Extreme Programming







Risiko: Das Grundproblem

Jedes Projekt der Softwareentwicklung hat *Risiken:*

- Der Termin wird nicht eingehalten
- Die Kosten werden nicht eingehalten
- Die Qualitätsziele werden nicht erreicht

Ziel: Diese Risiken minimieren!

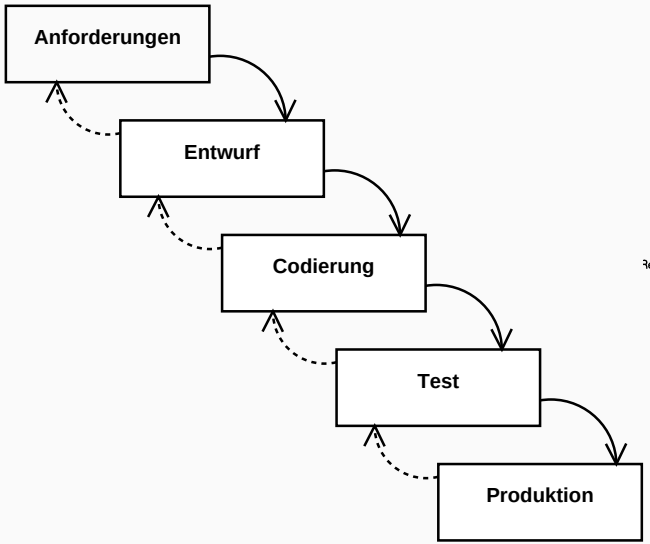




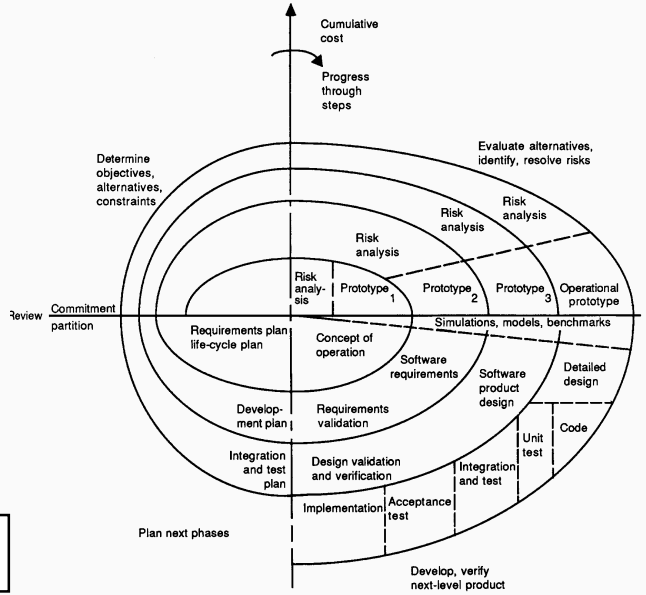


Vorgehensmodelle der Software-Entwicklung

Wasserfallmodell (1968)

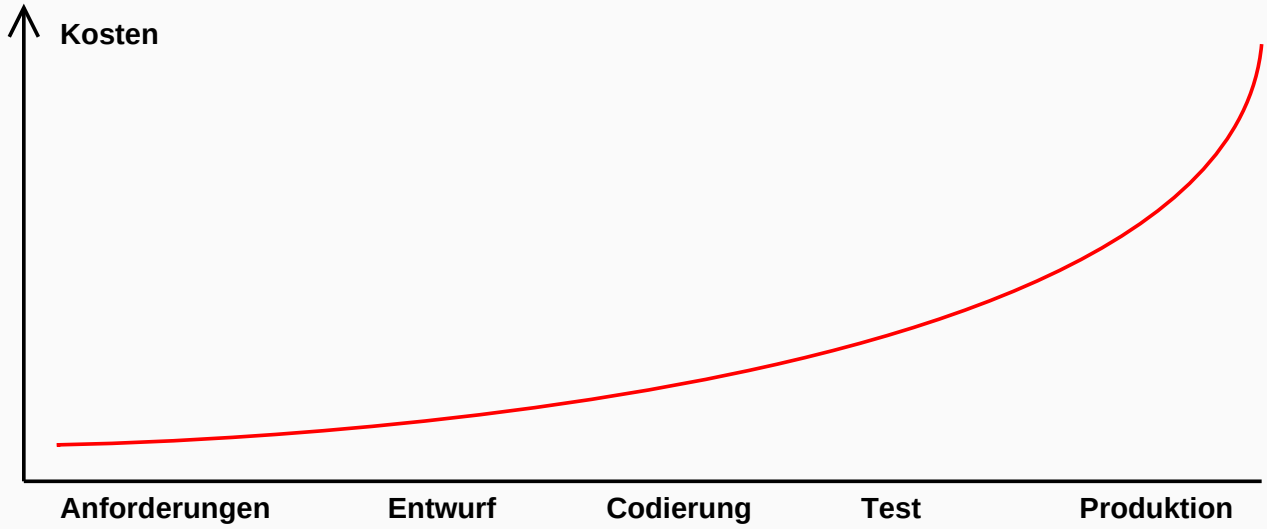


Spiralmodell (1988)



Änderungskosten

Annahme – Änderungskosten steigen *exponentiell:*



Folge: Große Sorgfalt bei Anforderungen und Entwurf

Wieviel Aufwand für den Entwurf?

Zunächst einmal: Große Sorgfalt bei Anforderungen und Entwurf ist gut.

Aber:

- *Vage Anforderungen* können oft erst durch Einsatz echter Systeme (z.B. Prototypen) geklärt werden
- Anforderungen und Entwurf benötigen *viel Zeit*, in der sich die Einsatzbedingungen ändern können
- Die *frühe Verfügbarkeit* eines Produkts kann über Wohl und Wehe einer Firma entscheiden

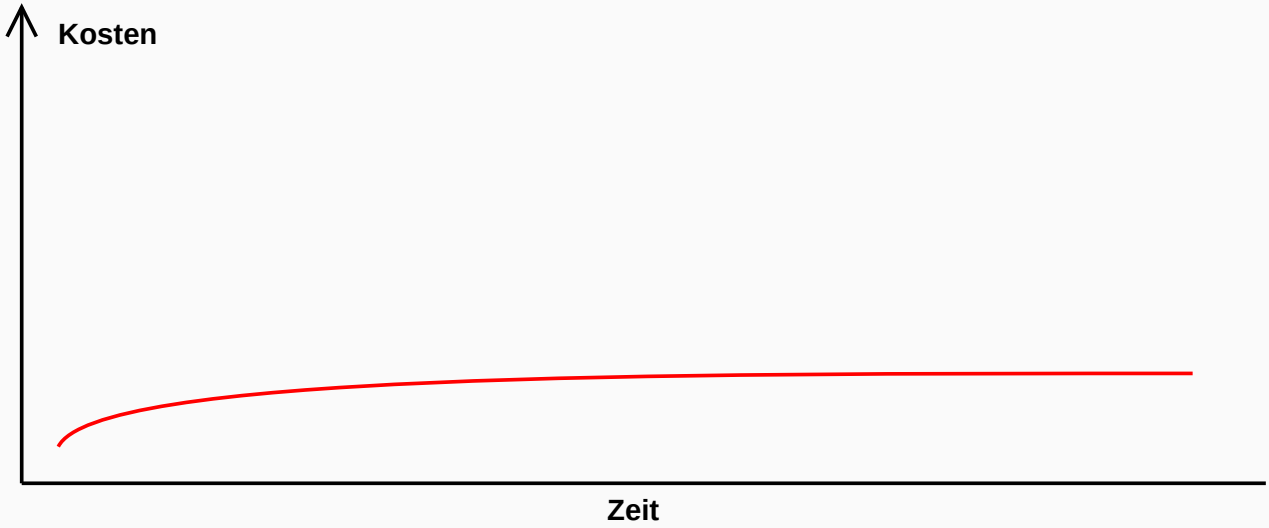






Flexibel durch niedrige Änderungskosten

Was wäre, wenn wir die Änderungskosten konstant halten könnten?



Folge: *Flexibilität während des gesamten Lebenszyklus*

Extreme Programming

Extreme Programming (XP) ist ein leichtgewichtiges Vorgehensmodell der Software-Entwicklung

- ✔ für *kleine bis mittelgroße* Teams,
- ✔ deren Anforderungen *vage* sind
- ✔ oder deren Anforderungen *sich schnell ändern.*

Ziel: Änderungskosten gering halten!







Übersicht

Extreme Programming setzt bewährte Techniken in *extremem Maße ein:*

- Kontinuierliche Reviews („Programmieren in Paaren“)
- Kontinuierliches Testen (Automatische Komponententests)
- Kontinuierliches Design und Redesign („Refactoring“)
- Kontinuierliche Rückmeldung durch kurze Release-Zyklen und ständiges Einbeziehen des Auftraggebers







Zweierkisten

Erkenntnis der 90er Jahre:

Programme gegenlesen lassen ist der beste Weg, die Qualität zu steigern!

Im Extreme Programming wird deshalb in *Paaren* programmiert:

- 1 Monitor, 1 Tastatur, 2 Programmierer
- Ein Partner schreibt, der andere liest gegen
- Ständiger Dialog und Rückmeldung
- Häufiger Rollen- und Tastaturtausch
- Keine Spezialisierung auf Rollen wie „Programmierer“, „Tester“, . . .

Paar-Erfahrungen

Es gibt zahlreiche Anekdoten von Programmierern, die in Paaren erfolgreich waren.

Beispiel: *C3 Chrysler Payroll System* (1997)

- 2000 Klassen, 20.000 Methoden
- Teils Einzel-, teils Paararbeit
- Installation termin- und ressourcengerecht
- Von den Fehlern, die in den ersten 5 Monaten des Einsatzes auftraten, stammten nahezu alle aus Einzelarbeit.

Paar-Erfahrungen (2)

1999 an der Universität von Utah: Kontrolliertes Experiment mit

- 13 Einzelprogrammierer (= Studenten)
- 14 Programmier-Paare
- 4 Aufgaben in 6 Wochen
- Programme wurden nach Einreichung getestet







Ergebnisse

Ergebnis 1: *Paare produzieren bessere Qualität*

Programm	Bestandene Testfälle Einzelprogrammierer	Bestandene Testfälle Programmier-Paare
1	73,4%	86,4%
2	78,1%	88,6%
3	70,4%	87,1%
4	78,1%	94,4%

Ergebnis 2: *Paare sind schneller fertig*

Programm	Zeitaufwand Einzelprogrammierer	Zeitaufwand Programmier-Paare
1	100%	79%
2	100%	59%
3	100%	60%

Aus: Williams et al., *Strengthening the Case for Pair-Programming,* IEEE Software, 2000

Weitere Erkenntnisse

Paar-Programmierer finden, daß die Arbeit in Paaren

- ✔ ihr Vertrauen in das Programm erhöht (96%)
- ✔ mehr Spaß macht als die Einzelarbeit (94%)

Außerdem: *Konzentration auf's Wesentliche* (= kein Web-Surfen)
⇒ *Pausen* sind wichtig!

Knackpunkt: *Chemie zwischen Partnern muß stimmen*

Zahlen aus: Laurie Williams, Pair Programming Questionnaire

Kontinuierliches Testen

Im XP werden alle Komponenten *automatisch getestet.*

```
// Test auf Hinzufügen zu Warenkorb
public void testProductAdd() {
    Product book = new Product("Refactoring", 53.95);
    bookCart.addItem(book);

    double expectedBalance = 23.95 + book.getPrice();
    double currentBalance = bookCart.getBalance();
    double tolerance = 0.0;
    assertEquals(expectedBalance, currentBalance, tolerance);

    int expectedItemCount = 2;
    int currentItemCount = bookCart.getItemCount();
    assertEquals(expectedItemCount, currentItemCount);
}
```

Ziel: Jederzeit die Funktionsfähigkeit des Systems sicherstellen!

Testfälle als Spezifikation

Im Extreme Programming werden Testfälle definiert, *bevor* die Komponente überhaupt implementiert wird:

- ✔ Test beschreibt die Modulfunktion aus exemplarischer Sicht
- ✔ Formaler als (informale) Spezifikation
- ✔ Tests laufen automatisch ab – selbst radikale Änderungen können so validiert werden
- ✔ Testfälle können auftretende Fehler dokumentieren (für jeden neuen Fehler wird sofort ein reproduzierender Testfall erstellt)
- ✘ Deckt (da Testen) nur Beispiele ab







Konsequenzen

Im Softwaretechnik-Praktikum an der Uni des Saarlandes verlangen wir,

- daß das Verhalten des Systems durch *Szenarios* beschrieben wird,
- daß diese Szenarios in ausführbare *Testfälle* gegossen werden,
- bevor implementiert wird!

Die Testbarkeit verbessert die Systemstruktur erheblich:

✔ Deutliche Entkoppelung der Subsysteme

✔ Deutliche Trennung von Funktionalität und Präsentation







Weg mit alten Zöpfen!

Durch konsequente Testbarkeit und Gegenlesen kann man auch große Änderungen am System vornehmen, ohne die Qualität zu gefährden.

Dies wird im XP eingesetzt, um das System stets *so einfach wie möglich* zu halten.

Ein Design ist genau dann richtig, wenn es

- ✔ alle Testfälle korrekt behandelt
- ✔ keine Redundanzen enthält
- ✔ alle wichtigen Absichten klar ausdrückt
- ✔ mit der minimalen Anzahl von Klassen und Methoden auskommt.

Systematisches Redesign

Im XP helfen *Refactoring-Verfahren* beim Umstrukturieren.

Refactoring (wörtl. „Refaktorisieren“) bedeutet das *Umstrukturieren* von Software in weitgehend unabhängige *Faktoren.*

Es gibt *Kataloge* von Refactoring-Verfahren:

- *Extract Method:* Code zu einer Methode zusammenfassen
- *Move Method:* Bewegen einer Methode von Klasse zu Klasse
- *Replace Magic Number with Symbolic Constant:* (klar)
- *Replace Conditional with Polymorphism:* Klassenhierarchie mit dynamischer Bindung einführen. . .

Es gibt auch erste Werkzeuge, die solche Methoden realisieren.

Beispiel: Extract Method

Es gibt ein Codestück, das zusammengefaßt werden kann.

Wandle das Codestück in eine Methode, deren Name den Zweck der Methode erklärt:

Alte Fassung

```
void printOwing(double amount) {
    printBanner();

    // print details
    print("name: " + _name);
    print("amount: " + amount);
}
```

Neue Fassung

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDe-
tails(double amount) {
    print("name: " + _name);
    print("amount: " + amount);
}
```

Vorteil: System bleibt stets wohlstrukturiert

Nachteile des Umstrukturierens

Wiederverwendung außerhalb des Projekts (= der aktuellen Testfälle) wird erschwert

Externe Dokumentation (= Entwurf) ist schnell obsolet; alles Wissen steckt im Code.

Programmierer müssen mit beständigen Änderungen leben (= gute Versionsverwaltung, Kommunikationshürden)







Stories

Im Extreme Programming beschreibt der Auftraggeber die Anforderungen über *Stories*.

Jede Story beschreibt einen konkreten Vorfall, der für den Betrieb des Produktes notwendig ist.

Beispiel – eine „Parkhaus“-Story:

- Der Fahrer bleibt an der geschlossenen Schranke stehen und fordert einen Parkschein an.
- Hat er diesen entnommen, öffnet sich die Schranke
- Der Fahrer passiert die Schranke, die sich wieder schließt.

Weitere Situationen (volles Parkhaus, Stromausfall. . .) werden ebenfalls durch eigene Stories beschrieben.

Entwurf? Nix Entwurf!

Das Extreme Programming kennt keine eigenen Entwurfsphasen:

- Die Funktionalität wird in Stories zusammengefaßt
- Jede Story wird 1:1 in einen Testfall umgesetzt
- Man nehme den *einfachsten Entwurf, der die Testfälle besteht* – und implementiere ihn
- Die Implementierung ist abgeschlossen, wenn alle Testfälle bestanden sind
- Treten bei der Abnahme weitere Fragen auf, gibt es *neue Stories* – und neue zu erfüllende Testfälle

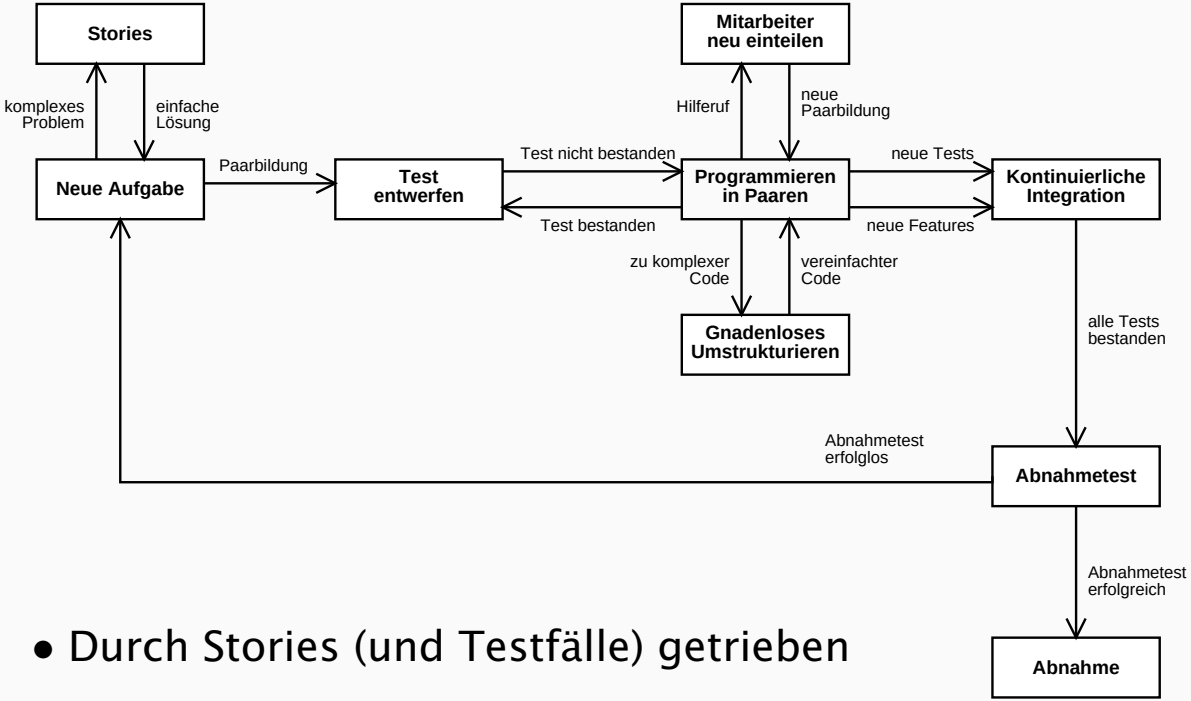
Der Kunde ist bei der gesamten Entwicklung dabei!







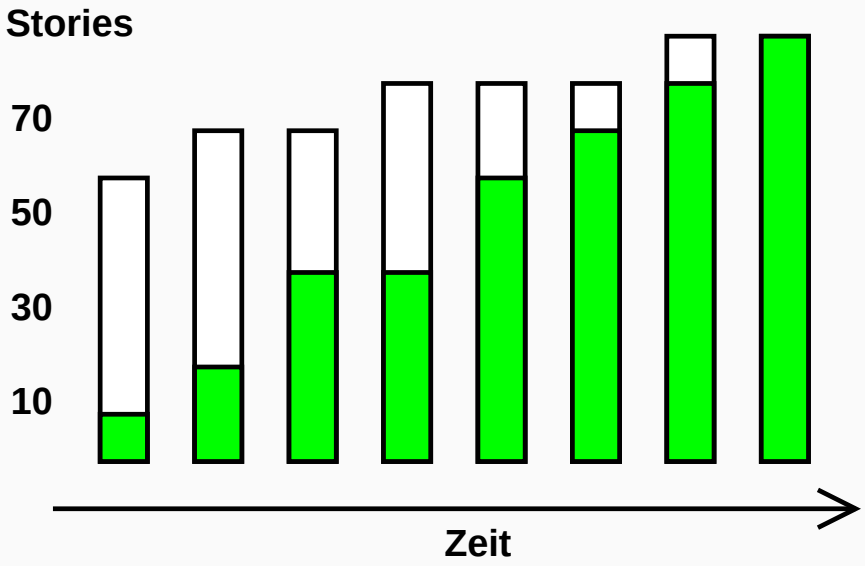
Das XP-Vorgehensmodell



- Durch Stories (und Testfälle) getrieben
- Programmieren steht im Mittelpunkt
- Zahlreiche Release-Zyklen

Projektfortschritt

Der Projektfortschritt wird durch die Anzahl der *erfolgreich getesteten Stories* ausgedrückt.



Kritik am Vorgehensmodell

Das XP-Vorgehensmodell wird oft als *unvollständig* kritisiert:

- ✘ Stories alleine reichen nicht aus, ein System zu beschreiben
- ✘ Keine Aussagen über Qualifikation, Integration von Qualitätsstandards (z.B. ISO 9000), Werkzeuge
- ✘ Keine Metriken oder Kostenschätzung – Ausnahme: Zahl (abgeschlossener) Stories

Folge: Ergänzung durch Konzepte „bewährter“ Entwurfsverfahren und Vorgehensmodelle

Zusammenfassung

Extreme Programming bringt frischen Wind in Vorgehensmodelle der Software-Entwicklung:

- Programmieren in Paaren
- Testfälle vor Implementierung + kontinuierliches Testen
- Kontinuierliches gnadenloses Umstrukturieren
- Vorgehen getrieben durch Stories und Testfälle







Extreme Programming – Vorbedingungen

Extreme Programming ist *geeignet...*

- ✔ wenn die Anforderungen *vage* sind
- ✔ wenn die Anforderungen *sich schnell ändern*
- ✔ wenn das Projekt klein bis mittelgroß ist (< 10–12 Programmierer)

Extreme Programming ist *ungeeignet...*

- ✘ wenn es auf *beweisbare* Programmeigenschaften ankommt
- ✘ wenn späte Änderungen zu teuer werden
- ✘ wenn häufiges Testen zu teuer ist
- ✘ wenn das Team zu groß oder nicht an einem Ort ist

Extreme Programming – Vor- und Nachteile

✔ Testfälle vor dem Codieren schreiben

✔ Programmieren in Paaren

- Kein Entwurf
- Keine externe Dokumentation

✘ Erschwerte Wiederverwendung

✘ Nur für kleine, hochqualifizierte Teams geeignet

✘ Erst wenig Erfahrung vorhanden

www.xprogramming.com
www.pairprogramming.com
www.refactoring.com
www.junit.org