Agile Softwareentwicklung

Softwareentwicklung besteht aus

- Endlose Diskussionen
- Planen
  - Zielfindung → Requirement Analyse (Pflichtenheft), Capture (Lastenheft)
  - Klassendiagramm → Architektur (Feindesign)
- Fehlersuche
- Codieren
- Optimieren
- Testen
- Organisation
- Schnittstellen – technisch und organisatorisch
- Ändern
- Spezifikation
- Machbarkeitsstudie → Riskassessment
- Dokumentation
- Aufwandsberechnung
- Produktpräsentation → auch Prototypen
- Configuration Management
  - Version Control

V-Modell-Submodelle

- SWE (Software Engineering)
- QS (Qualitätssicherung)
- KM (Konfigurationsmodelle)
- PM (Projektmanagement)

Produkte
Aktivitäten
Tailoring

Herkömmliches Projekt

Projektmanagement

- Planen: Ressourcen, Zeit, Aufwand, Kosten
- Supervision / Kontrolle / Steuerung
- Roadmap: Milestones, Baseline → Schnittstelle zum Kunden
- Replanning (z.B. Ressourcen verschieben)
- Konfliktmanagement: intern und extern
- Motivation / Reflektion / Koordination

Software Engineering / Software Development

- Requirement Capture / Analysis
- Verification (Unit-Tests, Abnahmetests, …) / Validation (Endabnahme aus Kundensicht) → AIV / AIT (Assembly, Integration, Verification, Test)
- Implementierung
- Design
- Dokumentation

Qualitätssicherung (QA) / Produktsicherung (PA)

- Freigabe von Produkten
- nach Prüfung
- Content Management
  - produktbezogen
  - prozessbezogen
  - Process Improvement
- RAMS:
  - Reliability
  - Availability
  - Maintainability
  - Safety

Manifesto for Agile Development

1. Individuals and interaction over process and tools
2. Working Software over comprehensive documentation
3. Customer Collaboration over contract negotiation
4. Responding to change over following a plan
   (Wenn geplant wird, dann muss mitgeplant werden, dass sich Plan ändern kann.)

Principles

- satisfy customer through early and continuous delivery of valuable software
- welcome changing requirements
- deliver working software frequently
- alle ein Team (Projektmanager, Kunde, Entwickler, …)
- motivated individuals
- face to face conversation
- working software is the primary measure for progress
- promote sustainable development
- continuous attention to technical excellence and good design
- simplicity
- self-organizing teams
- reflect on how to become more effective

Varianten von Agiler Software Entwicklung

- **XP: Extreme Programming**
- **RUP: Rational Unified Process**
- Crystal
- Adaptive Software Development
- Scrum
- FDD: Feature Driven Development
- DSDM: Dynamic System Development Methods
- **AM: Agile Modelling**
- **TDD: Test Driven Development**

(**Fett**gedruckte werden im Kurs behandelt)

XP: Extreme Programming

4 Variablen zur Kontrolle eines Projektes

- Cost → meist vorgegeben und nicht veränderbar
- Time ↗
- Quality (gefährlich bei Abstrichen)
- Scope (Funktionsumfang) ← aus XP-Sicht am einfachsten zu handhaben

XP Values

- Kommunikation:
  - pair programming
  - stand-up meetings
  - team room (Arbeitsraum)
    - customer integrated
    - shared white boards
    - pin board
- Einfachheit:
  - runs all tests
  - reveal all intentions
- Feedback:
  - team, customers
  - tests
  - pair programming
  - interaction planning
- Mut (mutig, zuzugestehen, was nicht geht, aber auch sagen, dass man das anders machen könnte):
  - to make changes to working code

XP Practices

1. Whole team
    a. programmer
    b. coach
    c. customer
    d. management (muss nicht dauerhaft anwesend sein)
    e. user (muss nicht dauerhaft anwesend sein)
2. Planning Game
    a. Releases → user stories (use cases)
    b. Iterations → 1-3 weeks, tasks
3. Small Release
4. Customer Tests
    a. user stories schreiben
    b. acceptance tests schreiben
5. Simple Design → test-first & refactoring
6. Pair Programming
7. Test-Driven Design → automatisch, kontinuierlich
8. Design Improvement → Refactoring
9. Collective Code Ownership
10. Continuous Integration
11. Coding Standards
12. Common Vocabulary
13. Sustainable Pace (einheitliche Geschwindigkeit des Teams bzw. der Entwickler)

11.02.04

RUP – Rational Unified Process

- develop iteratively
    - incremental growth
    - risk reduction through demonstrable progress
    - frequent releases
    - enabling continuous end-user involvement
    - focus on producing results
    - frequent status check
- Manage Requirements
    - elicit (entlocken), organize, document required functionality and constraints
    - track and document trade-offs, decisions
    - capture & communicate requirements → use case, scenarios
- component-based architecture
- Visually Model Software
    - capture
    - structure
    - behaviour

- Verify software quality → alle Teilnehmer und Verwendung objektiver Messwerte und Kriterien
  - korrekt im funktionalen Sinne
  - Reliability / Availability performance
  - Operability performance (z.B. Ergonomie der Schnittstellen)
- Control Change to Software
  - establish secure workspaces
  - Team working together as a single unit
  - automatic build, integration

Phasen des RUP

- Inception (Anfang): Request Capture, Risk Analysis
  - Vision document, initial use cases (10-20%)
  - project glossary, initial risk assessment
  - project plan (phases, iterations)
  - prototypes (optional)

  < Review
- Elaboration: Request Analysis, Architekturprototyp
  - use cases (ca. 80%), non-functional requirements, executable prototypes
  - preliminary user manual

  < Review
- Construction: Implementierung
  - components, integration, parallel
  - user manual, description of current release

  < Review
- Transition: Auslieferung an Kunden, Beta-Test
  - Migration from old system
  - Training

  < Review

Vergleich XP vs. RUP

XP	RUP
kleine Projekte	Projekte aller Größen
kurze Laufzeit	alle Laufzeiten
keine Phasen	phasenorientiert
"ohne Tools"	toolzentriert
ohne Doku-Overhead	Starke Betonung von Doku

Das Fahrstuhl-Beispiel

Ebenen	Kabinen	Steuereinheit
Anforderungsknöpfe	Stockwerksknöpfe	Anforderungslisten
Anzeigelemente	Anzeigeelemente	Scheduling
Sensoren für Fahrstuhlerkennung	Türsensoren/-aktuatoren	

Use Case vs. User Story

Use Case

- Nutzerbezogene Beschreibung eines Systemverhaltens
- Cockburn: behavioural requirements
  - low precision: actor – goal list
  - medium precision: title + success scenario
  - medium-high precision: extension, error conditions by name
  - high precision: detailed description (algorithms) + error conditions & handling

User Story

- funktionale Aspekte der Software, die in **einer** Iteration realisiert werden können
- chunk of functionality that is valuable to the customer
- verständlich
- je kürzer desto besser
- **mehrere** in eine Iteration

Use Cases bei Fahrstuhl-Beispiel

- Türfunktion & Sicherheit (normal und exceptional)
- Fahrstuhl fahren (normal)
- Fahrstuhl rufen
- Fahrstuhlwartung
- Abarbeitungsreihenfolge

Akteure

- „ich“ / „FB“
- Fahrstuhl
- Techniker
- Sensoren
- Knöpfe
- Tür
- Timer
- Anzeigeelement
- Motor
- Scheduler

Unsere User Stories

- Die Anzeigen geben die letzte gesicherte Position der Kabinen wieder (sowohl innerhalb der Kabine als auch in den Stockwerken).
- Die Türen der Kabine (und damit auch die des Stockwerks) öffnen sich ausschließlich dann, wenn sich die Kabine im zugehörigen Stockwerk befindet.
- Der Fahrstuhl kann erst dann fahren, wenn innere und äußere Tür geschlossen sind.
- Das Schließen der Tür wird entweder durch den Signalgeber (Timer) nach Ablauf der Wartezeit aktiviert (Zeitsignal) oder durch den Interrupt, der vom Schnellschließen-Knopf ausgelöst wird.
- Beim Schließen der Tür geht Sicherheit vor: beim Erkennen eines Hindernisses wird ein Schnellschließen ignoriert bzw. abgebrochen.
- Wenn ein Richtungsknopf gedrückt wird, hält ein Fahrstuhl, der in diese Richtung fährt.
- Wenn man einen Stockwerksknopf drückt, hält der Fahrstuhl in diesem Stockwerk.
- Wenn der Fahrstuhl in einem Stockwerk hält, geht die Tür auf.
- Wenn man den „Tür-Öffnen-Knopf“ drückt, geht die Tür auf oder bleibt auf,
- Richtungswechsel finden nur statt, wenn keine weiteren Anforderungen in seine aktuelle Fahrtrichtung anliegen.
- Anforderungen können von Richtungs- und Stockwerksknöpfen ausgelöst werden.
- Bei Fahrtrichtungswechsel ändert sich die Anzeige für die Fahrtrichtung
- Wenn keine Anforderungen vorhanden sind, dann werden alle Richtungsanzeigen im aktuellen Stockwerk aktiviert.

Agile Modellierung (AM)

Values

1. Effective, lightweight modelling
2. Avoid to be too code centric
3. Improve development after modelling

Principles – Practices – Goals

1. Assume simplicity
2. Content is more important than representation
3. Embrace change
4. Enabling the next effort is your secondary goal
5. Everybody can learn something from anybody else
6. Incremental change
7. Know your models
8. Local adaptation
9. Maximize stakeholder (Kunde, Firmenmanagement, Nutzer, Maintenance Team, …) investment
10. Model with the purpose
11. Multiple model

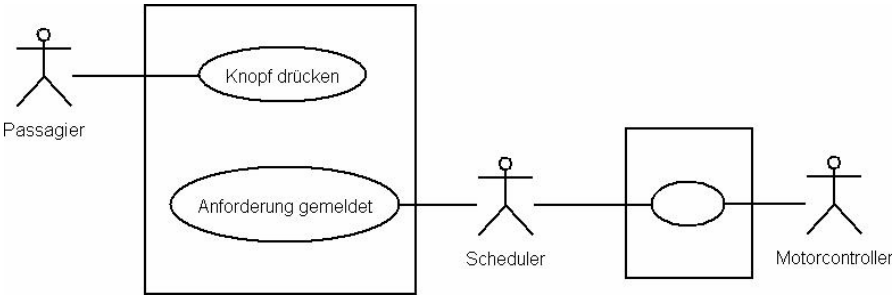
12. Open and honest communication
13. Quality work
14. Rapid feedback (Modelansätze werden diskutiert)
15. Software is your goal
16. Travel light (nur soviel Modelle, wie unbedingt nötig)
17. Work with peoples instincts (auf “Bauchgefühle” hören)

12.02.2004

UML Diagramme

Use Case Diagramm

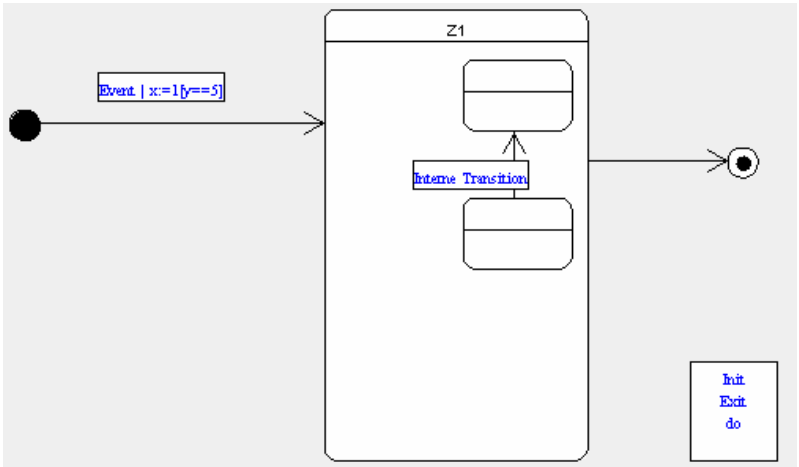
- Use Cases
- Actors



Activity Diagrams

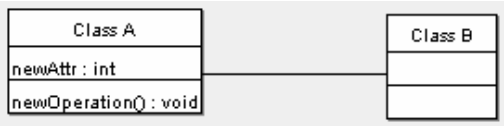
- Beschreiben abstrakter Abläufe
- Modellierung von komplexer Logik des Systems
    - einzelner Operation, Methode oder Use Case

State Diagrams



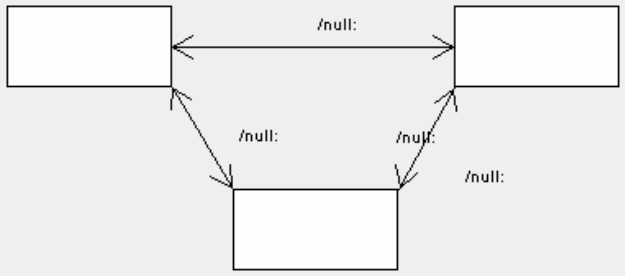
Sequenzdiagramme

Klassendiagramme



Collaboration Diagrams (Communication Diagrams)

- Beschreibung von Beziehungen zwischen Objekten
- Strukturelle Organisation



Composite Structure Diagrams

- Interne Struktur von Objekten
  - Klassen
  - Komponenten
  - Use Cases

Interaction Overview Diagrams

- Variante der Activity Diagrams
  - control flow im System
  - Knoten repräsentieren auch andere Interaction Diagrams

Timing Diagrams

- zeitbezogene Zustandsübergänge

Component Diagrams

- Interaktion zwischen den Komponenten
- Public Interfaces unserer Komponenten

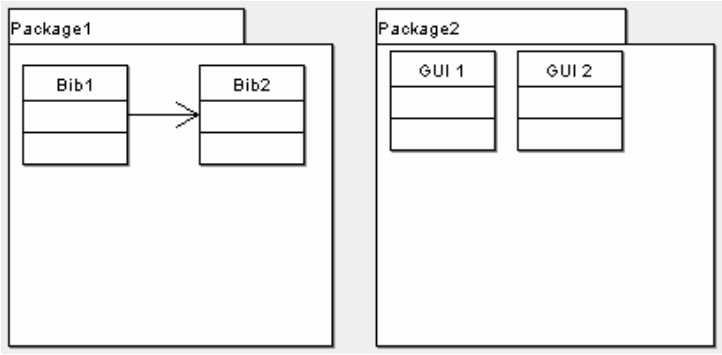
Deployment Diagram

- physikalische Struktur
- Hardware/Software und Middleware

Object Diagrams

- Objekte und ihre Beziehungen zu einem spezifischen Zeitpunkt
- Spezialisierung der Klassendiagramme

Package Diagrams



Gruppen-Aufgabe

Use Case Diagramm – Unsere Use Cases / User Stories
Sequenz Diagramm – Genereller Ablauf des Fahrstuhl
State Diagramm – Türmechanismus (Timer, …)
Activity – Scheduler parallel / synchronisiert mit Motorkontrolle und Türmechanismus
Deployment Diagramm

13.02.2004

Release / Iteration Planning

Story	Time Estimation (Ideal Week)	Assigned Iteration #	Assigned Release #
Wenn Knopf dann kommt Fahrstuhl.	2	1	1
Wenn Knopf, dann fährt er auch zum richtiges Stockwerk.	2	2	1
Wenn Fahrstuhl hält,	2	3	2

dann Tür auf.			
Wenn Fahrstuhl losfährt, dann sind alle Türen zu.	1	4	2
Alle Stockwerkstüren sind zu, wenn kein Fahrstuhl da ist.	1	5	2
Bei Hindernis nicht schließen und Fehler melden.	4	6	2
Mehrere Anforderungen innerhalb des Fahrstuhls möglich.	3	7	3
Interne angewählt Stockwerke werden in sinnvoller Reihenfolge angefahren.	4	8	3
Richtungswechsel finden nur statt, wenn keine weiteren Anforderungen in seine aktuelle Fahrtrichtung anliegen.	2	9	3
Mehrere Fahrstühle sind möglich.	4	10	4
Mehrere Anforderungen auch von außerhalb möglich.	3	11	5
Bei Fahrtrichtungswechsel ändert sich die Anzeige für die Fahrtrichtung	1		6
Wenn keine Anforderungen, dann alle Richtungsanzeigen im aktuellen Stockwerk aktiviert	1		6

…
…

Effort Estimation

- basierend auf eigenen Erfahrungen
- Teameffekt
- always use the same unit → ideal week
- Yesterdays weather
- Estimates are no commitments

Ideal Time

- Calendar Time = day – weeks – month
  8 Stunden 5 Tage 4 Wochen
- Calendar Effort = # Personen * Calendar Time
- Ideal Time = Zeit, in der man konzentriert an einer Task arbeitet
- Load Factor = $\frac{\text{calendar effort}}{\text{velocity}}$ pro Iteration
- Velocity = Ideal Time eines Entwicklers / Teams in einer Iteration

Softwareentwicklungsumgebung

- Compiler
- Editor
- Debugger
- Versionskontrolle
- Hierarchiebrowser
- Buildtool
- Dokumentation (low-level, z.B. javadoc)
- Modellierungswerkzeug
- Testtool
- Browser für Manuals
- Refactoring browser
- Libraries

Guter Code

- Maintainability
  - Lesbarkeit
  - Modularität – Komplexität
  - Dokumentation
- Coding Rules
  - Naming Conventions
  - # Lines of Code

Test Driven Development

Motivation

- Defizite traditioneller Testansätze
- nicht erkannte Fehler bei unzureichenden Tests
- teurere Fehlerbehebung bei späten Tests
- Post-mortem: erst der Code, dann Tests
- Independent Testing führt zum Übersehen von wichtigen Tests ← Unit
- Spez. basierte Tests häufig nicht aktuell
- nicht-automatisierte Tests werden nicht regelmäßig durchgeführt
- Fehlerbehebung führt neue Fehler ein, die die „alten" Tests nicht finden

TDD-Charakterisierung

- Der Programmierer testet; Code basierend auf Tests – test first
  - testability garantiert
  - vollständige Test coverage
  - Orthogonalität zwischen Codeanteilen
  - Code und Test synchronisiert
- automatisiert, regelmäßig, einheitlich
- vollständige Testüberdeckung (100% Codecoverage) → nachträglich eingebaute Fehler werden sofort aufgedeckt
- Tests gehören zur Lieferung → regression tests

TDD-Practices

1. Maintain an exhaustive suite of programmer tests
   a. unit tests zeigen, dass vorhandener Code korrekt ist
   b. customer tests: Tests aus Nutzersicht
   c. definieren, das der Code tun soll
   d. exhaustive = kein Stück Code wird nicht getestet
2. No code goes into production unless is has associated tests
3. Write tests first
4. Tests determine what code you need to write
5. Neuen Code nur, wenn es Probleme beim Testen gibt
   Programmierschritte
   1. RED: Test, der nicht funktioniert
   2. GREEN: Simple Code, der den Test durchlaufen lässt

Soziale Effekte von TDD

- QA proaktiv, nicht reaktiv durch weniger Fehler
- Projektmanagement kann Fortschritt und Kosten besser abschätzen ➔ Kundenzufriedenheit
- ENG kann enger zusammenarbeiten, bei tageweiser Planung

Fahrstuhl-Aufgabe

- Datenstruktur Fahrzielliste
    - repräsentiert Statue der angefragten Fahrziele
    - Abfrage des Status
    - Austragen von erreichten Zielen
    - Eintragen neuer Anfragen

Refactoring

- Änderung an existierendem Code, die das Verhalten nach außen nicht ändert
- improve internal structure → Duplikate entfernen
- Wann?
    - Duplikate entdeckt
    - When we perceive that the code or its intent is not clean
    - Code Smell

Duplikation

- separate Methoden auslagern
- in Hierarchie nach oben schieben
- Generalisieren

Unclear Intent

- code must be simple and understandable
- better names
- “think about the interfaces rather than the implementation”

Code Smells

- Indizien für nicht ausreichende Qualität
- Typische Code Smells:
    - comments
    - data classes
    - duplicate code
    - inappropriate intimacy
    - large classes / lazy classes / long methods
    - shot gun surgery
    - switch statements

How to refactor

- only if automated tests are in place
- in small steps
- help from good refactoring tools
- refactor of models

Typen

- Extract Classes – Interfaces – Method
- Replace Type Code with subclasses
- replace conditionals with polymorphism
- Template Methods
- Introduce Explaining Variables
- Replace constructor by factory method (Pattern)
- Replace inheritance (Vererbung) with delegations
- Replace magic numbers with symbolic constants

Zeiterfassung einer Iteration

Pair	Date	Hour
— —	———	———

Am Ende der Iteration in ein Spreadsheet eintragen.

- Acceptance Tests defined and passed
- Test Code zu Production Code
- # Integrationen
- Bug Density
- Story Process
- System Performance