**Extreme Programming**

**Seminararbeit**

Objektorientiertes Software Engineering



|  |  |
| --- | --- |
| Studiengang: | Wirtschaftsinformatik |
| Dozent: | Dr. sc. pol. Peter Knorr |
| vorgelegt von: | Torsten Abraham |
|  | Matthias Holdorf |
|  | Alexander Nikiforow |
|  | Maximilian Ruppel |
|  | Matthias Schroeder |
| vorgelegt am: | 27.01.2014 |

# Inhaltsverzeichnis

[Inhaltsverzeichnis I](#_Toc378580682)

[Abbildungsverzeichnis IV](#_Toc378580683)

[Tabellenverzeichnis V](#_Toc378580684)

[1. Einleitung 1](#_Toc378580685)

[1.1 Motivation und Ziele 1](#_Toc378580686)

[1.2 Aufbau der Arbeit 1](#_Toc378580687)

[2. Entstehungsgeschichte von XP 2](#_Toc378580688)

[2.1 Prozessmodelle 2](#_Toc378580689)

[2.2 Entstehung und Entwicklung von XP 3](#_Toc378580690)

[3. XP im Überblick 5](#_Toc378580691)

[4. Werte 7](#_Toc378580692)

[4.1 Kommunikation 7](#_Toc378580693)

[4.2 Einfachheit 7](#_Toc378580694)

[4.3 Feedback 8](#_Toc378580695)

[4.4 Mut 8](#_Toc378580696)

[5. Prinzipien 9](#_Toc378580697)

[5.1 Primäre Prinzipien 9](#_Toc378580698)

[5.2 Sekundäre Prinzipien 10](#_Toc378580699)

[6. Techniken 12](#_Toc378580700)

[6.1 Management-Techniken 12](#_Toc378580701)

[6.1.1 Kunde vor Ort (On-Site Cutomer) 12](#_Toc378580702)

[6.1.2 Planungsspiel (Planning Game) 13](#_Toc378580703)

[6.1.3 Kurze Releasezyklen (Short Releases) 17](#_Toc378580704)

[6.2 Team-Techniken 17](#_Toc378580705)

[6.2.1 Metapher (Metaphor) 18](#_Toc378580706)

[6.2.2 Gemeinsame Verantwortung (Collective Ownership) 18](#_Toc378580707)

[6.2.3 Fortlaufende Integration (Continous Integration) 18](#_Toc378580708)

[6.2.4 Programmierstandards (Coding Standards) 19](#_Toc378580709)

[6.2.5 Nachhaltiges Tempo (Sustainable Pace) 19](#_Toc378580710)

[6.3 Entwickler-Techniken 19](#_Toc378580711)

[6.3.1 Testen (Testing) 19](#_Toc378580712)

[6.3.2 Einfaches Design (Simple Design) 19](#_Toc378580713)

[6.3.3 Refactoring 20](#_Toc378580714)

[6.3.4 Programmieren in Paaren (Pair Programming) 20](#_Toc378580715)

[7. Rollen 22](#_Toc378580716)

[7.1 Programmierer 22](#_Toc378580717)

[7.2 Kunde 23](#_Toc378580718)

[7.3 Tester 24](#_Toc378580719)

[7.4 Tracker 24](#_Toc378580720)

[7.5 Coach 24](#_Toc378580721)

[7.6 Consultants 26](#_Toc378580722)

[8. Ablauf eines XP-Projektes 27](#_Toc378580723)

[8.1 Vorbereitung 28](#_Toc378580724)

[8.2 Planung 28](#_Toc378580725)

[8.3 Iteration 29](#_Toc378580726)

[8.4 Integration der Produktivversion 29](#_Toc378580727)

[8.5 Wartung 29](#_Toc378580728)

[8.6 Auflösung 30](#_Toc378580729)

[9. Zusammenfassung 31](#_Toc378580730)

[9.1 Vorteile und Chancen 32](#_Toc378580731)

[9.1.1 Vorteile für den Kunden 32](#_Toc378580732)

[9.1.2 Vorteile für den Entwickler 33](#_Toc378580733)

[9.1.3 Vorteile für das Management und das Projekt 33](#_Toc378580734)

[9.2 Risiken 34](#_Toc378580735)

[9.3 Bewertung 35](#_Toc378580736)

[9.4 Abgrenzung zu SCRUM 35](#_Toc378580737)

[Literaturverzeichnis 37](#_Toc378580738)

# Abbildungsverzeichnis

[Abbildung 1: Änderungskosten 6](#_Toc378557157)

[Abbildung 2: Notwendigkeit der Planung 6](#_Toc378557158)

[Abbildung 3: Storycard aus dem C3 Projekt von Kent Beck 14](#_Toc378557159)

[Abbildung 4: Verknüpfung unter den XP-Techniken 21](#_Toc378557160)

[Abbildung 5: Iterativer Ablauf von Extreme Programming mit jeweiligem Zeithorizont 27](#_Toc378557161)

# Tabellenverzeichnis

[Tabelle 1: Die 14 Prinzipien des XP 5](#_Toc378557178)

# Einleitung

## Motivation und Ziele

Diese Ausarbeitung ist im Rahmen des Kurses *Objektorientiertes Software Engineering* an der Fachhochschule Flensburg entstanden. Sie gibt einen Einblick in die agile Software-Entwicklungsmethode *Extreme Programming (XP)* und soll in erster Linie als Informationsquelle für die Kommilitonen im Kurs dienen.

## Aufbau der Arbeit

Die Arbeit ist in neun Kapitel aufgeteilt. Das vorliegende erste Kapitel ist die Einleitung, die Informationen zu Ziel, Motivation und Aufbau der Arbeit liefert. Das zweite Kapitel befasst sich mit Prozessmodellen und der Entstehungsgeschichte des *XP*. Daran anschließend gibt das dritte Kapitel einen Überblick über die wesentlichen Bestandteile des *Extreme Programming*, auf die in den darauf folgenden Kapiteln eingegangen wird. So widmet sich das vierte Kapitel den vier Werten des *XP*, das fünfte Kapitel den 14 Prinzipien. Im sechsten Kapitel werden die Techniken und Praktiken genauer vorgestellt, welche im *Extrem Programming* verwendet werden. Anschließend wird im siebten Kapitel auf die Rollen eines *XP*-Projektes eingegangen, im achten Kapitel der Ablauf eines *XP*-Projektes vorgestellt. Den Abschluss der Arbeit bildet Kapitel neun, in dem eine Bewertung der Methode *XP*, sowie eine Abgrenzung zur verwandten Methode *SCRUM* vorgenommen wird.

# Entstehungsgeschichte von XP

## Prozessmodelle

Bevor auf die Entwicklung und die einzelnen Bestandteile des *XP* eingegangen werden kann, muss grundsätzlich geklärt werden, was Prozessmodelle sind und wie sie sich grundlegend voneinander unterscheiden.

In der Software-Entwicklung dienen Prozessmodelle der Festlegung des Vorgehens und des Ablaufs zur Erstellung einer Software. Der Ablauf wird häufig in Phasen aufgeteilt – Planung, Analyse, Implementierung und Tests sind einige davon. Neben dem Ablauf beschreiben Prozessmodelle auch die an der Entwicklung beteiligten Personen beziehungsweise Rollen sowie die Art und den Umfang der anzufertigenden Dokumentation des Entwicklungsprozesses.[[1]](#footnote-1)

Zur Klassifizierung von Prozessmodellen können die Begriffe *schwergewichtig* und *leichtgewichtig* verwendet werden. Diese Angabe des *Gewichts* bezieht sich vor allem auf Art und Umfang der Dokumentation. Die schwergewichtigen Prozessmodelle sind durch eine detaillierte Dokumentation gekennzeichnet, wodurch spätere Änderungen an vorher definierten Anforderungen nur mit hohem Aufwand möglich sind. Die leichtgewichtigen Prozessmodelle hingegen dokumentieren nur das Nötigste und gehen von dem Ansatz aus, dass die Anforderungen zu Beginn des Projektes nicht vollständig bekannt sind, sondern sich während der Entwicklung ergeben bzw. ändern.[[2]](#footnote-2)

Zu den schwergewichtigen Prozessmodellen können unter anderem das V-Modell, das Phasenmodell und das Spiralmodell gezählt werden.

Die sogenannten *agilen Prozessmodelle,* zu denen auch das *XP* gehört, werden der Kategorie leichtgewichtig zugeordnet. Sie berücksichtigen alle das *Manifesto for Agile Software Development* – ein Werk aus dem Jahr 2001. Es beschreibt Grundsätze der agilen Softwareentwicklung. Zu den Autoren gehören unter anderem Kent Beck, der *XP* entwickelt hat, sowie viele weitere Verfechter der agilen Entwicklungsmethoden.[[3]](#footnote-3)

Der Wortlaut des Manifests lautet:

„We are uncovering better ways of developing  
software by doing it and helping others do it.  
Through this work we have come to value:

**Individuals and interactions** over processes and tools  
**Working software** over comprehensive documentation  
**Customer collaboration** over contract negotiation  
**Responding to change** over following a plan

That is, while there is value in the items on  
the right, we value the items on the left more.“[[4]](#footnote-4)

Nach diesem Manifest haben die Personen und Interaktionen der Personen, eine höhere Priorität als die verwendeten Prozesse und Werkzeuge. Der Stellenwert einer laufenden Software wird höher eingeschätzt, als die Anfertigung einer umfassenden Dokumentation. Die Kooperation mit dem Kunden ist laut Manifest wichtiger, als die Vertragsverhandlungen und es ist dem Manifest nach weitaus wichtiger auf Änderungen einzugehen, als sich strikt an einen vorab gefassten Plan zu halten.

## Entstehung und Entwicklung von XP

Ein Problem der schwergewichtigen Prozessmodelle ist die fehlende Flexibilität. Anforderungen werden zu Beginn des Projektes aufgenommen und als Grundlage für alle anschließenden Arbeitsschritte verwendet. Änderungen sind nicht vorgesehen; treten sie dennoch auf, führt dies zu Problemen. Diese fehlende Flexibilität führte zur Entwicklung von leichtgewichtigen, agilen Prozessmodellen, die von Änderungen der Anforderungen während des Projektes ausgehen.[[5]](#footnote-5)

Das sogenannte *Chrysler Comprehensive Compensation* Projekt (kurz C3) hatte als Ziel eine Lohn- und Gehaltssoftware zu erstellen. Das Projekt wurde mit einem schwergewichtigen Prozessmodell begonnen und drohte zu scheitern. Um das Scheitern zu verhindern entwickelte Kent Beck das leichtgewichtige, agile Prozessmodell *Extreme Programming* und machte das Projekt damit zum Erfolg. Das C3-Projekt aus dem Jahr 1996 gilt heute als erstes *XP*-Projekt.[[6]](#footnote-6)

Nach dem Erfolg im C3-Projekt wurden weitere Projekte nach der *XP*-Methode realisiert, wie bspw. das VCAPS-Projekt von Ford.[[7]](#footnote-7) Das erste Buch zur Methode *XP* „*Extreme Programming Explained: Embrace Change“* wurde von Kent Beck geschrieben und erschien im Jahr 1999. Darauf folgten weitere Publikationen von ihm, wie das Buch *„Planning Extreme Programming“* und im Jahr 2004 eine zweite Auflage des erstgenannten Werkes.

In der Neuauflage werden einige Entwicklungen eingebunden, die das Prozessmodell leicht abändern. Die Änderungen seien laut Hanser jedoch nur geringfügig.[[8]](#footnote-8)

# XP im Überblick

Bevor in den nächsten Kapiteln die Bestandteile des *Extreme Programming* vorgestellt werden, soll in diesem Kapitel, als eine Art Rahmen, ein allgemeiner Eindruck von *XP* vermittelt werden.

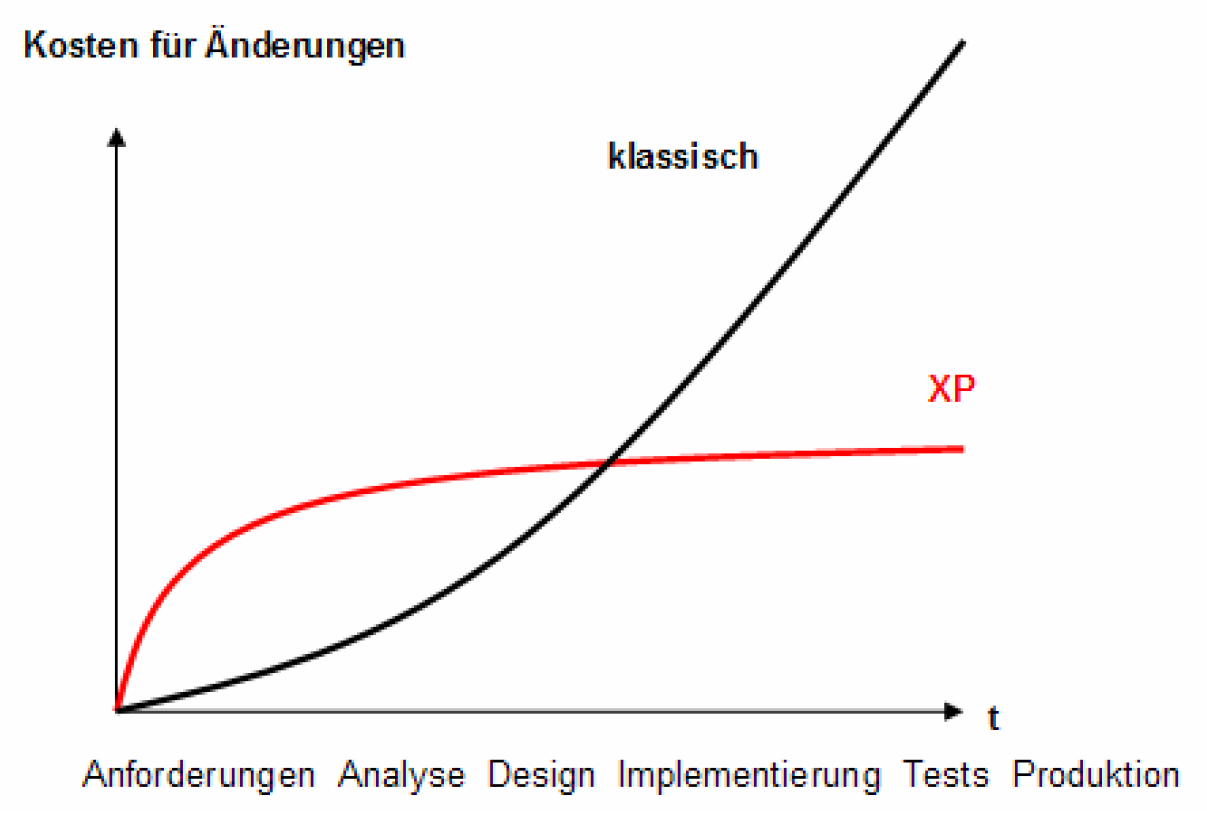
Wie im vorangegangenen Kapitel beschrieben, kann *Extrem Programming* zu den agilen Entwicklungsmethoden gezählt werden, die als *leichtgewichtige* Prozessmodelle bekannt sind und die Grundsätze des *Manifesto for Agile Software Development* berücksichtigen. Ein Unterschied zu anderen agilen Prozessmodellen ist die explizite Vorgabe anhand der *Rules and Practices* im *XP*, die definieren, wie bei Design, Planung, Koordination und Tests vorzugehen ist. Andere Prozessmodelle räumen bei genannten Punkten größere Freiräume ein. Den Regeln übergeordnet sind die *Werte* des *XP* bestehend aus *Kommunikation, Einfachheit, Feedback* und *Mut*.[[9]](#footnote-9) Eine ausführliche Beschreibung der Werte folgt in Kapitel 0. Aus diesen fünf Werten lassen sich die *14 Prinzipien* des *XP* ableiten:

|  |  |
| --- | --- |
| *Unmittelbares Feedback* | *Spielen, um zu gewinnen* |
| *Einfachheit anstreben* | *Gezielte Experimente* |
| *Inkrementelle Veränderungen* | *Offene, ehrliche Kommunikation* |
| *Veränderungen wollen* | *Die Instinkte der Mitarbeiter nutzen, nicht dagegen arbeiten* |
| *Qualitätsarbeit* | *An örtliche Gegebenheiten anpassen* |
| *Lernen lehren* | *Mit leichtem Gepäck reisen* |
| *Kleine Anfangsinvestitionen* | *Ehrliches Messen* |

**Tabelle 1: Die 14 Prinzipien des XP[[10]](#footnote-10)**

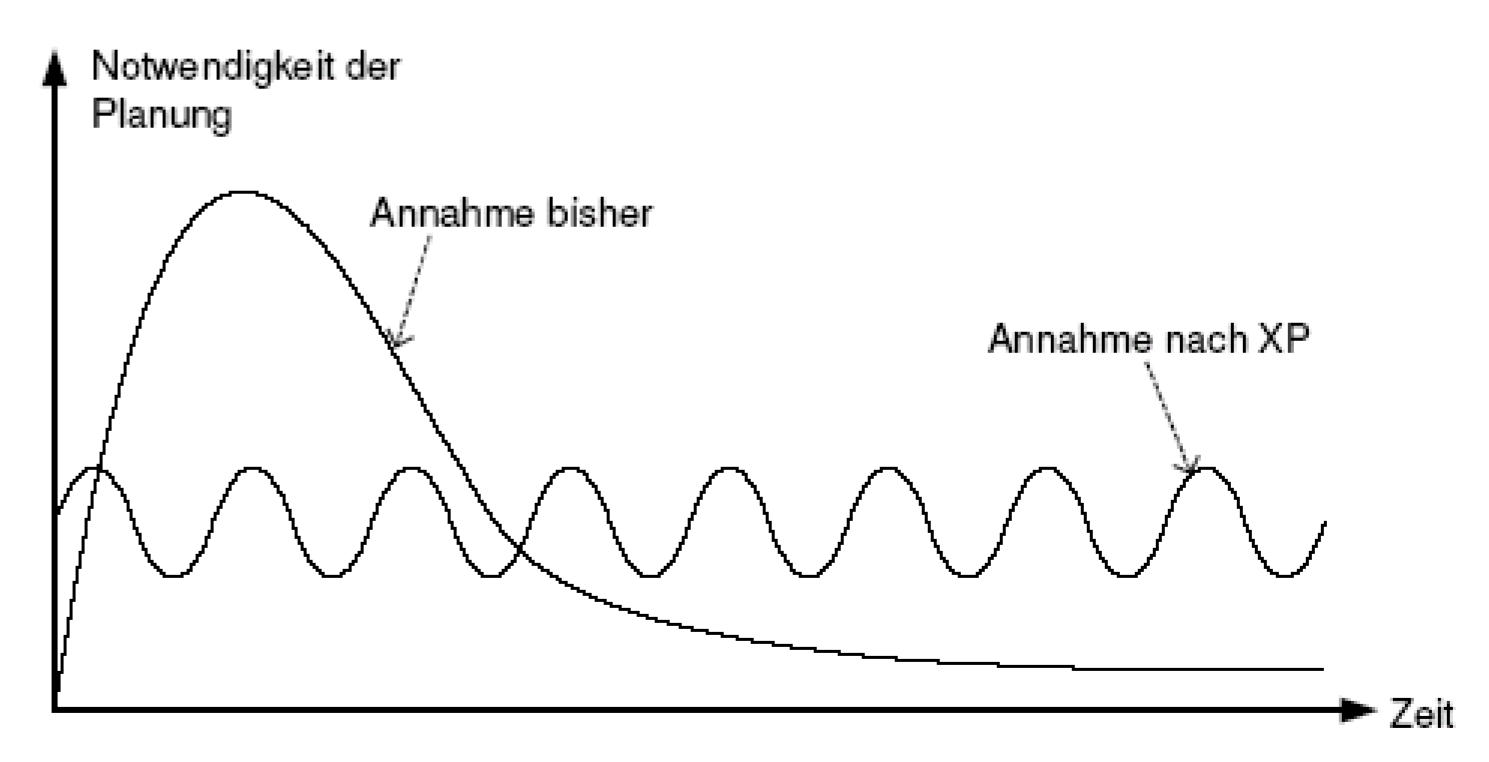
Eine ausführliche Beschreibung der Prinzipien ist in Kapitel 5 zu finden. Neben den Werten und Prinzipien umfasst *XP* einige Techniken und Praktiken, darunter das *Pair Programming,* das Erstellen von *User Stories* und die *Einbeziehung des Kunden* in das Team.[[11]](#footnote-11) Weitere Techniken und Praktiken sind in Kapitel 6 aufgeführt. Die Prozessschritte im *Extrem Programming* sind iterativ und ähneln damit dem schwergewichtigen *Spiralmodell.* In Zyklen werden einzelne *Inkremente* erstellt, die anschließend begutachtet und freigegeben werden können, woraufhin sie implementiert werden. Näheres zum Ablauf ist in Kapitel 8 erklärt.

Einen entscheidenden Vorteil des *XP* gegenüber klassischen Prozessmodellen ist exemplarisch in Abbildung 1 zu sehen. Dargestellt werden die Änderungskosten im Laufe der Zeit eines Projektes. Es lässt sich deutlich erkennen, dass die Änderungskosten zu einem späten Zeitpunkt im Projektverlauf, beim Einsatz klassischer Prozessmodelle, um einiges höher liegen, als beim Einsatz von *XP*, für das die Kostenkurve im Verlauf der Zeit relativ flach ist.



**Abbildung 1: Änderungskosten[[12]](#footnote-12)**

Die Begründung für den Verlauf der Kostenkurven ist exemplarisch in Abbildung 2 zu sehen. Dargestellt wird die Notwendigkeit der Planung im Verlauf des Projektes. Im Gegensatz zu anderen Prozessmodellen wird beim *XP* während des gesamten Projektverlaufs immer wieder neu geplant, wodurch Änderungen auch zu einem späten Zeitpunkt des Projektes berücksichtig werden können, ohne das die Kosten besonders ansteigen.



**Abbildung 2: Notwendigkeit der Planung[[13]](#footnote-13)**

# Werte

*XP* definiert vier abstrakte Werte, welche die Grundlage der Methode bilden. Für den erfolgreichen Einsatz innerhalb eines Softwareprojektes ist es von entscheidender Bedeutung, dass die Teilnehmer diese verinnerlichen und auch umsetzen. Es sind die folgenden vier Werte definiert:[[14]](#footnote-14)

* Kommunikation
* Einfachheit
* Feedback
* Mut

Darüber hinaus wird im Schrifttum häufig auch „Respekt“ als expliziter Wert aufgeführt.[[15]](#footnote-15) Dieser sollte jedoch in jedem Projekt einen essenziellen Bestandteil darstellen. Daher ist der Wert hier nicht als spezifischer Punkt für ein *XP-*Projekt aufgeführt. In den folgenden Abschnitten dieses Kapitels werden die vier zuvor genannten Werte des *Extrem Programming* näher betrachtet.

## Kommunikation

Viele Projekte scheitern an einer fehlenden oder schlechten Kommunikation der Beteiligten. Werden Fragestellungen oder Probleme nicht rechtzeitig offen und ehrlich angesprochen, kann dies zu Komplikationen im späteren Projektverlauf führen. Dies betrifft nicht nur die Kommunikation innerhalb des Teams, sondern auch die mit dem Kunden. Dabei ist fehlende Kommunikation keine zufällige Erscheinung, sondern ein teilweise antrainiertes Verhalten. Teilt ein Programmierer seinem Manager oder dem Kunden mit, dass es zu Problemen gekommen ist, hat dieser meistens keine positive Resonanz zu erwarten. Um diesem zu entgehen, werden Probleme oftmals verschwiegen. Beim *XP* soll dieser fehlenden Kommunikation durch verschiedene Instrumente entgegengewirkt werden. Trotz dessen kann auch während eines *XP*-Projektes die Kommunikation zeitweilig gestört sein. In solchen Fällen ist es die Aufgabe des *Coachs* zu intervenieren. Die offene und ehrliche Kommunikation stellt das Fundament des *XP* dar und verlangt den Beteiligten ein hohes Maß an sozialer Kompetenz ab.[[16]](#footnote-16)

## Einfachheit

Beim *XP* wird stets die einfachste Lösung für ein Problem angestrebt. Die Programmierer dürfen sich nicht mit Dingen beschäftigen, die sie vielleicht nächste Woche implementieren müssen. Es soll nur das implementiert werden, was den aktuellen Anforderungen entspricht. Dabei wird wie folgt vorgegangen: Heute lieber etwas Einfaches tun, und morgen etwas mehr zahlen, falls etwas geändert werden muss, anstatt heute etwas Kompliziertes tun, das vielleicht niemals eingesetzt wird. Durch einfache Lösungen wird zudem die Kommunikation im Team gefördert, da andere Programmierer den Code schneller verstehen und somit leichter Anmerkungen machen können.[[17]](#footnote-17)

## Feedback

Durch das Schreiben von Tests für alle logischen Komponenten des Systems erhalten die Programmierer fortwährend ein Feedback über die Qualität ihrer Arbeit. Durch die kontinuierliche Kommunikation mit dem Kunden, sowie dessen Akzeptanztests, geben sich die beiden Parteien Kunde und Entwickler gegenseitig Rückmeldungen. Das schnelle Feedback und die Tatsache, dass bereits zu einem frühen Zeitpunkt ein lauffähiger Prototyp der Software entsteht, führen dazu, dass für alle Beteiligten der Zielerreichungsgrad des Projektes zu jedem Zeitpunkt transparent ist.[[18]](#footnote-18)

## Mut

Ein bedeutender Bestandteil des *XP* ist der Mut. Treten kurz vor Ablauf einer Iteration negative Test-Resultate auf oder es wird komplizierter Code entdeckt, gehört Mut dazu, ein *Refactoring*[[19]](#footnote-19) zu einem solchen Zeitpunkt am entsprechenden Code vorzunehmen. In Konsequenz können durch diesen Mehraufwand die Ziele der Iteration möglicherweise nicht eingehalten werden. Dies muss der Programmierer seinen Kollegen, dem Manager und schließlich auch dem Kunden kommunizieren. Aufgrund dieser Gegebenheit entscheiden sich viele Programmierer den Code dennoch auszuliefern, worunter die Qualität und Wartbarkeit der Software maßgeblich leidet. Es ist ein ineinandergreifen der Werte und die Schaffung eines entsprechenden Umfeldes, welche es dem Programmierer erlaubt solch mutige Schritte zu gehen. Die Qualität einer Software ist ein Faktor, welcher nie zur Diskussion steht und daher stets zugunsten dieser gehandelt werden sollte.[[20]](#footnote-20)

# Prinzipien

Die in Kapitel vier erläuterten Werte (*Kommunikation, Einfachheit, Feedback und Mut*) beschreiben als eine Abstraktion den Stil des *Extrem Programming*. Um diese in der Praxis umzusetzen, sind aus ihnen konkrete Prinzipien abgeleitet. Dabei wird zwischen primären und sekundären Prinzipien unterschieden, auf welche im Folgenden näher eingegangen wird.[[21]](#footnote-21)

## Primäre Prinzipien

* *Unmittelbares Feedback*

Der Zeitraum zwischen einer Aktion und dem Feedback ist von entscheidender Bedeutung. Nach Erkenntnissen der Lernpsychologie hängt ein erfolgreiches Lernen unmittelbar von der Zeit zwischen einer Aktion und ihrer Rückmeldung ab. Es ist demnach effizienter zu lernen, wenn diese Zeitspanne relativ kurz ist.[[22]](#footnote-22)

* *Einfachheit anstreben*

In *XP* werden Programmierer dazu angehalten nur das aktuelle Problem zu betrachten. Dies widerspricht dem herkömmlichen Programmierstil, bei dem versucht wird, wiederverwendbare Softwaremodule zu erstellen. Es wird darauf vertraut, dass bei Bedarf die benötigte Funktionalität zu einem später Zeitpunkt hinzugefügt werden kann.[[23]](#footnote-23)

* *Inkrementelle Veränderung*

Probleme sollen durch viele kleine, wirkungsvolle Veränderungen gelöst werden und nicht durch eine große.[[24]](#footnote-24)

* *Veränderungen wollen*

Das Team sollte Veränderungen offen gegenüberstehen und keine Angst haben das bereits Erstellte aufzugeben, um einen anderen Weg einzuschlagen.[[25]](#footnote-25)

* *Qualitätsarbeit*

Niemand arbeitet gerne an einem schlechten Produkt. Daher ist es wichtig einen hohen Qualitätsmaßstab einzuhalten, sodass alle Beteiligten motiviert bleiben.[[26]](#footnote-26)

## Sekundäre Prinzipien

Neben den primären existieren zusätzlich sekundäre Prinzipen, welche bei der Umsetzung der Werte behilflich sein sollen.[[27]](#footnote-27)

* *Lernen lehren*

Es soll von doktrinären Aussagen und Vorgaben abgesehen werden. Stattdessen sollen Strategien gelehrt werden, wie bestimmte Ziele oder Anforderungen zu erreichen sind. Das Lernen und der Wille sich weiterzuentwickeln sind entscheidende Aspekte des *XP*.[[28]](#footnote-28)

* *Kleine Anfangsinvestitionen*

Knappe Ressourcen führen dazu, dass sich die Programmierer und der Auftraggeber auf das Wesentliche konzentrieren. Werden zu einem frühen Zeitpunkt viele Ressourcen eingesetzt, verleitet dies zu Tätigkeiten, welche im Zweifel nicht zielführend sind.[[29]](#footnote-29)

* *Spielen, um zu gewinnen*

Softwareentwicklungsprojekte sollen „gespielt“ werden, um zu gewinnen. Stattdessen ist es oft das Bestreben vieler Beteiligter sich selbst nichts zu Schulden kommen zu lassen und Verantwortung abzugeben. Die innere Einstellung und der Mut zur offenen Kommunikation sind jedoch entscheidend für den Projekterfolg. Es ist wichtig, dass sich jeder Einzelne voll einsetzt und nicht lediglich *Dienst nach Vorschrift* absolviert.[[30]](#footnote-30)

* *Gezielte Experimente*

Entscheidungen, die gefällt und nicht überprüft werden, bergen das Risiko falsch zu sein. Demnach müssen Entscheidung durch Experimente abgestützt werden, um das Risiko möglichst minimal zu halten.[[31]](#footnote-31)

* *Offene, ehrliche Kommunikation*

Probleme und Anregungen müssen von allen Beteiligten offenen kommuniziert werden. Dabei gilt es ein Umfeld zu kreieren, welches genau dies zulässt. Es gilt ein Gefühl von Sicherheit und Verständnis zu schaffen. Ein Programmierer muss seine Bedenken gegenüber Kollegen und dem Management äußern können, ohne negative Resonanz fürchten zu müssen.[[32]](#footnote-32)

* *Die Instinkte der Mitarbeiter nutzen, nicht dagegen arbeiten*

Es soll versucht werden die Ideen der Mitarbeiter für ein Projekt zu nutzen. Unterschiedliche Ansichten sind im *XP* nicht nur geduldetet, sondern sogar erwünscht. Dabei muss jedoch darauf geachtet werden, dass das aktuelle Problem betrachtet wird und nicht Probleme, die in der Zukunft auftreten könnten.[[33]](#footnote-33)

* *Verantwortung übernehmen*

Den Mitarbeitern soll es ermöglicht werden Eigenverantwortung zu übernehmen. Anhaltende Vorschriften darüber was und wie etwas getan werden muss, schwächen auf Dauer das Engagement des Teams oder des Einzelnen.[[34]](#footnote-34)

* *An örtliche Gegebenheiten anpassen*

*XP* muss immer an das jeweilige Projekt angepasst werden. Ein Vorgehen welches bei einem Projekt funktioniert hat, muss nicht zwangsläufig auch bei einem anderen Projekt funktionieren.[[35]](#footnote-35)

* *Mit leichtem Gepäck reisen*

Überflüssiger Ballast soll im Projekt vermieden werden, um schnell und flexibel auf die Gegebenheiten reagieren zu können.[[36]](#footnote-36)

* *Ehrliches Messen*

Es soll ehrlich gemessen und geschätzt werden. Eine ungefähre Angabe ist dabei besser, als eine genaue Angabe, falls es nicht möglich ist etwas präzise zu schätzen.[[37]](#footnote-37)

# Techniken

*Extreme Programming* macht sich nebst den Werten und Prinzipien noch Techniken zunutze. Diese Techniken basieren auf einer Vielzahl der zuvor definierten Werte und Prinzipien. Der Idee von *Extreme Programming* folgend, werden die Techniken klassifiziert und im Folgenden dem Anwendertyp zugeordnet..[[38]](#footnote-38) Bei diesen handelt es sich um das Management, das Entwicklerteam sowie den Kunden.

## Management-Techniken

Die Management-Techniken sollen eine umfangreiche Kommunikation zwischen dem Projektteam und dem Kunden gewährleisten. Dabei wird der Kunde aktiv in das Projekt eingebunden, um einer möglichen Fehlentwicklung entgegenzuwirken. Im Folgenden werden die Management-Techniken im Einzelnen vorgestellt.

### Kunde vor Ort (On-Site Cutomer)

Oftmals treten neue, nicht vorhersehbare Probleme und Fragen während des Entwicklungsprozesses auf. Manche von diesen können entscheidend für den Verlauf des Projektes sein. Bei *Extreme Programming* wird, anders als üblich, der Kunde direkt beim Auftreten des Problems kontaktiert. Dies hat den entscheidenden Vorteil, dass eine sofortige Korrektur möglich ist und der Fortschritt des (Teil-)Projektes nicht abhängig von einem wöchentlichen Meeting mit dem Kunden ist. Fragen und Probleme können umgehend geklärt werden.[[39]](#footnote-39)

Dazu muss ein Ansprechpartner für die Entwickler, der entweder der Kunde selber ist oder diesen zumindest vertritt, jederzeit erreichbar sein. Dieser sollte den Kunden in allen Aspekten repräsentieren (Vorstellungen, Ziele, Bedingungen etc.). Er muss folglich maximal in das Projekt integriert sein.

In der Praxis ist dieses Vorgehen i.d.R. schwer zu realisieren, da Kunden neben der Projektbetreuung andere Aufgaben und Verpflichtungen haben. Als alternative, aber ähnlich effektive Technik kann der Kunde ein „rotes Telefon“ stets mit sich tragen. Das Management oder das Entwickler-Team ist so in der Lage den Kunden trotz seiner physischen Abwesenheit umgehend zu kontaktieren. Dabei sind Aufstellungen von Regeln hilfreich, z. B. dass der Kunde innerhalb von einer Stunde zurückrufen muss und anschließend ein bis zwei Tage Bedenkzeit für eine qualifizierte Antwort hat.[[40]](#footnote-40)

### Planungsspiel (Planning Game)

Das Planungsspiel zielt auf die Kommunikation zwischen zwei grundlegend unterschiedlich beschaffenen Parteien: Die Geschäftsseite (welche den Kunden und das Management mit ökonomischem Fachwissen darstellen) und die Entwickler (welche die Programmierer mit technischem Fachwissen darstellen). Die geschäfts- und die technisch orientierten Seiten stehen zumeist im Konflikt zueinander, aufgrund der Diskrepanzen bezüglich ihrer Ziele und Vorstellungen. Die Geschäftsseite versucht den Wert der Software zu maximieren, bei geringstem Aufwand und minimalen Kosten (unter eventuell unrealistischen Umständen). Die technische Seite versucht eine Aufwandschätzung für sich mit „Polster“ zu erstellen, um bei auftretenden Problemen nicht sofort in Zeitverzug zu geraten. Die Entwickler werden aufgrund ihres praktischen Wissens und ihrer praktischen Erfahrung zum Planungsspiel hinzugezogen. So müssen sie Aufwandsschätzungen zur Umsetzung bestimmter Funktionen abgeben oder strategische Entscheidungen in Bezug auf technische Konsequenzen prüfen (z. B. die Wahl einer Programmierplattform oder Architektur). Die Programmierer müssen außerdem das Vorgehen der Entwicklung bestimmen: Es müssen kritische oder schwer planbare Implementierungen vor Anderen durchgeführt werden. Zuletzt ist für das Entwicklerteam die Zeitplanung von Bedeutung. Da sich *Extreme Programming* an Releasezyklen orientiert, sollte geklärt werden, welche Funktionen für welche Releases angestrebt werden. Neben dem Vorteil, dass alle Beteiligten über die genaue Zielvorgabe informiert sind, kann der Fortschritt überwacht werden – ähnlich wie ein Projektablaufplan, der allerdings keine Releasezyklen aufweist, sondern Meilensteine.[[41]](#footnote-41)

*Extreme Programming* basiert auf einem iterativen und inkrementellen Ablauf: Vor jeder Iteration werden die Aufgaben und Inhalte für die folgende Iteration besprochen. Das gemeinsame Ziel ist es zunächst eine lauffähige Version der Software zu erstellen, einen sogenannten Prototypen.

Das *Planungsspiel* wird mithilfe der sogenannten *Storycards* durchgeführt, dies ist in Abbildung 3 veranschaulicht. Zwei Spielgruppen nehmen daran teil: Die Geschäftsseite, die alle Beteiligten beinhaltet, die über das Projekt entscheiden, es steuern und kontrollieren. Als auch die Entwicklerseite, die alle Teilnehmer beinhaltet, die für die Umsetzung von Funktionen für die Software zuständig sind.

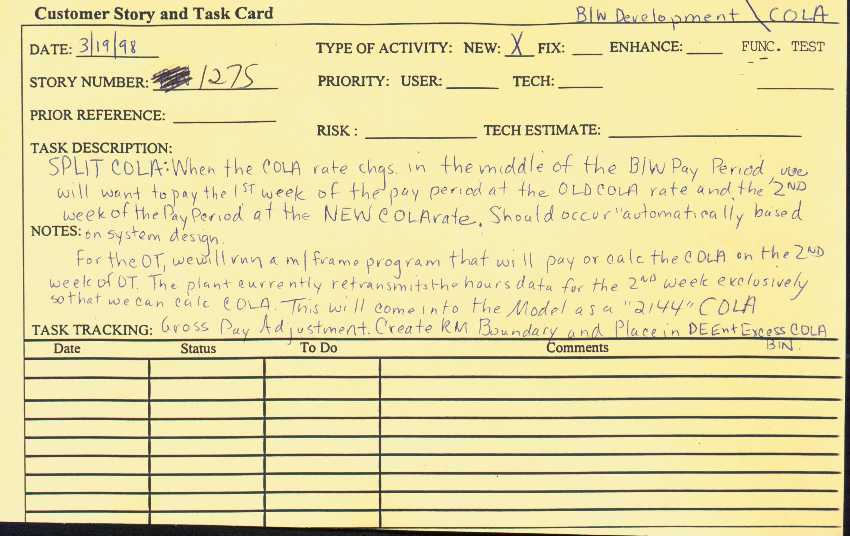


Abbildung 3: Storycard aus dem C3 Projekt von Kent Beck[[42]](#footnote-42)

Die Geschäftsseite muss über den Funktionsumfang der gesamten Software und der einzelnen Releases entscheiden. Aus letzterem ergibt sich die Notwendigkeit einer Einteilung nach Prioritäten: Essentielle Funktionen müssen in den ersten Iterationen umgesetzt werden, während weniger wichtige Funktionen in den nachfolgenden Iterationen umgesetzt werden können. Das *Planungsspiel* basiert auf drei Phasen:

**Phase 1 - Erforschung**

Die *Erforschung* konzentriert sich auf die Analyse des alten Systems, falls vorhanden, sowie die Erstellung neuer Ziele (Funktionen einer Software). Dazu wird diese Phase in drei Teilschritte unterteilt:

1. Erstellen einer Storycard

Die Geschäftsseite erstellt zunächst Storycards, welche alle gewünschten und relevanten Funktionen der Software beschreiben. Diese Leistungsmerkmale werden aufgelistet und ihr Zweck erläutert.

1. Einschätzung der Storycard

Die Entwickler analysieren die von der Geschäftsseite erstellten Storycards und schätzen den Aufwand auf Basis des notierten Funktionsumfangs. Es wird dabei von einer Ideal-Aufwandsschätzung ausgegangen. Die Schätzung der benötigten Zeit erfolgt unter der Prämisse, dass die Entwickler sich ausschließlich mit der jeweiligen Aufgabe auseinandersetzen dürfen. Im *XP*-Modell ist dies als „Ideal Engineering Time“ definiert.

1. Einteilung der Storycards

Oftmals ist der von der Geschäftsseite erstmals festgelegte Funktionsumfang zu hoch für eine Storycard oder sogar eine ganze Iteration. In diesem Teilschritt wird die ursprüngliche Storycard in einzelne aufgeteilt. Die Geschäftsseite und Entwickler arbeiten gemeinsam an der Vergabe von Prioritätsstufen für die einzelnen Aufgaben der aufzuteilenden Storycard.[[43]](#footnote-43)

**Phase 2 - Planung und Verpflichtung**

Diese Phase sieht die zeitliche Planung des Projektes vor. Hierbei werden Termine für die Lieferung von ausgewählten Funktionen (Storycards) festgelegt. Die Entwickler werden dabei verpflichtet zu dem festgelegten und zugestimmten Zeitpunkt die funktionstüchtigen Leistungsmerkmale zu liefern. Auch diese Phase unterteilt sich in vier Teilschritte:

1. Sortierung nach Wert

Nach der Aufteilung der Storycards nimmt die Geschäftsseite eine Kategorisierung der vorhandenen Storycards nach Funktionspriorität vor. Die drei Kategorien lauten:

* essentielle Leistungsfunktionen (ohne diese ist ein Systembetrieb unmöglich)
* Leistungsfunktionen, die wichtig sind, allerdings nicht für den Systembetrieb von elementarer Bedeutung
* Wunschleistungen, die, falls es der Aufwand und die Zeit zulassen, zu implementieren sind

1. Sortierung nach Risiko

Hierbei erfolgt, ähnlich wie zuvor, eine Einteilung der Storycards. In diesem Fall jedoch nach Risiko:

* Konsequenzen für die Leistungsmerkmale sind vollständig und detailliert einschätzbar
* eine moderate Einschätzung ist möglich
* die Einschätzung ist unmöglich

1. Sortierung nach Fortschrittsgeschwindigkeit

Die Entwicklungsseite schätzt hierbei den prognostizierten zeitlichen Fortschritt für die Umsetzung der Storycards unter idealer Entwicklungszeit.

1. Sortierung nach Umfang

Die Geschäftsseite hat die Möglichkeit in dieser Phase eine Storycard für eine Iteration festzulegen, indem sie ein Datum für die erfolgreiche Implementation, unter Berücksichtigung der Aufwandschätzung, auswählt.[[44]](#footnote-44)

**Phase 3 - Kontrolle und Steuerung**

Während des Projektbetriebs, genauer der Durchführung einer Iteration, findet eine ständige Kontrolle statt. Die dadurch gewonnenen Informationen können gegebenenfalls für die Anpassung (also Steuerung) der Iteration und Planung verwendet werden. Die Kontroll- und Steuerungsphase fasst ihrerseits vier Teilschritte zusammen:

1. Iteration

Für jede Iteration legt die Geschäftsseite fest, welche Funktionen (Storycards) in der Iteration umgesetzt werden sollen. Jede Iteration sollte in ein bis zwei Wochen umsetzbar sein. Für die erste Iteration gilt dabei, dass diese essentielle Funktionen beinhalten muss, die einen – wenn auch rudimentären – Systembetrieb ermöglichen.

1. Plankorrektur

Stellt das *XP*-Team fest, dass es für die aktuelle Iteration nicht alle zuvor festgelegten Funktionen umsetzten kann, sollte mit der Geschäftsseite (speziell dem Kunden) eine Auswahl der Funktionen der aktuellen Storycards erfolgen, die für diese Iteration unbedingt erfüllt werden sollten. Die nicht umgesetzten Funktionen müssen in der nächsten Iteration nachgeholt werden.

1. Anpassung der Leistungsmerkmale

Es ist üblich, dass während der Entwicklung, also der Durchführung einer Iteration, der Kunde (bzw. die Geschäftsseite) eine Storycard anpassen oder eine gänzlich Neue einfügen möchte. Für beide Optionen erfolgt durch die Entwickler eine Aufwandsschätzung, Planung und anschließend die Aufnahme der Storycard in eine Iteration.

1. Neueinschätzung

Häufen sich die Verzögerungen oder treten vermehrt ungeahnte Probleme auf, kann die Geschäftsseite die Aufwandschätzung für die ausstehenden Aufgaben neu initiieren. Dazu werden sämtliche zuvor beschriebenen Planungs- und Verpflichtungsphasen erneut für die ausstehenden Aufgaben durchgeführt.[[45]](#footnote-45)

### Kurze Releasezyklen (Short Releases)

Der Umfang des Funktionszuwachses sollte pro Release möglichst klein gehalten werden. Kurze Releasezyklen haben viele Vorteile: Der Kunde bemerkt den praktischen Fortschritt und kann bereits während der Entwicklung Anregungen zu der Software geben. Die Entwickler können daraufhin vorzeitig Änderungswünsche des Kunden übernehmen und im nächsten Release integrieren. Somit werden Fehlerkorrekturen bereits während der eigentlichen Entwicklung vorgenommen, sodass nicht eine finale Version ausgeliefert wird, die möglicherweise komplett überarbeitet werden muss. Dennoch sollten die kurzen Releasezyklen zu vollständig umgesetzten Geschäftsanforderungen führen. Ein Release muss stets aus vollständig implementierten Funktionen bestehen.[[46]](#footnote-46) Üblicherweise werden Releases im Zeithorizont von ein bis zwei Monaten geplant. Der Entwickler hat so u.a. die Möglichkeit auf sich schnell ändernde Anforderungen des Kunden zu reagieren.[[47]](#footnote-47)

## Team-Techniken

Die Team-Techniken dienen der Optimierung des gemeinsamen Arbeitens an Projekten. Menschen neigen zu individuellen Mustern, die bei einer Teamarbeit jedoch teils den reibungslosen Ablauf stören können. Um das gemeinsame Arbeiten an Aufgaben zu erleichtern, schlägt Kent Beck spezielle Techniken vor.

### Metapher (Metaphor)

Die Entwicklung von Funktionen und Programminhalten orientiert sich stets an einer gemeinsamen Metapher. Diese wurde zuvor von allen Teilnehmern abgestimmt und festgelegt. Die Metapher illustriert die vollumfängliche Funktionsweise der Software. Die Metapher beschreibt im Projekt verwendete Bestandteile. So muss beispielsweise dargelegt werden, dass in einem Softwareprojekt einer Investmentbank ein „Arbeitsplatz“ stets mit einem Computer ausgestattet ist, also der Arbeitsplatz ein Computer ist. Weiterführend muss klar sein, dass beispielsweise eine „Bonitätsprüfung“ stets definierte externe Quellen (z.B. Schnittstellen) einbezieht.[[48]](#footnote-48) Die Metapher wird nicht nur für fachspezifische Ausdrücke, sondern auch für Begriffe des allgemeinen Sprachegebrauchs verwendet. Es muss sichergestellt werden, dass die Teilnehmer am Projekt kein abweichendes Verständnis für einen Begriff haben.[[49]](#footnote-49)

### Gemeinsame Verantwortung (Collective Ownership)

Jeder Entwickler ist in der Lage sämtliche Teile des Programmcodes zu ändern. Folglich kann jede Funktionsintegration von jedem Entwickler vorgenommen werden.[[50]](#footnote-50) Die Aufgabenzuteilung kann somit unter den Entwicklern wechseln. Bei einem *XP*-Projekt wird jedoch ausdrücklich die „aufgezwängte Verantwortung“ vermieden: Programmierer sollen sich mit dem entwickelten Programmcode identifizieren und ein eigenes Interesse an dessen Fortschritt und Qualität haben. Eine erzwungene Verantwortung führt nicht zu diesem Effekt und ist daher nicht erwünscht.[[51]](#footnote-51)

### Fortlaufende Integration (Continous Integration)

Das gemeinsame Softwareprojekt wird immer dann aktualisiert (eine Integration neu entwickelter Funktionen), wenn eine Programmieraufgabe (*Task*) einer Storycard erfüllt wurde. Dies sollte bestenfalls mehrmals am Tag geschehen. Andernfalls werden umfangreiche Funktionen integriert, die möglicherweise viele Abhängigkeiten aufweisen und einen höheren Aufwand bei der Integration erfordern. Des Weiteren sollen alle Entwickler mit der möglichst aktuellen Version der Software arbeiten, da es sonst zu Konflikten bei einer lang herausgezögerten Integration kommen kann.[[52]](#footnote-52) Die Integration ist erst dann abgeschlossen, wenn alle Tests zu 100 Prozent erfolgreich ausgeführt werden. Ist dies nicht der Fall, muss der Entwickler entweder die Fehler beheben oder – falls sich die Probleme als komplexer herausstellen – den zu integrierenden Programmcode verwerfen und neu erstellen.

### Programmierstandards (Coding Standards)

Die Programmierer verwenden einen einheitlichen Standard bzw. Stil zur Entwicklung der Software. Nur so kann u. a. die Anwendbarkeit der Technik „Gemeinsame Verantwortung“ sichergestellt werden.[[53]](#footnote-53)

### Nachhaltiges Tempo (Sustainable Pace)

Die Entwickler müssen effizient arbeiten. Das bedeutet nicht, dass die Entwicklung so schnell wie möglich fortschreiten soll. Es muss gleichermaßen auf die Qualität der Ergebnisse geachtet werden. Daher wird an einer 40-Stunden-Woche festgehalten, da dauerhafte Überstunden zu einer Leistungsabnahme führen. Als Ausnahme dürfen Überstunden lediglich für den Zeitraum einer Woche geleistet werden. Im Falle einer Überbelastung nimmt u. a. die Kreativität der Personen ab, die allerdings ausdrücklich bei einem *XP-*Projekt gewünscht ist.[[54]](#footnote-54)

## Entwickler-Techniken

Die Entwickler-Techniken beschreiben konkrete Praktiken, welche den Vorgang der Implementierung der Software unterstützen. Dabei steht vor allem Einfachheit, Qualität und die Kommunikation im Vordergrund.

### Testen (Testing)

Die Programmierer schreiben fortlaufend Tests, die eine erfolgreiche und korrekte Entwicklung der Tasks sicherstellen. Ergebnisse werden nur dann integriert, wenn alle Tests erfolgreich sind. Der Tester schreibt zusätzlich in Kooperation mit dem Kunden Akzeptanztests, um zu überprüfen, ob die definierten Leistungen erfüllt werden. Mit dem Entwicklungsfortschritt nimmt die Zahl der Tests folglich fortwährend zu. *Kent Beck* merkt dazu an, dass paradoxerweise mit wachsender Anzahl von Tests die Fähigkeit eines Softwareprojektes steigt, Änderungen anzunehmen.[[55]](#footnote-55) Das Schreiben von Tests ist zwar wichtig, sollte jedoch nicht für Funktionen in Betracht gezogen werden, die nicht fehlerhaft ablaufen können (z. B. simple Hilfsfunktionen).

### Einfaches Design (Simple Design)

Die Struktur des Softwaresystems sollte zu jedem Zeitpunkt so simpel wie möglich entworfen werden. *Kent Beck* beschreibt vier Kriterien,[[56]](#footnote-56) welche es für jeden Teil des Designs, zu jedem Änderungszeitpunkt, zu prüfen gilt:

* alle Tests müssen erfolgreich sein
* Programmcode enthält keine (versteckten) Redundanzen
* Programmcode ist kommentiert und erfüllt definierte Standards (zum besseren Nachvollzug durch andere Entwickler)
* simpelste Struktur, die möglich ist (wenige und einfache Klassen sowie Methoden)

Ist dies sichergestellt, können andere Entwickler sich leichter und schneller in nicht selbst geschriebenen Programmcode einlesen und an diesem weiterarbeiten. Ein weiterer Aspekt ist die Fehleranfälligkeit von kompliziertem Code, durch nicht beachtete Seiteneffekte oder Abhängigkeiten.

### Refactoring

*Refactoring* wird von *XP*-Entwicklern vorgenommen, sobald es das Design des Programmcodes erfordert. Falls unnötig komplizierter Programmcode im Softwareprojekt gefunden wird, sollte dieser entfernt und durch simpleren ersetzt werden.[[57]](#footnote-57) Im Detail läuft dies wie folgt ab: Bevor ein Entwickler eine neue Funktion implementieren möchte, wird zunächst geprüft, ob der bereits vorhandene Programmcode so verändert werden kann, dass das Implementieren neuer Funktionen einfacher wird. Im zweiten Schritt wird der neue Programmcode implementiert. Im Anschluss ist zu prüfen, ob der nun vorhandene Programmcode neustrukturiert werden kann, sodass er einfacherer wird. *Kent Beck* rechtfertigt diesen Mehraufwand (eine über das Minimum aufzuwendende Leistung) durch die Nachhaltigkeit und zukünftige leichtere Implementation von Programmcode.[[58]](#footnote-58)

### Programmieren in Paaren (Pair Programming)

Die Entwicklung der Software findet in Paaren statt. Zwei Entwickler sitzen nebeneinander vor einem Computer und entwickeln gemeinsam den Programmcode. Abwechselnd ist einer für das strategische und der andere für das operative Vorgehen verantwortlich. Der operative Entwickler führt die tatsächliche technische Implementation durch, also das Schreiben des Programmcodes, während der strategische Entwickler stets prüft, ob:

* Die Implementation des Programmcodes in dieser Weise sinnvoll und nötig ist,
* die allgemeine Programmfunktion benötigt wird oder Teile entfallen können,
* der Ansatz des operativen Entwicklers erfolgsversprechend ist,
* welche Abhängigkeiten zu anderen Programmteilen bestehen, sodass deren Komponententests fehlschlagen könnten und ob die Programmcodestruktur vereinfacht werden kann oder gänzlich neu implementiert werden sollte.[[59]](#footnote-59)

*Refactoring* wird angewendet, um u. a. ein simples Design (nachträglich) zu erreichen. Daher gilt auch hier, dass ein Mehraufwand für ein simpleres Design einer schnellen Implementation mit schlechter strukturiertem Design vorzuziehen ist.[[60]](#footnote-60)

Ein weiteres Ziel des *Pair Programming* ist die Steigerung der Qualität des Programmcodes und das gewonnene Wissen über das Softwareprojekt auf mehrere Köpfe zu verteilen. Daher sollte in Betracht gezogen werden, Programmier-Paare regelmäßig neu zusammenzusetzen.

Letztendlich verdeutlichen die erläuterten Techniken die gegenseitige Abhängigkeit – mit Ausnahme des Testens, welches auch effektiv einzeln angewendet werden kann. Keine dieser Techniken ist neu bzw. durch *Extreme Programming* neu eingeführt. Vielmehr verbindet *XP* die aufgeführten Techniken sinnvoll miteinander, sodass sie sich ergänzen und optimal zusammenwirken. In einem *XP*-Projekt sollten daher stets alle dieser Techniken angewandt werden, da andernfalls keine optimale Wirkung erzielt werden kann. In Abbildung 4 stellt *Kent Beck* die gegenseitigen Abhängigkeiten der einzelnen Techniken dar. Das Konstrukt wirkt auf den ersten Blick komplex, doch stellt es lediglich die zuvor beschriebenen Techniken in einer grafischen Struktur dar

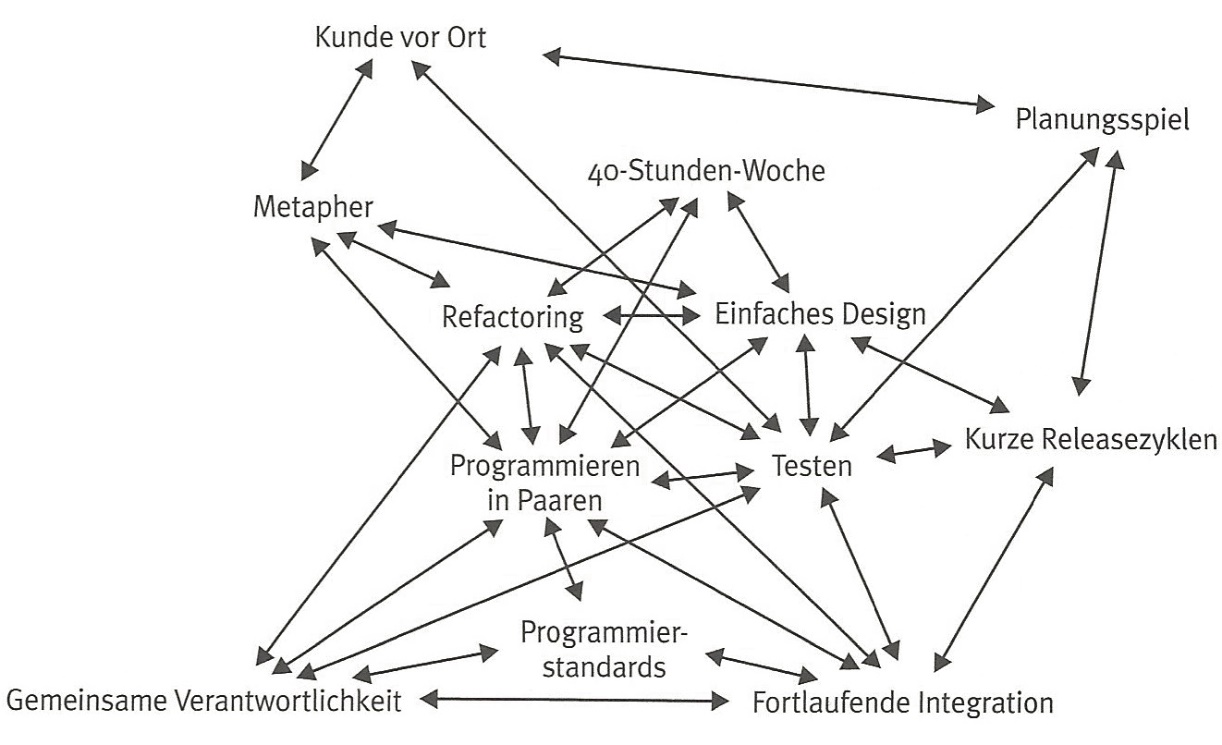


Abbildung 4: Verknüpfung unter den XP-Techniken[[61]](#footnote-61)

# Rollen

*Extreme Programming* erzielt eine umfassende Stakeholder-übergreifende Zusammenarbeit. Ein Softwareprojekt involviert mehrere Interessensgruppen, die sich in ihrem Aufgaben- und Zuständigkeitsbereich, ihrer Expertise und ihrer Kommunikationssprache (bezogen auf fachspezifische Begriffe) unterscheiden. Dennoch verfolgen alle ein übergeordnetes Ziel: das Umsetzen des Projektes. Doch existieren nebst diesem Hauptziel weitere Ziele und Randbedingungen: Der Kunde versucht in der Regel das Projekt in einer möglichst kurzen Zeitspanne und zu einem minimalen Kostenfaktor zu realisieren. Der Dienstleistungsanbieter hingegen versucht die Zufriedenheit des Kunden zu erreichen (z.B. Erfüllung aller Soll- und – wenn möglich – Kann-Kriterien). Das Management muss dabei zwischen den Entwicklern und dem Kunden Vorgänge und Absprachen koordinieren. Die Entwickler benötigen Zeit, um Funktionen zu implementieren. Üblicherweise sind diese bereit mehr Zeit aufzuwenden, um die Qualität der Software zu steigern. Dies bedeutet letztlich jedoch Mehrkosten für den Kunden. *Extreme Programming* beinhaltet weitere Gruppen, die für fest definierte Aufgaben zuständig sind. Im Folgenden werden diese erläutert.[[62]](#footnote-62)

Ein Teilnehmer nimmt nach *Extreme Programming* nur eine Rolle ein. Doch auch hier gilt, wenn unter den realen Umständen anderes verlangt wird, kann davon abgewichen werden: Es ist nur wichtig, dass alle Teilnehmer die Rollen einer Person auseinander halten können. Wenn eine Person bspw. sowohl eine Management- als auch eine Programmierer-Rolle hat, muss allen bewusst sein, in welchem Fall die Person von geschäftsrelevanten oder technischen Inhalten spricht. Daher muss in solchen Fällen berücksichtigt werden, dass es zu zusätzlichen Komplikationen kommen kann.[[63]](#footnote-63)

## Programmierer

Die Programmierer nehmen die wichtigste Gruppe ein. Bei *XP* sind sie in nahezu allen Techniken und Vorgängen integriert. Bis auf wenige Ausnahmen sind die Programmierer in allen Vorgängen integriert und nehmen daher an allen relevanten Besprechungen teil. Der Vorteil hierbei ist, dass die Programmierer direkt mit dem Kunden kommunizieren können und somit ohne Umwege die Ziele, Wünsche und Vorstellungen erfahren. Werden diese Aspekte über einen Vermittler an die Entwickler herangetragen kann es zu Verfälschungen, Auslassungen und Erfindungen von Informationen durch die subjektive Interpretation des Vermittlers kommen. Aus diesem Grund nehmen die Programmierer am Planungsspiel teil. Dies ermöglicht eine direkte Auseinandersetzung mit dem Kunden und somit eine klare Kommunikation.[[64]](#footnote-64)

**Anforderungen**

Ein Programmierer, der in einem *XP-*Projekt tätig ist, muss über mehr Fähigkeiten als ein konventioneller Programmierer verfügen. Für die zum Einsatz kommende Technik des *Pair Programming* muss ein Entwickler in der Lage sein mit anderen uneingeschränkt zu kommunizieren. Die Kommunikation steht in *XP*-Projekten ohnehin an oberster Stelle: Ziele, Aufgaben, Fortschritte, Probleme und vieles mehr müssen im Team kommuniziert werden. Des Weiteren muss ein *XP*-Programmierer in der Lage sein, einfach zu denken. Das bedeutet, dass nicht zwangsweise komplexe und elegante Lösungen, die möglicherweise ein besonderes Fachwissen und Erfahrung erfordern im Programmcode umgesetzt werden müssen, sondern vielmehr simple, saubere (im Sinne von Fehlerfreiheit und Verständlichkeit) und zielgerichtete Lösungen.[[65]](#footnote-65)

## Kunde

Anders als bei klassischen Vorgehensmodellen wird der Kunde stark einbezogen und ist ständig verfügbar und vor Ort. Dies hat zur Folge, dass der Kunde weitaus mehr als in anderen Modellen üblich Verantwortung tragen muss. Die Entwickler ziehen bei Unklarheiten den Kunden umgehend zu Rate, der daraufhin eine präzise Antwort formulieren muss. Darüber hinaus übernimmt der Kunde eine Rolle des Management: Er muss Aufgaben (Funktionen einer Software) priorisieren. Damit geht die detaillierte Integration in das Projektmanagement einher, die u. a. eine ständige Anwesenheit und Fortschrittskontrolle erfordert. Der Kunde kann folglich nicht – wie bei anderen Vorgehensweisen möglich – das Softwareprojekt in Auftrag geben und sich erst wieder zum Release-Zeitpunkt damit auseinander setzen.[[66]](#footnote-66)

**Anforderungen**

Neben dem Programmierer stellt der Kunde ebenfalls eine entscheidende Rolle dar. Dieser muss über spezielle Kompetenzen verfügen: Zum einen kommen Techniken wie „Storycards“ zum Einsatz, die vom Kunden präzise formuliert werden müssen und zum anderen ist er stärker als bei anderen Vorgehensmodellen in das Projekt involviert. Dabei hat er jedoch die besondere Rolle, dass er zwar Einfluss auf das Projekt ausüben, es jedoch nicht steuern kann. Der Kunde muss sich mit den Entwicklern direkt in ein Gespräch begeben (regelmäßig wiederholend) um Kompromisse zwischen Idealvorstellungen und den möglichen Umsetzungen zu finden. Zusammenfassend muss der Kunde bereit sein, sich für das Projekt mitverantwortlich zu fühlen und einen erheblichen Beitrag zu leisten. Die Prämisse für ein solches Interesse ist die Notwendigkeit der zu entwickelnden Software: Wenn es sich um ein Softwareprojekt handelt, dessen Ergebnis (die Software) möglicherweise nicht wichtig für das Unternehmen ist oder möglicherweise nicht eingesetzt wird, fehlt dem Kunden i.d.R. das nötige Interesse und damit das für *Extreme Programming* unerlässliche Engagement.[[67]](#footnote-67)

## Tester

Die Tester unterstützen den Kunden bei der Definierung von Funktionstests, die zur Überprüfung der geforderten Funktionen der Software dienen. Für jeden Releasezyklus werden stets neue Tests hinzugefügt um geänderte oder neu implementierte Funktionen zu prüfen. Einmal erstellte Tests werden in der Regel nicht wieder entfernt. Es sei denn, die zu testende Funktion ändert sich in einem Folgerelease radikal. Dadurch werden Regressionstests durchgeführt.[[68]](#footnote-68)

**Anforderungen**

Der Tester muss in der Lage sein, zusammen mit dem Kunden die Funktionstests zu definieren. Der *Tester* ist folglich für die Erstellung (gemeinsam mit dem Kunden) und die Ausführung der Tests zuständig. Die auftretenden Probleme müssen mit dem Kunden und den Entwicklern kommuniziert werden.[[69]](#footnote-69)

## Tracker

Der *Tracker* misst sämtliche Daten rund um das Projekt. Dazu zählen die Geschwindigkeit des Projektfortschrittes, das Fehlschlagen von Tests sowie die Häufigkeit der Kontaktaufnahme zum Kunden. All diese Daten werden dazu verwendet, um Probleme im Projekt frühzeitig zu erkennen und um Prognosen für die Iterationen anzufertigen.[[70]](#footnote-70)

## Coach

Der *Coach* ist vor allem bei der erstmaligen Durchführung eines Projektes nach dem *Extreme Programming* Modell wichtig: Er ist mit allen Aspekten und Abläufen von *Extreme Programming* vertraut und leitet die Teilnehmer so an, dass sie ihre entsprechende Rolle einnehmen und nach den genannten *Extrem Programming* Werten, Prinzipien und Techniken handeln. Bei bereits etablierten *XP* Teilnehmern prüft der *Coach* ausschließlich, dass alle Abläufe ohne Konflikte und Probleme funktionieren.[[71]](#footnote-71)

Der *Coach* ist – äquivalent zu einem Coach einer Sportmannschaft – dafür zuständig, die Stärken und Schwächen der Teammitglieder zu identifizieren und daraus ein optimales Einsatzgebiet zu finden bzw. zu schaffen. Nach der erstmaligen Identifikation können Teammitglieder in bestimmten Funktionen und Aufgabenbereichen gefördert werden, um sie zu Experten zu machen. Neben der Spezifizierungstaktik können die Rollen auch getauscht werden, um z. B. die Kreativität zu fördern, indem man sich von der Gewohnheit an das alltägliche Vorgehen loslöst. Anders als bei Sportarten, formt der *Coach* jedoch nicht die Teilnehmer um sie an das „Spielmodell“ anzupassen: Bei *XP* wird das Ablaufmodell an die Teilnehmer angepasst. *XP* ist nicht als starres Vorgehensmodell zu sehen, es handelt sich vielmehr um Vorschläge von *Kent Beck*. Funktioniert ein Prinzip, eine Technik oder ein Vorgehen nicht optimal, kann es angepasst, ersetzt oder auch entfernt werden. Die Hauptsache ist, dass der erfolgreiche Projektfortschritt gesichert wird.[[72]](#footnote-72)

**Anforderungen**

Der *Coach* muss, anders als die übrigen Teammitglieder, ein fundiertes Fachwissen über *Extreme Programming* verfügen. Ihm muss bewusst sein, welche Vorgänge von *XP* möglicherweise zu Komplikationen in einem speziellen Projekttyp führen und wie diese durch alternative Herangehensweisen gelöst werden können. Neben dem Fachwissen über *Extrem Programming* muss der *Coach* Charakterzüge eines Krisenmanager aufweisen: Er muss in der Lage sein weiterhin effektiv und effizient zu arbeiten, auch wenn sich das Projekt in einem kritischen Zustand befindet. Während andere Teammitglieder in Panik geraten könnten muss der *Coach* Ruhe bewahren und durch probate Mittel einen Ausweg aus der Situation finden und das Projekt zum Ziel führen. Zwar greift der *Coach* in das *XP*-Vorgehen ein, sobald er dieses für nötig hält, doch sollte stets die Eigendynamik des Teams im Vordergrund stehen. Das bedeutet, sofern keine wesentlichen Probleme beim *XP*-Ablauf auftreten, sollte der *Coach* nicht zwingend eingreifen. Mit der zunehmenden Erfahrung eines *XP*-Teams muss der *Coach* seltener und weniger tiefgreifend in die Vorgänge intervenieren.[[73]](#footnote-73)

Neben dem *Coach* muss ein *Terminmanager* für die Einhaltung von gesetzten Terminen, zeitlichen Horizonten und Deadlines verantwortlich sein. Dieser ist folglich in einer eher kontrollierenden Position und prüft, dass die Iterationen nach Plan verlaufen. Unter Umständen müssen die Planungen angepasst oder es muss in den Ablauf von *XP* eingegriffen werden, sodass schneller zeitliche Ziele erreicht werden. Letzteres ist allerdings nicht im Sinne von *Extreme Programming* und sollte daher selten praktiziert werden. Unter Zeitdruck bzw. radikales Einhalten von Zeiten kommt es in der Regel zu Qualitätsverlust oder Reduzierung des ursprünglich geplanten Funktionsumfangs.[[74]](#footnote-74)

## Consultants

*Consultants* sind externe Mitarbeiter, die zeitweise in das Projekt integriert werden. Diese erledigen entweder eine spezielle Aufgabe direkt, welche besonderes Fachwissen erfordern oder aber geben jenes an die Mitarbeiter weiter.[[75]](#footnote-75)

# Ablauf eines XP-Projektes

Das typische *XP*-Projekt basiert auf einer kurzen Vorbereitungsphase, der Entwicklung der Software – also die Produktionsphase – und anschließende Stilllegung des Projektes.

Für die zeitliche Planung sieht *Kent Beck* bestimmte Zeithorizonte vor. Die Abbildung 5 zeigt den typischen iterativen Ablaufplan eines *XP*-Projektes. Jeder Teilschritt kann umgehend wiederholt werden. Aus diesem Grund handelt es sich bei *XP* um ein stark dynamisches Ablaufmodell:

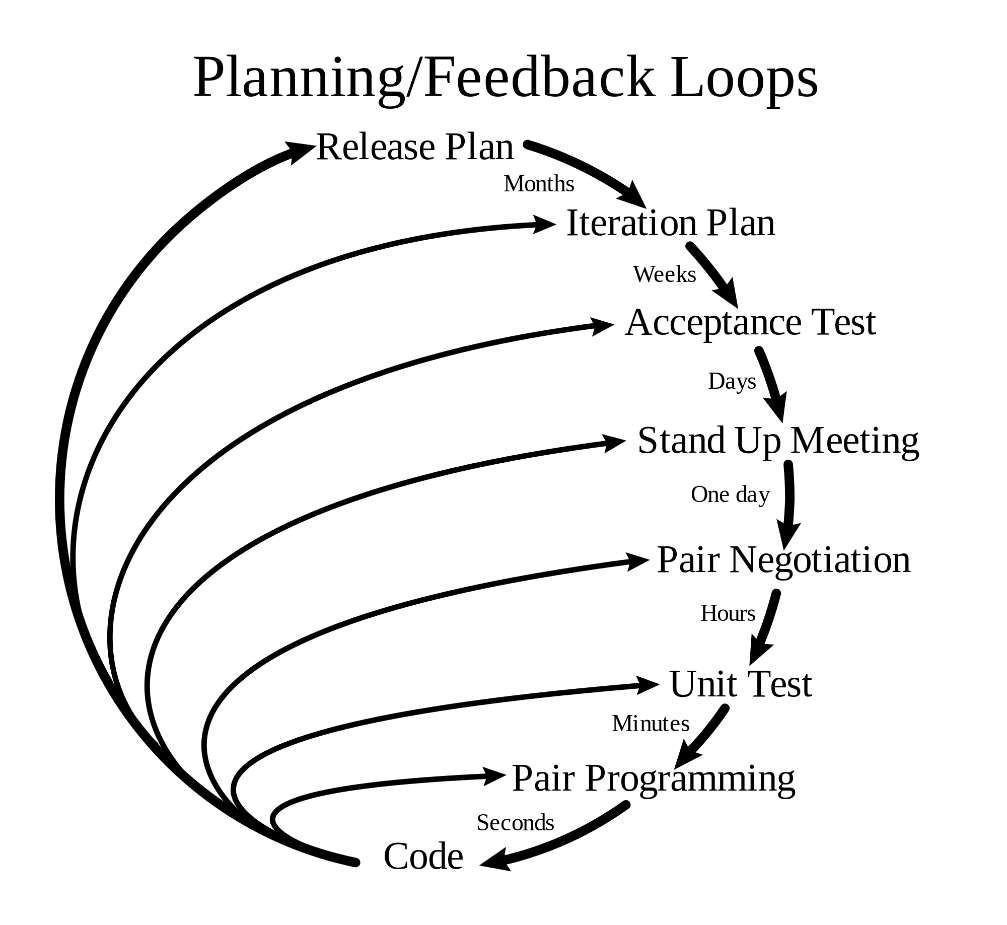


Abbildung 5: Iterativer Ablauf von Extreme Programming mit jeweiligem Zeithorizont[[76]](#footnote-76)

Für den *Release Plan* (stellt die Vorbereitungsphase dar) sollten zwei bis sechs Monate eingeplant werden. Ist diese Phase kürzer, leidet darunter die geschäftsrelevante Planung. Die Iterationen (Funktionsimplementierung durch die Entwickler) sollten innerhalb von ein bis zwei Wochen erledigt werden. Der *Tester* erstellt und führt die Funktionstests mit dem Kunden aus. Für diese sollten ein bis drei Tage eingeplant werden. Die „spontanen“ Meetings sollten innerhalb eines Tages durchgeführt werden. Über die Zusammensetzung neuer Programmier-Paare sollte in wenigen Stunden entschieden werden. Das Entwickeln von Komponententests erfolgt fortwährend innerhalb von Minuten. Die Absprachen zwischen den Programmierer-Paaren finden während der Entwicklung statt.

## Vorbereitung

*Extreme Programming* setzt eine Vorbereitungsphase voraus, die, je nach Vertrautheit der Teammitglieder, im Umfang variiert. Dies bewegt sich in der Regel zwischen wenigen Wochen bis hin zu ein paar Monaten. Der Kunde und oftmals auch das Projektmanagement streben einen möglichst zeitnahen Übergang zur Produktionsphase an. Aus ökonomischer Sicht betrachtet, bedeutet die Vorbereitungsphase einen Aufwand dem kein direkter Nutzen gegenübersteht (also Geldausgaben ohne dabei Geldeinnahmen zu haben). Dennoch ist die Vorbereitung für ein *XP*-Projekt essentiell: Werden vorab Strukturen, Ablaufpläne, Ziele, Kriterien und mögliche Probleme identifiziert und analysiert, bedeutet dies eine bessere Grundlage für die Produktionsphase, welche zwangsweise Probleme und ungeahnte Konflikte zum Vorschein bringt. Die Vorbereitungsphase ist erst dann abgeschlossen, wenn jeder Teilnehmer die Aussage tätigen kann, dass alle Informationen aufbereitet wurden und alle Ziele sowie Rahmenbedingungen eindeutig sind. Dazu zählt auch, dass der Kunde in Zusammenarbeit mit den übrigen Teilnehmern erste Storycards erstellt, von denen er vollständig überzeugt ist und ebenfalls sagen kann, dass er mit einer Software, die genau so entwickelt wird, zufrieden ist und all seinen Anforderungen entspricht. Dieser Prozess kann einige Zeit in Anspruch nehmen, da der Kunde i.d.R. zuerst lernen muss, wie „gute“ Storycards für *XP* geschrieben werden.

Für die Entwickler gilt es in der Vorbereitungsphase alle zur Verfügung stehenden Technologien einzubeziehen und sämtliche relevante Lösungsalternativen für die Problemstellung des Kunden prototypenhaft auszuarbeiten. Dazu kann ein Programmierer entweder selber mehrere Lösungsalternativen entwickeln, oder zwei oder mehr Programmierer-Teams erhalten die gleiche Aufgabe und entwickeln unabhängig voneinander individuelle Lösungsalternativen.[[77]](#footnote-77)

## Planung

Die Planungsphase dient dem Abschätzen und Erstellen von zeitlichen Abläufen. In der initialen Planung stimmen Kunde und Programmierer sich auf den Funktionsumfang mit der höchsten Priorität – in der Regel umfasst dies lediglich wenige Funktionen – und den dafür gesetzten Zeithorizont ab. Dies wird durch das Anwenden des *Planungsspiels* erreicht. Für die Koordination des zeitlichen Ablaufs bietet sich ein Projektablaufplan an. Die Erstellung des ersten Plans nimmt laut *Kent Beck* üblicherweise zwei bis sechs Monate in Anspruch.[[78]](#footnote-78)

## Iteration

Die Entwicklung der Software findet bei *XP* durch Iterationen statt. Die erste Iteration ist von besonderer Bedeutung, da hier grundlegende Architekturziele verfolgt werden: Die Storycards dieser Iteration sollten die gesamte Softwarestruktur abbilden, sodass die Entwickler mit der ersten Iteration bereits die Basis für die Software erstellen können. Dabei müssen noch nicht alle Funktionen implementiert werden, diese werden durch spätere Iterationen hinzugefügt, doch muss bedacht werden, dass der Grundstein durch die Architekturerstellung dafür zunächst vorhanden sein muss. Für die folgenden Iterationen macht es Sinn, die Funktionen mit der höchsten Priorität als erstes umzusetzen. Der gesamte Ablauf wird dabei vom *Tracker* überwacht. Verläuft das Projekt nicht wie geplant, greift der *Coach* ein und kann im Zweifel Prozesse anpassen oder Pläne nachkorrigieren lassen. Bevor die Iteration abgeschlossen ist, muss der Kunde die Funktionstests ausführen. Wenn alle Tests erfolgreich sind, kann die Iteration abgeschlossen werden und ein neues Release der Software publiziert werden. Die letzte Iteration bedeutet gleichzeitig, dass die Software fertiggestellt ist und vollständig produktiv eingesetzt werden kann.[[79]](#footnote-79)

## Integration der Produktivversion

Nähert sich das Softwareprojekt dem Ende verlagert sich der Aufgabenbereich der Storycards und der Iterationen: Die Entwickler gehen von der Implementation neuer Funktionen in die Phase des *Refactoring* der Software über. In dieser Phase werden die Iterationen zeitlich verkürzt um schnellere Releasezyklen zu erreichen. Ein weiteres Merkmal der Endphase ist, dass die Tests zunehmen, um jegliche Szenarien und Funktionen genauestens zu prüfen.[[80]](#footnote-80)

## Wartung

Die *Wartung* findet einhergehend mit einer neuen Iteration statt: Für jede Iteration werden zwar neue Aufgaben (Tasks) festgelegt, die es zu implementieren gilt, doch gibt der Kunde Feedback zu dem vorherigen Release. Folglich wird auch an bereits implementierten Funktionen gearbeitet. Hierbei kann es sich um Fehlerkorrekturen, Design-Optimierungen – also *Refactoring* – oder die Integration neuer Technologien handeln. Den Entwicklern muss klar sein, dass ein Teil des Aufwands jeder Iteration auf die Wartung entfällt. Ebenso muss der Kunde nachvollziehen, dass es Zeit bedarf sein Feedback zu verarbeiten und umzusetzen. Daher dürfen nicht zu viele neue Storycards für die Folge-Iteration geschrieben werden.[[81]](#footnote-81)

## Auflösung

Das Softwareprojekt kann dann beendet werden, wenn der Kunden keine weiteren Inhalte für Storycards, also Änderungen oder Entwicklung von Funktionen für die Software ausweisen kann. In diesem Fall schließt sich eine Dokumentationserstellung an, die für zukünftige Veränderungen der Software einen leichten Einstieg ermöglichen.

Neben der positiven Auflösung – also dem erfolgreichen Softwareprojekt – gibt es auch Auflösungen die aus Projektfehlschlägen resultieren. Angesichts einer Erfolgsrate von 39% im Jahr 2012 ist es nicht selbstverständlich, dass ein Projekt vollständig umgesetzt wird. In 43% der Fälle wurden nicht alle Anforderungen erfüllt und in 18% schlug das Projekt sogar fehl.[[82]](#footnote-82) Schlägt ein Projekt fehl, sollten die Teammitglieder in einem nachträglichen Meeting eine Problemanalyse durchführen. So kann aus Fehlern gelernt werden und in den nächsten Projekten auf eine größere Erfahrung zurückgegriffen werden.[[83]](#footnote-83)

# Zusammenfassung

*XP* ist nicht als pauschale Lösung für alle Herausforderungen und Probleme in der Softwareentwicklung einzusetzen. Dennoch stellt es einen entscheidenden Schritt für die Akzeptanz und Weiterentwicklung agiler Softwareentwicklungsmodelle dar. *Kent Beck* hat *XP* aus einer realen und praktischen Problemsituation heraus entwickelt. Mit *Extrem Programming* lassen sich schwer kontrollierbare Projekte umsetzen. Eine zentrale Herausforderung sind die sich stetig ändernden Anforderungen des Kunden. In einem klassischen Ablaufmodell führt dies zwangsweise zu Verzögerungen und Konflikten. *XP* dagegen erlaubt einen dynamischen Verlauf, durch die in Teilaufgaben zerlegten Anforderungen, welche in Iterationen bearbeitet werden. In Abgrenzung zu klassischen Modellen kann *XP* nach individuellen Nöten angepasst werden, sodass es sich durch eine hohe Eigendynamik auszeichnet.[[84]](#footnote-84)

*Extrem Programming,* als agile Softwareentwicklungsmethode, legt den Fokus auf die Softwareentwicklung und orientiert sich weniger an starren Ablaufmustern, wie es die klassischen Vorgehensmodelle vorsehen. Der gesamte Projektablauf erfolgt in Iterationen, welche durch vordefinierte Werte, Prinzipien, Techniken und Rollen angeleitet werden.[[85]](#footnote-85) Daher muss ein gewisses Grundverständnis bestehen, um *Extrem Programming* anzuwenden. Dennoch handelt es sich um ein sehr eigendynamisches Modell: Die von *Kent Beck* vorgeschlagenen Bestandteile können nach eigenem Ermessen angepasst, ersetzt oder entfernt werden, sollten diese den Projektablauf stören. *XP* konzentriert sich stärker auf Aspekte des Projektmanagement, als auf die eigentlichen Inhalte der Softwareentwicklung. Die uneingeschränkte Kommunikation unter den Projektteilnehmern steht an oberster Stelle, sodass alle Teilnehmer zu jederzeit über den aktuellen Zielerreichungsgrad, die Aufgaben und die Probleme informiert sind. Dabei geht *Kent Beck* mit *Extrem Programming* noch einen Schritt weiter und integriert auch den Kunden direkt in das Projektgeschehen. Die Anwesenheit des Kunden, oder die eines Vertreters dessen, wird während des gesamten Softwareprojektes vorausgesetzt. *XP* sieht die Priorisierung essenzieller Softwarefunktionen vor, anhand derer frühzeitig ein Prototyp der Anwendung entwickelt werden kann. Mit *XP* wird viel Wert auf eine lauffähige Softwareversion gelegt. Dies wird durch die mehrfachen Komponenten- und Funktionstests erzwungen. Ein wichtiges Ziel ist es, dass der Kunde stets die letzte, aber vollständig funktionierende Softwareversion testen kann. Das Risiko eines kompletten Projektfehlschlags lässt sich so erheblich reduzieren. Die eigentliche Programmierung erfolgt in einem sowohl freien, als auch stark kontrollierten Stil. Beim *Pair Programming* kann Programmcode weitgehend nach Belieben, unter dem Aspekt eines möglichst einfachen Designs, implementiert werden. Dabei kommt *Refactoring* während und nach der Entwicklung zum Einsatz, um ein möglichst simples Design zu erreichen. Im Anschluss wird die Implementation einer Vielzahl von Komponenten- und Funktionstests unterzogen, welche sicherstellen, dass ausschließlich funktionierender Programmcode in eine Releaseversion integriert wird. Der iterative Ablauf erlaubt eine schnelle Integration von Änderungswünschen. Dabei kann es sich um gänzlich neue Anforderungen oder aber Änderungswünsche des Kunden handeln.

*Extrem Programming* kann besser mit menschlichen Fehlern umgehen. Der dynamische und iterative Ablauf erlaubt die nachträgliche Korrektur der Planung, welche beispielsweise auf menschlicher Fehleinschätzung des Aufwands beruht. Der Ablauf eines *XP*-Projektes gerät dadurch nicht ins Stocken, sondern sieht dies vielmehr als Gelegenheit daraus zu lernen.

## Vorteile und Chancen

In einem *XP*-Projekt wird viel Wert auf die Kommunikation zwischen allen Beteiligten gelegt.[[86]](#footnote-86)

Dieser Aspekt resultiert aus der Tatsache, dass unterschiedliche Interessensgruppen an dem Projekt teilnehmen, welche divergierende Ziele, Vorstellungen und Meinungen haben. Im Folgenden werden die Vorteile und Chancen von *XP* für die einzelnen Gruppen genauer betrachtet.

### Vorteile für den Kunden

Der Kunde kann Einfluss auf das Projektgeschehen ausüben, indem er Funktionen der Software priorisiert, sodass ihm unerlässlich erscheinende Funktionen zuerst implementiert werden. Ein weiterer Vorteil für den Kunden ist die regelmäßige Bereitstellung von Releaseversionen. Dem Kunden ist somit der Fortschritt des Projektes zu jeder Zeit transparent. Darüber hinaus lassen sich Fehlentwicklungen verhindern und Änderungswünsche bereits während des Entwicklungsprozesses umsetzen. Anders als bei klassischen Vorgehensmodellen ist der Kunde bei *XP* ständig vor Ort und ein Teil des Projektteams, wodurch er Kenntnis über den Projektverlauf hat. Die Gefahr, dass eine Softwarelösung entwickelt wird, welche nicht den Anforderungen des Kunden entspricht, ist folglich ausgeschlossen. Der Kunde trägt somit einen erheblichen Teil der Verantwortung. Das Ergebnis der Software ist abhängig von seinen Interventionen, sowohl im negativen als auch positiven Sinne. Der Kunde neigt jedoch bei der Bewertung eines regelmäßig aktualisierten Prototypens dazu, den Funktionsumfang zu erweitern. Tritt dies ein, muss nach dem *YAGNI-*Prinzip[[87]](#footnote-87) gehandelt werden, um das Projekt nicht unnötig zu verlängern und um nicht zu stark vom ursprünglichen Projektplan abzuweichen.

### Vorteile für den Entwickler

*Kent Beck* setzt keine besonderen technischen Fähigkeiten für die *XP‑*Entwickler voraus. Dennoch müssen sie mit anderen Teilnehmern kommunizieren können.[[88]](#footnote-88) Der Projektablauf findet reibungsloser statt, wenn die Teilnehmer bereits Erfahrung mit *XP* gesammelt haben. Andernfalls ist der *Coach* für die Weiterbildung und Integration aller Teilnehmer zuständig. Da *Extrem Programming* stark auf Kommunikation setzt, sind eher solche Programmierer erwünscht, welche fähig sind im Team zu kommunizieren. Diese Eigenschaft – die Kommunikationsfähigkeit – wird in *XP* über die des Spezialwissens gesetzt. Die Entwickler tauschen ihr Wissen untereinander aus, was nicht zuletzt durch das *Pair Programming* verdeutlicht wird. Für spezielles Fachwissen können *Consultants* kurzfristig hinzugezogen werden, die entweder ihr Wissen im Team teilen oder spezielle Aufgabe direkt bearbeiten.[[89]](#footnote-89) Das iterative Vorgehen ist für die Entwickler von Vorteil, da sie so den Überblick über das oftmals komplexe Projekt behalten. Sie können viele kleine Erfolge erzielen und sind dadurch motiviert weiter am Projekt zu arbeiten.

### Vorteile für das Management und das Projekt

*Extrem Programming* ermöglicht durch seine Eigendynamik einen reibungsloseren Ablauf als klassische Ablaufmodelle. *XP* ist weniger anfällig gegenüber den typischen Probleme eines Projektes wie Verzögerungen, Konflikten oder nachträgliche Anforderungsänderungen. Dank der Anpassungsfähigkeit (jeder Aspekt von *XP* lässt sich abändern) kann *XP* in vielen Projekten zur Anwendung kommen. Die Priorisierung essenzieller Funktionen zur Erstellung eines lauffähigen Prototyps der Software mindert das Risiko des Fehlschlags eines Projektes, wenngleich noch nicht alle Funktionen implementiert sind. Im Softwareprojekt können darüber hinaus Kosten eingespart werden, da die Planung den tatsächlich aktuellen Anforderungen entspricht. Das Problem klassischer Vorgehensweisen, dass die Planung einem alten Stand entspricht, wird bei *Extrem Programming* umgangen: Nur tatsächlich für das Projekt relevante Aufgaben werden erledigt. *Kent Beck* verzichtet weitgehend auf bürokratische Prozesse, sodass versucht wird diese auf ihr Minimum zu reduziert, um direkt das gewünschtes Ziel zu erreichen. Ein prägnantes Beispiel dafür ist das Planungsspiel, in dem sich der Kunde direkt mit dem Entwickler abstimmt. Die Entwickler haben ebenfalls einen Mehrwert von *XP*, da statt des Schreibens umfangreicher Dokumentationen das (dazugelernte) Wissen zwischen allen Teilnehmern auf verbaler Ebene geteilt wird.

## Risiken

*XP* basiert größtenteils auf Selbstorganisation. Die Teilnehmer müssen daher mit den Aspekten des Ablaufmodells vertraut sein. Dazu zählen die Werte, Prinzipien, Techniken sowie einzelne Abläufe. Für ein Team ohne jegliche Vorkenntnisse muss mit einer längeren Vorbereitungsphase gerechnet werden. Dies bedeutet aus ökonomischer Perspektive, dass Geldausgaben keinen Geldeinnahmen gegenüberstehen. Ein weiteres Problempotenzial ist die Tatsache, dass *Extrem Programming*, wenn Teilnehmer des Projektes einzelne Aspekte dessen nicht akzeptieren, sogar ein Hindernis für den Projektablauf darstellen kann. Vor dem Einsatz sollte folglich geklärt werden, ob sich jeder Teilnehmer auf das Vorgehensmodell einlässt.[[90]](#footnote-90)

*XP* setzt auf die Kommunikation zwischen allen Teilnehmern. Programmierer die ihr Wissen, ihren Fortschritt und ihre Probleme mit anderen Teilnehmern nicht teilen, wirken sich negativ auf den Projektverlauf aus. Da der Aspekt einer ausgeprägten Kommunikation für viele Entwickler neu ist, muss geprüft werden, ob alle Programmierer in der Lage sind, uneingeschränkt zu kommunizieren. Eine weitere Schwierigkeit für Programmierer stellt das *Refactoring* und das Programmieren eines einfachen Designs dar. Programmcode nochmals zu ändern, auch wenn dieser bereits funktioniert, wird von vielen Programmierern vermieden. Doch genau dies ist für *XP* wichtig. Es wird ein einheitlicher, sich an Standards orientierter Programmcode erstrebt, welcher ohne große Einarbeitung von allen Entwicklern nachvollzogen werden kann.

Der Projektleiter – oder *Coach* – muss mit der Ablaufweise von *XP* vertraut sein und akzeptieren, dass er möglichst wenig Einfluss auf das Projekt auszuüben hat. Der Kunde darf über die Prioritäten der einzelnen Funktionen entscheiden. Die Entwickler können mit diesem über die Machbarkeit und Rahmenbedingungen diskutieren. Anders als bei klassischen Modellen gibt es keinen Projektleiter, welcher Aufgaben festlegt und sie anschließend den Entwicklern zuteilt.

Der Kunde muss als solcher akzeptiert werden. Trotz dessen muss dieser soweit wie möglich an *Extrem Programming* herangeführt werden, damit er den Ablauf versteht und seine Rolle erfüllen kann. Die regelmäßigen Releasezyklen sind ein Vorteil, doch kann der Kunde daraufhin das Gefühl bekommen, dass noch genügend Zeit für die Implementation zusätzlicher Funktionen besteht und erweitert daraufhin drastisch den Funktionsumfang. Der Kunde muss verstehen, dass dies im Zweifel nicht möglich ist und sich auf die tatsächlich benötigten Funktionen beschränken.

Eine weitere Einschränkung stellt die Größe des Entwicklerteams und damit oftmals einhergehend die Größe des Projektes dar: Die Koordination innerhalb des Projektes findet oftmals verbal und eher in einem lockeren Stil statt. Auf Dokumentation und starre, terminlich und inhaltlich fixierte Besprechungen wird weitestgehend verzichtet. Diese Art der Kommunikation und Planung funktioniert allerdings nur bis zu einem bestimmten Grad: Wenn zu viele Teilnehmer am Projekt beteiligt sind, nimmt der „Overhead“ für die mündlichen und spontanen Absprachen stark zu. Es wird schwierig, das Wissen ohne eine zentrale Datenbank über das gesamte Team zu verteilen, insbesondere dann, wenn begonnen wird, externe Teilnehmer in das Projekt einzubeziehen. In der Praxis werden daher maximal zehn Personen an einem *XP*-Projekt beteiligt.[[91]](#footnote-91)

*XP* ist nicht für alle Arten von Aufträgen geeignet. Für einen Festpreis-Auftrag ist das Prinzip der Iterationen und der nachträglichen Anpassungen von Anforderungen seitens des Kunden nicht erwünscht. Ein Festpreis-Projekt definiert schließlich, dass eine zu entwickelnde Software einen festgelegten Funktionsumfang und damit auch Preis widerspiegelt.[[92]](#footnote-92)

## Bewertung

In jedem Projekt herrschen divergierende Einflüsse, Probleme und Konfliktpotenziale, aus diesem Grund ist *XP* nicht für jeden Projekttyp geeignet. Dennoch stellt es eine entscheidende Basis für agile Softwareprojektmodelle dar. Die Methode entstand aus einer realen und praktischen Problemsituation heraus und nicht theoretisch an einem Institut. Dank seiner hohen Eigendynamik, den starken Fokus auf den Menschen und die Kommunikation sollte *XP*, sofern es die Rahmenbedingungen zulassen, bei der Auswahl des Vorgehensmodells stets in Erwägung gezogen werden.

## Abgrenzung zu SCRUM

*SCRUM* und *XP* sind agile Methoden und werden deshalb nicht selten verwechselt. Die Unterschiede sind oftmals nur sehr fein, jedoch entscheidend.[[93]](#footnote-93) Beide Methoden zeichnen sich durch die kurzen Entwicklungszeiten aus, die aus mehreren Iterationen bestehen.[[94]](#footnote-94) Um einen Einblick in die Unterschiede der beiden Methoden zu bekommen, werden im Folgenden vier Unterschiede zwischen *SCRUM* und *Extrem Programming* erläutert:[[95]](#footnote-95)

* Bei *SCRUM* wird in sogenannten Iterationen, auch Sprints genannt, gearbeitet, die einen Zeitraum von einer Woche bis zu einem Monat betragen können. In *XP*-Projekten wird zwar auch in Iterationen gearbeitet, diese betragen in der Regel jedoch nur ein bis zwei Wochen.
* Vor einer Iteration werden bei beiden Methoden die zu implementierenden Funktionen festgelegt. Während bei *SCRUM* das Arbeitspaket feststeht und nicht mehr verändert werden kann, ist es bei *Extrem Programming* hingegen möglich Funktionen zu ersetzen. Es muss lediglich darauf geachtet werden, dass sich der Umfang nicht im großen Ausmaß verändert.
* In beiden Methoden werden die zu entwickelnden Funktionen durch den Kunden bestimmt. Während jedoch bei *SCRUM* die zu implementierenden Funktionen durch die Programmierer selbst priorisiert werden, bestimmt beim *XP* der Kunde weitgehend die Reihenfolge dieser.
* Bei *XP* werden Praktiken, wie die testgetriebene Entwicklung, *Pair Programming* oder schlichtes Design vorgeschrieben. Bei *SCRUM* ist dies nicht der Fall.

# Literaturverzeichnis

Ambler, Scott W. (op. 2002): Agile modeling. Effective practices for eXtreme programming and the unified process. New York: John Wiley.

Beck, Kent (2004): Extreme Programming. Die revolutionäre Methode für Softwareentwicklung in kleinen Teams ; [das Manifest]. [Nachdr.]. München [u.a.]: Addison-Wesley (Programmer's choice).

Breitling, Holger (2012): Extreme Programming: XP Revisited. Online verfügbar unter http://jaxenter.de/artikel/Extreme-Programming-XP-Revisited, zuletzt aktualisiert am 23.01.2014.

Dornberger, Rolf; Habegger, Thomas (2004): Extreme Programming. Eine Übersicht und Bewertung. Olten: Fachhochsch. Solothurn Nordwestschweiz (Fachhochschule Solothurn Nordwestschweiz, Hochschule für Wirtschaft : Reihe A, Discussion papers, 2004,05).

Göbel, Albrecht (2006): Extreme Programming (XP). Ausarbeitung zum Referat vom 20.11.2006. TU Berlin, Berlin. Online verfügbar unter http://swt.cs.tu-berlin.de/lehre/sepr/ws0607/referate/Vorgehensmodelle-XP\_Ausarbeitung.pdf, zuletzt geprüft am 25.01.2014.

Hanser, Eckhart (2010): Agile Prozesse: von XP über Scrum bis MAP. Berlin [u.a.]: Springer (eXamen.press).

Jeffries, Ron (1999): Story and Task Cards. Online verfügbar unter http://xprogramming.com/articles/story\_and\_task\_cards/, zuletzt aktualisiert am 23.01.2014.

Salinger, Stephan: Exteme Programming (XP): Die Metapher. Seminar "Ausgewählte Beiträge zum Software Engineering" WS 2004/2005. Freie Universität Berlin.

Schmidt, Tommy: Extreme Programming. Herkunft, Methodik und Einsatz. Studienarbeit. Fachhochschule Brandenburg, Brandenburg. Online verfügbar unter http://www.tommy-schmidt.de/fileadmin/user\_upload/Studienarbeit\_XP\_SchmidtT.pdf, zuletzt geprüft am 25.01.2014.

scrum-kompakt.de (2014): Extreme Programming (XP) – Scrum Kompakt. Online verfügbar unter http://www.scrum-kompakt.de/grundlagen-des-projektmanagements/extreme-programming-xp/, zuletzt aktualisiert am 20.01.2014, zuletzt geprüft am 20.01.2014.

THE STANDISH GROUP (2013): CHAOS MANIFESTO 2013. Think Big, Act Small. THE STANDISH GROUP. Online verfügbar unter http://versionone.com/assets/img/files/ChaosManifesto2013.pdf.

Westphal, Frank (2001): Extreme Programming. Online verfügbar unter http://www.frankwestphal.de/ExtremeProgramming.html, zuletzt aktualisiert am 24.01.2014.

Wikipedia (Hg.) (2014): Extreme programming - Wikipedia, the free encyclopedia. Online verfügbar unter http://en.wikipedia.org/w/index.php?oldid=592527507, zuletzt aktualisiert am 26.01.2014, zuletzt geprüft am 27.01.2014.

1. Vgl. Beck 2004, S. 1 [↑](#footnote-ref-1)
2. Vgl. Beck 2004, S. 2–3 [↑](#footnote-ref-2)
3. Vgl. Beck 2004, S. 7–10 [↑](#footnote-ref-3)
4. Ambler op. 2002 [↑](#footnote-ref-4)
5. Vgl. Beck 2004, S. 9–10 [↑](#footnote-ref-5)
6. Vgl. Beck 2004, S. 13 [↑](#footnote-ref-6)
7. Vgl. Dornberger und Habegger 2004a, S. 24 [↑](#footnote-ref-7)
8. Vgl. Beck 2004, S. 20–21 [↑](#footnote-ref-8)
9. Vgl. Beck 2004, S. 13 [↑](#footnote-ref-9)
10. In Anlehnung an Beck 2004, S. 37–42 [↑](#footnote-ref-10)
11. Vgl. Beck 2004, S. 44–45 [↑](#footnote-ref-11)
12. Dornberger und Habegger 2004, S. 7 [↑](#footnote-ref-12)
13. Dornberger und Habegger 2004, S. 7 [↑](#footnote-ref-13)
14. Vgl. Beck 2004, S. 29 [↑](#footnote-ref-14)
15. Vgl. Dornberger und Habegger 2004, S. 12 [↑](#footnote-ref-15)
16. Vgl. Beck 2004, S. 29–30 [↑](#footnote-ref-16)
17. Vgl. Beck 2004, S. 30–31 [↑](#footnote-ref-17)
18. Vgl. Beck 2004, S. 31–32 [↑](#footnote-ref-18)
19. Refactoring: Strukturverbesserung unter Beibehaltung des Programmverhaltens zur Verbesserung der Lesbarkeit, Verständlichkeit, Wartbarkeit und Erweiterbarkeit [↑](#footnote-ref-19)
20. Vgl. Beck 2004, S. 33–34 [↑](#footnote-ref-20)
21. Vgl. Beck 2004, S. 37 [↑](#footnote-ref-21)
22. Vgl. Beck 2004, S. 37 [↑](#footnote-ref-22)
23. Vgl. Beck 2004, S. 38 [↑](#footnote-ref-23)
24. Vgl. Beck 2004, S. 38 [↑](#footnote-ref-24)
25. Vgl. Dornberger und Habegger 2004a, S. 13 [↑](#footnote-ref-25)
26. Vgl. Beck 2004, S. 38 [↑](#footnote-ref-26)
27. Vgl. Beck 2004, S. 38 [↑](#footnote-ref-27)
28. Vgl. Beck 2004, S. 39 [↑](#footnote-ref-28)
29. Vgl. Beck 2004, S. 39 [↑](#footnote-ref-29)
30. Vgl. Beck 2004, S. 39 [↑](#footnote-ref-30)
31. Vgl. Beck 2004, S. 40 [↑](#footnote-ref-31)
32. Vgl. Beck 2004, S. 40 [↑](#footnote-ref-32)
33. Vgl. Beck 2004, S. 40 [↑](#footnote-ref-33)
34. Vgl. Beck 2004, S. 41 [↑](#footnote-ref-34)
35. Vgl. Beck 2004, S. 41–42 [↑](#footnote-ref-35)
36. Vgl. Beck 2004, S. 42 [↑](#footnote-ref-36)
37. Vgl. Beck 2004, S. 42 [↑](#footnote-ref-37)
38. Vgl. scrum-kompakt.de 2014 [↑](#footnote-ref-38)
39. Vgl. Beck 2004, S. 61–62 [↑](#footnote-ref-39)
40. Vgl. Hanser 2010, S. 30 [↑](#footnote-ref-40)
41. Vgl. Beck 2004, S. 54–56 [↑](#footnote-ref-41)
42. Jeffries 1999 [↑](#footnote-ref-42)
43. Vgl. Beck 2004, S. 89–90 [↑](#footnote-ref-43)
44. Vgl. Beck 2004, S. 90 [↑](#footnote-ref-44)
45. Vgl. Beck 2004, S. 91 [↑](#footnote-ref-45)
46. Vgl. Breitling 2012 [↑](#footnote-ref-46)
47. Vgl. Beck 2004, S. 56 [↑](#footnote-ref-47)
48. Vgl. Beck 2004, S. 56–57 [↑](#footnote-ref-48)
49. Vgl. Salinger, S. 17 [↑](#footnote-ref-49)
50. Vgl. Breitling 2012 [↑](#footnote-ref-50)
51. Vgl. Hanser 2010, S. 44 [↑](#footnote-ref-51)
52. Vgl. Beck 2004, S. 60 [↑](#footnote-ref-52)
53. Vgl. Breitling 2012 [↑](#footnote-ref-53)
54. Vgl. Beck 2004, S. 60–61 [↑](#footnote-ref-54)
55. Vgl. Beck 2004, S. 57 [↑](#footnote-ref-55)
56. Vgl. Beck 2004, S. 57 [↑](#footnote-ref-56)
57. Vgl. Westphal 2001 [↑](#footnote-ref-57)
58. Vgl. Beck 2004, S. 58 [↑](#footnote-ref-58)
59. Vgl. Beck 2004, S. 58–59 [↑](#footnote-ref-59)
60. Vgl. Dornberger und Habegger 2004b, S. 27–28 [↑](#footnote-ref-60)
61. Vgl. Beck 2004, S. 70 [↑](#footnote-ref-61)
62. Vgl. Dornberger und Habegger 2004b, S. 27–28 [↑](#footnote-ref-62)
63. Vgl. Beck 2004, S. 140 [↑](#footnote-ref-63)
64. Vgl. Beck 2004, S. 140–141 [↑](#footnote-ref-64)
65. Vgl. Beck 2004, S. 141–142 [↑](#footnote-ref-65)
66. Vgl. Beck 2004, S. 142–143 [↑](#footnote-ref-66)
67. Vgl. Dornberger und Habegger 2004b, S. 22 [↑](#footnote-ref-67)
68. Vgl. Beck 2004, S. 144 [↑](#footnote-ref-68)
69. Vgl. Beck 2004, S. 144 [↑](#footnote-ref-69)
70. Vgl. Chromatic 2003, S. 63–64 [↑](#footnote-ref-70)
71. Vgl. Beck 2004, S. 145–146 [↑](#footnote-ref-71)
72. Vgl. Beck 2004, S. 145–146 [↑](#footnote-ref-72)
73. Vgl. Chromatic 2003, S. 64 [↑](#footnote-ref-73)
74. Vgl. Beck 2004, S. 144 [↑](#footnote-ref-74)
75. Vgl. Dornberger und Habegger 2004b, S. 23–24 [↑](#footnote-ref-75)
76. Wikipedia 2014 [↑](#footnote-ref-76)
77. Vgl. Beck 2004, S. 131–133 [↑](#footnote-ref-77)
78. Vgl. Beck 2004, S. 133 [↑](#footnote-ref-78)
79. Vgl. Beck 2004, S. 133–134 [↑](#footnote-ref-79)
80. Vgl. Beck 2004, S. 134–135 [↑](#footnote-ref-80)
81. Vgl. Beck 2004, S. 135–136 [↑](#footnote-ref-81)
82. Vgl. THE STANDISH GROUP 2013, S. 5 [↑](#footnote-ref-82)
83. Vgl. Beck 2004, S. 137 [↑](#footnote-ref-83)
84. Vgl. Hanser 2010, S. 13 [↑](#footnote-ref-84)
85. Vgl. Beck 2004, S. XVII [↑](#footnote-ref-85)
86. Vgl. Hanser 2010, S. 14 [↑](#footnote-ref-86)
87. *YAGNI* ist ein Akronym und steht für „You Ain’t Gonna Need It“, was im Deutschen bedeutet „Du wirst es nicht brauchen“. In *Extreme Programming* bedeutet dieses Prinzip, dass zu implementierende Funktionen zunächst einer genauen Analyse und Bewertung unterzogen werden sollen, sodass sichergestellt wird, dass die Funktionen wirklich später benötigt und angewendet werden. [↑](#footnote-ref-87)
88. Vgl. Hanser 2010, S. 141 [↑](#footnote-ref-88)
89. Vgl. Hanser 2010, S. 146–147 [↑](#footnote-ref-89)
90. Vgl. Schmidt, S. 16–17 [↑](#footnote-ref-90)
91. Vgl. Wikipedia 2014 [↑](#footnote-ref-91)
92. Vgl. Göbel 2006, S. 16–17 [↑](#footnote-ref-92)
93. Vgl. Cohn 2007 [↑](#footnote-ref-93)
94. Vgl. Hariyadi 2008 [↑](#footnote-ref-94)
95. Vgl. Cohn 2007 [↑](#footnote-ref-95)