Software Entwicklung

Jil Zerndt, Lucien Perret
May 2024

Einführung und Überblick

Software Engineering

- Disziplinen: Anforderungen, Architektur, Implementierung, Test und Wartung.
- Ziel: Strukturierte Prozesse für Qualität, Risiko- und Fehlerminimierung.

Modellierung in der Softwareentwicklung

- Modelle als Abstraktionen: Anforderungen, Architekturen, Testfälle.
- Einsatz von UML: Skizzen, detaillierte Blueprints, vollständige Spezifikationen.

Wrap-up und Ausblick

- Solide Analyse- und Entwurfskompetenzen sind essenziell.
- Iterativ-inkrementelle Modelle fördern agile Entwicklung.
- Nächste Lerneinheit: Detaillierte Anforderungsanalyse.

1 Vorlesung 01

1.1 Charakteristiken von Wasserfall, iterativ-inkrementellen und agilen Softwareentwicklungsprozessen

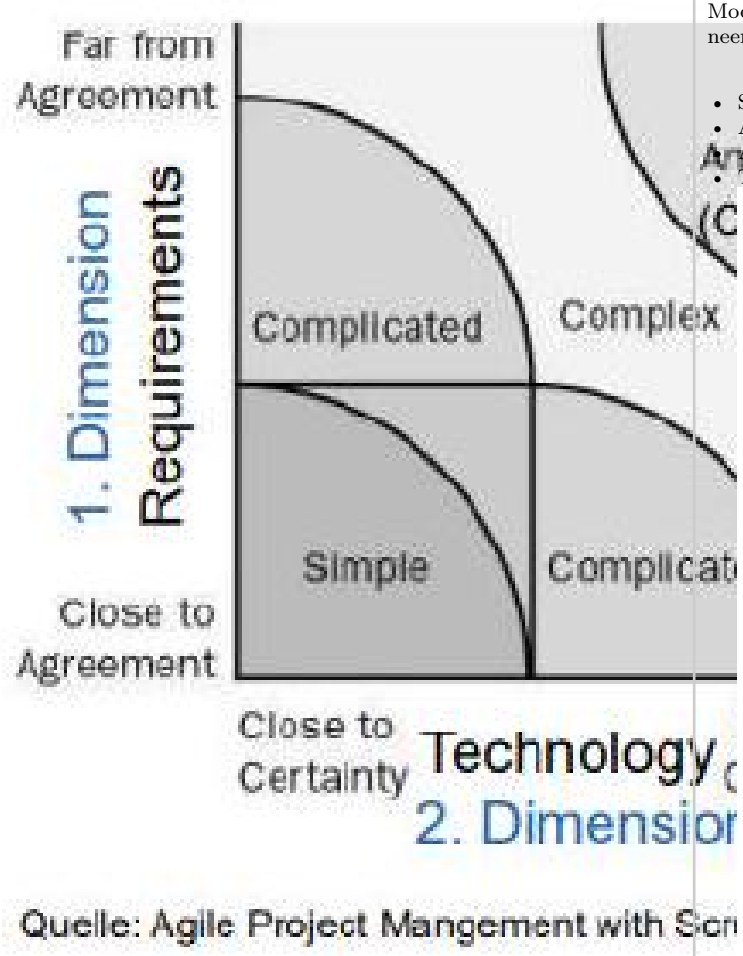


Abbildung 1: Klassifizierung Software-Entwicklungs-Probleme

Prozesse im Softwareengineering Kernprozesse

- Anforderungserhebung
- Systemdesign/technische Konzeption
- Implementierung
- Softwaretest
- Softwareeinführung
- Wartung/Pflege

Unterstützungsprozesse

- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

Begriffe Warum wird modelliert: Um Analyse- und Designentwürfe zu diskutieren, abstimmen und zu dokumentieren bzw. zu kommunizieren.
Modell: Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhanden Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).
Original: Das Original ist das abgebildete oder zu schaffende Gebilde.

Modellierung: Modellierung gehört zum Fundament des Software Engineerings

- Software ist vielfach (immer?) selbst ein Modell
- Anforderungen sind Modelle der Problemstellung
- Architekturen und Entwürfe sind Modelle der Lösung
- Testfälle sind Modelle des korrekten Funktionierens des Code

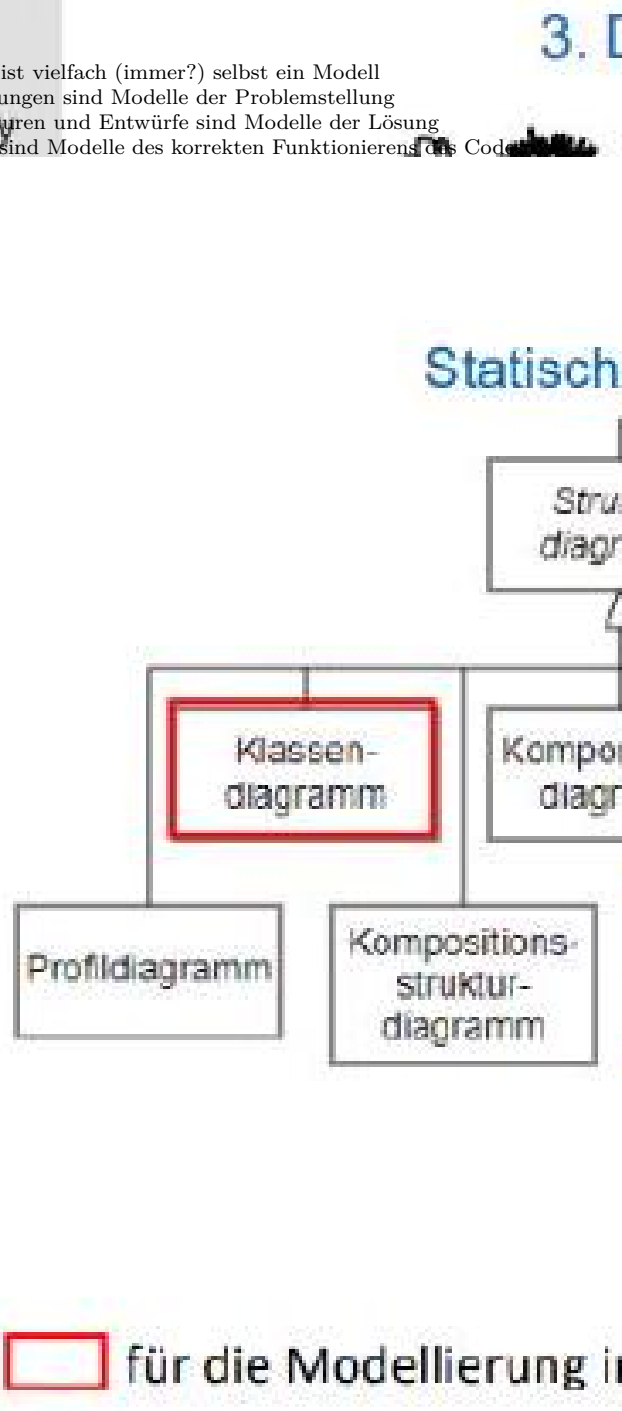


Abbildung 2: Diagramme UML Übersicht

1.1.1 Code and Fix

Vorgehen, bei dem Codierung oder Korrektur im Wechsel mit Ad-hoc-Tests die einzigen bewussten ausgeführten Tätigkeiten der Software-Entwicklung sind: Schnell, Agil, Einfach am Anfang, Schlecht Planbar, Schlecht Wartbar, Änderungen s. Aufwändig

1.1.2 Wasserfallmodell

Die Software-Entwicklung wird als Folge von Aktivitäten/Phasen betrachtet, die durch Teilergebnisse (Dokumente) gekoppelt sind. Die Reihenfolge der Aktivitäten ist fest definiert. : gut planbar, klare Aufteilung in Phasen, Schlechtes Risikomanagment, nie alle Anforderungen zu Anfang bekannt

1.1.3 Iterativ-inkrementelle Modelle

Software wird in mehreren geplanten und kontrolliert durchgeführten Iterationen schrittweise (inkrementell) entwickelt: Flexibles Modell, Gutes Risikomanagement, Frühe Einsetzbarkeit, Planung upfront hat Grenzen, Kunde Involviert über ganze Entwicklung

Agile Softwareentwicklung Basiert auf interativ-inkrementellen Prozessmodell, Fokussiert auf gut dokumentierten und getesteten Code statt auf ausführlicher Dokumentation

1.2 Zweck und den Nutzen von Modellen in der Softwareentwicklung

- Modell von Requirements (close to/ far from Agreement) & Technology (known / unknown)
  Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhanden Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).

1.2.1 Unified Modelling Language (UML)

UML ist die Standardsprache für die graphische Modellierung von Anforderungen, Analyse und Entwürfen im Software Engineering (objektorientierte Modellierung). (As a sketch, blueprint, programminglanguage)

1.3 Artefakte in einem iterativinkrementellen Prozess illustrieren und einzuordnen

Abbildung 3: Incremental Model

Abbildung 4: Usability und User Experience (UX)

Usability = Deutsch: Gebrauchstauglichkeit
User Experience = Usability + Desirability
Customer Experience = Usability + Desirability + Brand Experience

2.2 Usability-Anforderungen

Die Effektivität, Effizienz und Zufriedenheit mit der die adressierten Benutzer ihre Ziele erreichen in ihren spezifischen Kontexten.

Wichtigste Aspekte

- Benutzer
- Seine Ziele/Aufgaben
- Sein Kontext
- Softwaresystem (inkl. UI)

Effektivität

- Der Benutzer kann alle seine Aufgaben vollständig erfüllen
- Mit der gewünschten Genauigkeit

Effizienz Der Benutzer kann seine Aufgaben mit minimalem/angemessenem Aufwand erledigen

- Mental
- Physisch
- Zeit

Zufriedenheit Mit dem System / der Interaktion

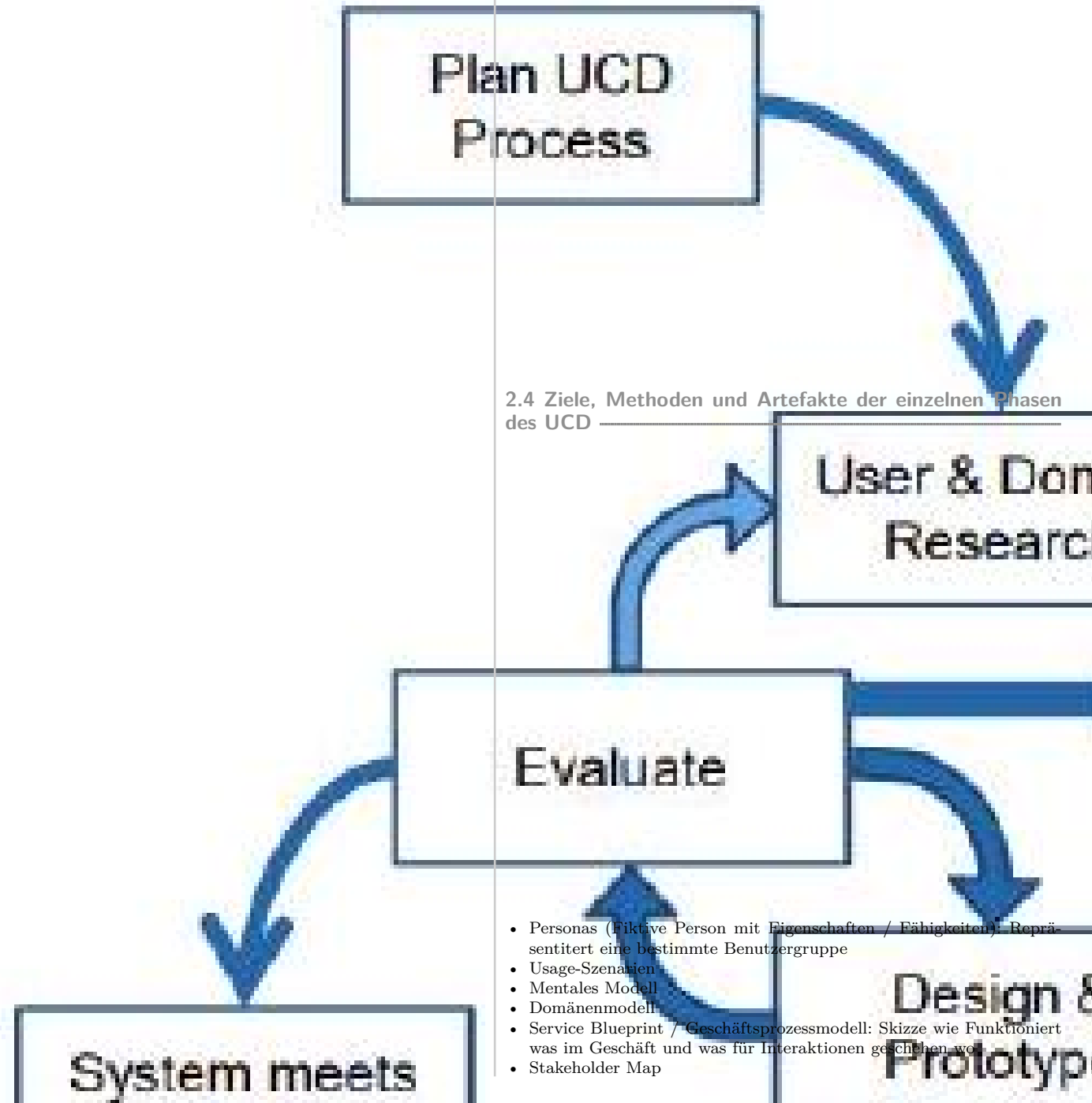
- Minimum: Benutzer ist nicht verärgert
- Normal: Benutzer ist zufrieden
- Optimal: Benutzer ist erfreut

2.2.1 7 wichtige Anforderungsbereiche bezüglich Usability

- Aufgabenangemessenheit
- Lernförderlichkeit
- Individualisierbarkeit
- Erwartungskonformität
- Selbstbeschreibungsfähigkeit
- Steuerbarkeit
- Fehlertoleranz

2.3 User-Centered Designs

Abbildung 5: Usercentered Design



2.4 Ziele, Methoden und Artefakte der einzelnen Phasen des UCD

- Personas (Fiktive Person mit Eigenschaften / Fähigkeiten): Repräsentitert eine bestimmte Benutzergruppe
- Usage-Szenarien
- Mentales Modell
- Domänenmodell
- Service Blueprint / Geschäftsprozessmodell: Skizze wie Funktioniert was im Geschäft und was für Interaktionen geschehen wo
- Stakeholder Map

Abbildung 6: Stakeholdermap Example

- Zusätzlich: UI-Skizzen der wichtigsten Screens, Wireframes (UIPrototypes), UI-Design, weitere Usability - Anforderungen

3 Vorlesung 03

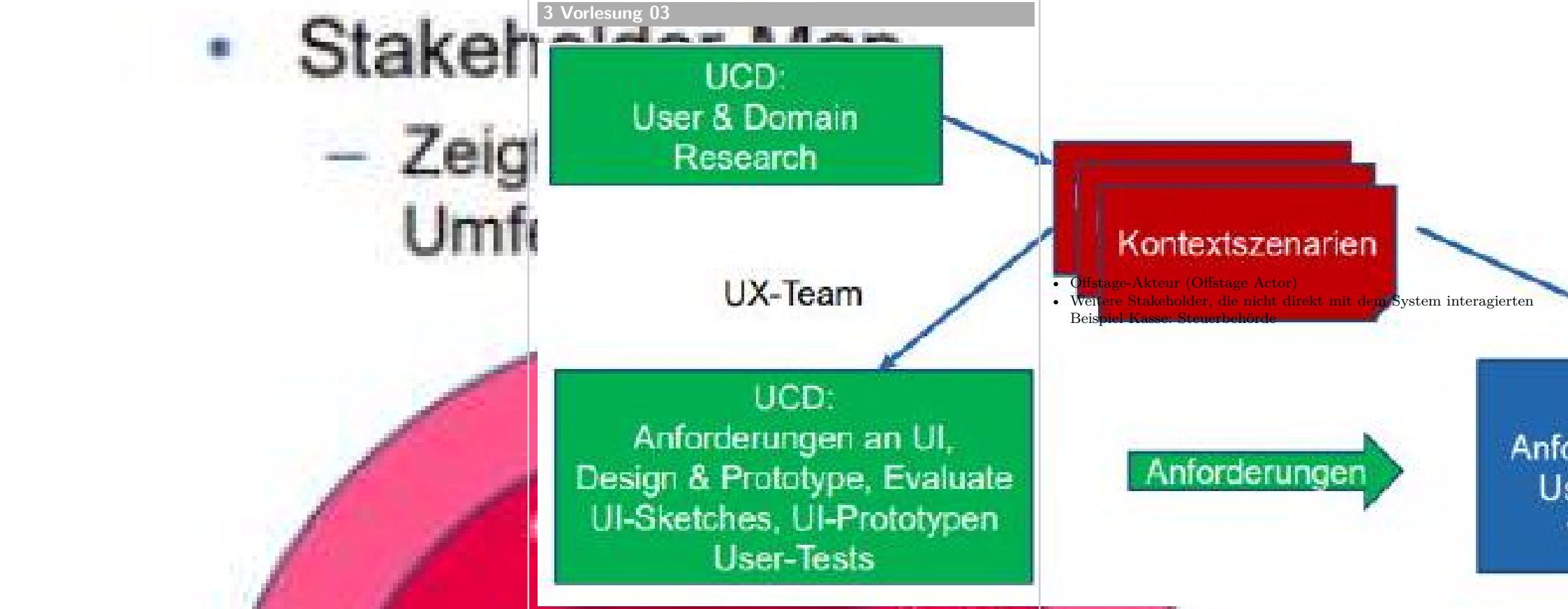


Abbildung 7: Anforderungsanalyse Übersicht

3.1 Anforderungen aus Artefakten des UCD

Anforderungen (Requirements): Forderungen bezüglich (Leistungs-) Fähigkeiten oder Eigenschaften die das System unter gegebenen Bedingungen erfüllen muss (explizit oder implizit)

- Meist sind nie alle Anforderungen im Voraus vollständig bekannt, entwickeln sich im Laufe des Projekts
- Müssen mit den Benutzern und Stakeholdern erarbeitet werden

3.2 Anforderungen in Form von Use Cases

Textuelle Beschreibung einer konkreten Interaktion eines Benutzers mit zukünfigem System (Beschreiben aus Sicht des Akteurs, Impkizite und Explizite Anforderungen, Ziel des Akteurs, Kontext)

3 Arten von Akteuren

- Primärakteur (Primary Actor)
- Initiiert einen Anwendungsfall, um sein (Teil-)Ziel zu erreichen
  Erhält den Hauptnutzen des Anwendungsfalls
  Beispiel Kasse: Kassier
- Unterstützender Akteur (Supporting Actor)

Hilft dem SuD bei der Bearbeitung eines Anwendungsfalls
Beispiel Kasse: externer Dienstleister wie Zahlungsdienst für Kreditkarten

- Offstage-Akteur (Offstage Actor)
- Weitere Stakeholder, die nicht direkt mit dem System interagierten
  Beispiel Kasse: Steuerbehörde

Abbildung 8: Arten von Akteuren

- Aus Sicht des Akteurs
- Aktiv formuliert (Titel auch aktiv)
- Konkreter Nutzen
- Mehr als eine einzelne Interaktion / UseCase
- im Essentiellen, nicht Konkreten Stil (Logik, nicht Umsetzung)

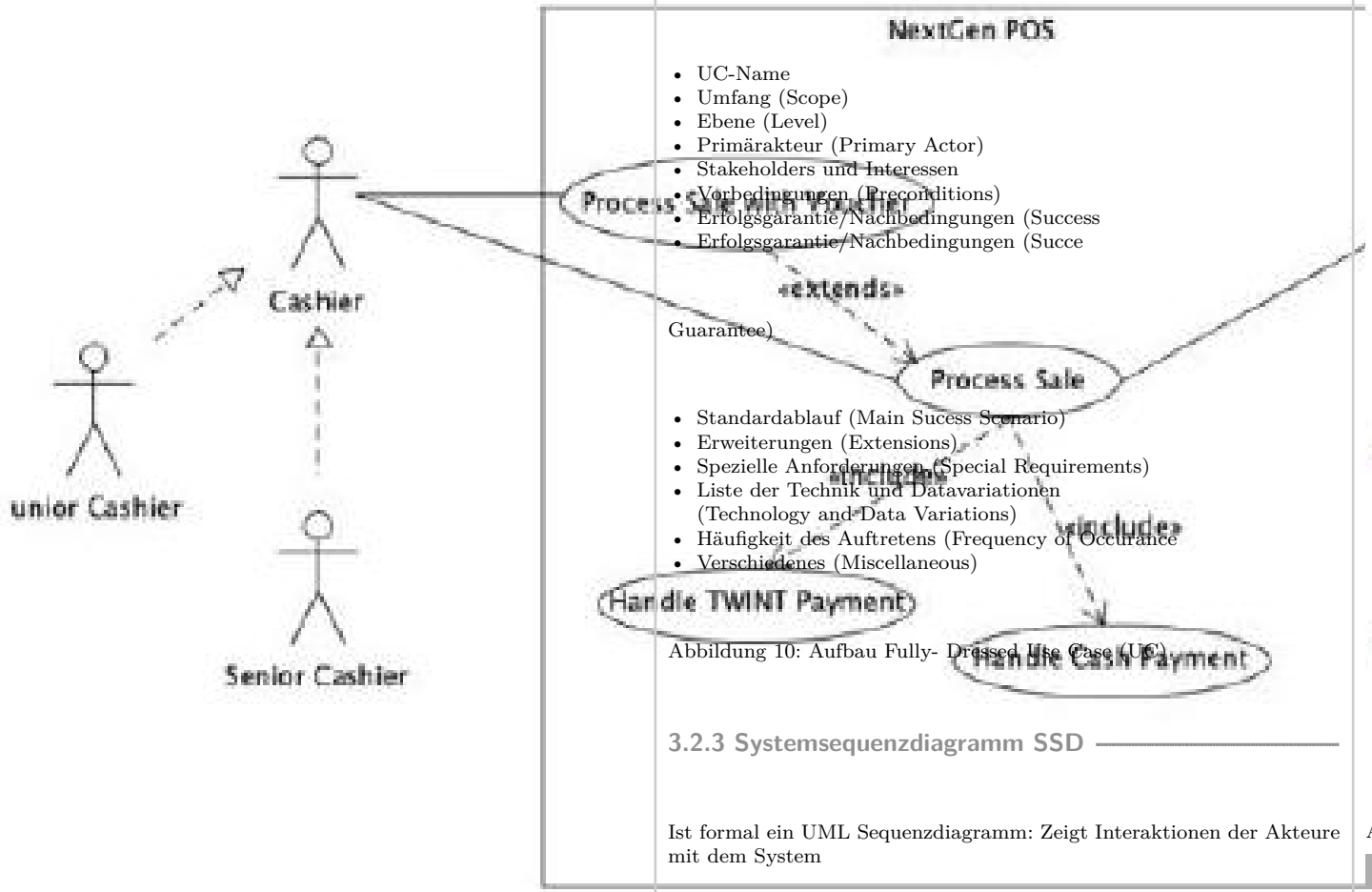


Abbildung 9: Use Case Diagramm

3.2.1 Brief UC

Kurze Beschreibung des Anwendungsfalls in einem Paragraphen

- Nur Erfolgsszenario
- Sollte enthalten:
- Trigger des UCs
- Akteure
- Summarischen Ablauf des UCs
- Wann?: Zu Beginn der Analyse

3.2.2 Casual UC

Informelle Beschreibung des Anwendungsfalls in mehreren Paragraphen

- Erfolgsszenario plus wichtigste Alternativszenarien
- Sollte enthalten:
- Trigger des UCs
- Akteure
- Interaktion des Akteurs mit System
- Wann?: Zu Beginn der Analyse

Formaler Aufbau

- UC-Name
- Umfang (Scope)
- Ebene (Level)
- Primärakteur (Primary Actor)
- Stakeholders und Interessen
- Vorbedingungen (Preconditions)
- Erfolgsgarantie/Nachbedingungen (Success
- Erfolgsgarantie/Nachbedingungen (Succe

Guarantee)

- Standardablauf (Main Sucess Scenario)
- Erweiterungen (Extensions)
- Spezielle Anforderungen (Special Requirements)
- Liste der Technik und Datavariationen (Technology and Data Variations)
- Häufigkeit des Auftretens (Frequency of Occurance
- Verschiedenes (Miscellaneous)

Abbildung 10: Aufbau Fully- Dressed Use Case (UC)

3.2.3 Systemsequenzdiagramm SSD

Ist formal ein UML Sequenzdiagramm: Zeigt Interaktionen der Akteure mit dem System

- Welche Input-Events auf das System einwirken
- Welche Output-Events das System erzeugt

Ziel:
Wichtigste Systemoperationen identifizieren, die das System zur Verfügung stellen muss (API) für einen gegebenen Anwendungsfall

- Formal wie Methodenaufruf, evtl mit Parametern, Details zu Parametern sollen im Glossar erklärt werden
- Durchgezogener Pfel für Methodenaufruf
- Rückgabewert kann fehlen falls unwichtig, indirektes Update des UI, deshalb gestrichelte Linie

SSD können auch Interaktionen zwischen SuD und externen unterstützenden System zeigen

- Links ist Primärakteur aufg
  - Hier Cashier
    - Inkl. seiner Benutzerschnittstell
  - Initiiert die Systemoperatione
    - UI findet zusammen mit Akteur dieser tun möchte
    - UI ruft sodann entsprechende S auf
- Mitte das System (:System
  - Muss die Systemoperationen Verfügung stellen
- Rechts
  - Sekundärakteure, falls nötig

Abbildung 11: Systemsequenzdiagramm (SSD)

Use Case Realisation

7 Vorlesung 07

7.1 Use Cases und Use-Case-Realization

Die Planung erfolgt anhand von Use-Cases, Realisierung v Use-Cases
Der wichtigste Teil sind die detaillierten Szenarien (Standardszenario und Erweiterungen), und davon die Systemantworten. Diese müssen schlussendlich realisiert werden.

7.1.1 Vergleich SSD

UI statt System, Systemoperationen sind Elemente die realisiert werden

7.1.2 Warum UML

- Zwischenschritt bei wenig Erfahrung
- Ersatz für Programmiersprache, Kompakt
- auch für Laien zu verstehen

7.2 Vorgehen UC Realization

1. Use Case auswählen, offene Fragen klären, SSD ableiten
2. Systemoperation auswählen
3. Operation Contract (Systemvertrag) erstellen/überlegen
4. Aktuellen Code/Dokumentation analysieren
   (a) DCD überprüfen/aktualisieren

(b) Vergleich mit Domänenmodell durchführen
(c) Neue Klassen gemäß Domänenmodell erstellen
5. Falls notwendig, Refactorings durchführen
(a) Controller Klasse bestimmen
(b) Zu verändernde Klassen festlegen
(c) Weg zu Klassen festlegen
i. Weg mit Parametern wählen
ii. Klassen ggf. neu erstellen
iii. Verantwortlichkeiten zuweisen
iv. Varianten bewerten
(d) Veränderungen programmieren
(e) Review durchführen

Design Patterns

8 Vorlesung 08

8.1 Allgemeiner Aufbau u Zweck von Design Pattern (DP)

- bewährte Lösungen f wiederkehrende Probleme schnell finden
- effiziente Kommunikation
- immer tradeoff zw. Flexibilität u Kompatibilität
- Programm wird nicht besser mit DP

8.1.1 Adapter

Ermöglicht die Zusammenarbeit von Objekten mit inkompatiblen Schnittstellen. (Überall wo Dienste angesprochen werden, die austauschbar sein sollten)

8.1.2 Simple Factory

(eigene Klasse) erstellt Objekte, die aufwändig zu erzeugen sind

8.1.3 Singleton

Stellt sicher, dass eine Klasse nur eine Instanz hat und einen globalen Zugriffspunkt darauf bereitstellt.

8.1.4 Dependency Injection

Ermöglicht es, einem abhängigen Objekt die benötigten Abhängigkeiten bereitzustellen. (Ersatz f Facotry Pattern)

8.1.5 Proxy

Bietet einen Platzhalter (mit demselben Interface) für ein anderes Objekt, um den Zugriff darauf zu kontrollieren. Proxy Objekt leitet alle Methodenaufrufe zum richtigen Objekt weiter

- Remote Proxy: Stellvertreter für ein Objekt in einem anderen Adressraum und übernimmt die Kommunikation mit diesem.
- Virtual Proxy: Verzögert das Erzeugen des richtigen Objekts bis zum ersten Mal, dass es benutzt wird.
- Protection Proxy: Kontrolliert den Zugriff auf das richtige Objekt.

8.1.6 Chain of Responsibility

Ermöglicht es einem Objekt, eine Anfrage entlang einer Kette potenzieller Handler zu senden, bis einer die Anfrage behandelt. (wenn unklar im vorraus welcher Handler zuständig sein wird)

9 Vorlesung 09

9.1 Decorator

9.1.1 Problem

Ein Objekt (nicht eine ganze Klasse) soll mit zusätzlichen Verantwortlichkeiten versehen werden.

9.1.2 Solution

Ein Decorator, der dieselbe Schnittstelle hat wie das ursprüngliche Objekt, wird vor dieses geschaltet. Der Decorator kann nun jeden Methodenaufruf entweder selber bearbeiten, ihn an das ursprüngliche Objekt weiterleiten oder eine Mischung aus beidem machen.

9.2 Observer

9.2.1 Problem

Ein Objekt soll ein anderes Objekt benachrichtigen, ohne dass es den genauen Typ des Empfängers kennt.

9.2.2 Solution

Ein Interface wird definiert, das nur dazu dient, ein Objekt über eine Änderung zu informieren. Dieses Interface wird vom «Observer» implementiert. Das
«Observable» Objekt benachrichtigt alle registrierten «Observer» über eine Änderung.

9.3 Strategy

9.3.1 Problem

Ein Algorithmus soll einfach austauschbar sein.

9.3.2 Solution

Den Algorithmus in eine eigene Klasse verschieben, die nur eine Methode mit diesem Algorithmus hat.
Ein Interface für diese Klasse definieren, das von alternativen Algorithmen implementiert werden muss

9.4 Composite

9.4.1 Problem

Eine Menge von Objekten haben dasselbe Interface und müssen für viele Verantwortlichkeiten als Gesamtheit betrachtet werden.

9.4.2 Solution

Sie definieren ein Composite, das ebenfalls dasselbe Interface implementiert und Methoden an die darin enthaltenen Objekte weiterleitet

9.5 State

9.5.1 Problem

Das Verhalten eines Objekts ist abhängig von seinem inneren Zustand.

9.5.2 Solution

Das Objekt hat ein darin enthaltenes Zustandsobjekt.
Alle Methoden, deren Verhalten vom Zustand abhängig sind, werden über das Zustandsobjekt geführt.

9.6 Visitor

9.6.1 Problem

Eine Klassenhierarchie soll um (weniger wichtige) Verantwortlichkeiten erweitert werden, ohne dass viele neue Methoden hinzukommen.

9.6.2 Solution

Die Klassenhierarchie wird mit einer Visitor-Infrastruktur erweitert. Alle weiteren neuen Verantwortlichkeiten werden dann mit spezifischen VisitorKlassen realisiert.

9.7 Facade

9.7.1 Problem

Sie setzen ein ziemlich kompliziertes Subsystem mit vielen Klassen ein. Wie können Sie seine Verwendung so vereinfachen, dass alle Team-Mitglieder es korrekt und einfach verwenden können?

9.7.2 Solution

Eine Facade (Fassade) Klasse wird definiert, welche eine vereinfachte Schnittstelle zum Subsystem anbietet und die meisten Anwendungen abdeckt.

9.8 Abstract Factory

9.8.1 Problem

Sie müssen verschiedene, aber verwandte Objekte erstellen, ohne ihre konkreten Klassen anzugeben. Wie können Sie die Erstellung der Objektfamilien zentralisieren und von ihrer konkreten Implementierung abstrahieren?

9.8.2 Solution

Das Abstract Factory Muster definiert eine Schnittstelle zur Erstellung von Familien verwandter oder abhängiger Objekte, ohne ihre konkreten Klassen zu benennen. Es bietet Methoden, um Objekte der verschiedenen Produktfamilien
zu erstellen, und ermöglicht es, ganze Produktfamilien konsistent zu verwenden.

9.9 Factory Method

9.9.1 Problem

Es gibt eine Oberklasse, aber die genauen Unterklassen sollen durch eine spezielle Logik zur Laufzeit bestimmt werden. Wie können Sie die Instanziierung dieser Klassen so gestalten, dass sie flexibel und erweiterbar bleibt?

9.9.2 Solution

Das Factory Method Muster definiert eine Schnittstelle zur Erstellung eines Objekts, lässt aber die Unterklassen entscheiden, welche Klasse instanziiert wird. Dadurch wird die Erstellung der Objekte auf Unterklassen delegiert, wodurch die Klasse flexibler und erweiterbar wird.

9.10 Command

9.10.1 Problem

Sie müssen eine Anforderung in Form eines Objekts kapseln, um parameterisierte Clients, Warteschlangen oder Log-Requests sowie Operationen rückgängig machen zu können. Wie können Sie dies strukturieren?

9.10.2 Solution

Das Command Muster kapselt eine Anforderung als Objekt, wodurch Sie Parameter für Clients, Warteschlangen oder Log-Requests einfügen und Operationen rückgängig machen können. Es besteht aus einem Kommando-Objekt, das eine bestimmte Aktion mit ihren Parametern, Empfänger und eventuell einer Methode zur Rückgängigmachung enthält.

9.11 Template Method

9.11.1 Problem

In einer Methode einer Oberklasse gibt es einige Schritte, die in Unterklassen unterschiedlich implementiert werden sollen, während die Struktur der Methode erhalten bleiben muss. Wie können Sie die Schritte anpassbar machen?

9.11.2 Solution

Das Template Method Muster definiert das Skelett eines Algorithmus in einer Methode, wobei einige Schritte von Unterklassen implementiert werden. Es ermöglicht Unterklassen, bestimmte Schritte des Algorithmus zu überschreiben, ohne die Struktur des Algorithmus zu verändern.

Implementation, Refactoring und Testing

10 Vorlesung 10

10.1 Quellcode aus Design Artefakten ableiten

10.1.1 Umsetzungs-Reihenfolge: Variante Bottom-Up

Kurze Erklärung: Implementierung beginnt mit Basisbausteinen, die schrittweise zu größeren Teilen kombiniert werden.
Vorgehen: Start mit Basisfunktionalitäten, dann schrittweise Erweiterung und Integration.
Eigenschaften: Gründlich, bietet solide Basis, gut für sich ändernde Anforderungen.

10.1.2 Umsetzungs-Reihenfolge: Variante Agile

Kurze Erklärung: Flexible, inkrementelle Entwicklung in kurzen Iterationen. Vorgehen: Kontinuierliche Lieferung funktionsfähiger Teile in Sprints, Anpassung an sich ändernde Anforderungen.
Eigenschaften: Hohe Flexibilität, schnelles Feedback, geeignet für sich ändernde Anforderungen.

10.2 Codier-Richtlinien

Abmachung für:

- Fehlerbehandlung
- Codierrichtlinien (Gross/Kleinschreibung, Einrücken, Klammernsetzung, Prüf programme)
- Namensgebung f. Klasse, Attribute, Methoden, Variablen

10.3 Implementierungs- / Umsetzungsstrategie

Code-Driven Development

- Zuerst die Klasse implementieren

TDD: Test-Driven Development

- Zuerst Tests für Klassen/Komponenten schreiben, dann den Code entwickeln

BDD: Behavior-Driven Development

- Tests aus Benutzersicht beschreiben
- Zum Beispiel durch die Business Analysten mit Hilfe von Gherkin

10.4 Laufzeit Optimierung, Optimierungsregeln

- Optimiere nicht sofort, sondern analysiere zuerst, wo Zeit tatsächlich verbraucht wird.
- Verwende Performance-Monitoring-Tools, um Zeitfresser zu identifizieren.
- Besonders kritisch sind Datenbankzugriffe pro Objekt über eine Liste.
- Überprüfe und optimiere Algorithmen, z.B. Collections.sort() in Java 7.
- Bedenke, dass moderne Compiler bereits viel Optimierung leisten.
- Ziehe Berechnungen aus Schleifen heraus, da die Java VM und Just-In-Time-Compilation diese optimieren.

10.5 Refactoring

Definition: Strukturierte Methode zum Umstrukturieren vorhandenen Codes, ohne das externe Verhalten zu ändern. Ziele:

- Verbesserung der internen Struktur durch viele kleine Schritte.
- Trennung vom eigentlichen Entwicklungsprozess.
- Verbesserung des Low-Level-Designs und der Programmierpraktiken.

Methoden zur Code-Verbesserung:

- DRY: Vermeidung von dupliziertem Code.
- Klare Namensgebung für erhöhte Lesbarkeit.
- Aufteilung langer Methoden in kleinere, klar definierte Teile.
- Strukturierung von Algorithmen in Initialisierung, Berechnung und Ergebnisverarbeitung.
- Verbesserung der Sichtbarkeit und Testbarkeit.

Code Smells:

- Duplizierter Code
- Lange Methoden
- Klassen mit vielen Instanzvariablen oder viel Code
- Auffällig ähnliche Unterklassen
- Fehlen von Interfaces oder hohe Kopplung zwischen Klassen

Unterstützung durch:

- Automatisierte Tests zur Sicherstellung der Funktionsfähigkeit nach Refactoring.
- Moderne Entwicklungsumgebungen, die abhängige Arbeitsschritte automatisieren.

Refactoring Patterns:

- Umbenennung von Methoden, Klassen und Variablen für klarere Bezeichnungen.
- Verschieben von Methoden in Super- oder Subklassen.
- Extrahieren von Teilfunktionen in separate Methoden oder Konstanten.
- Einführung erklärender Variablen zur Verbesserung der Lesbarkeit.

10.6 Testing

10.6.1 Grundlegende Testarten

- Funktionaler Test (Black-Box Verfahren): Überprüft die Funktionalität des Systems, ohne den internen Code zu kennen.
- Nicht funktionaler Test (Lasttest etc.): Testet nicht-funktionale Anforderungen wie Leistung, Skalierbarkeit, usw.
- Strukturbezogener Test (White-Box Verfahren): Überprüft die interne Struktur des Codes, um sicherzustellen, dass alle Pfade abgedeckt sind.
- Änderungsbezogener Test (Regressionstest etc.): Überprüft, ob durch Änderungen im Code keine neuen Fehler eingeführt wurden.
- Integrationstest
- Systemtest
- Abnahmetest
- Regressionstest

10.6.2 Wichtige Begriffe

- Testling, Testobjekt: Objekt, das getestet wird
- Fehler: Fehler, den der Entwickler macht
- Fehlerwirkung, Bug: Jedes Verhalten, das von den Spezifikationen abweicht
- Testfall: Satz von Testdaten zur vollständigen Ausführung eines Tests
- Testtreiber: Programm, das den Test startet und ausführt

10.6.3 Merkmale

Was wird getestet?

- Einheit / Klasse (Unit-Test)
- Zusammenarbeit mehrerer Klassen
- Gesamte Applikationslogik (ohne UI)
- Gesamte Anwendung (über UI)

Wie wird getestet?

- Dynamisch: Testfall wird ausgeführt (Black-Box / White-Box Test)
- Statisch: Quelltext wird analysiert (Walkthrough, Review, Inspektion)

Wann wird der Test geschrieben?

- Vor dem Implementieren (Test-Driven Development, TDD)
- Nach dem Implementieren

Wer testet?

- Entwickler
- Tester, Qualitätssicherungsabteilung
- Kunde, Endbenutzer