Code vorzunehmen, soll er diese Verbesserungen vornehmen, unabhängig davon, ob er der Autor der entsprechenden Stelle ist oder nicht. Die Anzahl der zu vollbringenden Verbesserungen möglichst gering zu halten ist eines der Ziele des **Programmierens in Paaren**. In XP entwickeln immer zwei Entwickler an einem Rechner. Dieses Verfahren soll eine bessere Qualität des Codes garantieren. Diesem Umstand wird unter anderem dadurch Rechnung getragen, dass durch das 4-Augen-Prinzip ständige Code-Reviews durchgeführt werden.

Zusammengefasst kann man von folgender Aufgabenverteilung in einem Entwicklerpaar sprechen: Derjenige an der Tastatur implementiert die gerade zu bearbeitende Anforderung, der Andere unterstützt dabei und stellt eher strategische Überlegungen an. Ein Wechsel der Rollen soll jederzeit möglich sein, z.B. wenn der Entwickler, der nicht an der Tastatur sitzt, eine bessere Idee für die Implementierung einer Funktion hat.

Im Anschluss an die Implementierung einer Story erfolgt im Rahmen der **Fortlaufenden Integration** ein Einspielen dieser auf einem eigens dafür vorgesehen Integrationsrechner. Erst nach dem erfolgreichen Ausführen aller Tests (siehe *Testen*) erfolgt die Freigabe des neuen Codes. Dadurch existiert immer ein lauffähiges System, welches den tatsächlichen Stand der Entwicklung widerspiegelt.

Die letzte zu nennende Methode des Teamworks ist die **40-Stunden-Woche**. Dieses Konzept bedeutet schlicht und einfach, dass kein Mitglied des Teams mehr arbeiten soll als in seinem Vertrag steht. Zum Thema Überstunden gibt es eine einfache Regel: Überstunden innerhalb einer Woche sind erlaubt, nie aber in zwei aufeinander folgenden Wochen.

Der Grund liegt auf der Hand: In einer Woche, in der z.B. ein Release ansteht, ist es legitim, durch den erhöhten bzw. zusätzlichen Aufwand etwas mehr Zeit als üblich zu investieren. Allerdings sagt diese Regel (oder im speziellen K. Beck) auch, dass für

den Fall, dass einzelne Entwickler oder das gesamte Team in min. zwei aufeinander folgenden Wochen Überstunden machen, ein Problem vorliegt, dessen Ursachen nicht im Zeitmangel begründet sind. Ein möglicher Grund könnte z.B. eine falsche Aufwandsschätzung sein.

3.3 Programmieren

Der dritte Bereich umfasst die Methoden, die mittelbar und unmittelbar mit der tatsächlichen Programmierung in Zusammenhang stehen. Da Schwerpunkt dieser Ausarbeitung, werden diese Methoden in ausführlicherer Form vorgestellt als die voran gegangenen.

3.3.1 Programmier-Standards

Die Notwendigkeit von definierten Standards in einem XP-Team lässt sich bereits an sehr kleinen Beispielen veranschaulichen. Die Abbildungen 5 und 6 zeigen jeweils eine Implementierung eines Währungsrechners.

```
public class CurrencyCalculator {

    public static final  int    DM_TO_EURO       = 0;
    public static final  int    EURO_TO_DM       = 1;
    private static final double RATE_OF_EXCHANGE = 1.95583;

    private int mode;

    public CurrencyCalculator(int _mode) {
        mode = _mode;
    }

    public double caluclateValue(double value) {
        switch (mode) {
            case DM_TO_EURO:
                return value * RATE_OF_EXCHANGE;
            case EURO_TO_DM:
                return value / RATE_OF_EXCHANGE;
        }
        return -1d;
    }
}
```

Abbildung 5: Beispiel-Code nach den Java Code Conventions

```
public class Rechner {
public static final int de = 0;
public static final int ed = 1;
public double rechne(double v) {
switch (m)
{case de: return v*k;
case ed: return v/k;}
return -1d;}
private static final double k = 1.95583;
public Rechner(int _m)
{m = _m;}
private int m;
}
```

Abbildung 6: Beispiel-Code ohne Konventionen

Bereits auf den ersten Blick wird ein wesentlicher Unterschied deutlich: Die Lesbarkeit des Codes aus Abbildung 5 ist wesentlich höher. Zurückzuführen ist dies auf die enge Anlehnung an die Java Code Conventions von Sun Microsystems, die zumindest in Ansätzen jedem Java-Programmierer geläufig sind.

Bei näherer Betrachtung fallen weitere Unterschiede auf, z.B. in der Anordnung der verschiedenen Komponenten oder auch in der Wahl der Namen.

Im Wesentlichen bedingt durch die Gemeinsame Verantwortlichkeit definiert das Entwickler-Team in XP deshalb Standards z.B. für

- Benennung von Klassen, Methoden und Variablen (→ Anlehnung der Namen an die Metapher, verwendete Sprache, Art der Zusammensetzung von mehreren Wörtern in einem Namen, z.B. calculate_values vs. calculateValues)
- Positionen von Variablen- und Methoden-Deklarationen (z.B. Reihenfolge: Klassenvariablen, Konstruktoren, Methoden oder Sortierung nach Sichtbarkeit: public, protected, default, private)
- Formatierung (Position von Klammern: { am Ende der Zeile oder in der nächsten Zeile, Einrückungen, Zeilenumbruch)
- Kommentare
- Aber auch: Entwicklungsumgebungen, Shortcuts, etc.

Das Ziel dieser Bemühungen um Standards ist, dass das Erkennen des Autors eines Codesegments nicht möglich sein soll.

3.3.2 Testen

Das Testen von Programmen hat im Wesentlichen zwei Gründe. Zum einen stärken erfolgreiche Tests das Selbstvertrauen der Entwickler in ihre Arbeit, zum anderen bieten Tests Sicherheit für den Kunden, dass das entwickelte System tatsächlich wie gefordert funktioniert.

In XP ist das Testen eine tragende Säule. Tests werden soweit als möglich[3] automatisiert. Bei automatisierten Tests handelt es sich um solche, die ohne weiteres Zutun des Entwicklers berechnete Werte mit zuvor definierten Erwartungswerten abgleichen, und dem Entwickler lediglich über Erfolg oder Misserfolg informieren. Dank dieser Automatisierung können sämtliche verfügbaren Tests mit minimalem Zeitaufwand immer dann ausgeführt werden, wenn eine Änderung oder Erweiterung des Systems vorgenommen wurde. Fehler sind dadurch schnell zu lokalisieren, der Zeitaufwand für Fehlersuche und –korrektur wird deutlich reduziert.

Für die Automatisierung der Tests gibt es inzwischen zahlreiche Werkzeuge, das für Java am weitesten verbreitete ist das JUnit-Framework.

Der Implementierung einer Story oder Änderung ist immer das Schreiben eines entsprechenden Testfalls vorangestellt. Konkret wiederholt sich immer der folgende Ablauf:

1. Schreiben eines Testfalls für die zu implementierende Story
2. Implementierung eines Klassen- bzw. Methoden-Skeletts (Beschränkung auf Methodenköpfe, soweit als möglich), vergleichbar mit der Implementierung einer abstrakten Klasse

[3] Beck geht davon aus, dass alles automatisch getestet werden kann.

3. Ausführen des Tests
4. Implementierung der fehlenden Methodenrümpfe
5. Ausführen des Tests

In Punkt 3 ist natürlich nahezu ausgeschlossen, dass der Test erfolgreich ist. In Punkt 1 werden automatisch schon Rahmenbedingungen für die weitere Implementierung festgelegt, unter anderem z.B. Klassen- und Methodennamen, Parameter usw.

Zum besseren Verständnis nun ein Beispiel für das Testen in XP. Grundlage ist folgende Beispiel-Story:

> **Freie Kapazitäten anzeigen (# 1202/7)**
> *Nach Eingabe von zwei Daten prüft das System, welche Zimmer im gesamten Zeitraum frei sind und gibt eine Auflistung dieser aus. Ebenfalls gezeigt werden soll die Anzahl der freien Betten im gewählten Zeitraum.*

Im ersten Schritt wird der entsprechende Testfall geschrieben, mit dem die korrekte Funktion der zu implementierenden Story anhand von Erwartungswerten überprüft werden kann. In Abbildung 7 ist ein solcher Testfall exemplarisch dargestellt.

```
class FreeCapacityDetector_Test extends TestCase {

      private FreeCapacityDetector capDetector;
      private List freeRooms0 = …
      private List freeRooms1;
      private int freeBeds0 = …
      private int freeBeds1;

      public void testGetFreeCapacities() {
            freeRooms1 = capDetector.getFreeRooms();
            freeBeds1 = capDetector.getFreeBeds();
            assertEquals(freeRooms0, freeRooms1);
            assertEquals(freeBeds0; freeBeds1);
      }

      protected void setUp() {
            capDetector = new FreeCapacityDetector(
                                    DateFormat.parse(„24.03.02“),
                                    DateFormat.parse(„26.03.02“));
     }
}
```

Abbildung 7: Beispiel-Code TestCase

Zum Verständnis: Der in Abbildung 7 dargestellte Testfall benutzt das JUnit-Framwork, welches unter anderem eine Klasse `TestCase` zur Verfügung stellt. Diese wiederum implementiert `assertEquals(param1, param2)`-Methoden, die zwei übergebene Parameter auf Gleichheit prüfen, wobei der eine Parameter das erwartete, der andere Parameter das berechnete Ergebnis ist.

Aus dem Beispiel wird deutlich, dass beim schreiben des Testfalls schon einige Entscheidungen für die spätere Implementierung getroffen werden. So werden der Klassenname, die Parameter für den Konstruktor, die entsprechenden Methodennamen sowie deren Rückgabewerte festgelegt.

Im nächsten Schritt wird, wie in Abbildung 8 dargestellt, ein Klassenskelett implementiert.

```
class FreeCapacityDetector {

	private List freeRooms;
	private int freeBeds;

	public FreeCapacityDetector(Date dateFrom, Date dateTo) {}

	public List getFreeRooms() {
		return freeRooms;
	}

	public int getFreeBeds() {
		return freeBeds;
	}
}
```

Abbildung 8: Beispiel-Code Klassenskelett

Die eigentliche Klasse (bzw. Funktion) wird genau soweit implementiert, dass sie kompilierbar und der Test ausführbar ist. Damit ist gewährleistet, dass im Testfall gewählte Namen usw. richtig übernommen wurden. Nach dem Kompilieren erfolgt eine erste Ausführung des in Schritt 1 implementierten Testfalls.

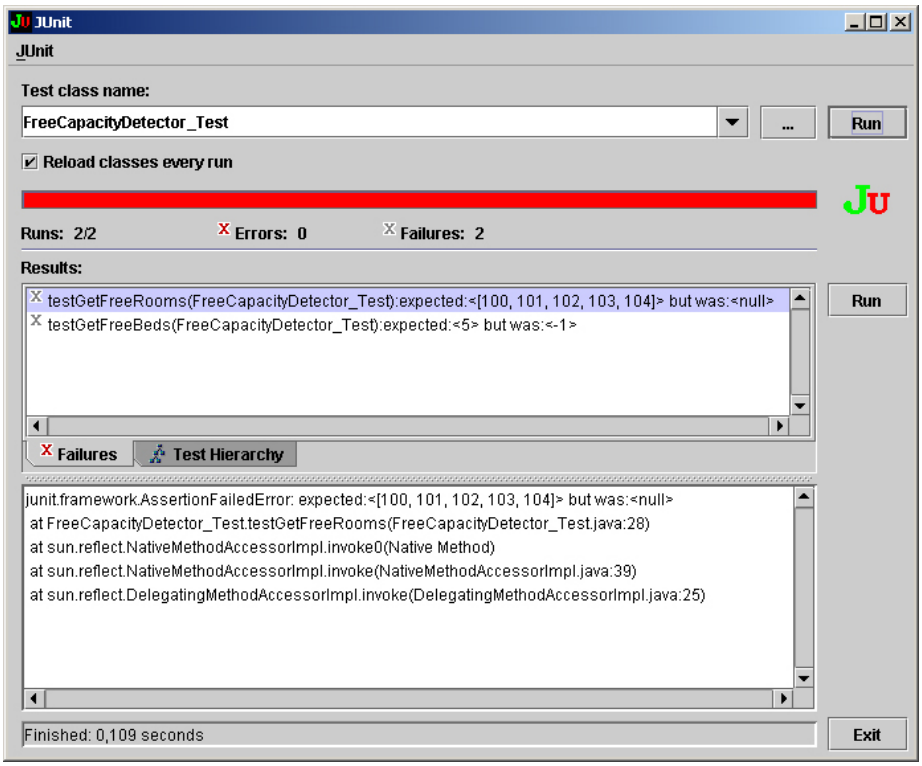

Abbildung 9: JUnit-GUI nach dem ersten Test

Abbildung 9 zeigt die Swing-GUI des JUnit-Frameworks nach der ersten Ausführung. Da bisher im Wesentlichen nur Methodenköpfe implementiert wurden, ist es nicht weiter verwunderlich, dass beide Tests fehlgeschlagen sind. Der nächste Schritt ist nun die Implementierung der Methodenrümpfe, also der eigentlichen Funktionalität

der Story. In Abbildung 10 ist der „fertige“ Code skizziert, zu Gunsten des Platzbedarfs werden die Methodenrümpfe lediglich als Kommentare dargestellt.

```
class FreeCapacityDetector {

	private List freeRooms;
	private int freeBeds;

	public FreeCapacityDetector(Date dateFrom, Date dateTo) {
		//do sth with these dates
	}

	public List getFreeRooms() {
		//do sth to fill freeRooms
		return freeRooms;
	}

	public int getFreeBeds() {
		//do sth to set freeBeds
		return freeBeds;
	}
}
```

Abbildung 10: Beispiel-Code mit implementierten Methodenrümpfen

Nach der Implementierung der Methodenrümpfe wird der Test erneut ausgeführt. JUnit zeigt lediglich, dass die ausgeführten Tests erfolgreich waren, das heißt, dass die berechneten Werte mit den „hart codierten“ Erwartungswerten übereinstimmen (vgl. Abbildung 11). Ein manueller Abgleich der erwarteten und berechneten Werte ist nicht notwendig.

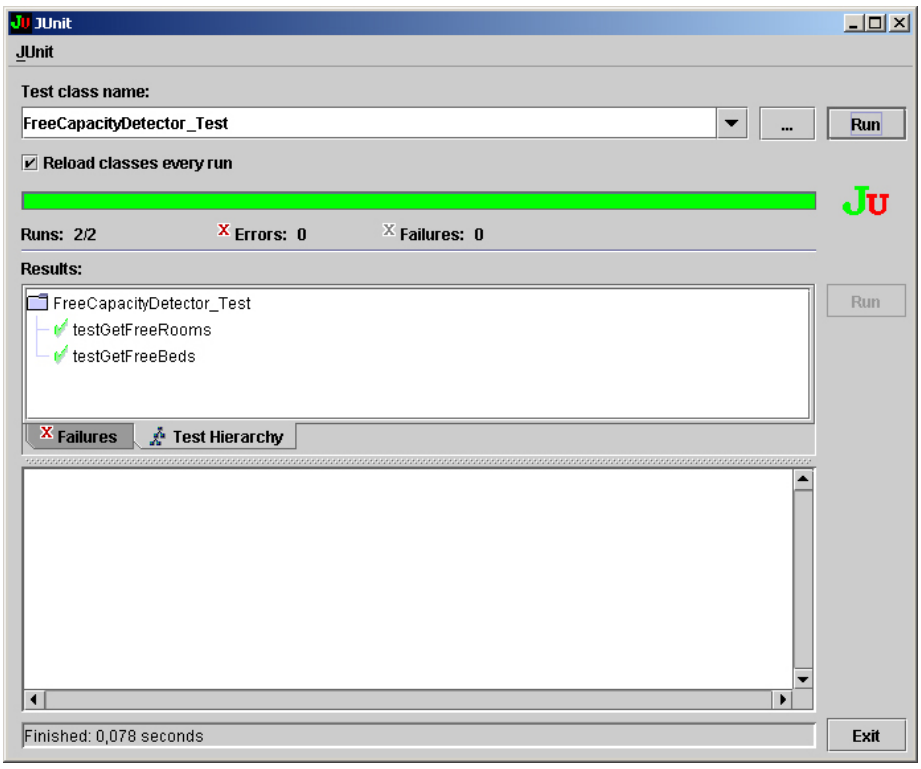

Abbildung 11: JUnit-GUI nach dem zweiten Test

3.3.3 Refactoring

Unter Refactoring versteht man die Anwendung einzelner oder mehrerer Transformationen, mit denen sich Änderungen im und am System vornehmen lassen. Die Besonderheit dieser Änderungen ist, dass sich das beobachtbare Verhalten des Systems nicht ändert, heißt, dass sich die Auswirkungen für einen Benutzer des Systems nicht bemerkbar machen. Derartige Änderungen zielen darauf ab, die interne Struktur der Software zu verbessern.

Gängige Transformationen sind zum Beispiel das *extrahieren von Methoden* (ein Teil einer Methode wird selbst zur Methode; Anwendung: zur besseren Lesbarkeit durch kürzere Methoden oder auch zur Vermeidung duplizierten Codes), die *Umbenennung von Variablen* (Anwendung: Wahrung der Konsistenz bei der Namensgebung) oder auch das *ändern von Parametern* (Anwendung: einer Methode wurden bislang Parameter übergeben, die nicht verwendet werden oder aber eine Methode benötigt fortan mehr und/oder andere Parameter).

Viele Refactorings werden durch diverse Werkzeuge unterstützt, die eine weitestgehend automatische Durchführung der Refactorings ermöglichen.

Eng verknüpft mit dem Anwenden einer Transformation ist das anschließende Ausführen der vorhandenen Tests, um sicher zu gehen, dass sich das beobachtbare Verhalten tatsächlich nicht geändert hat.

Für einen tieferen Einblick in das Thema Refactoring sei an dieser Stelle auf die übrigen Arbeiten des Seminars verwiesen.

3.3.4 Einfaches Design

Beck benennt in [B00] folgende vier Regeln für ein einfaches Design:

1. Das System besteht alle Tests
2. Es enthält keine Redundanzen
3. Es spiegelt die Intention der Entwickler wieder
4. Es hat die geringste mögliche Anzahl von Klassen und Methoden

Die Reihenfolge der Nennungen legt auch gleichzeitig die Prioritäten fest. Nach Beck spricht man vom einfachsten Design, wenn die Anzahl der Klassen und Methoden soweit wie möglich reduziert wird, ohne die Regeln eins bis drei zu verletzen.

Eine weitere Forderung von XP bezüglich des Designs ist, dass Funktionen genau dann implementiert werden, wenn sie gebraucht werden. Analog gilt der Umkehrschluss, dass Funktionen, die jetzt nicht benötigt werden, auch nicht implementiert werden. XP stellt sich damit gegen die aus dem Unified Process bekannte Forderung *für heute implementieren, für morgen entwerfen*. Der Unterschied wir deutlich in den folgenden Darstellungen:

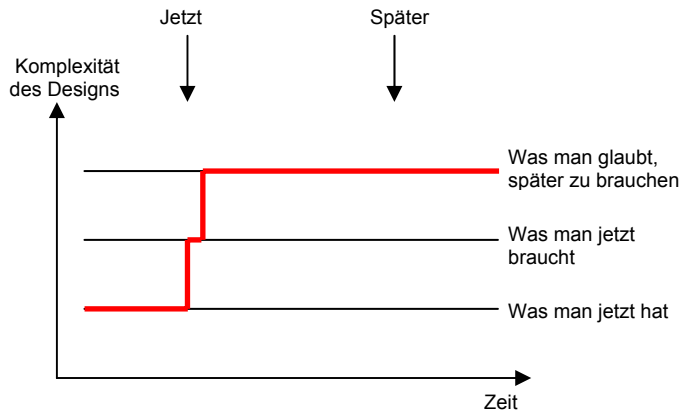

Abbildung 12: Designentwicklung im Unified Process

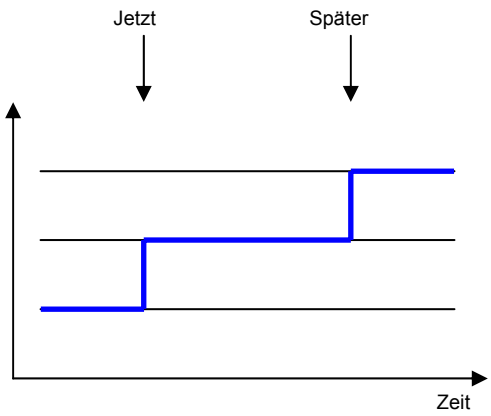

Abbildung 13: Designentwicklung in XP

3.4 Abhängigkeiten der Methoden untereinander

Die folgende Darstellung beschreibt die Abhängigkeiten der zwölf in XP verwendeten Konzepte und Methoden untereinander. Ein Doppelpfeil zwischen zwei Methoden bedeutet, dass die eine von der anderen abhängig ist. Zum Beispiel funktioniert die gemeinsame Verantwortlichkeit nur, wenn sich jedes Mitglied des Entwickler-Teams problemlos im Code zurechtfinden kann, was das Definieren von Programmier-Standards voraussetzt.

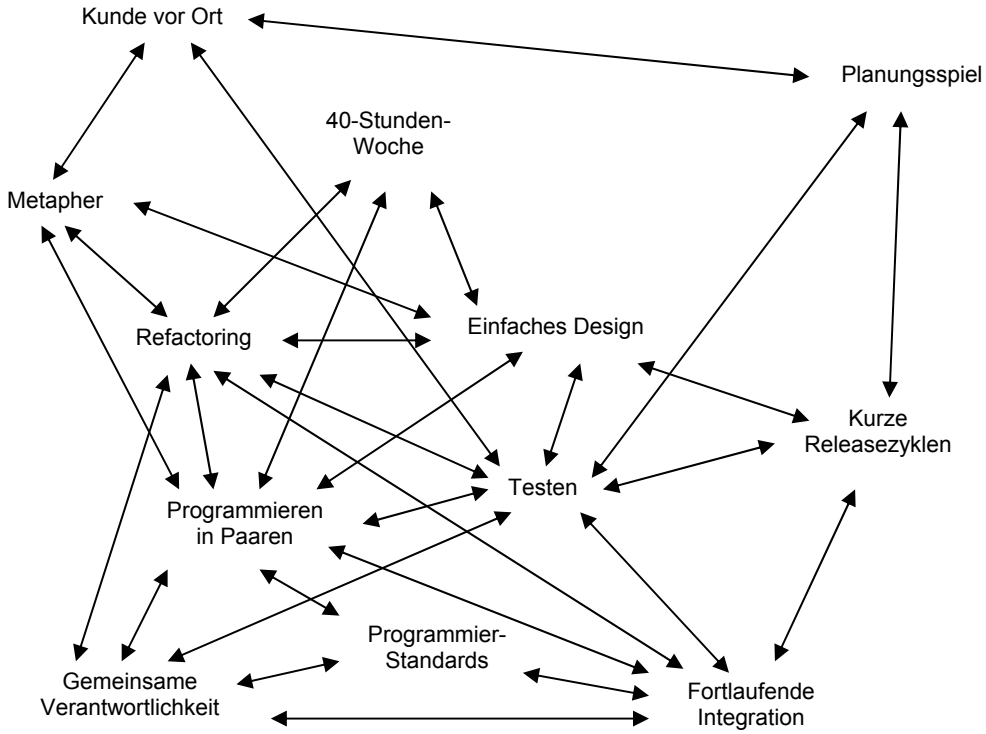

Abbildung 14: Abhängigkeiten der XP-Methoden

Besonders interessant im Rahmen dieses Seminars ist vor allem die Rolle des Refactorings im Zusammenspiel mit den übrigen Methoden des eXtreme Programmings. In der folgenden Tabelle sind die Relationen stichpunktartig zusammen gefasst.

	Refactoring in Abhängigkeit zur Methode …
Metapher	Die Metapher ist auch Vorgabe für die Architektur des Systems, somit definiert sie den Rahmen, in dem man sich beim Refactoring bewegen darf.
Gemeinsame Verantwortlichkeit	Entdeckt ein Entwickler eine Möglichkeit, etwas im System zu verbessern, tut er dies, unabhängig davon, ob er der Autor des zu ändernden Codeabschnitts ist oder nicht.
Testen	Eine ausreichende Zahl an Testfällen reduziert das Risiko mit Refactoring unerwünschte Nebeneffekte zu erzielen. Auswirkungen des Refactorings werden durch die automatisierten Tests unmittelbar im Anschluss bekannt.
Programmieren in Paaren	Durch ständige Code-Reviews besteht eine deutlich geringere Fehleranfälligkeit beim Refactoring.
Fortlaufende Integration	kleine Refactorings begünstigen Fortlaufende Integration dahingehend, dass je kleiner ein Refactoring ist, umso schneller eine weitere Integration durchgeführt werden kann.
Einfaches Design	Manchmal muss bei der Weiterentwicklung das System temporär komplizierter gemacht werden, um die zu implementierende Anforderung zu erfüllen. Ein anschließendes Refactoring soll dann für eine Vereinfachung des Designs sorgen.
40-Stunden-Woche	Kleine Refactorings in Kombination mit automatisierten Tests bedeuten selbst im Fehlerfall einen geringeren Zeitaufwand. Deshalb besteht in der Regel keine Notwendigkeit für Überstunden, um im Falle eines Fehlers das System wieder in einen lauffähigen Zustand zu versetzen.

4 Beispiele aus der Praxis

Zur Vertiefung der in XP verwendeten Konzepte und Methoden wurden zwei Beispiele für den Einsatz von XP in der Praxis in diese Ausarbeitung mit aufgenommen.

4.1 Projekt „Kermit“ (aus: [LRW02])

4.1.1 Kontext

Bei diesem Projekt ging es um die Erweiterung des Großrechnersystems eines internationalen Dienstleistungsunternehmens mit dem Ziel, länderspezifische Vorgänge für KfZ-Rückkaufverträge mit komplexen Vertragsbedingungen abzubilden.

Für die ersten zwei Monate standen zwei Entwickler zwei Monate lang für jeweils zwei Tage/Woche zur Ermittlung der notwendigen Fachkompetenz und anschließend

drei Entwickler über einen Zeitraum von sechs Monaten für jeweils zwei bis drei Tage/Woche zur Entwicklung des Kernsystems zur Verfügung.

4.1.2 XP-Einsatz

- Die automatisierten Tests erwiesen sich hilfreich zum „Ausprobieren“ von vom Kunden zur Verfügung gestellten Praxisbeispielen der komplexen Vertragsmodelle.
- Das Programmieren in Paaren und die gemeinsame Verantwortlichkeit kompensierten den dreimonatigen Ausfall eines Entwicklers aufgrund eines Bandscheibenvorfalls.
- Das Zerlegen von „großen“ Refactorings in genügend „kleine“ wurde als Schwierigkeit benannt.
- Aufgrund der von Anfang an sehr klaren Anforderungen des Kunden wurde auf das Planungsspiel verzichtet.
- Zur Unterstützung der Entwickler stand eine entsprechende Abteilung des Kunden ständig als „Kunde vor Ort“ per Telefon und Mail zur Verfügung.
- Eine „passende“ Metapher ermöglichte einfache Kommunikation mit dem Kunden.
- Um die gemeinsamen Vorstellungen über das System zu etablieren, wurde erst ein Prototyp entwickelt, welcher dann anschließend in kurzen Releasezyklen erweitert und um weitere Vertragsmodelle ergänzt wurde.

4.1.3 Bewertung

- Nach Auffassung der Entwickler hätte es eine früher einsetzbare Version für eine schnellere Rückkopplung über den Einsatz im Produktivbetrieb geben müssen (statt des Prototypings)
- Insgesamt wurde das Projekt sowohl vom Kunden als auch vom Systemhaus als positiv eingeschätzt
- Ledigleich die Festpreiskonstellation wurde seitens des Systemhaus als „harte Einschränkung“ für XP-Projekte bennant.

4.2 Erfahrungen mit XP im universitären Umfeld

Bei diesem Projekt handelt es sich um eine Studie der Universität Karlsruhe im Rahmen eines Praktikums über die Dauer von einem Semester.

An dem Praktikum nahmen zwölf Studenten der Informatik im Hauptstudium teil, als Programmiersprache wurde Java eingesetzt.

In den ersten drei Wochen beschäftigten sich die in zwei Gruppen á sechs Personen aufgeteilten Teilnehmer mit Übungen zu den XP-Methoden, in den restlichen Wochen wurde eine Verkehrssimulation zur Visualisierung eines Straßennetzes mit Ampeln, Vorfahrtsregeln etc. entwickelt.

Die Schwerpunkte der Beobachtung lagen auf den Methoden *Programmieren in Paaren*, *Iterationsplanung (Planungsspiel und Kurze Releasezyklen)*, *Testen* und *Refactoring*. Darüber hinaus wurde die Skalierbarkeit eines Teams im XP-Einsatz untersucht.

Es erfolgte eine Datenerhebung in Form von Fragebögen vor, während und nach Abschluss des Praktikums, die Ergebnisse sind in den folgenden Abschnitten zusammengefasst.

4.2.1 Erfahrungen: Programmieren in Paaren

Das Programmieren in Paaren wurde von den Teilnehmern überwiegend als positiv bewertet. Als Vorteil wurde das „vom Partner lernen“ in den Vordergrund gestellt, allerdings nahm dieses mit der Zeit ab.

Als Problem wurde die Strukturierung der Arbeit benannt. Die Verteilung der Aufgaben bei der Paarprogrammierung sein nicht klar. Teilweise benutzten die Paare zwei PCs, einen für die Implementierung, den zweiten z.B. zum präsent halten von Dokumetationen.

Insgesamt blieb unklar, ob Vorteile wie höhere Qualität nicht durch weniger personalintensive Ansätze erreicht werden können (z.B. paarweise Code-Reviews nach der Implementierung).

4.2.2 Erfahrungen: Iterationsplanung

Der Ansatz „highest priority first“ erwies sich als problematisch: Die Teilnehmer planten für die Zukunft, was nach Angabe der Verantwortlichen des Projekts auf die bisherige Ausbildung zurückzuführen sei, da die Teilnehmer bislang ausschließlich den Unified Process als Modell der Softwareentwicklung kennen gelernt hatten.

Es stellte sich die Frage, ob der Minimaler Entwurf lediglich eine Frage des Trainings oder generell eine schlechte Idee sei. Dieses Vorgehen wurde von den Teilnehmern als „Entwurf mit Scheuklappen“ bezeichnet.

4.2.3 Erfahrungen: Testen

Die Integration der Tests verlief von Anfang an relativ problemlos. Das ging soweit, dass der Verzicht auf die Implementierung eines Testfalls dazu führte, dass ein Team nicht weiterarbeiten konnte, bis der resultierende Fehler behoben war.

Die graphische Darstellung wurde im Rahmen des Projekts aus Zeitmangel nicht automatisiert getestet.

Insgesamt fühlten sich die Teilnehmer durch den Einsatz automatisierter Tests sicherer und bewerteten das Testen als beste Praxis von XP.

4.2.4 Erfahrungen: Refactoring

Während des Projekts wurde abgesehen von den Übungen innerhalb der ersten drei Wochen kein weiteres Refactoring durchgeführt. Als Gründe nannten die Verantwortlichen folgende Punkte:

- Die Teilnehmer hatten zu gründlicher entworfen
- Das Programm wurde ohne Testfälle umgeschrieben
- Zu geringe Projektgröße

4.2.5 Erfahrungen: Skalierbarkeit

Die erste Beobachtung hierbei ist relativ intuitiv: mehr Teilnehmer beim Planungsspiel bedeuten einen größeren Kommunikations- und damit auch Zeitaufwand. Das

Lösen abstrakter Probleme sei aufgrund des geringeren Kommunikationsaufwands und gruppendynamischer Eigenschaften wie zum Beispiel die geringere Hemmschwelle sich in kleineren Gruppen zu äußern in kleinen Teams wesentlich effektiver.

Aufgrund dieser Beobachtungen sei die Teamgröße in XP-Projekten ein kritischer Faktor. Die Verantwortlichen vertreten die Auffassung, die optimale Teamgröße liegt zwischen sechs und acht Personen. [4]

4.2.6 Zusammenfassung

- Das Programmieren in Paaren wurde schnell übernommen und als reizvolle Art der Softwareentwicklung empfunden.
- Der iterative Entwurf in kleinen Schritten ist schwierig.
- Die Test-first Entwicklung braucht Anlaufzeit, und ist manchmal schwer zu verwirklichen (zum Beispiel bei visuellen Komponenten wie GUIs).
- XP ist definitiv nur für kleine Teams geeignet.

5 Schlussbemerkungen

Diese Ausarbeitung umfasst mit den Konzepten und Methoden nur einen Ausschnitt aus dem XP umfassenden Themenspektrum. Auch wurde versucht, möglichst objektive Beschreibungen der Methoden zu liefern, eine intensive Diskussion von Vor- und Nachteilen der Methoden im speziellen und von XP im allgemeinen würde sicherlich den Rahmen einer solchen Ausarbeitung sprengen und ist auch für die Einordnung des Refactorings in eXtreme Programming nicht notwendig.

6 Empfehlungen zur Vertiefung des Themas

Zur Vertiefung des Seminar-Themas sei verwiesen auf die weiteren Ausarbeitungen, in Bezug auf weiterführende Konzepte in XP auf die in den Quellenangaben aufgeführte Literatur.

Insbesondere ist empfehlenswert, sich zur intensiveren Auseinandersetzung mit XP auf die Themen Werte, Prinzipien und Rollen zu konzentrieren.

Für die Diskussion von Vor- und Nachteilen ist unter anderem das Internetangebot der Gesellschaft für Informatik e.V. heranzuziehen.

[4] Im Vergleich dazu: Kent Beck spricht von einer optimalen Größe bei zwei bis zehn Personen.

7 Quellenangaben

[B00] K. Beck: *Extreme Programming*, Addison-Wesley, 2000

[JAH01] Jeffries, Anderson, Hendrickson: *Extreme Programming installed*, Addison-Wesley, 2001

[LRW02] Lippert, Roock, Wolf: Software entwickeln mit eXtreme Programming. Erfahrungen aus der Praxis, dpunkt Verlag, 2002

[S02] M. Sauer: Extreme Programming - Grundsätze für Agile Software-Entwicklung, Uni Erlangen, 2002
http://www3.informatik.uni-erlangen.de/Lehre/UML-Seminar/SS2002/xprog.pdf

[wwwW] D.Well: Extreme Programming: A gentle introduction
http://www.extremeprogramming.org

[wwwH] Hype Softwaretechnik GmbH
http://www.hype.de

[MT01] M. Müller, W. Tichy: Erfahrungen mit eXtreme Programming im universitären Umfeld, ObjektSpektrum, Ausgabe 03/2001