eXtreme Programming

(und was wir davon lernen können!)

eXtreme Programming

Eine Einführung – basierend auf

- Kent Beck:
  eXtreme Programming eXplained
  Addison Wesley (2000)

- http://www.extremeprogramming.org

Was ist XP?

eXtreme Programming (XP) ist sowohl

- Programmiermethode

als auch

- Philosophie

Was ist neu an XP?

XP nimmt altbekannte, erprobte, *selbstverständliche* Programmierstrategien und „dreht alle Regler auf das Maximum“ –

alles was gut ist, wird ins Extrem gebracht!

XP sorgt für das beste Zusammenspiel zwischen diesen Techniken

Was ist gut?

- *Code review* ist gut, also programmieren wir zu zweit
- Testen ist gut, also testen wir immer (*unit testing)*, auch die Kunden (Funktionstests)
- Design ist wichtig, also machen wir es ständig (*refactoring)*
- Einfachheit ist gut, also machen wir das einfachste, was geht (was gerade so die Tests erfüllt)
- Architektur ist wichtig, also arbeiten alle ständig an der Gesamtarchitektur (Metapher!)
- Integration ist wichtig, also wird das Gesamtsystem täglich gebaut
- Kurze Iterationszyklen sind gut, also machen wir sie so kurz wie möglich (Sekunden-Minuten-Stunden)

Was haben wir davon?

Programmierer:

- Man arbeitet stets an wichtigem Code
- Man ist nicht allein
- Man kann wirklich etwas beitragen
- Man entscheidet qualifiziert

Kunden und Manager:

- Man kriegt das meiste für sein Geld
- Man sieht den Fortschritt
- Man kann ein Projekt auch später noch umlenken

Autofahren lernen

Das Schwierige am Auto fahren ist nicht, das Auto in die richtige Richtung zu bringen.

Richtig Auto fahren heißt, stets aufmerksam zu sein, und immer wieder ein wenig zu korrigieren, so dass man auf der Straße bleibt.

Werte

XP wird durch vier Werte gesteuert:

- Kommunikation
- Schlichtheit
- Feedback
- Mut

Alle diese Werte werden durch die Regeln und Praktiken des XP unterstützt

Kommunikation

- Probleme entstehen meistens, wenn die Kommunikation im Team oder mit dem Kunden (Tutor!) nicht funktioniert

Dagegen helfen

- Unit testing, programmieren zu zweit, Aufgabenabschätzung
- Ein *Coach* passt auf, dass die Kommunikation nicht abbricht

Schlichtheit

- Was ist das einfachste, was funktionieren könnte?
- Das ist schwierig!
- Nicht zuviel vorausdenken (Ängste!)
- Wette: Lieber billig etwas Einfaches umsetzen und das Komplizierte dadurch teurer machen, und darauf spekulieren, dass man das Komplizierte nie braucht.

Feedback

- Optimismus ist Mist
- Jeder muss jederzeit wissen, was das System kann
- Kunden müssen wissen, ob sie sinnvolle „Stories“ bringen
- Das System soll so schnell wie möglich „in Produktion“

Mut

- Mut zur Entscheidung: Wenn etwas fundamental verbockt ist, dann muss es geändert werden!
- Mut zum Fortwerfen von Code!
- Ausprobieren: Mögliche Implementationen einfach mal anfangen, und das Beste weitermachen!

Grundlegende Prinzipien

- Schnelles Feedback
  - Lernen geht nur mit direktem feedback
- Von Einfachheit ausgehen
  - Alles ist einfach lösbar (98% – 2%)
- Inkrementelle Änderungen
  - Kleine Schritte gehen besser
- *Embrace Change*
  - Optionen offenhalten, dringende Probleme lösen
- Qualitätsarbeit
  - Niemand möchte schlecht arbeiten (1 oder 1+)

Regeln und Praktiken

- Planung
  - Projektplanung und Teamplanung
- Coding
  - Das wichtigste überhaupt!
- Design
  - Meta-Coding
- Testing
  - Programmierer lieben Tests

Planung

- *Release plan (the planning game)*
  - Drei Phasen
    - *Exploration*
      - Was soll das System können?
    - *Commitment*
      - Was werden wir als nächstes machen?
    - *Steer*
      - Sorge dafür, dass das auch passiert
  - Zwei Spieler
    - *Business (**B**):* Management, Kunden
    - *Development (**D**):* Techniker, Programmierer

TPG: Explorationsphase

- **B:** Stories schreiben (Karteikarten)
  - was soll das System können?
- **D:** Aufwand für jede Story abschätzen
  - (in *„Ideal Engineering Time“)*
- **B:** Stories aufteilen
  - Falls **D** Aufwand nicht schätzen kann oder Teile verschieden wichtig

TPG: Commitment

- Sort by value (**B**)
  - Absolut wichtig – wertvoll – nice to have
- Sort by risk (**D**)
  - Präzise geschätzt – ganz OK zu schätzen – kann nicht geschätzt werden
- Set velocity (**D**)
  - Setze IET/Monat fest
- Choose scope
  - Business wählt Karten aus

TPG: Steering Phase (I)

- Iteration (alle 1–3 Wochen)
  - **B** wählt für diesen Zeitraum Stories aus
  - Erste Iteration muss ein lauffähiges System hervorbringen
- Recovery
  - **D** stellt fest, dass die Velocity falsch gewählt wurde und bitte **B** um eine neue Auswahl der Stories mit der neuen Geschwindigkeit

TPG: Steering Phase (II)

- New story
  - **B** stellt fest, dass noch eine neue Story benötigt wird, **D** schätzt den Aufwand und **B** darf austauschen
- Reestimate
  - Wenn **D** feststellt, dass der Gesamtplan nicht mehr realistisch ist, dann werden alle verbliebenen Stories und die Velocity neu eingeschätzt.

Iterationsplanung

- Jede Iteration wird von **D** geplant.
- Der Aufbau ist wie bei der Gesamtplanung
  - Exploration
  - Commitment
  - Steering

Exploration (Iterationsphase)

- Aufgaben *(tasks)* aufschreiben
  - Kleiner als stories
  - Eine task kann mehrere Stories bedienen
  - Eine task kann gar keine Story abdecken
- Tasks zusammenlegen und trennen
  - Mehrtägige Aufgaben aufteilen
  - Kurze (1h) Aufgaben kombinieren

Commitment Phase (It.)

- Aufgabe annehmen
- Aufgabe einschätzen
  - Wieviele IED (ideal engineering days)?
  - Darf höchstens ein paar Tage sein
- *Load factor* setzen
  - Gemessen: Wieviele IED/Tag?
- Austarieren
  - Haben alle gleich viel Arbeit?
  - Haben wir zuviel Arbeit?

Steering phase (It.)

- Implementiere eine Aufgabe
  - Suche einen Partner, schreibe die Tests, sorge dafür, dass sie laufen, integriere
- Fortschritt notieren
  - Alle 2-3 Tage: Wie lange schon und noch?
- Recovery
  - Task o. story reduzieren oder weglassen
- Story überprüfen
  - Funktionstests schreiben und durchführen

Weitere Planungsregeln

- „Stehendes Meeting“ jeden Morgen
- Leute austauschen
  - Paare wechseln, jeder soll möglichst viel vom Code kennen
- XP anpassen, wenn es nötig wird
  - Die Regeln sollen helfen. Wenn sie es nicht tun, dann muss man sie ändern!
  - Aber: Das Team muss darüber entscheiden, man kann sich nicht einfach „ausklinken“

Coding

- Pair programming
  - Zwei Leute an einem Computer produzieren nicht so viel Code wie zwei Leute an zwei Computern, dafür aber mit weniger Fehlern
  - Fliegende Wechsel an Keyboard & Maus!
  - Man muss sich daran gewöhnen
  - Fördert Kommunikation, Schlichtheit und Mut
  - Achtung: Beide sollten gleich gut sein!

Coding

- Zuerst die Tests schreiben
  - Fördert Schlichtheit und Feedback
  - Spezifiziert die Anforderungen
- Coding standards
  - Wichtig für refactoring, gemeinsames Lesen und Verstehen, CVS
- Optimierung kann warten
  - Nicht raten, messen!
  - Make it work – make it right – make it fast.
  - Keep it simple (KISS)

Integration

- Es integriert immer nur ein Paar gleichzeitig
- Es wird so oft wie möglich integriert
  - Findet Kompatibilitätsprobleme so schnell wie möglich
- Der Code gehört allen
  - Jeder muß Fehler beheben
  - Es gibt keinen „Chefarchitekten“
  - Die Tests geben die Sicherheit!

No overtime

- Eine Woche hat 40 Stunden. Punkt.
- Überstunden sind verboten
- „Projects that require overtime to be finished will be late no matter what you do“
- Natürlich kann man Ausnahmen machen. Aber nicht ständig!

On-site customer

- Jemand, der das System benutzen wird, muss immer persönlich erreichbar sein – für Fragen
- Dadurch wird das System schneller fertig und besser
- Das lohnt sich!

Design

- **EINFACH!**
  - Alle Tests laufen
  - Keine Programmlogik ist doppelt
    - Auf parallele Klassenhierarchien achten
  - So wenig Klassen und Methoden wie möglich
  - Keine Features „auf Vorrat“ programmieren (YAGNI!)
  - Design und Programmieren gleichzeitig!

Refactoring

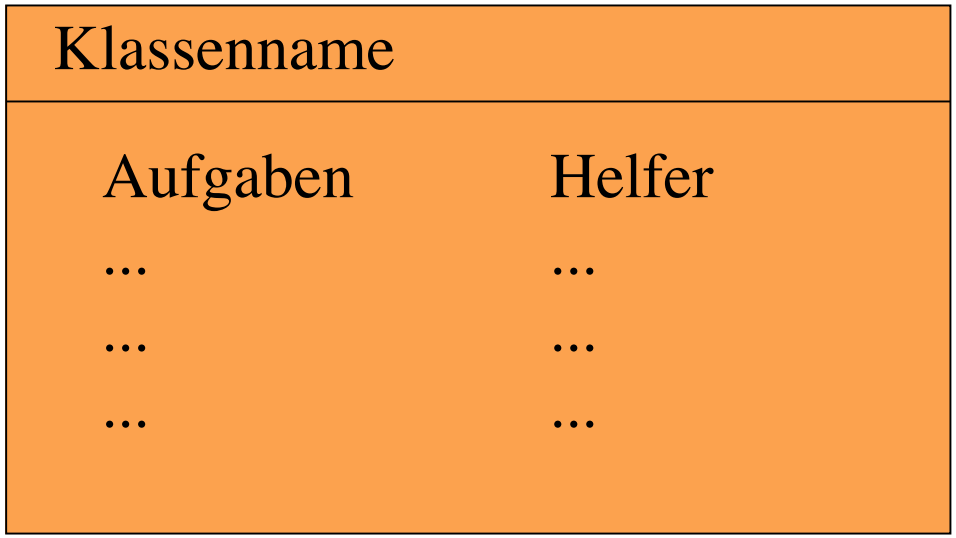
- Kann das bestehende Design geändert werden, so dass
  1. die Funktionalität erhalten bleibt?
  2. alle Tests immer noch laufen?
  3. das Programm einfacher wird?
- Das kann mehr Arbeit sein
- Beispiel: Code duplizieren schreit nach refactoring

Metapher

- Das Team braucht einen Weg, über das Gesamtsystem zu sprechen. Dafür braucht es eine gut gewählte Metapher
- Beispiele:
  - Fließband
  - Schreibtisch
  - Tabellenkalkulation

CRC Cards

- CRC =
  Class, Responsibilities, Collaboration
- Karteikarten
- Damit wird herumge-schoben



Tests

- Unit tests
  - Vom Programmierer geschrieben
  - Nur wenn alle unit tests laufen, darf es eine release geben
  - You break it, you fix it
- Functional/Acceptance tests
  - Vom Kunden geschrieben
- Bug finden heißt Test schreiben
  - Dann ist es einfacher ihn zu beheben

Überblick

- The planning game
- Small releases
- Metapher
- Einfaches Design
- Testen
- Refactoring
- Pair programming
- Collective Ownership
- Kontinuierliche Integration
- 40-Stunden-Woche
- On-site customer
- Coding standards

Warum funktioniert es?

Ein Beispiel:

- Einfaches Design (nur für heute planen)
  - Refactoring erlaubt einfache Änderungen
  - Es gibt eine Metapher, so dass zukünftige Änderungen absehbar sind
  - Programmieren zu zweit, damit es einfach, nicht dumm wird.

Und warum noch?

Noch ein Beispiel:

- Pair programming
  - Coding standards vermeiden blöde Streits
  - Alle sind frisch und ausgeruht
  - Erst werden Tests geschrieben, damit klar ist, was gemacht werden muss
  - Es gibt eine gemeinsame Metapher, so dass man besser miteinander reden kann
  - Das Design ist so einfach, dass beide es verstehen

Und die 40-Stunden Woche?

- Das Planning Game sorgt für sinnvolle Aufgaben und produktive Arbeit
- Durch Planen und Testen werden hässliche Überraschungen vermieden
- XP lässt einen so schnell programmieren, dass das Projekt sowieso nicht schneller laufen kann
- Und außerdem muß man ausgeschlafen sein!

Fazit

- XP ist gut und schwierig
- XP produziert gute Software
- Man kann nicht immer XP machen
- Man möchte in Firmen arbeiten, die XP praktizieren

Können wir XP für Torfu benutzen?