Kapitel 3
Extreme Programming (XP)

Als Kent Beck 1996 sein Prozessmodell Extreme Programming im Rahmen des Chrysler-Projekts *C3* entwickelte, war das Projekt bereits so gut wie gescheitert. Man war also bereit, auf neue Ansätze zu hören und gab somit Beck die Möglichkeit, seine zuerst informelle agile Praxis als neues Prozessmodell zu formulieren. Nach Becks eigener Aussage war das C3-Projekt ideal für seinen neuen Ansatz, weil sich die Spezifikationen schnell änderten und weil der Kunde keine endgültige Vorstellung vom Funktionsumfang hatte. In der ersten Phase war C3 sehr erfolgreich und wurde 1997 produktiv. Somit wurde C3 zum „Flaggschiff" für XP.

Kent Beck entwickelte sein Prozessmodell Extreme Programming (XP) (Beck 2000) im Rahmen des Chrysler-Projekts *Chrysler Comprehensive Compensation* (*C3*), das die Entwicklung einer Lohn- und Gehaltslisten-Software zum Ziel hatte (Beck u. Andres 2004). Das Projekt war zum Zeitpunkt des Eintretens von Beck bereits so gut wie gescheitert. Leider ist dies sehr oft erst die Voraussetzung, neuen Ansätzen eine Chance zu geben. Man gab also Beck die Möglichkeit, seine zuerst informellen agilen Projektpraktiken als neues Prozessmodell zu formulieren und das Smalltalk-Projekt neu zu starten. Nach Becks eigener Aussage war das Projekt ideal für seinen neuen Ansatz,

- weil sich die Spezifikationen schnell änderten,
- weil der Kunde keine endgültige Vorstellung vom Funktionsumfang hatte,
- weil eine hohe Produktivität der Programmierer gefordert war,
- weil das Projekt ein (auch zeitlich) hohes Risiko in sich trug,
- weil die Projektgruppe inkl. Managern, Programmierern und Kunden eine Stärke von weniger als zwölf Mitarbeitern hatte
- und weil die zu erstellende Software automatisiertes Testen erlaubte.

Im Gegensatz zu anderen Verfechtern des Agilen Manifests, wie z.B. Alistair Cockburn, dem Begründer des Crystal-Prozessmodells (Kap. 4), schreibt Kent Beck die Vorgehensweisen im Projekt strikt vor. XP definiert „Rules and Practices" zu Planung, Design, Kodierung und Testen, auf deren Einhaltung bestanden wird. Über allen Regeln stehen die fünf Werte: *Kommunikation, Einfachheit, Feedback, Mut* und *Respekt*.

E. Hanser, *Agile Prozesse: Von XP über Scrum bis MAP*, eXamen.press,
DOI 10.1007/978-3-642-12313-9_3, © Springer-Verlag Berlin Heidelberg 2010

3.1 Die fünf Werte

3.1.1 Kommunikation

Entscheidend für den Erfolg eines agilen Projekts ist die Kommunikation. Wir unterscheiden mindestens zwei Arten: die Kommunikation mit dem Kunden und die interne Kommunikation im Entwicklerteam.

Die interne Kommunikation ist die Grundlage für ein gemeinsames Vorgehen im Projekt. Nur wenn die Teammitglieder miteinander kommunizieren, ist die Grundlage gegeben, dass jedem Teammitglied die gemeinsamen Ziele des Teams bekannt sind. Außerdem kann sehr effizient auf abweichende Projektziele[16] reagiert werden. Interne Teamkommunikation gibt es nicht zum Nulltarif. Wie später noch gezeigt wird, ist ein vernünftiger psychologischer Prozess unabdingbar. Dies kann nur durch eine entsprechende Auswahl der Teammitglieder erreicht werden.

Die Kommunikation mit dem Kunden ist deutlich schwieriger als die interne Teamkommunikation. Oft sträubt sich das Team, den Kunden ins Team „aufzunehmen“, d.h. ihn als Teammitglied *vor Ort* zu akzeptieren, so wie es von Beck gefordert wird (Beck 2000). Immer wieder fällt auf, dass das Team Angst davor hat, dem Kunden zu sehr Einblicke in die projektinternen Abläufe zu gewähren. Oft hört man das Argument, der Kunde traute dem Team keine professionelle Software-Entwicklung zu, wenn er wüsste, wie intern „gebastelt“ würde. In der Praxis ist meist das Gegenteil der Fall. Der Kunde freut sich über die interne Sichtweise auf das Projekt, die ihm gewährt wird. Er fühlt sich ernst genommen und hat bereits in frühen Projektphasen Anteil an Erfolg und Misserfolg. Seine Achtung vor den Fähigkeiten des Teams wächst, und er entwickelt Verständnis für Probleme im Projektablauf.

3.1.2 Einfachheit

XP sucht immer nach der einfachsten Lösung, zumindest zum aktuellen Zeitpunkt. Dies betrifft nicht nur das Design einer Software, sondern auch den Ablauf, die Planung des Projekts. Bewusst beschäftigt man sich *nicht* mit Dingen, die man erst in späteren Releases implementieren muss. Man wählt immer die einfachste Lösung eines Problems, z.B. bei der Implementierung eines Algorithmus. Der Autor hat selbst miterlebt, wie in einem Messtechnik-Projekt ein genialer generischer Algorithmus zur dynamischen Erzeugung von Objekten zur Hardwarekapselung nach ein paar Jahren nicht mehr erweiterbar war. Der „Vater“ des Algorithmus hatte das Projektteam verlassen und kein anderes Teammitglied hatte dessen Code hinreichend gut verstanden. Berücksichtigt man, dass insgesamt nur etwa fünf Hardware-Objekte mit diesem Algorithmus dynamisch erzeugt wurden, war die Komplexität des generischen Algorithmus im Nachhinein sicherlich nicht zu rechtfertigen.

[16] sog. *Moving Targets.*

3.1.3 Feedback

Feedback (Rückkopplung) hat mehrere Facetten. Einerseits ist das Feedback des Systems gemeint. Wenn man Testfälle (*Test Cases*) schreibt und diese mit Erfolg durchlaufen werden, erhält man vom Software-System das Feedback, dass die spezifizierte Funktionalität korrekt implementiert wurde. Die Summe aller Testfälle ersetzt damit mehr und mehr die klassische funktionale Spezifikation. Änderungen der Spezifikation müssen nicht mehr zwingend schriftlich niedergelegt werden. Das Feedback des Systems, dass ein angepasster Testfall erfolgreich durchlaufen wurde, ist der Beweis, dass die Änderung der Anforderung erfolgreich implementiert wurde.

Feedback ist aber auch wichtig in der Kunden-Lieferanten-Beziehung. Im klassischen Ansatz entwickelt der Kunde eine Benutzeranforderung, die *User Requirements Specification* (URS), manchmal auch Lastenheft genannt. Dies erfordert aber vom Kunden eine ingenieursmäßige oder vergleichbare Ausbildung. Wenn der Kunde diese Qualifikation nicht hat, muss er einen entsprechenden Consultant engagieren. Da die funktionale Anforderung, die *Functional Specification* (FS) oder auch Pflichtenheft, vom Lieferanten geschrieben wird, entsteht oft die Situation, dass zwei aus Kundensicht fachfremde Parteien sowohl Lasten- als auch Pflichtenheft schreiben. Dem Kunden obliegt bestenfalls die Aufgabe, die entstandenen Dokumente zu unterzeichnen. Zeigen sich im Verlauf der Implementierung der Software im Funktionsumfang Abweichungen von ursprünglichen (aber nicht dokumentierten) Vorstellungen des Kunden, kommt es zu Differenzen. Da diese meist in späteren Projektphasen auftreten, wird es immer schwieriger die Wünsche des Kunden in die Software „einzubauen“. Agile Ansätze, insbesondere XP, versuchen, dieses Problem durch Feedback zu vermeiden. Der Lieferant, also das Software-Entwicklungsteam, verpflichtet sich, dem Kunden *möglichst schnell* den aktuellen Stand der Software vorzustellen. Idealerweise kann der Kunde immer einen Stand sehen, der nicht älter als ein Iterations-Schritt, also in der Regel nicht älter als wenige Wochen ist. Auf der anderen Seite muss der Kunde ein *qualifiziertes Feedback* geben. Dazu gehört, dass es sich um den *richtigen* Kunden handelt. (Diese Forderung ist eigentlich trivial!) Er muss später einer der Anwender der Software sein und muss über die Kompetenz verfügen, Entscheidungen über den Funktionsumfang sowohl fachlich, als auch verantwortlich zu fällen. Das Entwicklerteam darf also vom Kunden die fachliche Kompetenz der Zielgruppe der Software erwarten – aber keine Kompetenz im Bereich des Software-Engineerings! Die Kommunikation muss *in der Sprache des Kunden* erfolgen. Dies erfolgt am besten an Hand der aktuellen Implementierung (Release). Nur wenn der Kunde Anwender ist und mit der Software arbeiten kann, kann er auch letztendlich beurteilen, ob sie seinen Bedürfnissen entspricht. Dabei ist es aus der Erfahrung des Autors nicht so entscheidend, ob die Software fehlerfrei läuft. Man kann dem Kunden fehlerhafte Features der Software erklären und ihn auf die nächste Release verweisen. Meistens kann der Kunde trotz einiger Detailfehler durchaus beurteilen, ob die Software seine Bedürfnisse abdeckt. Es ist also wichtiger, schnell das Feedback des Kunden auf neue Funktionen der Software zu erhalten, als ihm eine fehlerfreie Software vorzustellen. Dasselbe gilt auch für die

Projektplanung. Dem Kunden eine ehrliche Terminplanung vorzustellen, ermöglicht diesem ein ehrliches Feedback. Dem Kunden so tiefe Einsichten in das Projekt zu geben, fällt dem Entwicklerteam in der Regel schwer, wie schon im Abschnitt über Kommunikation dargestellt wurde. Es gehört eine gehörige Portion Vertrauen und Mut dazu!

3.1.4 Mut

Schnell sich ändernde Projekte resultieren zwangsläufig in zweitklassigen Designs. In Laboren mit studentischen Arbeitsgruppen zeigt sich dieser Effekt genauso wie in Projekten in der Unternehmenspraxis. Oft wird Code solange angepasst, bis es einfacher wäre, alles wegzuwerfen und neu anzufangen. Dasselbe gilt für das Design. Wenn das Team am Anfang des Projekts das endgültige Ziel noch nicht vor Augen hat, kann es dieses im Design auch nicht berücksichtigen. Trotzdem ist die Bereitschaft Designs anzupassen und Code ggf. neu zu schreiben sehr gering verbreitet. Oftmals wird bei fehlschlagenden Testfällen ein Design oder ein Stück Code solange angepasst, bis pro neuer Änderung mehr Testfälle schiefgehen als erfolgreiche Testfälle dazukommen. Und trotzdem scheut sich das Team vor der Konsequenz. Das Problem ist die Rechtfertigung u.a. vor Vorgesetzten – vielleicht auch vor dem Kunden. Es wird augenscheinlich, dass man mehr Zeit für die nächste Iteration braucht. Trotzdem ist *Refactoring*, also die Umstrukturierung des Designs oder des Codes unabdingbar, um auf Dauer die Fehlerrate bei den Testfällen zu reduzieren. Martin Fowler fordert sogar, 10% des Codes pro Iteration neu zu strukturieren (Fowler u. Scott 2000). Dem Autor erscheint dieser Wert allerdings etwas hoch gegriffen.

Mut fordert XP aber auch an anderer Stelle. So fordert Kommunikation und Feedback in der Kunden-Lieferantenbeziehung Mut von beiden Seiten, wie oben beschrieben. Aber auch die Anpassung des Prozessmodells als Ganzes kann großen Mut erfordern. Wenn gewisse Rules and Practices von XP nicht funktionieren, so stehen auch sie zur Disposition: *Fix XP when it breaks*, wie Kent Beck selbst schreibt (Beck 2000). Doch dazu später mehr.

Wichtig ist, dass die drei Werte Kommunikation, Einfachheit und Feedback den Mut bereits implizit voraussetzen. Nur ein mutiges Team und ein mutiger Kunde werden die Anforderungen eines agilen Projekts bewältigen können.[17]

3.1.5 Respekt

Im Gegensatz zu den o.g. vier Werten (Beck 2000) fügte Beck diesen Wert erst später hinzu (Beck u. Andres 2004). Respekt ist ein wichtiger Wert für den Umgang miteinander. Sowohl innerhalb des Entwicklerteams als auch in der Beziehung zum

[17] Und diese Aussage gilt auch unabhängig von XP.

Kunden ist Respekt die Basis für einen vertrauensvollen gegenseitigen Umgang. Aus der Sicht des Autors ist Respekt die Grundlage für ein erfolgreiches Projekt.

3.2 Die 14 Prinzipien

XP definiert 14 Prinzipien, die aus den fünf Werten abgeleitet werden. Alle konkreten XP-Praktiken müssen auf die Einhaltung dieser Prinzipien geprüft werden:

- Menschlichkeit
- Wirtschaftlichkeit
- Wechselseitiger Vorteil
- Selbstähnlichkeit
- Verbesserung
- Vielfältigkeit
- Reflexion
- Fluss
- Gelegenheit
- Redundanz
- Fehlschlag
- Qualität
- Kleine Schritte
- Akzeptierte Verantwortung

Im Folgenden sollen die 14 Prinzipien kurz erläutert werden:

3.2.1 Menschlichkeit

Es sind Menschen, die Software entwickeln. Also müssen auch die Arbeitsbedingungen menschenwürdig sein. Persönliche Weiterentwicklungsmöglichkeiten der Software-Entwickler, ihre Integration und ihre Akzeptanz im Team müssen gewährleistet sein. Die Entwickler müssen sich in ihrer Projektumgebung wohlfühlen.

3.2.2 Wirtschaftlichkeit

Software-Entwicklung muss wirtschaftlich sein. Die reine „Schönheit" der technischen Lösung nützt niemandem (obwohl Techniker manchmal dazu tendieren). Auch Software-Entwickler müssen akzeptieren, dass das Unternehmen letztlich mit ihrer Lösung Geld verdienen will und muss.[18]

[18] Nicht umsonst beschäftigen sich in der heutigen Zeit Technik-Studenten an deutschen Hochschulen vermehrt mit betriebswirtschaftlichen Themen.

3.2.3 Wechselseitiger Vorteil

Alles, was im Team getan wird, sollte allen beteiligten Gruppen Vorteile bringen. Man sollte also immer versuchen, „Win-Win"-Situationen zu erreichen. Dokumentiert man z.B. aufwändig im Code, hat man als Nachteil einen aktuellen zeitlichen Mehraufwand. Demgegenüber steht der Vorteil der leichteren Lesbarkeit (zu einem *späteren* Zeitpunkt!). Der eigene Nachteil steht also in Relation zu einem später möglichen Vorteil einer bislang unbekannten Person. Dies muss gegeneinander abgewogen und in Balance gebracht werden.[19]

3.2.4 Selbstähnlichkeit

Selbstähnlichkeit in der Natur ist gegeben, wenn eine Struktur auf unterschiedlichen Größenskalen gleich ist. Dies kennt man z.B. von ferromagnetischen Strukturen. Es gilt aber auch für die Struktur einer erfolgreichen Lösung eines Problems. Man darf sich nicht scheuen (sondern muss es zum Prinzip erheben), solche erfolgreichen Lösungen auch in andere Kontexte zu kopieren. Hat man ein erfolgreiches Prinzip entdeckt, kann es auf unterschiedlichen (zeitlichen) Skalen funktionieren.

3.2.5 Verbesserung

Ständige Verbesserung ist ein Grundprinzip der Software-Entwicklung. Iteratives Vorgehen birgt die Möglichkeit der ständigen Verbesserung der Lösung. Agile Prozesse unterliegen der Verbesserung: Praktiken, die sich nicht bewähren, werden vom agilen Team abgeschafft und durch bessere Praktiken ersetzt.

3.2.6 Vielfältigkeit

Teams müssen aus einer Vielfalt von Mitgliedern mit unterschiedlichen Eigenschaften und Fähigkeiten bestehen. Das birgt Konfliktpotenzial, aber auch die Möglichkeit völlig neue Lösungen zu entwickeln. Verschiedene Menschen entwickeln beispielsweise verschiedene Designs, aus denen dann das beste ausgesucht werden kann. Die Zusammensetzung des Teams ist entscheidend für den Erfolg des Teams (Kap. 6 und 7).

---

[19] Ein Ausweg, den Kent Beck erwähnt, wäre die Einführung gut dokumentierter automatisierter Tests (Beck u. Andres 2004) anstatt (aufwändiger) Code-Dokumentation. Die Tests haben den Vorteil, dass sie dem jetzigen wie dem zukünftigen Entwickler nützen. Außerdem hält ein regelmäßiges Refactoring das Design einfach. Wir haben also eine „Win-Win"-Situation zwischen aktuellem und zukünftigem Entwickler. Der Autor empfiehlt allerdings, nicht *völlig* auf die Code-Dokumentation zu verzichten.

3.2.7 Reflexion

Ein gutes Team denkt darüber nach, wie es arbeitet und wie es seine Arbeit verbessern könnte. Es ist sinnvoll, dass das Team diese Reflexion formalisiert, d.h. bewusst Sitzungen durchführt mit dem Ziel, den Prozess zu verbessern. Aber auch informelle Gespräche zwischen Teammitgliedern dienen diesem Zweck. Agile Prozesse sind immer in Bewegung und auf dem Pfad der ständigen Verbesserung.

3.2.8 Fluss

Die iterative Entwicklung von Software soll einen möglichst kontinuierlichen Fluss der ausgelieferten Releases zur Folge haben. Nur so kann dem Kunden ständig ein *aktuelles* (Zwischen-) Resultat der Entwicklung gezeigt werden. Dies steht im Widerspruch zur Entwicklung in einem „großen Rutsch“, der *Big Bang Entwicklung* der 1970er und 1980er-Jahre.[20]

3.2.9 Gelegenheit

Jedes Problem birgt die Gelegenheit für eine Verbesserung. Wenn ein Team Probleme hat, muss es sie aktiv angehen. Es reicht nicht aus, nur zu „überleben“, das Team muss besser werden. Aus diesem Prinzip ergibt sich die XP-Praxis des *Pair Programming* (Kap. 3.3.3): Macht ein Programmierer zu viele Fehler, kann man dieses Problem durch Paar-Programmierung abstellen.[21]

3.2.10 Redundanz

Wenn man kritische Probleme mehrfach angeht, kann man aus den entstehenden Lösungen die beste auswählen. So können unterschiedliche Design-Ansätze bis zum Prototypen entwickelt werden, um dann den effektivsten Ansatz auszuwählen. Auch beim Testen kann Redundanz sinnvoll sein, beispielsweise durch Anhängen einer Testphase *nach* Ablieferung der fertigen Release.[22] Auch das Arbeiten in Programmiererpaaren ist eine redundante XP-Praxis.

---

[20] Phasen- oder Wasserfall-Modell (Kap. 1).

[21] Aussage von Kent Beck in (Beck u. Andres 2004). Denkbar wäre natürlich auch die Verbesserung des Schulungswesens.

[22] Im regulierten Umfeld z.B. der Pharmaindustrie auch *Operational Qualification* genannt.

3.2.11 Fehlschlag

Manchmal muss man Fehlschläge hinnehmen, um den besten Weg zu finden. Wen man bei mehreren Designvorschlägen nicht in der Lage ist, durch Diskussion den besten zu bestimmen, muss man die Designvorschläge ausprobieren und den Fehlschlag eines oder mehrerer Vorschläge hinnehmen. Wichtig ist, dass man dabei Erkenntnis gewinnt.[23]

3.2.12 Qualität

Qualität ist nicht verhandelbar. Ein Team wird nicht schneller, bloß weil es eine mindere Qualität akzeptiert. Meist wird die Produktivität durch Zunahme der Qualität höher. Qualität der Arbeit stärkt auch das Selbstvertrauen des Teams. Man ist stolz auf das Geleistete.

3.2.13 Kleine Schritte

Iterationen sollten in kleinen Schritten erfolgen. Insbesondere in agilen Projekten, ist es wichtig, Arbeitsfortschritte (auch *kleine* Fortschritte) schnell zeigen zu können, um Feedback zu erhalten. Kleine, regelmäßige Schritte bei der Integration der Software verringern den Overhead, der durch die Integration entsteht. Nur wenn Integration selbstverständlich ist, wird sie regelmäßig durchgeführt.

3.2.14 Akzeptierte Verantwortung

Verantwortung kann nicht zugewiesen werden, sie muss akzeptiert werden. Das Team muss Verantwortung übernehmen für das Produkt, das im agilen Projekt entsteht. Wenn keiner Verantwortung übernimmt, wird das Projekt scheitern.

3.3 Prozessschritte und traditionelle XP-Praktiken

Der von Kent Beck vorgeschlagene Satz von XP-Praktiken hat sich seit den ersten Publikationen verändert bzw. erweitert. Die in (Beck 2000) genannten „traditionellen“ XP-Praktiken, die auch auf der Website von Don Wells beschrieben sind (Wells 2009), unterscheiden sich zum Teil von den Praktiken, die im neueren Buch (Beck u. Andres 2004) erläutert werden.[24] Dies wird mit der Weiterentwicklung von XP

[23] Dieses Prinzip korreliert natürlich mit dem Prinzip der Redundanz.

[24] Sie werden in diesem Buch als „erweiterte“ Praktiken bezeichnet.

und der wachsenden Erfahrung mit diesem agilen Prozessmodell begründet. Allerdings ist es meist ein Unterschied „auf den ersten Blick“, der größer erscheint, als er tatsächlich ist.

In diesem Kapitel sollen die traditionellen XP-Praktiken erläutert werden. Die vier Phasen des Wasserfallmodells tauchen dabei leicht abgewandelt als Kategorien der *Rules and Practices* auf:

- Planung
- Design
- Kodieren
- Testen

Alle von Kent Beck vorgeschlagenen XP-Regeln und Praktiken werden diesen Kategorien zugeordnet (Wells 2009). Allerdings lässt XP die Reihenfolge der Regeln und Praktiken offen. Natürlich ist klar, dass das Projektteam mit Planung und Design beginnt, bevor implementiert werden kann. Ist jedoch das Projekt im Gang, ist keine klare Reihenfolge erkennbar. Agilität bedeutet eben auch Freiheit bei der Reihenfolge der Prozessschritte.

Allerdings gibt es, wie schon vom Spiralmodell (Kap. 1.3) bekannt, Iterationen und Inkremente, die geplant bzw. designt werden müssen. Die Software wird dann Stück für Stück als *Inkremente* in kleinen Prozessschritten, den Iterationen, implementiert. Jedes Inkrement kann vom Kunden begutachtet und freigegeben werden.

Es soll nicht verschwiegen werden, dass Kent Beck die traditionellen Regeln und Praktiken einige Jahre später anpasste. Trotzdem finden sie sich nach wie vor auf der offiziellen XP-Webseite wieder (Wells 2009) und haben somit ihre Gültigkeit. Da sie zentral sind und viel über das Entstehen von XP verraten, sollen die „traditionellen“ Regeln und Praktiken im Folgenden näher beleuchtet werden.

3.3.1 Planung

In der Planungsphase kennt XP folgende traditionellen Regeln und Praktiken:

3.3.1.1 User Stories

User Stories sind wie auch die aus dem Unified Process (UP) bekannten *Use Cases* oder Anwendungsfälle (Fowler u. Scott 2000) eine moderne Spezifikationstechnik. Im Gegensatz zur klassischen statischen Spezifikation des Funktionsumfangs wird in beiden Fällen beschrieben, was im Rahmen des Software-Projekts *getan* werden soll. Neu an dieser Betrachtungsweise ist der dynamische Aspekt. Während aber der Use Case eine spezielle Art der Minispezifikation darstellt und die Summe der Use Cases das System spezifizieren, handelt es sich bei der User Story mehr um ein Arbeitspaket, welches insbesondere zur Risikoanalyse, Aufwandschätzung und Iterationsplanung benutzt wird (Abb. 3.1 und 3.2). Es geht also nicht darum, möglichst

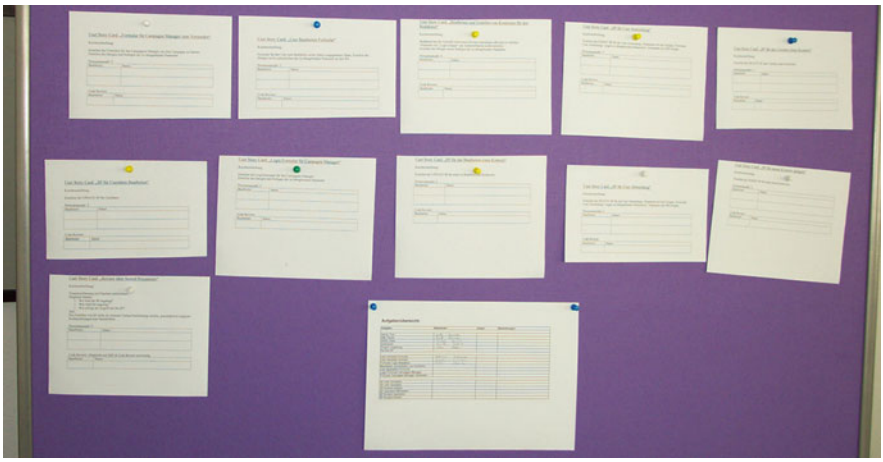

Abb. 3.1 User Story Card Board aus dem studentischen Labor

User Story Card: „Datenbank“

Kurzbeschreibung:

- Prüfen der vorhanden Datenbank
- Erweitern der Datenbank mit fehlenden Tabellen
- Aufzeichnen des Schemas der Datenbank auf ein Flipchartblatt

Personenanzahl: 2

Bearbeiter:	Status
Bensch, Kanat	Syssatzke Tabelle wurde modifiziert. Bitte neue Spalten beachten und bei Validierung die DB-Regeln beachten!

Code Review: (Absprache mit QM, ob Review notwendig)

Bearbeiter:	Status

Abb. 3.2 Details einer User Story Card

genau einen Anwendungsfall zu beschreiben, sondern lediglich das Thema in zwei bis drei Halbsätzen anzureißen. Details werden dann später während der Implementierung in Zusammenarbeit mit dem Kunden ausgearbeitet. Der Kunde ist also in diesem Sinne die „wandelnde Spezifikation“.

Das Ausarbeiten der User Stories erfordert insbesondere die beiden XP-Werte Kommunikation und Feedback (Kap. 3.1).

3.3.1.2 Release-Planung – das Planungsspiel

Bei der Planung einer auszuliefernden Software-Release unterscheidet XP zwischen zwei teilnehmenden Parteien: der *Geschäftsseite* und der *Entwicklung*. Die Geschäftsseite benötigt die Software, die Entwicklung, also das Team, entwickelt die Software. Bei der Planung können Interessenskonflikte der beteiligten Parteien offen zu Tage treten. Aus der Erfahrung des Autors tendieren Entwickler dazu, in alle ihre Aufwandschätzungen „Sicherheit" einzubauen, während die Geschäftsseite gerne den besten anzunehmenden Projektverlauf als Standard erklärt. Etwas unklar ist die Rolle des Kunden, der eigentlich zur Geschäftsseite gehört, aber aufgrund seiner ständigen Nähe zum Team, als Teammitglied, mit der Zeit auch immer mehr die Sorgen und Nöte der Entwickler verstehen wird. Trotzdem wird er im Zweifelsfall versuchen, die Interessen seines Unternehmens zu vertreten, in der Regel also auf eine schnelle Auslieferung drängen.

Das XP-Planungsspiel kennt vier Variablen: *Umfang, Ressourcen, Zeit und Qualität*. Dabei spielt die Qualität aus der Sicht des Autors eine Sonderrolle. Sie ist keine echte Variable. Ziel muss immer sein, Software maximaler Qualität zu erzeugen, da die Auswirkungen schlechter Qualität, wie unzufriedene Kunden, notwendig werdende Nachbesserungen etc., immer Einfluss auf die anderen Variablen haben. Es bleiben also drei Variablen, von denen die Geschäftsseite maximal zwei beeinflussen darf. Die dritte Variable „gehört" der Entwicklung.

Im Rahmen des Planungsspiels entwirft die Geschäftsseite User Story Cards (Abb. 3.2), allerdings in der Regel unter Mithilfe einiger Mitglieder des Entwicklungsteams, wie in Kap. 6 noch näher erläutert werden wird. Als Regel gibt Kent Beck vor, dass User Stories so definiert werden sollten, dass ihre Bearbeitung etwa ein bis drei Wochen dauern sollte. Ein Release-Plan sollte aus ungefähr 80 ± 20 User Stories bestehen (Wells 2009).

Im eigentlichen Planungsspiel werden die User Stories unter Abwägung der Marktanforderungen, des Risikos und anderer Einflüsse zeitlich so angeordnet, dass alle Parteien mit dem sich ergebenden Projektablaufplan einverstanden sind (Abb. 3.3 und 3.4).

Neben den XP-Werten Kommunikation und Feedback ist insbesondere der Mut gefordert, den sich ergebenden Konflikten nicht aus dem Weg zu gehen. Insbesondere die Entwickler seien gewarnt, ihre Termine auf Aufforderung der Geschäftsseite nach „unten" zu korrigieren, ohne gleichzeitig den Funktionsumfang der Software zu reduzieren. Zu optimistische Termine rächen sich, insbesondere weil dann die Geschäftsseite in der Regel nicht mehr weiß, dass sie es war, die diese gefordert hatte.

3.3.1.3 Kleine Releases

Kleine Releases garantieren schnelles Feedback. Wenn man Releases plant, die einen sichtbaren Kundennutzen repräsentieren, wie z.B. ein neues Eingabeformular, dann kann der Kunde auch sofort seine Meinung äußern. Da der Kunde Teammitglied ist, spricht natürlich auch nichts dagegen, wenn er eine fertiggestellte User

Abb. 3.3 Planungsspiel im studentischen Labor

Abb. 3.4 User Story Card Board im Planungsspiel

Story sofort am Entwicklerarbeitsplatz begutachtet. Allerdings weiß der Autor aus der Erfahrung im studentischen Labor, dass es wichtig ist, dass der Kunde auf einem Integrationsrechner den aktuellen Stand der Release mit allen aktuellen User Stories sehen kann. Doch dazu mehr in Kap. 6 und 7.

3.3.1.4 Messung der Projektgeschwindigkeit

Addiert man die geschätzten Aufwände für die User Stories pro Iteration und vergleicht man sie mit den tatsächlichen Aufwänden für die von den User Stories abgeleiteten Aufgaben (Tasks), so erhält man ein Maß für den Fortschritt des Projekts. Im Grunde macht man einen einfachen Soll-Ist-Vergleich, wie er aus der Projektplanung bekannt ist.

Selbstverständlich kann man etwas professioneller auch nach *Function Point* (Hürten 1999) oder *COCOMO II* (Boehm et al. 2000) schätzen. Der Autor bedient sich gerne solcher komplexeren Methoden. Einfachheit des Prozesses impliziert nicht unbedingt Einfachheit der Schätzmethode. Lohnenswert ist auch immer ein Blick auf die Methoden im GPM-Infocenter.[25]

3.3.1.5 Iterationen und Iterationsplanung

Beck schlägt Iterationslängen von ein bis drei Wochen vor. Die Iteration wird gemeinsam mit dem Kunden im Rahmen eines *Iteration Planning Meetings* geplant. Üblicherweise wählt sich der Kunde eine oder mehrere User Stories, die für ihn wichtig sind, aus um sie vom Team realisieren zu lassen. Die Aufgabe des Meetings ist eine einvernehmliche Einigung auf Programmieraufgaben (Tasks), die sich aus der oder den ausgewählten User Stories ergeben.

Im Rahmen der Iterationsplanung wählen sich die Programmierer einzelne Tasks zur Realisierung aus. Der Autor hat gute Erfahrungen damit gemacht, dass die Programmierer ganze User Stories wählen, weil damit die Verantwortung für einen für den Kunden wichtigen Teilaspekt der Software übernommen wird. So wird letztlich die Kommunikation mit dem Kunden gefördert.

Die Programmierer machen für die ausgewählten Tasks eine Detailplanung, deren geschätzte Aufwände wiederum in die Messung der Projektgeschwindigkeit eingehen. Ebenfalls machen sich die Entwickler gleich zu Anfang Gedanken über geeignete Testfälle zur Messung der Erreichung des Ziels der Iteration. Die Test Cases ergeben sich in der Regel direkt aus der bearbeiteten User Story (Abb. 3.5).

3.3.1.6 Move People Around

Agile Vorgehensweisen weisen eine große Schwäche auf. Aufgrund der eher dürftigen Dokumentation ist viel Wissen in den Köpfen der Entwickler oder des Kunden vorhanden. Die Frage stellt sich, wie viele Teammitglieder ausscheiden dürfen, oder

[25] Siehe dazu (GPM 2009) oder (Drews u. Hillebrand 2007).

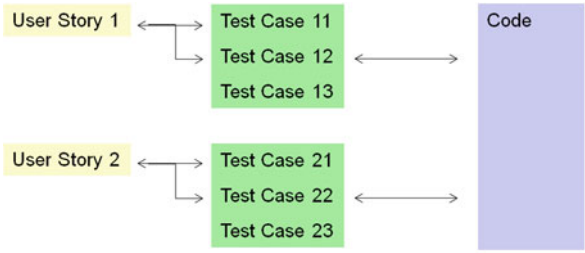

Abb. 3.5 Beziehung zwischen User Stories, Test Cases und Code

wie John Coplien etwas zugespitzt sagt, „wie viele von einem LKW überfahren werden dürfen“ (Williams u. Kessler 2002), bevor das Projekt nicht mehr vernünftig weitergeführt werden kann. Beck schlägt deswegen den sog. *Truck-Faktor* vor, der von 0 bis 100% die Wahrscheinlichkeit des Scheiterns des Projekts angibt, wenn ein bestimmtes Teammitglied aus dem Projekt ausscheidet.

Wenn alle Teammitglieder in jedem Bereich der Software mindestens einmal mitarbeiten, dann kann dieser Truck-Faktor reduziert werden. Damit braucht ein agiles Projekt aber Generalisten als Teammitglieder. Der Vorteil von Spezialisten, die komplexe Detailaufgaben erledigen können, kann so nicht ausgenutzt werden. Die Erfahrung im studentischen Labor des Autors zeigt, dass es Programmierer gibt, die keinerlei Interesse daran haben, die *ganze* Software zu kennen. Hier muss bei XP Überzeugungsarbeit geleistet werden. Generell hat der Autor aber keine guten Erfahrungen damit gemacht, Entwickler zu Ihrem „Glück“ zu zwingen.[26] Es braucht hier sicher *Mut*, um die für das Team beste Lösung zu finden.

3.3.1.7 Stand-Up Meeting

Auch für diese XP-Praxis gibt es keine überzeugende deutsche Übersetzung. Wie oft haben sich Entwickler und Ingenieure schon beschwert über endlose Sitzungen ohne nennenswertes Ergebnis. Ein guter Freund des Autors berichtete von einer Sitzung bei einem deutschen Automobilbauer, die mit einer stundenlangen Anfahrt begann und in stundenlanger Langweile endete. Fazit war ein vergeudeter Tag! Eine bewährte agile Technik dagegen (und beileibe keine Erfindung von XP) ist das „Meeting im Stehen“ (und eben nicht die „Sitzung“). Oft werden diese täglichen Meetings vor einem Kaffeeautomaten abgehalten. Ziel ist, möglichst schnell und präzise den Projektstand zu reflektieren, gefundene Probleme aufzuzeigen oder Lösungen von allgemeinem Interesse zu kommunizieren. Oft bilden sich solche Meetings auch spontan, nach der Erfahrung des Autors ist es jedoch besser, eine fixe Uhrzeit abzumachen (nicht zu früh!).

Denkbar ist auch die Variante, dass pro Sitzung ein Entwickler ein „Freilos“ bekommt, das es ihm ermöglicht, wegen anderer wichtiger Aufgaben an dieser Sitzung

[26] Siehe auch *Collective Code Ownership* in Kap. 3.3.3.

nicht teilzunehmen. Wichtig ist aber, dass dieser Entwickler erst sein zweites Freilos erhält, wenn vorher alle anderen ebenfalls in den Genuss gekommen sind. So ist sichergestellt, dass kein Entwickler leichtfertig mit diesem Freilos umgeht.

3.3.1.8 Fix XP When It Breaks

Im Sinne einer vernünftigen Prozessverbesserung muss es möglich sein, von Regeln und Praktiken Abstand zu nehmen, die sich in der Praxis nicht bewähren. So lässt auch XP diese Möglichkeit der Korrektur zu. Wichtig ist jedoch, dass eine Prozessverbesserung kein schleichender Vorgang ist. Es muss zu jedem Zeitpunkt klar sein, welche Regeln und Praktiken gelten und welche nicht mehr. Im studentischen Labor neigte das Team dazu, von einer „impliziten Prozessverbesserung“ zu sprechen. Das ist natürlich Unfug und zeigt eher die Faulheit des Teams, Änderungen am Prozess öffentlich zu machen und somit auch dem Kunden mitzuteilen.

Es gehört Mut dazu, Änderungen des Prozesses durchzuführen und zu veröffentlichen. Die Frage, die man sich stellen muss, ist, ab der wievielten Änderung der „gefahrene“ Prozess nicht mehr XP ist. Der Autor wird im weiteren Verlauf zeigen, dass hier eine übergeordnete Sichtweise auf den Prozess Erfolg verspricht (Kap. 7).

3.3.2 Design der Software

Zur Modellierung der Software bietet XP die nachfolgenden traditionellen Regeln und Praktiken an. Auch hier sieht man, dass die vorgeschlagenen Praktiken nicht neu sind. Es ist eher die Zusammensetzung der Regeln und Praktiken, die XP ausmacht.

3.3.2.1 Einfachheit des Designs – Implementiere Funktionalität nicht früher als nötig

Insbesondere im agilen Design ist Einfachheit gefragt. Da bei agilen Projekten aus prinzipiellen Gründen zu Beginn nicht klar sein kann, wie das endgültige Design aussehen wird, macht es auch keinen Sinn, Klassen oder Schnittstellen zu designen, die nicht auch in der gleichen Iteration implementiert werden. Damit entsteht das Problem, dass die Gesamtarchitektur des Systems auch erst kurz vor Schluss des Projekts sichtbar wird. Was aber noch nicht bekannt ist, kann auch nicht designt werden. Ein XP-Design stellt immer den aktuellen Wissensstand des Teams dar. Soll z.B. für ein PC-gestütztes Messsystem neben einem TCP/IP-Bus in einer späteren Iteration auch ein aktueller Labor-Bus realisiert werden, so darf dieser im ersten Design noch nicht auftauchen. Es besteht durchaus die Wahrscheinlichkeit, dass der Labor-Bus im weiteren Verlauf des Projekts als überflüssig erachtet wird.[27] Also: Funktionalität nicht früher als nötig einbauen.

[27] Dies ist jedenfalls die Erfahrung des Autors.

3.3.2.2 Systemmetapher

Laut Kent Beck ist eine Systemmetapher eine „Geschichte, mit der jeder – Kunde, Programmierer, Manager – die Funktionsweise des Systems veranschaulichen kann“ (Beck 2000). Im originalen Chrysler-Projekt C3, das die Entwicklung einer Lohn- und Gehaltslisten-Software zum Ziel hatte und das erste dokumentierte XP-Projekt war, war die Systemmetapher ein Bild aus der Automobilproduktion. Die Software wurde mit einer Automobilproduktionslinie verglichen, weil man davon ausging, dass alle Chrysler-Mitarbeiter mit dieser Begriffswelt vertraut waren.

Die Suche nach einer Metapher gestaltet sich im Allgemeinen schwierig, so dass der Autor im studentischen Labor auf diese Praxis verzichtet. Im Rahmen des Labors wurde bisher immer der „klassische“ Ansatz der *Systemarchitektur* verfolgt, also die Idee, die wichtigen Zusammenhänge im Software-System darzustellen und die Details der Software zu vernachlässigen, was mit dem Begriff des Modells korrespondiert, wie er z.B. bei Balzert Verwendung findet (Balzert 2001 u. 1998). Natürlich wird auch bei diesem Ansatz eine Metapher verwendet, indem die Modellierung der Software mit der Modellierung eines Hauses verglichen wird – dies steckt bereits in der Verwendung des Begriffs *Architektur*!

3.3.2.3 CRC-Karten

Class, Responsibilities and Collaboration (CRC) Cards sind ein äußerst praktisches Design-Werkzeug. Ganz schlicht in der Form wird pro Klasse in maximal vier Halbsätzen dargestellt, was die Klasse leisten soll bzw. mit welcher anderen Klasse sie dazu in Wechselwirkung stehen muss.[28] So kann pro Iteration ein Satz von korrespondierenden Klassen gebildet und in Form von Karteikarten verwaltet werden. Wenn man diese Karteikarten auf einer Pinwand anordnet und durch Assoziationen verbindet, entsteht ein erstes UML-Diagramm.[29] Zu erwähnen ist noch, dass die UML-Klassen in der nicht-objektorientierten Welt durchaus auch Datenbanktabellen oder, ganz allgemein, Software-Module sein können (Abb. 3.6).

3.3.2.4 Risikominimierung durch „Spike Solutions“

Eine *Spike Solution* lässt sich am besten als prototypenhafte Implementierung der Lösung eines aufgetauchten (technischen) Problems bezeichnen. Mit diesem „Durchstich“ wird die Lösbarkeit bewiesen. Meist taugt jedoch dieser Prototyp nicht als Komponente der endgültigen Software. Es ist meist besser, den „quick and dirty“ entwickelten Code wieder wegzuwerfen. Es gehört allerdings Mut dazu, funktionierenden Code allein aufgrund eines mangelhaften Designs aufzugeben. Die Erfahrung zeigt jedoch, dass es sich rächt, schlecht designten Code weiter zu nutzen.

---

[28] z.B. durch Aufruf einer entsprechenden Methode.

[29] Unified Modeling Language, ein sehr gutes Buch dazu ist (Fowler u. Scott 2000).

class *MailVersand*	
Verantwortlichkeit	**Beziehung zu Klasse**
Mail-Kampagne zuordnen	Campaign
Newsletter zuordnen	Newsletter
Abonnenten zuordnen	User
Mail versenden	

Abb. 3.6 CRC-Karte aus einer eMail-Versand-Software

3.3.2.5 Refactoring

Refactoring lässt sich am besten mit *Umstrukturierung* des Designs oder des Codes übersetzen (Fowler u. Scott 2000). Dahinter steckt das Problem der *Software-Entropie*: Entwickelt man nur lange genug, wird aus jedem noch so guten Design ein schlechtes Design. Bleiben wir beim Beispiel des Messsystems: Nachdem eine TCP/IP-Schnittstelle implementiert wurde, wird unabhängig davon (zu einem späteren Zeitpunkt) ein Labor-Bus hinzugefügt.[30] Angenommen, diese verschiedenen Schnittstellen werden jeweils durch eine Klasse repräsentiert, wird spätestens jetzt klar, dass ein optimales Design ein *Interface*, also eine „Schnittstelle" beinhalten sollte, von dem die o.g. Klassen abgeleitet werden. Das (wild) gewachsene System der zwei unabhängigen Schnittstellen-Klassen wird durch das saubere Design *Interface vererbt an Klassen* ersetzt. Dieser Ansatz hat aber auch einen weiteren Vorteil. Sollte sich in einer späteren User Story die Implementierung einer weiteren Bus-Schnittstelle, z.B. Profibus, als notwendig erweisen, kann der Entwickler die entsprechende Klasse vom Interface „Schnittstelle" ableiten.

Refactoring macht das Design besser und reduziert damit auf Dauer die Fehlerrate bei den Testfällen. Martin Fowler fordert sogar, 10% des Codes pro Iteration neu zu strukturieren (Fowler u. Scott 2000). Vergleicht man mit der Praxis des studentischen Labors, scheint dem Autor dieser Wert allerdings zu hoch gegriffen. Über die Jahre hinweg kann es aber in einem studentischen Projekt durchaus passieren, dass mehr als die Hälfte des Codes nochmals angepasst wird.

Leider funktioniert Refactoring ohne die entsprechende Werkzeug-Unterstützung nicht vernünftig. Mit Hilfe eines guten *Model Driven Development*-Tools (MDD) reduziert sich eine Design-Änderung nach der Erfahrung des Autors auf wenige Maus-Klicks. Deshalb fordert Refactoring in

[30] z.B. IEEE-488 / IEC-625.

einem agilen Projekt fast zwangsweise den Einsatz von *Computer Aided Software Engineering (CASE)*. Ansonsten wird nur die systemweite Namensänderung einer Variablen zu einem fast ausweglosen Unterfangen.[31]

Zum Refactoring gehört Mut! Mut, lauffähige Software aus „ästhetischen Gründen“ umzuschreiben. Dies beginnt mit Variablen-Namen und endet mit den o.g. Design-Anpassungen. Allerdings verbessert ein klares Design auf die Dauer die Resultate der Software-Tests (Fowler u. Scott 2000).

3.3.3 Kodieren

Die traditionellen Regeln und Praktiken, die XP in der Kodierungsphase zu Verfügung stellt, sind zum Teil neu und wenigstens teilweise nicht unumstritten.

3.3.3.1 Der Kunde ist immer verfügbar

XP fordert die permanente Mitgliedschaft des *Kunden im Team*. Dies soll eine schnelle Reaktion auf sich ändernde Kundenwünsche ermöglichen, selbst in späten Projektphasen. Spezifikationsdetails können „face to face“ mit dem Kunden besprochen werden. Die geforderte dauernde Anwesenheit des Kunden im Projekt ist in der Praxis naturgemäß ein Problem, da erfahrene Kunden kaum vollständig einem Software-Projekt zur Verfügung stehen. Die Praxis zeigt jedoch einen Mittelweg auf: Der Autor hat gute Erfahrungen mit einem „roten Telefon“ gemacht, welches den Kunden mit dem Team verbindet. Dies kann z.B. ein Prepaid-Handy o.Ä. sein, dessen Nummer idealerweise nur dem Entwicklerteam bekannt ist. Wenn dieses Handy klingelt, setzt ein zweistufiger Mechanismus ein: Der Kunde meldet sich möglichst direkt, spätestens innerhalb von einer Stunde, um den Anruf entgegenzunehmen. Innerhalb eines Arbeitstags gibt er die endgültige Antwort. Diese Zeitspannen reichen üblicherweise sowohl den Entwicklern als auch dem Kunden aus. Die Entwickler können normalerweise einen Tag Antwortzeit überbrücken, der Kunde kann innerhalb eines Tags in der Regel eine qualifizierte Antwort geben.

Ein anderer Einwand gegen die ständige Präsenz des Kunden ist, dass er auf diese Weise zu viele Interna des Projekts (auch gemachte Fehler) mitbekommen könnte. Würde dadurch nicht sein Vertrauen in die Qualität der Entwickler-Mannschaft sinken? Und wird dadurch die Projekt-Kommunikation insbesondere mit dem Kunden wirklich besser? Die Praxis zeigt, dass der Kunde es dem Team in aller Regel dankt, wenn er integriert wird und auch an den Entscheidungsprozessen beteiligt wird. Sein Vertrauen in das Team wächst! Allerdings ist die Kommunikation zwischen Team und Kunde kein Selbstläufer. Der Autor, der im studentischen Labor

[31] Man stelle sich auch noch vor, die Variable heiße *i*.

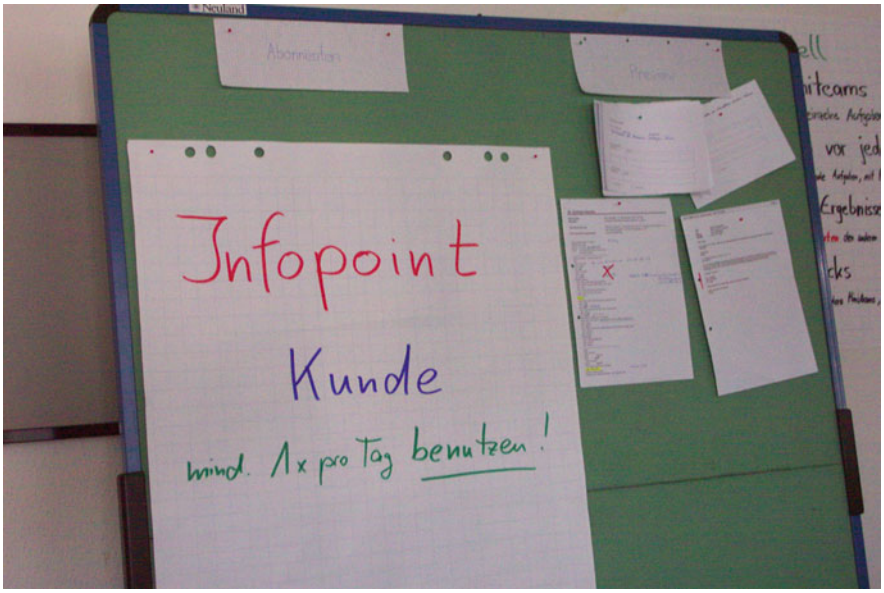

Abb. 3.7 Kunde bemüht sich um Kommunikation im studentischen Labor

immer die Rolle des Kunden inne hat, musste sich in der Vergangenheit immer wieder Gehör verschaffen bzw. das Team aktiv auffordern, mit ihm zu kommunizieren (Abb. 3.7).

3.3.3.2 Kodierungsstandards

Coding guidelines sind aus der modernen Software-Entwicklung nicht mehr wegzudenken. Es gibt *keinen* Grund, auf sie zu verzichten. Sie vereinheitlichen den Code und ermöglichen dem Teammitglied, den Code der anderen leicht zu verstehen. Im studentischen Labor „überleben“ solche Guidelines an der Wand über Jahre (Abb. 3.8). Von drei Nachfolgeteams hatte keines das Bedürfnis, die Richtlinien des studentischen Teams aus dem Jahr 2005 zu verändern oder anzupassen. Kodierungsstandards sind die Grundlage für ein erfolgreiches Refactoring.

3.3.3.3 Unit Test zuerst programmieren

“Test first!“ ist ein wichtiger Grundsatz des XP (Beck u. Andres 2004). Wenn man *Modultests* (Unit tests) am Anfang einer Iteration schreibt, kann man damit die betrachtete User Story präzisieren. Zur Erinnerung: User Stories bestehen in der Regel aus maximal drei Halbsätzen und haben nicht ansatzweise den Informationsgehalt einer klassischen Spezifikation. Zusammen mit dem Kunden wird bei der Implementierung der User Story eine Minispezifikation geschaffen,

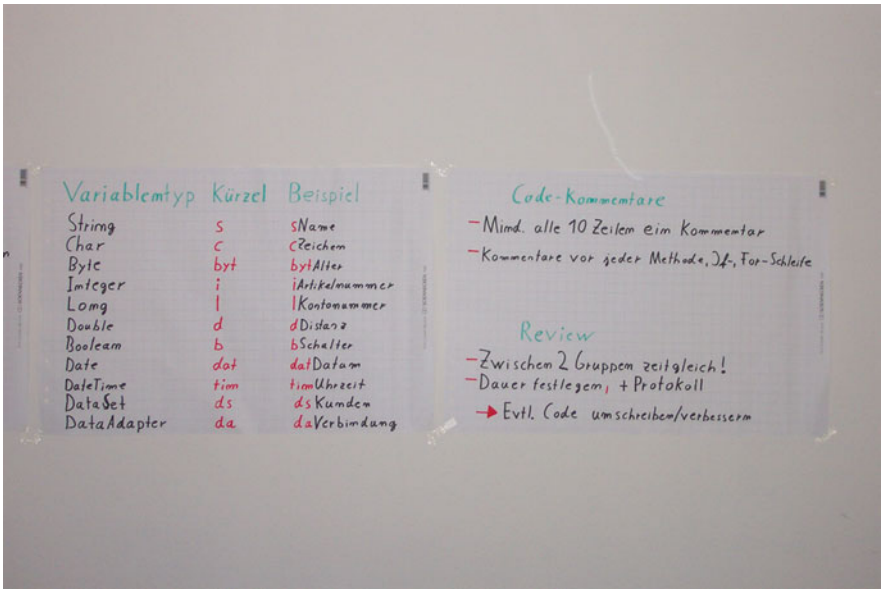

Abb. 3.8 Coding Guidelines aus dem Jahr 2005

welche allerdings nirgends detailliert niedergeschrieben wird. Im studentischen Labor findet man im Wiki[32] lediglich Ausschnitte der Detailspezifikation und des Feindesigns. Es fällt deutlich leichter, die Entwickler zu motivieren parallel zur (nicht schriftlich niedergelegten) Minispezifikation Testfälle zu generieren, die das Erreichen des spezifizierten Ziels dokumentieren sollen. Diese können auch problemlos mit dem Kunden abgestimmt werden und ersetzen so die schriftliche Detailspezifikation (Abb. 3.9). Naturgemäß befinden sich diese Testfälle auf dem Niveau des Software-Moduls (also auf der Ebene der Software-Units). Übergeordnete Systemtests, in denen auch das Umfeld der Software getestet werden kann,[33] sind dabei ausgeklammert, bzw. auf der Zeitachse nach hinten verschoben (Kap. 3.3.4).

3.3.3.4 Pair Programming

Die Idee der Programmierung in Paaren besagt, dass der ganze Code des Software-Projekts von Gruppen – der Autor nennt diese Gruppen auch *Mini-Teams* – bestehend aus zwei Entwicklern, programmiert werden soll. Dabei *teilen* sich die

[32] Ein Wiki (Wiki Web) ist eine im World Wide Web verfügbare Seitensammlung, die von den Benutzern nicht nur gelesen, sondern auch online geändert werden kann. Wikis ähneln damit einfach zu bedienenden Content Management Systemen, siehe http://de.wikipedia.org/wiki/Wiki

[33] Performance-Tests, Stresstests usw.

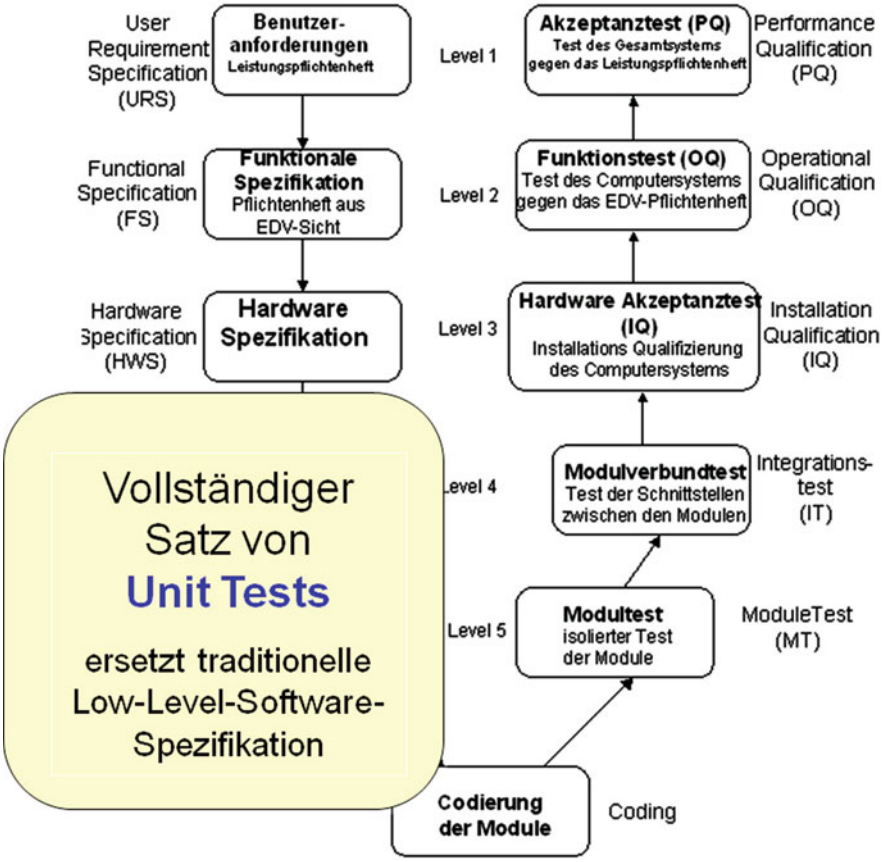

Abb. 3.9 Detail-Spezifikationen des V-Modells werden durch Unit-Tests ersetzt

beiden Entwickler einen Computer (Abb. 3.10). Periodisch wechselnd schieben sie die Tastatur hin und her. Der eine denkt taktisch, er implementiert z.B. die Methode einer Klasse, während der andere daneben sitzt und strategisch denkt: Braucht man diese Klasse überhaupt, oder sollte sie abgeändert werden?

Beck behauptet, dass das Programmieren in Paaren die Qualität des erzeugten Codes erhöht, was dem Projekt sicherlich nützt. Interessant ist aber, dass er ebenfalls (ohne einen dem Autor bekannten Beweis) behauptet, dass das Paar genauso schnell entwickelt wie zwei unabhängige Entwickler an zwei verschiedenen Rechnern. Dies deckt sich nicht mit den Erfahrungen des Autors im studentischen Labor. In Low-Level-Implementierungen auf Detailniveau ist es eher so, dass der Spezialist programmiert und der Partner zuschaut. Solange es aber um Design-Entscheidungen geht, macht das Arbeiten in Mini-Teams Sinn. In Kap. 3.3 wird der Frage nachgegangen, ob diese Mini-Teams sinnvollerweise aus zwei Mitgliedern bestehen müssen.

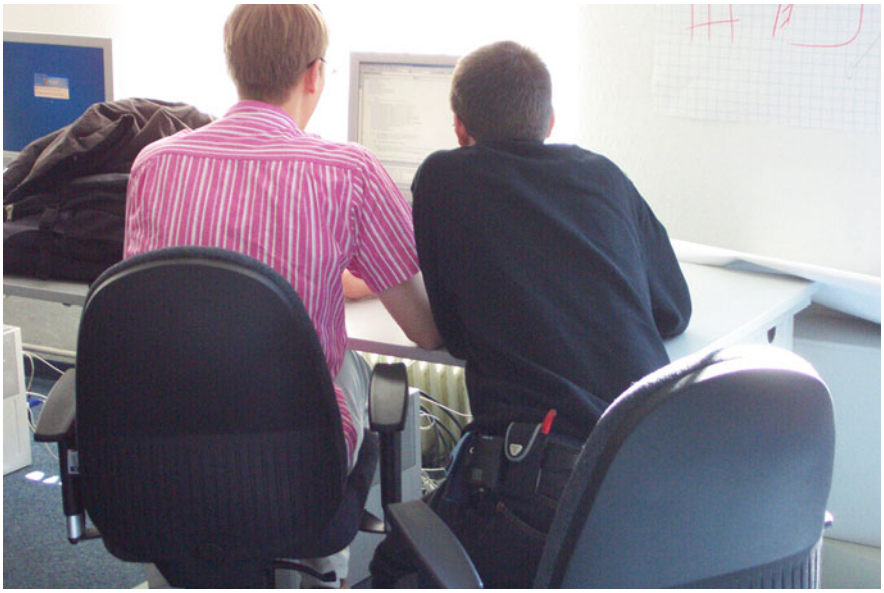

Abb. 3.10 Ein XP-Paar im studentischen Labor

3.3.3.5 Code-Integration durch Paare – Integriere oft!

XP fordert, dass die Paare ihren Code selbst integrieren. Allerdings darf dies nur durch jeweils *ein* Paar geschehen. Dieses Paar hält ein *Token*, was es nur ihm ermöglicht zu integrieren. Dieses Token kann z.B. ein freier Integrationsrechner sein. Die Idee dahinter ist, dass ein Paar seinen neu entwickelten Code am besten kennt und deshalb auftauchende Integrationsprobleme am besten lösen kann. Problematisch wird es, wenn das Paar für die Integration des eigenen Codes Änderungen am Code anderer Paare durchführen muss. Aufgrund der nachfolgenden Regel *Collective Code Ownership* geht XP davon aus, dass zumindest ein Grundverständnis des Codes anderer Paare vorhanden ist. Und durch die vorhandenen Testfälle ist es leicht, herauszufinden, ob die Änderung am „fremden" Code Fehler produziert oder nicht.

Durch diese Praxis der *sequentiellen Integration* durch die Paare wird verhindert, dass abwesende oder verhinderte Teammitglieder zum Engpass für den Projektfortschritt werden. Den Einsatz von Integratoren oder Integrationsteams lehnt Kent Beck explizit ab. Dies würde ebenfalls die Projektgeschwindigkeit reduzieren (Wells 2009). Dies entspricht in keinster Weise der Erfahrung des Autors im studentischen Labor oder in der Praxis! Die Wahrheit ist, dass sich alle Mini-Teams, auch Paare, vor der Integration zu „drücken" versuchen. Kein Mini-Team von Entwicklern ist erpicht darauf, in „fremdem" Code Änderungen durchzuführen. Der Autor machte als Kunde des studentischen Labor-Teams die Erfahrung, dass

jedes Mini-Team voller Stolz seine realisierte User-Story vorführte, aber immer nur auf dem *eigenen* Rechner.[34]

Beck fordert eine kontinuierliche Code-Integration (*Continuous Code Integration*)**.** „Integriere oft!“ ist sein Leitspruch. Dabei strebt er Integrationszyklen unter einem Tag an. Dieses ehrgeizige Ziel ist ohne automatisierte Tests niemals zu erreichen.

3.3.3.6 Collective Code Ownership

Dem Team als Ganzem gehört der Code! Damit ist in XP das Team auch gleichzeitig der Chefarchitekt. Alle Teammitglieder können so ihren Beitrag zu einer guten Software-Architektur leisten. Sicherlich weiß das Team als Ganzes mehr als ein einzelner Architekt. Gute Ideen gehen nicht mehr unter. Keiner wird mehr zum Engpass für eine schnelle und effiziente Software-Entwicklung.

Funktionieren kann diese Regel nur, wenn vor der Architektur die Ausarbeitung der *Unit Tests* steht. Nachdem die Paare sich pro Release „ihre“ User Stories gewählt haben, müssen sie Unit Tests schreiben, die die korrekte Funktionalität der User Story beweisen. Damit kann jedes Paar Änderungen am Code anderer Paare durchführen, ohne Gefahr zu laufen, unbewusst Fehler in den Code einzubauen. Wenn der originale Unit Test des anderen Paars immer noch fehlerfrei läuft, gilt die Änderung als erfolgreich.

Die Anwendung dieser Regel erfordert Mut! Die Praxis im studentischen Labor zeigt, wie schon früher erwähnt, dass Entwickler üblicherweise wenig Interesse zeigen, „fremden Code“ zu verändern. Man stößt oft auf regelrechtes Desinteresse daran. Von einem Team von ca. 20 Personen war nie mehr als ein Drittel am Gesamtcode interessiert. Auch diese Regel wird in Kap. 6 hinterfragt werden.

3.3.3.7 Optimiere erst zum Schluss

Dies ist keine spezielle XP-Regel. Auch im *Unified Process* findet Optimierung erst in der *Transition-Phase* statt (Jacobsen et al. 1999). Beim Optimieren wird der Debug-Code eliminiert, der während der Entwicklung wertvolle Hinweise auf Programmierfehler gibt. Außerdem kann erst ganz zum Schluss klar sein, wo Performance-Engpässe im System entstehen. Diese sollten dann gezielt angegangen werden. Nur der Verdacht auf einen möglichen Engpass reicht nicht aus.

3.3.3.8 Keine Überstunden

Wie der Autor aus eigener Erfahrung weiß, zerstören anhaltende Überstunden die Motivation des Teams. Wenn man täglich zehn Stunden arbeitet und mehrere Wochen die Wochenenden opfert, ist nach maximal einem Monat „die Luft raus“. Das Team will nicht mehr, die Arbeitsleistung nimmt ab. Wie man schon in der

[34] In Kap. 6 soll diese Problematik näher beleuchtet werden.

Projektmanagement-Vorlesung lernt, kann man am Tag kaum mehr als 6,5 Stunden produktiv am Projekt arbeiten. Die restliche Zeit wird für andere Dinge gebraucht.[35]

3.3.4 Testen der Software

Wie schon in Kap. 3.3.3 klar wird, durchzieht das Testen der Software den ganzen Entwicklungsprozess. Durch das Prinzip *Test first!* wird das Schreiben von Unit Tests sogar an den Anfang der Implementierung einer User Story gestellt. Trotzdem gibt es Tests, die naturgemäß am Ende der Implementierung einer Release stehen, die Akzeptanztests.

3.3.4.1 Unit Tests für den gesamten Code! Wenn ein Fehler gefunden wird, muss ein Test generiert werden! Alle Unit Tests müssen vor der Release erfolgreich sein!

In diesen Regeln wird nochmals explizit die Wichtigkeit von Unit Tests herausgestellt. Wie schon in Kap. 3.3.3 dargestellt, können am Anfang einer Iteration geschriebene Unit Tests die betrachteten User Stories präzisieren. Testfälle können problemlos mit dem Kunden abgestimmt werden und damit „klassische“ schriftliche Detailspezifikation ersetzen (Abb. 3.9). Unit Tests befinden sich auf dem Niveau des Software-Moduls (Software-*Unit*). Wenn aber während der Tests oder im laufenden Betrieb Fehler gefunden werden, muss zum Fehler ein Test generiert werden. Man darf davon ausgehen, dass dieser Test noch nicht bestand, weil sonst der Fehler entdeckt worden wäre.[36] Alle Unit Tests müssen *vor* einer Release erfolgreich durchgeführt sein.

3.3.4.2 Akzeptanztests

Bei allen Überlegungen zu Software-Tests wurde bisher ausgeklammert, dass gewisse Fehler, z.B. bezüglich Performance, Internet-Anbindung oder allgemeiner Belastung des Systems, nicht in Unit Tests gefunden werden können. Das ausgelieferte System besteht am Ende aus Software *und* Hardware. Und dieses Komplettsystem muss auch getestet werden. Auch die Akzeptanztests resultieren aus den User Stories. Der Unterschied zu den Unit Tests ist, dass die Akzeptanztests nicht mehr eindeutig einem Software-Modul zugeordnet werden können (und sollen). Mit Hilfe der Akzeptanztests kann der Kunde „seine“ User Stories testen, sie ersetzen die (in der Regel nicht sauber niedergeschriebenen) High-Level-Spezifikationen des Systems (Abb. 3.11). Akzeptanztests sind *Black Box Tests*. Zu ihrer Durchführung muss der Code nicht vorliegen. Hier setzt auch die *Qualitätssicherung* an, die zwar ansonsten bei XP nicht explizit in Erscheinung tritt, die aber nach Erfahrung des Autors in einem agilen Projekt unabdingbar ist.

---

[35] Abfragen von E-Mails, Verwaltungsaufgaben usw.

[36] Zumindest bei Unit Tests ist das richtig.

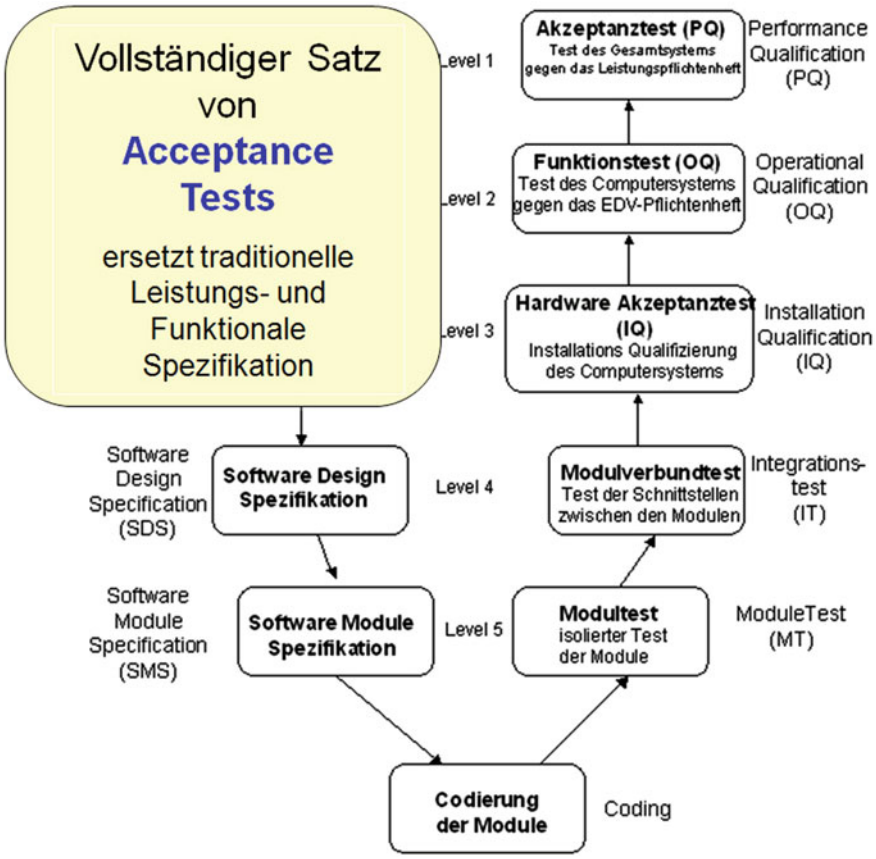

Abb. 3.11 High-Level-Spezifikationen des V-Modells werden durch Akzeptanz-Tests ersetzt

Auch Akzeptanztests sollten in XP möglichst automatisiert werden (Wells 2009). Dies ist aber aller Erfahrung nach weitaus schwerer als die Automatisierung der Unit Tests. Im studentischen Labor ist das den Teams in fünf Jahren bisher noch nicht gelungen.

Es ist sinnvoll ein *Score* für diese Tests einzuführen, so dass das Entwicklerteam (und jeder andere) jederzeit den Stand der Tests sieht. Sinnvoll erachtet der Autor auch ein System aus *Ampeln*, die visuell (rot und grün) den Erfolg der wichtigsten Tests z.B. auf einer Intranet-Seite anzeigen.

3.4 Erweiterte XP-Praktiken

Die bisher beschriebenen „traditionellen“ XP-Praktiken wurden bereits 2000 von Kent Beck vorgeschlagen (Beck 2000) und haben noch immer ihre Berechtigung.

Sie werden auch heute noch gepflegt und in Projekten angewandt (Wells 2009). Trotzdem hat Kent Beck die traditionellen XP-Praktiken in seinem neueren Buch (Beck u. Andres 2004) durch einen neuen Satz von 13 Primärpraktiken und 11 Folgepraktiken ersetzt. Dies wird mit der Weiterentwicklung von XP und der wachsenden Erfahrung mit diesem agilen Prozessmodell begründet. Der Unterschied zwischen den traditionellen und den in diesem Kapitel beschriebenen „erweiterten" XP-Praktiken ist allerdings nicht so groß, wie er auf den ersten Blick erscheint. XP bleibt sich treu!

3.4.1 Primärpraktiken

Die von Kent Beck 2004 neu definierten 13 Primärpraktiken (Beck u. Andres 2004) sind:

- Team sitzt räumlich zusammen
- Komplettes Team
- Informative Arbeitsumgebung
- Energized Work
- Pair Programming
- Stories
- Wöchentlicher Zyklus
- Vierteljährlicher Zyklus
- Freiraum
- Ten-Minute Build
- Continuous Code Integration
- Test First Entwicklung
- Inkrementelles Design

Einige der Primärpraktiken sind bereits als traditionelle Praktiken bekannt und brauchen nicht noch einmal beschrieben werden. Dies sind *User Stories* (Kap. 3.3.1) und *Inkrementelles Design* (Kap. 3.3.2) sowie *Pair Programming*, *Continuous Code Integration* und *Test First Entwicklung* (Kap. 3.3.3). Interessant sind die „neuen" XP-Praktiken:

3.4.1.1 Team sitzt räumlich zusammen

Diese Primärpraxis ist nicht wirklich neu. Schon die geforderte paarweise Arbeit (mit wechselnder Besetzung) verlangt eine räumliche Nähe des Teams. Neu ist, dass XP diese agile Selbstverständlichkeit explizit fordert.

3.4.1.2 Komplettes Team

Das Team muss alle Fähigkeiten vereinen, die zur Erfüllung der gestellten Aufgabe notwendig sind. Dieser wichtige Aspekt soll in Kap. 7 ausführlich behandelt werden.

3.4.1.3 Informative Arbeitsumgebung, Energized Work, Freiraum

Die Arbeitsumgebung soll den Bedürfnissen des Teams angepasst sein, aber auch den aktuellen Stand des Projekts widerspiegeln. Es ist z.B. sinnvoll, die User Story Cards so auf Pinwänden anzuordnen, dass sofort klar wird, wie viele davon erledigt und wie viele noch offen sind (Abb. 3.12). Innerhalb weniger Sekunden kann sich jeder Besucher vom Stand des Projekts überzeugen.

Eine gute und angepasste Arbeitsumgebung „beflügelt" das Team und befähigt es zu „Energized Work", also zu einer engagierten Arbeit voller Energie. Nur so kann das Team Außergewöhnliches leisten.

Ebenso wichtig ist auch der *Freiraum* der einzelnen Teammitglieder. Sie sollten auch eigene Aktivitäten entwickeln können, sich also in einem (begrenzten) Teil ihrer Zeit mit Dingen beschäftigen, die sie interessieren (und die in aller Regel mit ihrem Thema im Projekt zu tun haben). Das Team profitiert von den neu gewonnenen Ideen.

3.4.1.4 Ten-Minute Build

Neu ist eine klare Aussage Kent Becks zu den Zeitskalen im Projekt. Er weist daraufhin, dass ein Build des Software-Systems inklusive aller automatisierten Tests nicht länger als zehn Minuten dauern sollte. Dies ist eine vernünftige Größe, weil ein längeres Warten dazu führt, dass die Zahl der Builds reduziert wird. Der Autor

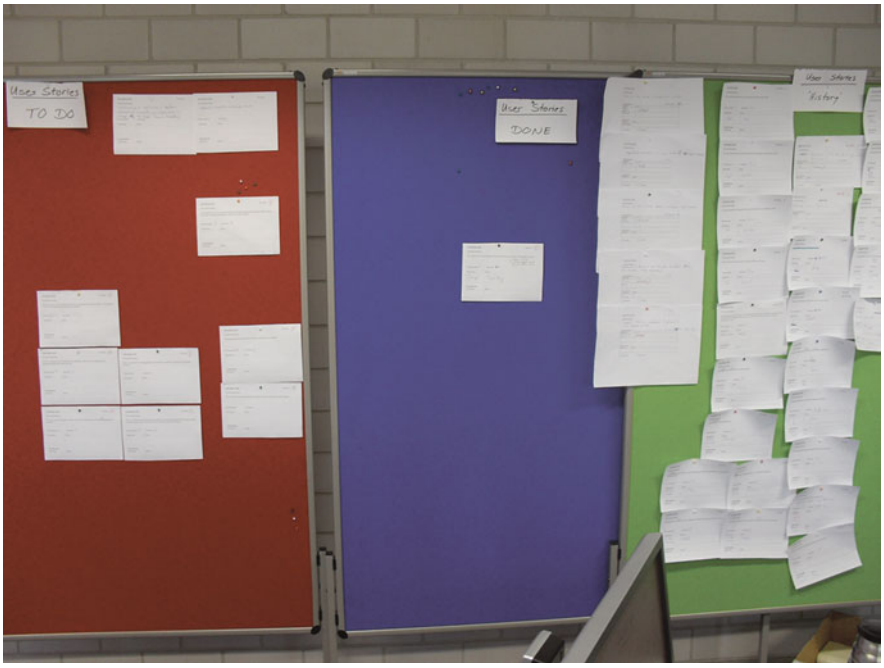

Abb. 3.12 Die Anordnung der User Story Cards zeigt den aktuellen Stand des Projekt (2009)

arbeitete in den 1980er Jahren in einem Projekt, bei dem ein Build über eine Stunde lief. Das Resultat waren Builds in Abständen von jeweils einigen Tagen mit der Gefahr von Schnittstellenproblemen zwischen den separat implementierten User Stories. Ausgelöst durch den Preisverfall leistungsfähiger Hardware, lässt sich diese Problematik heutzutage leicht in den Griff bekommen.

3.4.1.5 Wöchentlicher Zyklus

Die zweite Zeitskala, die XP vorgibt, bezieht sich auf die Länge der Iterationen. Kent Beck schlägt im Gegensatz zu den traditionellen Praktiken (Kap. 3.2) eine kürzere Zeitskala vor. Spricht er anfangs von zwei bis drei Wochen, bevorzugt er nunmehr eine Woche, weil sich damit alles auf Freitag fokusiert. Freitag ist das Ende eines Arbeitsabschnitts. Es liegt somit nahe, Iterationen auf eine Woche hin zu planen.[37] Man beginnt die Woche planend, schreibt darauf die automatisierten Tests und verbringt den Rest der Woche damit, den Code so zu implementieren, dass die zuvor geschriebenen Tests erfüllt werden.

3.4.1.6 Vierteljährlicher Zyklus

Die nächstgrößere, vorgegebene Zeiteinheit ist das Vierteljahr. In Analogie zur (natürlichen) Länge einer Jahreszeit können größere Einheiten der Software implementiert werden. Es bietet sich an, vierteljährliche Releases durchzuplanen und durchzuführen.[38]

Auch Reflexionssitzungen finden in XP einmal im Vierteljahr statt. Wichtig ist, nicht nur Probleme zu benennen, sondern auch Lösungen anzustoßen, also das „Big Picture“ des Projekts nicht aus den Augen zu verlieren.

Um es noch einmal ganz deutlich zu machen: Die Aussage zu den Zeitskalen ist *neu* und in den ersten Ansätzen zu XP noch nicht in dieser Deutlichkeit zu finden.[39]

3.4.2 Folgepraktiken

XP kennt 11 Folgepraktiken, die allerdings erst zum Einsatz kommen sollen, wenn das Team die Primärpraktiken beherrscht:

- Einbeziehung des Kunden
- Inkrementelle Lieferung
- Teamkontinuität
- Teams können schrumpfen
- Ursachenanalyse

---

[37] Obwohl Beck das im entsprechenden Kapitel nicht explizit verlangt (Beck u. Andres 2004).

[38] Obwohl Beck auch das nicht explizit verlangt (Beck u. Andres 2004).

[39] Vergleiche dazu (Beck 2000) und (Beck u. Andres 2004).

- Shared Code
- Eine Codebasis
- Tägliche Lieferung
- Vertrag mit verhandelbarem Umfang
- Pay-Per-Use

Auch einige der Primärpraktiken sind bereits als traditionelle Praktiken bekannt. Die *Einbeziehung des Kunden* ins Team[40] war von Anfang an eine Forderung von Kent Beck. Ebenso wichtig war immer eine *inkrementelle Lieferung* kleiner Releases.[41] *Shared Code* korrespondiert mit der traditionellen Praxis *Collective Code Ownership* (Kap. 3.3.3). Neu ist die explizite Erwähnung der nachfolgenden Folgepraktiken:

3.4.2.1 Teamkontinuität

Effektive Teams sollten, soweit es geht, zusammengehalten werden, denn ein Team ist mehr als die Summe der einzelnen Teammitglieder. Ein gutes „Klima“ in einem Team führt dazu, dass sich die einzelnen Teammitglieder wohlfühlen und damit ihre optimale Leistung abrufen können. Das Problem, dass Teams auseinandergerissen werden, besteht hauptsächlich in großen Entwicklungsorganisationen.

3.4.2.2 Teams können schrumpfen

Mit der Zeit wachsen die Fähigkeiten eines Projektteams und damit die Bearbeitungsgeschwindigkeit der definierten User Stories. In einem solchen Fall ist es sinnvoller, den „Workload“ konstant zu halten und die Teamgröße zu verkleinern.[42]

3.4.2.3 Ursachenanalyse

Wenn ein Bug die Entwicklung unentdeckt überstanden hat, muss nach den Ursachen geforscht werden. Zuerst sollte auf System-Ebene ein (idealerweise automatisierter) Test entwickelt werden, der den Fehler zeigt. Danach sollte auf Modul-Ebene der korrespondierende Test-Case implementiert werden.[43] Ist der Fehler dann behoben, sollte nach den Ursachen geforscht werden, warum der Fehler nicht entdeckt wurde. War die Person, die den Fehler hätte finden können, nicht Mitglied des Teams? Diese und ähnliche Ursachen müssen anschließend geklärt werden.

---

[40] „Der Kunde ist immer verfügbar“ (Kap. 3.3.3).

[41] „Kleine Releases“ (Kap. 3.3.1) und „Integriere oft!“(Kap. 3.3.3)

[42] Im Rahmen der studentischen Projekte hat der Autor diese Möglichkeit allerdings nicht. Die Teamgröße ist definiert durch die Kursgröße.

[43] Dies wird allerdings nur bei funktionalen Fehlern möglich sein. Performance-Probleme sind in der Regel auf Modulebene nur schwer zu identifizieren.

3.4.2.4 Eine Codebasis, tägliche Lieferung

Das Team und das Projekt haben immer eine aktuelle Codebasis. Natürlich wird jedes XP-Paar eine User Story zuerst lokal implementieren. Spätestens aber nach ein paar Stunden sollte der Code mit den anderen wieder synchronisiert werden. Ansonsten wird der Aufwand der Code-Integration zu groß.[44] Aus demselben Grund ist vorzusehen, dass jede Nacht ein neuer *Build* der Software generiert wird. Nur eine ständige Synchronisation der Arbeit eines Programmierers mit dem „Produktionscode", verhindert Integrationsrisiken.

3.4.2.5 Vertrag mit verhandelbarem Umfang

In Software-Entwicklungsverträgen sollten Zeitaufwand, Kosten und Qualität definiert sein. Der genaue Funktionsumfang sollte aber kontinuierlich angepasst werden können. Soviel Freiraum sollte der Vertrag vorsehen. Es entspricht dem agilen Gedanken, dass Spezifikationen in Form von User Stories mit der Zeit entwickelt werden. Diese Praxis ist also eine direkte Folge des agilen Ansatzes. Nichtsdestotrotz haben Auftraggeber große Mühe mit der Realisierung eines solchen Freiraums in Verträgen.

3.4.2.6 Pay-Per-Use

Pay-Per-Use bedeutet, dass der Kunde nur bezahlt, wenn er die Software nutzt. Bei jeder Nutzung der Software entstehen somit angemessene Gebühren. Über die Nutzungsrate der Software entsteht letztlich eine Aussage über die Kundenzufriedenheit. Diese kann vom Entwicklungsteam bei der Weiterentwicklung berücksichtigt werden. Je näher das Produkt an den Bedürfnissen des Kunden ist, umso höher ist die Nutzung und sind somit die Gebühren, die der Kunde bereit ist zu zahlen.[45]

Problematisch ist dieser Ansatz allerdings bei speziellen Kundenentwicklungen. Hier erwartet der Kunde mit Recht Kostensicherheit, während das Entwicklungsteam seine Kosten gedeckt sehen will. (Vielleicht sollte man Pay-Per-Use auch nur im übertragenen Sinn verstehen!)

3.4.3 Unterschiede zwischen erweiterten und traditionellen XP-Praktiken

Seit seinem ersten Buch (Beck 2000) hat Kent Beck einige der traditionellen *Rules and Practices* aus der Liste der XP-Praktiken herausgenommen. Es sind dies hauptsächlich nicht XP-spezifische Praktiken wie

[44] Siehe dazu auch Kap. 6.4.

[45] Dieser Ansatz ist auch als *Application Service Providing* (ASP) bekannt.

- das Messen der Projektgeschwindigkeit
- Stand Up Meetings
- der Einsatz von CRC-Karten
- das Implementieren von Prototypen (Spike Solutions)
- Refactoring einer Software
- Optimieren am Schluss
- keine Überstunden

Diese Praktiken gehören heutzutage zum „Good Engineering“ und werden von der Mehrzahl der Entwickler nicht mehr als XP-spezifisch angesehen.

Andere traditionelle Praktiken haben sich (meist nur leicht) gewandelt oder werden nicht mehr explizit erwähnt:

- Das Planungsspiel (Kap. 3.3.1) weicht dem Vertrag mit verhandelbarem Umfang.
- Die Iterationsplanung mit kleinen Releases (Kap. 3.3.1) wird präzisiert zum wöchentlichen Zyklus. Beck reduziert die ursprünglich zwei- bis dreiwöchigen Iterationen auf *eine Woche*, um das Team auf Freitag als Abgabetag zu fokusieren.
- *Move People around* (Kap. 3.3.1) muss nicht mehr separat erwähnt werden. Diese Praxis dient der Verbreitung des Wissens um den Code im Team. Pair Programming und Teamkontinuität bewirken letztlich dasselbe.
- Die Regel *Fix XP when it breaks* findet man in den erweiterten XP-Praktiken nicht mehr. Das Heikle an dieser Regel ist festzustellen, ab wann der Prozess *kein* XP mehr ist.[46]
- Interessant ist das Verschwinden der Forderung nach einer *Systemmetapher* (Kap. 3.3.2). Aus dem studentischen Labor ist bekannt, dass sich die Suche nach einer Metapher im Allgemeinen als schwierig erweist. Aus diesem Grund wird dort auf diese Praxis verzichtet. Es ist davon auszugehen, dass diese generelle Schwierigkeit Kent Beck bewogen hat, auf diese Praxis zu verzichten.

3.5 Berühmte XP-Praktiken

Wie schon erwähnt, sind viele der XP-Regeln und -Praktiken nicht neu. Es ist eher ihre Kombination und ihre strikte Einhaltung, was XP ausmacht. Man kann die XP-Praktiken durchaus in zwei Kategorien einteilen: Regeln, die jeder moderne Projektmanager sofort akzeptiert und meist auch schon einsetzt, und Regeln, die Widerspruch hervorrufen:

[46] Hier sei auf Kap. 6 und 7 verwiesen.

Kein erfahrener und zeitgemäßer Projektleiter bezweifelt die Wichtigkeit einer vernünftigen Versions- und Release-Planung, kleiner Releases, einer iterativ-inkrementellen Vorgehensweise und eines einfachen Designs mit Namenskonventionen. Ebenso gehört heutzutage ein ausführliches und automatisiertes Testen „ab dem ersten Tag“ bereits auf Modul-Ebene zu den modernen Software-Projekttechniken.

Die andere Gruppe der XP-Regeln und -Praktiken gibt mehr Anlass zu Diskussionen, wie der Autor aus eigener Erfahrung weiß und im studentischen Labor verifizieren konnte. Diese Regeln und Praktiken, die XP berühmt gemacht haben, werden im studentischen Labor (Kap. 6) auf ihre Praxistauglichkeit überprüft:

3.5.1 Erstellen von User Stories

Die Anforderungen sollen nicht in Form von Dokumenten, sondern in Form von kurzen, dreisätzigen Mini-Spezifikationen vom Kunden in seiner „Sprache“ abgefasst werden. Der Aufwand für die Implementierung des zugehörigen Software-Moduls soll drei Wochen nicht überschreiten. Kann der Kunde das? Wie kann er drei Wochen Aufwand abschätzen? Macht eine solche zeitlich feine Gliederung überhaupt Sinn? Braucht er Unterstützung? (Und wenn ja, von wem?)

3.5.2 Paarweises Programmieren

Zwei Programmierer arbeiten zusammen an *einem* Computer. Beck verspricht eine Erhöhung der Software-Qualität bei gleicher Effizienz wie bei einem Team von zwei Programmierern an zwei Computern. Im Paar soll ein Programmierer taktisch denken, sich also mit dem Funktionsumfang einer aktuellen Klassenmethode beschäftigen, während der andere strategisch denkt, sich also mit der Frage beschäftigt, ob obige Klassenmethode überhaupt in diese Klasse passt. Die Programmierer wechseln regelmäßig, ggf. alle paar Stunden, ihre Rollen. Diese Forderung wird wohl in den wenigsten Projekten erfüllt. Die Frage nach dem Sinn dieser XP-Regel soll im Weiteren untersucht werden.

3.5.3 Collective Code Ownership, Shared Code

Jedes Team-Mitglied soll Verantwortung für den gesamten Code tragen und diesen auch entsprechend verstehen. Jeder, der eine Änderung am Code eines anderen Paares braucht, kann diese selbstständig durchführen, da das automatisierte Testen mit einer entsprechenden Versionskontrolle es ermöglicht, den „Urzustand“ der Software vor der Änderung wieder herbeizuführen. Dieser Punkt ist heikel! Jeder der schon einmal mit entsprechenden Tools gearbeitet hat, weiß, dass man hierzu ein tiefes Verständnis der zu ändernden Software haben oder die Bereitschaft zum

Erlangen dieses Verständnisses mitbringen muss. Auch diese Regel soll untersucht werden.

3.5.4 Kontinuierliche Code-Integration, eine Codebasis

Integration in kurzen Abständen, idealerweise kontinuierlich, dient dem Zweck, immer eine aktuelle Codebasis zu haben. Die Integration des Codes wird von den XP-Paaren geleistet, die eigenverantwortlich nacheinander an einem Integrationsrechner die Projektmodule zu einem lauffähigen Release verknüpfen. Hier muss die Frage geklärt werden, ob alle Paare genügend Willen und Know-how mitbringen, um eine in der Regel komplexe Integration erfolgreich durchzuführen.

3.5.5 Kunde im Team, Einbeziehung des Kunden

Beck fordert die permanente Mitgliedschaft des *Kunden im Team* (zumindest in den traditionellen Praktiken und in den erweiterten Praktiken/Folgepraktiken). Dies soll eine schnelle Reaktion auf sich ändernde Kundenwünsche ermöglichen, selbst in späten Projektphasen. Spezifikationsdetails können „face to face“ mit dem Kunden besprochen werden. Die geforderte dauernde Anwesenheit des Kunden im Projekt ist in der Praxis naturgemäß ein Problem, da erfahrene Kunden kaum vollständig einem Software-Projekt zur Verfügung stehen. Ein anderer Einwand ist, dass der Kunde auf diese Weise zu viele Interna des Projekts mitbekommen könnte. Und wird dadurch die Projekt-Kommunikation insbesondere mit dem Kunden wirklich besser? Ein Indiz der Problematik dieser Praxis ist, dass Kent Beck sie in seinen erweiterten Praktiken in die Folgepraktiken für erfahrene Teams einordnet.

Die hier diskutierten XP-Praktiken wurden intensiv im studentischen Labor untersucht und werden im Kap. 6 intensiv diskutiert.

Literatur

Balzert H (2001 u. 1998) Lehrbuch der Software-Technik. Band 1 und 2. Amsterdam: Elsevier-Verlag

Beck K (2000) Extreme Programming. Reading, MA: Addison Wesley

Beck K, Andres C (2004) Extreme Programming Explained – Embrace Change, 2nd Ed. Boston, MA: Addison-Wesley

Boehm B, Abts C, Brown W, Chulani S, Clark B (2000) Software Cost Estimation with Cocomo II. Upper Saddle River, NJ: Prentice Hall

Drews G, Hillebrand N (2007) Lexikon der Projektmanagement-Methoden. Haufe, München

Fowler M, Scott K (2000) UML konzentriert, 2. Auflage. Bonn: Addison-Wesley

GPM (2009) http://www.gpm-infocenter.de. Zugriff Oktober 2009

Jacobsen I, Booch G, Rumaugh J (1999) The Unified Software Development Process. Reading, MA: Addison-Wesley

Hürten R (1999) Function-Point Analysis – Theorie und Praxis. Renningen-Malmsheim: expert-Verlag

Wells D. (2009) www.extremeprogramming.org. Zugriff Oktober 2009

Williams L, Kessler R (2002) Pair Programming Illuminated. New York, NY: Addison-Wesley