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.

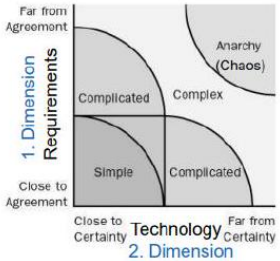
Wrap-up

- Solide Analyse- und Entwurfskompetenzen sind essenziell.
- Iterativ-inkrementelle Modelle fördern agile Entwicklung.

Softwareentwicklungsprozesse

Klassifizierung Software-Entwicklungs-Probleme

Wir betrachten Wasserfall, iterativ-inkrementelle und agile Softwareentwicklungsprozesse.



Quelle: Agile Project Mangement with Scrum, Ken Schwaber, 2003

Prozesse im Softwareengineering Kernprozesse

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

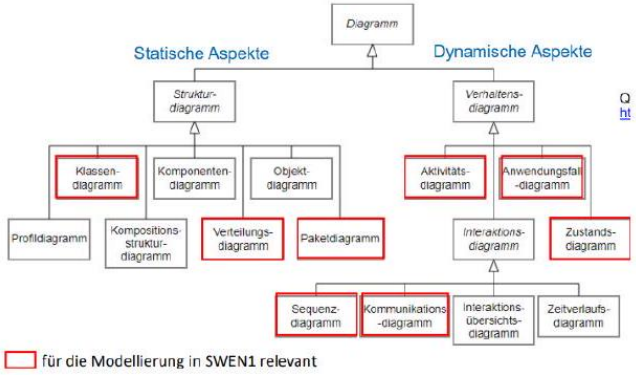
Unterstützungsprozesse

- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

Modelle und Diagramme

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 Codes usw.



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

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

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

Zweck und den Nutzen von Modellen in der Softwareentwicklung

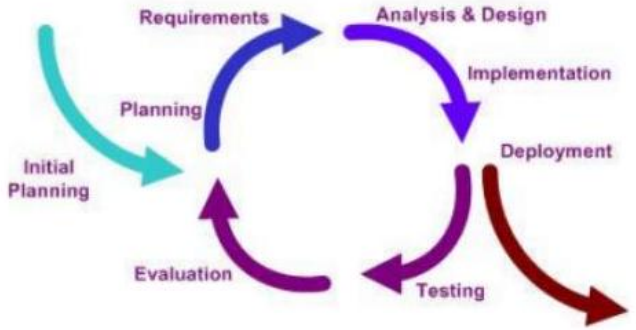
- Modell von Requirements (close to/ far from Agreement) & Technology (known / unknown)
  Ein Modell ist ein konkretes oder gedankliches Abbild eines vorhanden Gebildes oder Vorbild für ein zu schaffendes Gebilde (hier Softwareprodukt).

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)

Incremental Model

Artefakte in einem iterativ-inkrementellen Prozess illustrieren und einordnen



Anforderungsanalyse

2 Vorlesung 02

2.1 wichtigste Begriffe des Usability-Engineering

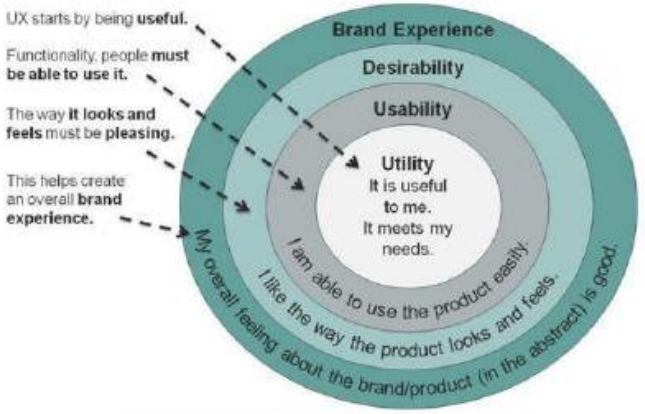


Abbildung 4: Usability und User Experience (UX)
Usability = Deutsch: Gebrauchstauglichkeit
User Experience = Usability + Desirability

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

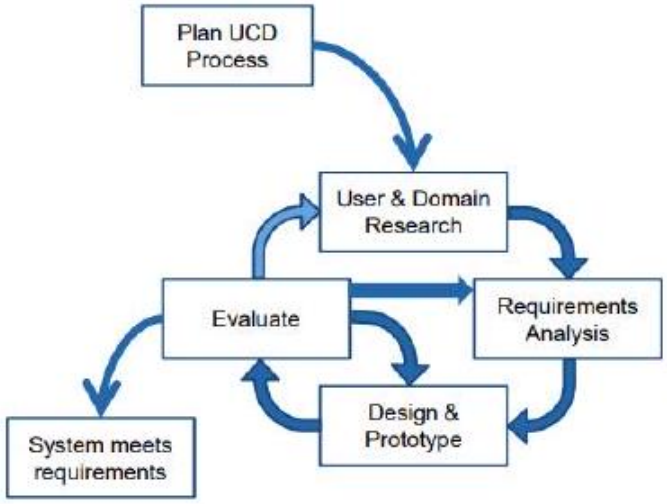


Abbildung 5: Usercentered Design

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

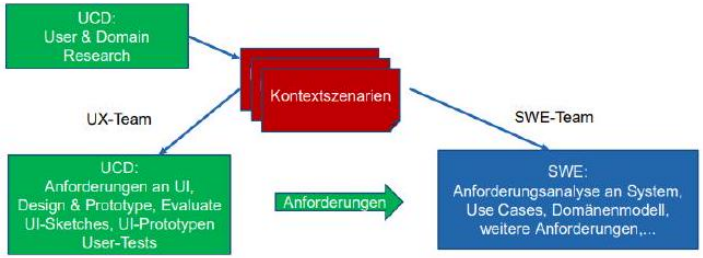


Abbildung 7: Anforderungsanalyse Übersicht

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)
- Initiert 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)

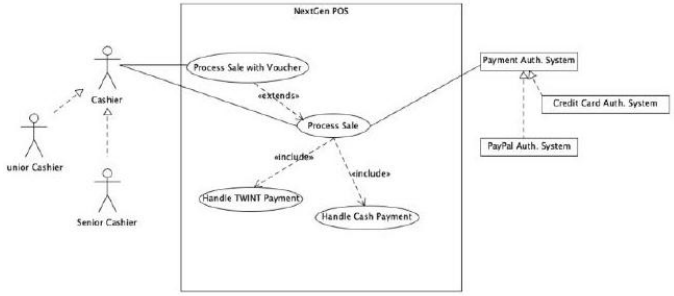


Abbildung 9: Use Case Diagramm

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

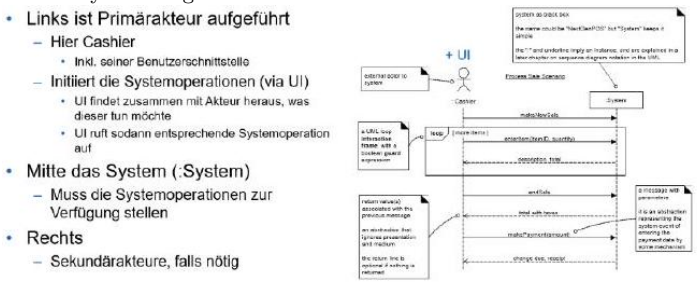


Abbildung 11: Systemsequenzdiagramm (SSD)

Domänenmodellierung

4 Vorlesung 04

4.1 UML Klassendiagramm = Domänenmodell (vereinfachtes UML Klassendiagramm)

Konzepte werden als Klassen modelliert, Eigenschaften als Attribute (ohne Typangabe), Assoziationen mit Multiplizitäten als Beziehung zw. Konzepten (wenn notwendig noch Aggregation (Beschriftung d Pfeile))

4.1.1 Konzepte: Substantive

- Physische Objekte
- Kataloge
- Container von Dingen
- Andere beteiligte Systeme
- Rollen von beteiligten Personen
- Artefakte (Pläne, Finanzen, Arbeit, Verträge)
- Zahlungsinstrumente
- Keine Softwareklassen

4.1.2 Attribute: sollen einfach/wichtig sein

- Transaktion
- Teil zum Ganzen
- Beschreibung/ Protokoll zum Gegenstand
- Verwendung

Attribute an Stelle von Assoziationen Verwenden Sie Assoziationen und nicht Attribute, um Konzepte in Beziehung zueinander zu setzen.

4.2 Analysemuster

4.2.1 Beschreibungsklassen

Artikel, Physischer Gegenstand, Diensleitung: hat Preis, Serie Nummer u Code

4.2.2 Generalisierung / Spezialisierung

Wenn 100% Regel: alle instanzen eines spezialisierten Konzepts sind auch Instanzen des generalisierten Konzepts und ÏS A"
Assoziationen und Attribute dienen umgekehrt als Begründung für eine gemeinsame generalisierte Klasse.

4.2.3 Komposition

4.2.4 Zustände

Sollen durch eigene Hierarchie dargestellt werden

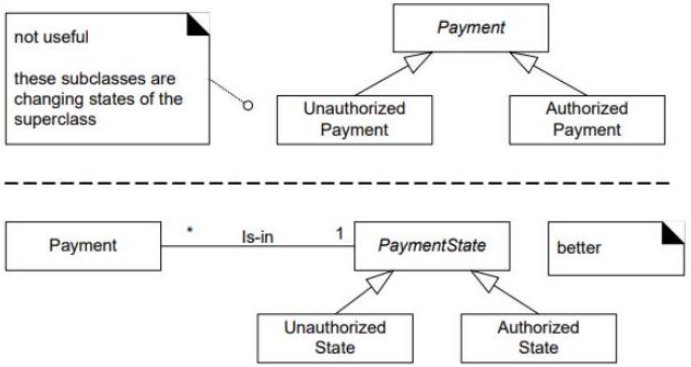


Abbildung 12: Zustände Domänenmodell(DM) Beispiel

4.2.5 Rollen (Manager etc.)

Dasselbe Konzept (aber selten dieselbe Instanz) kann unterschiedliche Rollen einnehmen.
Dargestellt als Konzepte / Assoziationen

4.2.6 Assoziationsklasse

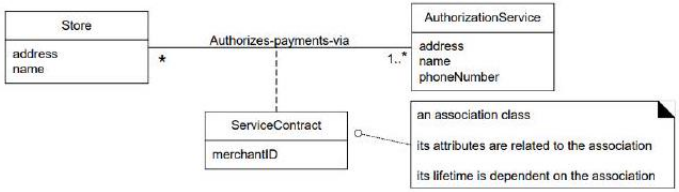


Abbildung 13: Assoziationsklasse Beispiel

4.2.7 Masseinheiten / Zeitintervalle

Oft Sinnvollerweise als Konzept modelliert

Softwarearchitektur und Design

5 Vorlesung 05

5.1 Ubersicht Business Analyse vs Architektur vs Entwicklung

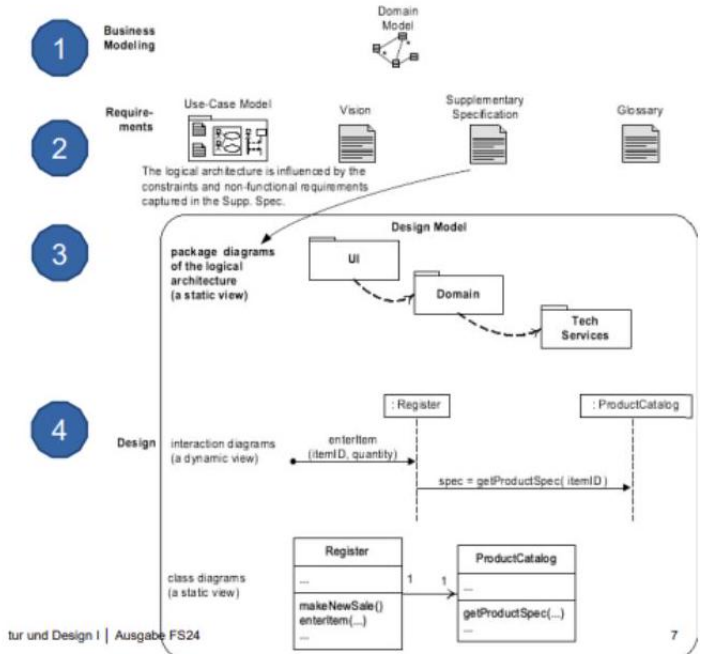


Abbildung 14: Übersicht Business Analyse vs Architektur vs Entwicklung
1. Domänenmodell (Business Modelling) Kontext Diagramm (Business Analyst)
2. Requirement (Business Analyst)
3. Logische Architektur (Software Architekt)
4. Umsetzung (Entwicklung)

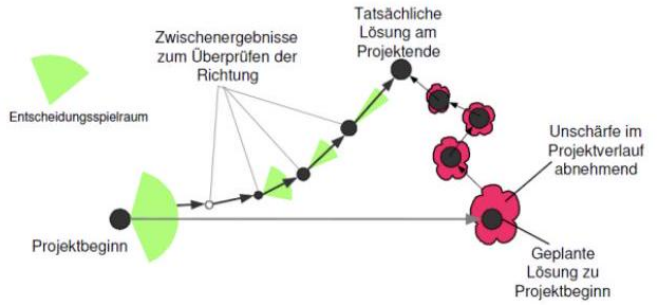


Abbildung 15: Entstehung Archtektur

5.2 Architektur aus Anforderungen

Die Architektur muss heutige und zukünftige Anforderungen erfüllen können und Weiterentwicklungen der Software und seiner Umgebung ermöglichen

5.2.1 Architekturanalyse

Analyse der funktionalen und nichtfunktionalen Anforderungen:

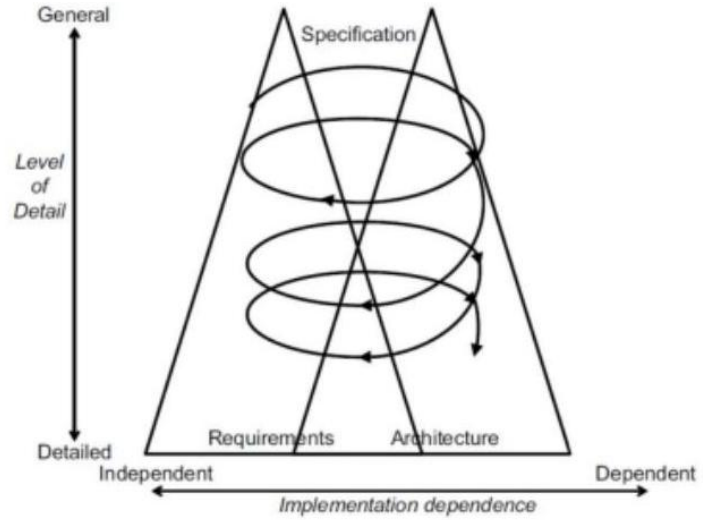


Abbildung 16: Twin Peak Model

Entwurfsentscheidungen sollten in erster Linie aus den Anforderungen abgeleitet werden, Architekturentscheidungen und die Konsequenzen daraus müssen mit den Stakeholdern abgestimmt werden.

5.2.2 ISO 25010

- ISO 25010 provides a hierarchical structure for non-functional requirements.
- It defines main characteristics, sub-characteristics, and metrics.
- Each non-functional requirement in ISO 25010 is associated with metrics.
- Metrics include a description of the requirement, a measurement method to check requirement fulfillment, and guidance for interpreting results.
- This allows for more precise and measurable formulation of requirements, which can be verified later.
  5.2.3 Difference from FURPS + (Functionality, Usiability, Reliability, Performance, Supportability (Anpassungsfähigkeit, Wartbarkeit, etc), + = Implementation, Interface, Operations, Packaging, Legal)
- FURPS+ is an acronym and not a standard.
- FURPS+ includes Functionality, Usability, Reliability, Performance, Supportability, and other terms.

5.2.4 Grundprinzip: Modulkonzept

Modul (Baustein, Komponente): Güte wird gemessen mit Kohäsion und Kopplung

- Möglichst autarkes Teilsystem (wenig Kopplung nach aussen)
- Hat eine klare minimale Schnittstelle gegen aussen
- Software-Modul enthält alle Funktionen und Datenstrukturen, die es benötigt
- Modul kann sein: Paket, Programmierkonstrukt, Library, Komponente, Service

5.2.5 Architektur beschreiben

Architektur umfasst verschiedene Aspekte, die je nach Sichtweise wichtig sind

N+1 View Model + 1 View: Use Cases

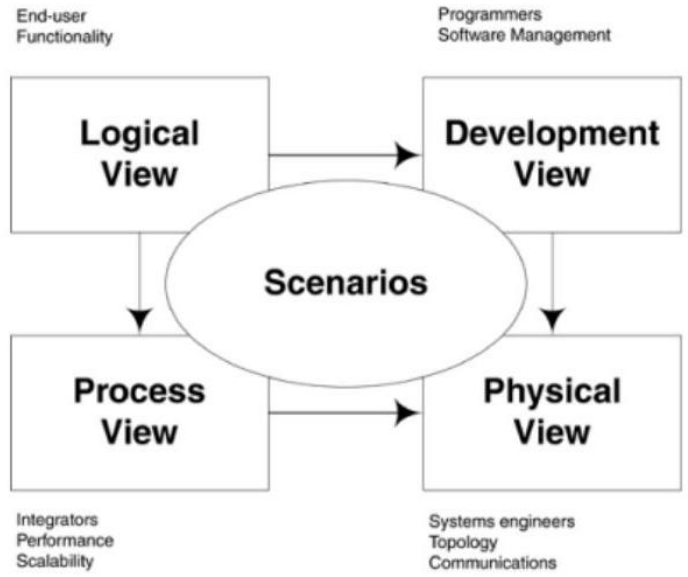


Abbildung 17: View Model

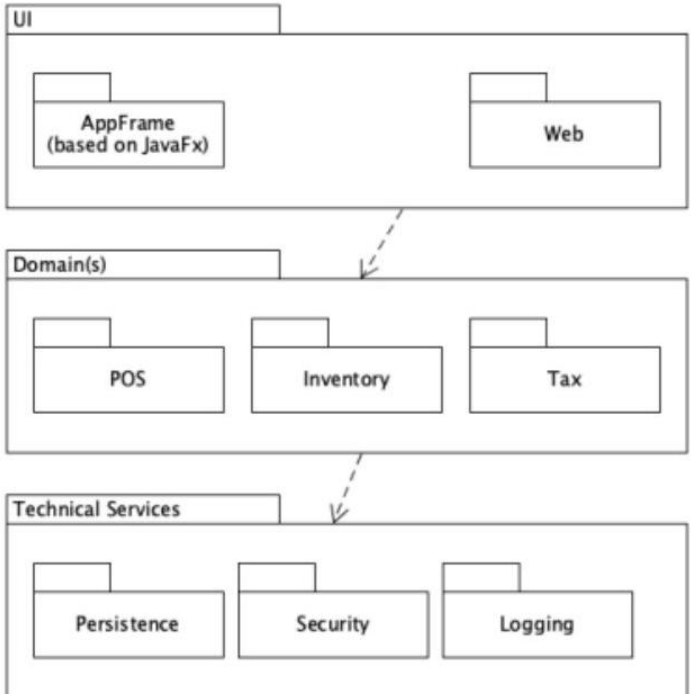


Abbildung 18: UML- Paketdiagram

- Mittel, um Teilsysteme zu definieren
- Mittel zur Gruppierung von Elementen

Paket enthält Klassen und andere Pakete

- Ähnlich, aber allgemeiner als Java Packages

Abhängigkeiten zwischen Paketen

Pattern	Beschreibung
Layered Pattern	Strukturierung eines Programms in Schichten
Client-Server Pattern	Ein Server stellt Services für mehrere Clients zur Verfügung
Master-Slave Pattern	Ein Master verteilt die Arbeit auf mehrere Slaves
Pipe-Filter Pattern	Verarbeitung eines Datenstroms (filtern, zuordnen, speichern)
Broker Pattern	Meldungsvermittler zwischen verschiedenen Endpunkten
Event-Bus Pattern	Datenquellen publizieren Meldungen an einen Kanal auf dem Event-Bus. Datensenken abonnieren einen bestimmten Kanal
MVC Pattern	Eine interaktive Anwendung wird in 3 Komponenten aufgeteilt: Model, View – Informationsanzeige, Controller – Verarbeitung der Benutzereingabe

Abbildung 19: Architekturpatterns

6 Vorlesung 06

6.1 Zweck und Anwendung von Statischen und Dynamischen Modellen im Design

- Statische Modelle, wie beispielsweise das UML-Klassendiagramm, unterstützen den Entwurf von Paketen, Klassennamen, Attributen und Methodensignaturen (ohne Methodenkörper).
- Dynamische Modelle, wie beispielsweise UMLInteraktionsdiagramme, unterstützten den Entwurf der Logik, des Verhaltens des Codes und der Methodenkörper.

Statische u Dynamische ergänzen sich, werden parallel erstellt

6.2 Objektentwurf mit UML-Klassen-, UML-Interaktions-, UML-Zustands- und UML-Aktivitätsdiagrammen

6.2.1 UML-Klassendiagramm

Notationselemente:

- Klasse, aktive Klasse
- Attribut
- Operation
- Sichtbarkeit von Attributen und Operationen
- Assoziationsname, Rollen an den Assoziationsenden
- Multiplizität (Bezieht sich auf die Objekte der betreffenden Klasse)
- Navigierbarkeit in Assoziationen
- Datentypen und Enumerationen
- Generalisierung / Spezialisierung
- Abstrakte Klassen
- Assoziation, Assoziationsklasse: Komposition, Aggregation
- Interface, Interface Realisierung

6.2.2 UML-Interkationsdiagramm

Modellieren die Kollaborationen bzw. den Informationsaustausch zwischen Objekten (Dynamik).

Sequenzdiagramm Notationselemente:

- Lebenslinie
- Aktionssequenz
- Synchrone Nachricht
- Antwortnachricht
- Gefundene, verlorene Nachricht
- Kombiniertes Fragment
- Erzeugungs-, Löschereignis
- Selbstaufruf
- Interaktionsreferenz
- Lebensline mit aktiver Klasse

- Asynchrone Nachricht

Kommuninaktionsdiagramm

- Lebenslinie (Box)
- Synchrone Nachricht (= Aufruf einer Operation) (Nummeriert)
- Antwortnachricht (= Rückgabewert)
- Bedingte Nachrichten «[]»
- Iteration «*»

6.2.3 UML-Zustandsdiagramm

Welche Zustände kann ein Objekt, eine Schnittstelle, ein Use Case, ... bei welchen Ereignissen annehmen?

- Start-, Endzustand
- einfacher Zustand
- Zusammengesetzter bzw. geschachtelter Zustand
- Flache und tiefe Historie
- Transition
- Orthogonaler Zustand
- Parallelisierungsknoten
- Synchronisationsknoten
- Einstiegpunkt
- Ausstiegspunkt
- Unterzustandsautomat

6.2.4 UML-Aktivitätsdiagramm

Wie läuft ein bestimmter Prozess oder ein Algorithmus ab?

- Aktivität
- Aktionsknoten (Aktion)
- Objektknoten (Objekt)
- Entscheidungs- und Vereinigungsknoten
- Kante
- Initialknoten
- Aktivitätsendknoten
- Partition (auch Swimlane genannt)
- Parallelisierungsknoten
- Synchronisationsknoten
- SendSignal-Aktion
- Ereignis- bzw. Zeitereignisannahmeaktion
- CallBehavior-Aktion

6.3 Responsibility Driven Design (RDD)

Denken in Verantwortlichkeiten, Rollen und Kollaborationsbeziehungen für den Entwurf von Softwareklassen.
RDD kann auf jeder Ebene des Designs angewendet werden (Klasse, Komponente, Schicht).
Verantwortlichkeiten werden durch Attribute und Methoden implementiert.

6.4 Prinizpien für Klassenentwurf: GRASP, SOLID

6.4.1 SOLID: Missing from Slides

6.4.2 GRASP (General Responsibility Assignment Software Patterns)

- welche Klassen und Objekte wofür zuständig
- für erleichterung Kommunikation d Entwickler
- grundlegenden Prinzipen bzw. Pattern
- Information Expert: Ein Objekt sollte die Verantwortung für eine Aufgabe übernehmen, wenn es die benötigten Informationen dazu besitzt.
- Creator: Ein Objekt sollte für die Erstellung von anderen Objekten verantwortlich sein, wenn eine starke Beziehung zwischen ihnen besteht.
- Controller: Ein Objekt sollte die zentrale Steuerungslogik in einem System repräsentieren.
- Low Coupling: Objekte sollten lose miteinander gekoppelt sein, um die Flexibilität und Wiederverwendbarkeit des Systems zu erhöhen.
- High Cohesion: Eine Klasse sollte nur zusammengehörige Funktionen und Daten enthalten, um ihre Verständlichkeit und Wartbarkeit zu verbessern.
- Polymorphism: Objekte sollten so entworfen werden, dass sie anhand ihrer Schnittstellen verwendet werden können, unabhängig von ihrer spezifischen Implementierung.
- Pure Fabrication: Künstliche Klassen sollten erstellt werden, um eine hohe Kohäsion und niedrige Kopplung zu erreichen, wenn keine natürliche Klasse die Verantwortung übernehmen kann.
- Indirection: Zwischen Objekten sollten indirekte Verbindungen hergestellt werden, um die Flexibilität und Wartbarkeit zu erhöhen.
- Protected Variations: Mechanismen sollten eingeführt werden, um Variationen in den Systemkomponenten zu schützen und unerwünschte Auswirkungen von Änderungen zu minimieren.

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

Verteilte Systeme

11 Vorlesung 11

11.1 Verteiltes System Definition + Einsatz

Ein verteiltes System besteht aus einer Sammlung autonomer Computer (Knoten) und Softwarebausteinen (Komponenten), die über ein Netzwerk miteinander verbunden sind und gemeinsam als ein einziges System arbeiten. Sie werden in verschiedenen Bereichen eingesetzt, darunter Datenbanken, CloudComputing, verteilte Anwendungen usw.

- Oft sehr gross
- Sehr datenorientiert: Datenbanken im Zentrum der Anwendung
- Extrem interaktiv: GUI, aber auch Batch
- Sehr nebenläufig: Grosse Anzahl an parallel arbeitenden Benutzern
- Oft hohe Konsistenzanforderungen

11.2 Verteiltes System Konzepte + Architekturstil

Verteilte Systeme basieren auf verschiedenen Konzepten und Architekturstilen:

11.2.1 Kommunikationsverfahren

Kommunikationsverfahren umfassen Methoden, mit denen die einzelnen Knoten in einem verteilten System miteinander kommunizieren können. Dazu gehören beispielsweise Remote Procedure Calls (RPC), Message Queuing und Publish-Subscribe-Systeme.

11.2.2 Fehlertoleranz

Fehlertoleranz ist ein wichtiger Aspekt verteilter Systeme, der sicherstellt, dass das System auch bei Ausfällen oder Fehlern in einzelnen Komponenten weiterhin zuverlässig arbeitet. Hierzu werden Mechanismen wie Replikation, Failover und Fehlererkennung eingesetzt.

11.2.3 Fehlersemantik

Die Fehlersemantik beschreibt das Verhalten eines verteilten Systems im Falle von Fehlern oder Ausfällen. Dies umfasst Aspekte wie Konsistenzgarantien, Recovery-Verfahren und Kompensationsmechanismen.

11.3 Design- und Implementierungsaspekte von Client-ServerSystemen

Client-Server-Systeme sind eine häufige Architektur für verteilte Systeme, bei der Clients Anfragen an einen zentralen Server senden, der diese verarbeitet und entsprechende Antworten zurückgibt. Design- und Implementierungsaspekte umfassen unter anderem die Aufteilung von Funktionalitäten zwischen Client und Server, die Wahl der Kommunikationsprotokolle und die Skalierbarkeit des Systems.

11.4 Verteiltes System Architektur + Design Patterns

Die Architektur verteilter Systeme kann durch verschiedene Design Patterns strukturiert werden, um wiederkehrende Probleme effizient zu lösen. Dazu gehören Patterns wie Master-Slave, Peer-to-Peer, Publish-Subscribe, sowie verschiedene Replikations- und Verteilungsstrategien.

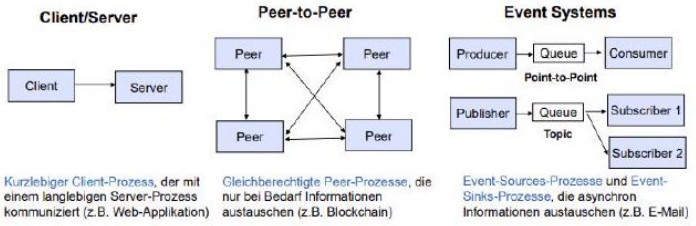


Abbildung 20: Architekturmodelle

11.5 Gängige Technologien (Middleware) f. Informationssysteme und Internet-basierte Systeme

Für die Entwicklung verteilter Systeme stehen verschiedene Middleware-Technologien zur Verfügung, die die Kommunikation und Integration von verteilten Komponenten erleichtern. Dazu gehören Messaging-Broker wie Apache
Kafka, Middleware-Frameworks wie CORBA (Common Object Request Broker Architecture) und RESTful Web Services.

Persistenz

Um was geht es?

- Wie kann ich meine Java Objekte dauerhaft speichern?

Java

- Welche Arten von Datenspeicherung gibt es?
- Welche Design Patterns stehen für die Realisierung von Persistenz in einer Applikation zur Verfügung?
- Wie kann ich mit Hilfe von den Java APIs JDBC (Java Database Connectivity) und JPA (Java Persistence API) meine Objekte dauerhaft in einer Datenbank speichern?

Applikation

`DB API`

Treiber

Lernziele LE 12 - Persistenz

- Sie sind in der Lage
- die Varianten der Datenspeicherung zu nennen,
- die unterschiedlichen Design Patterns für die Persistenz zu erklären,
- mit Hilfe des Design Patterns DAO (Data Access Object) und JDBC eine Persistenz in Java umzusetzen,
- mit JPA ein Objekt-Relationales-Mapping (O/R-Mapping) in Java anzuwenden.

1. Einführung in Persistenz

2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapping mit DAO
5. O/R-Mapping mit JPA
6. Wrap-up und Ausblick

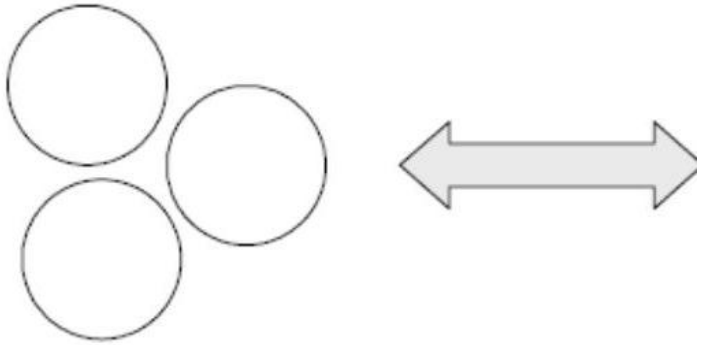
Problemstellung Persistenz (1/2)

- In vielen Applikationen müssen unterschiedliche Daten verarbeitet, verwaltet und dauerhaft, d.h. über das Programmende hinaus gesichert werden.
- Letzteres bezeichnet man als Persistenz.
- Die dauerhafte Speicherung erfolgt in Datenbankmanagementsystemen (DBMS).
- Übliche Datenbanksysteme sind sogenannte Relationale Datenbanksysteme (RDBMS) und sogenannte NoSQL-Datenbanken.
- NoSQL-Datenbanken speichern Daten ohne fixes Schema und in verschiedenen Formaten (Dokument Stores, Key-Value Stores, Graph DB, ...).

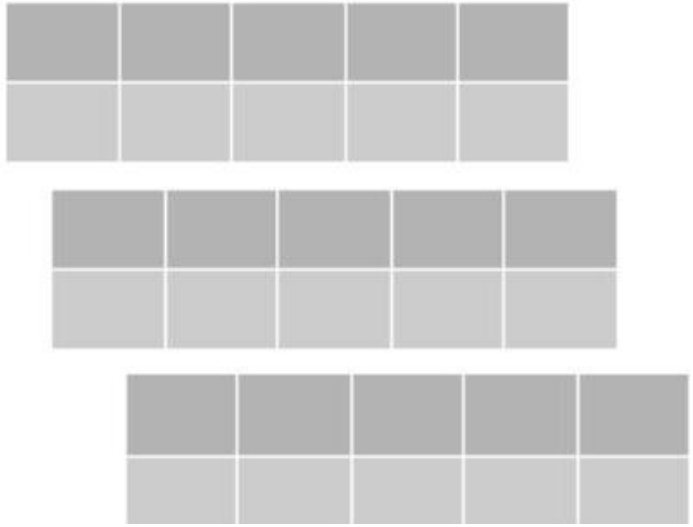
Problemstellung Persistenz (2/2)

- Die Abbildung zwischen Objekten und Datensätzen in Tabellen einer relationalen Datenbank wird auch als O/R-Mapping (Object Relational Mapping, ORM) bezeichnet.
- Verhältnismässig viel Java-Code wird benötigt, um die Datensätze des Ergebnisses zu verarbeiten und in Java-Objekte zu transformieren.
- Es besteht ein Strukturbruch (engl. Impedance Mismatch) aufgrund der unterschiedlichen Repräsentationsformen von Daten (flache Tabellenstruktur -Java-Objekte).

Objektorientiertes Programm



Relationale Datenbank



Denkpause

Aufgabe 12.1 (5')

Diskutieren Sie in Murmelgruppen folgende Frage:

- Was ist aktuell die vorherrschende Technologie zum Speichern von Daten im Enterprise-Umfeld?
- Recherchieren Sie dazu unter https://db-engines.com/en/ranking.
- Was sind die Gründe für dieses Ranking?

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapper mit DAO
5. O/R-Mapper mit JPA
6. Wrap-up und Ausblick

Herausforderung: Der O/R-Mismatch (1/2)

- Der O/R-Mismatch ist ein Fakt.
- Der O/R-Mismatch folgt aus konzeptionellen Unterschieden der zugrundeliegenden Technologien.
- Es gibt viele verschiedene Möglichkeiten (Patterns) den O/R-Mismatch zu überwinden.
- Active Record, O/R-Mapping resp. O/R-Mapping Frameworks oder Repositories (aus Domain Driven Design, DDD) sind ein möglicher Lösungsansatz.

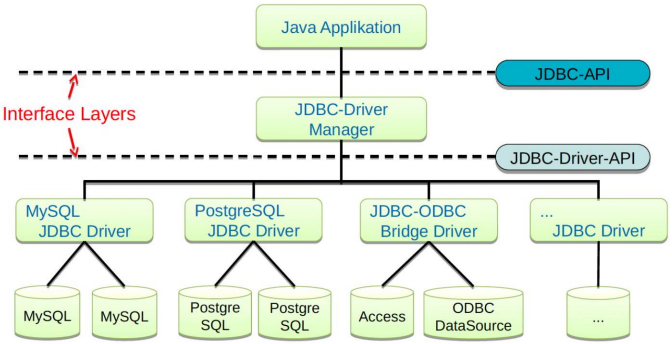
Herausforderung: Der O/R-Mismatch (2/2)

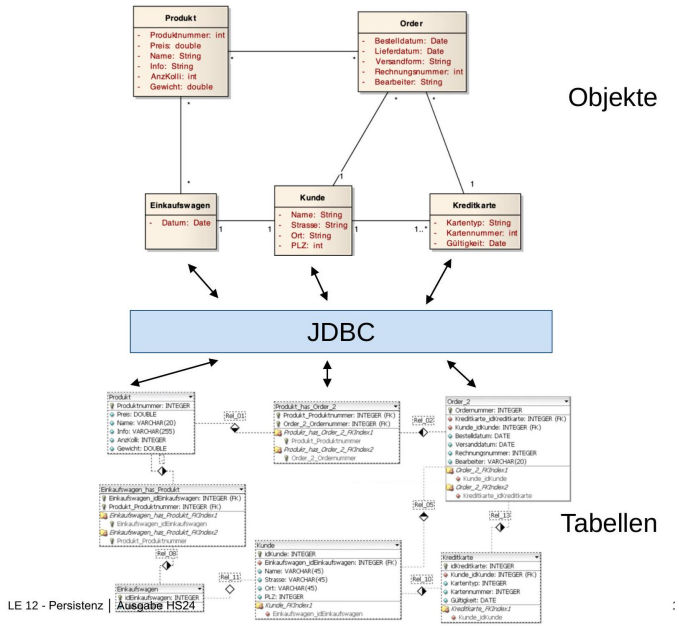
- Typen-Systeme
- Null
- Datum/Zeit
- Abbildung von Beziehungen
- Richtung
- Mehrfachbeziehungen
- Vererbung
- Identität
- Objekte haben eine implizite Identität
- Relationen haben eine explizite Identität (Primary Key)
- Transaktionen

JDBC: Java Database Connectivity (1997)

School of Engineering

- JDBC verbindet die Objektwelt mit der relationalen Datenbankwelt
- Herausforderung: Objekte vs. Tabellen,
- Verschiedene Datentypen etc.
- Die Programmierung ist aufwändig

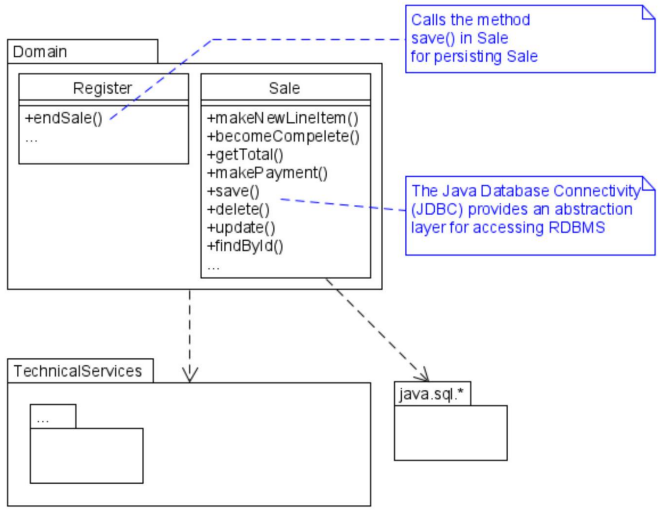




Design Pattern für Persistenz

Für eine Persistenz-Strategie muss eine Entscheidung getroffen werden, wo die Zuordnung (Mapping) zwischen Objekten und Tabellen stattfinden soll:

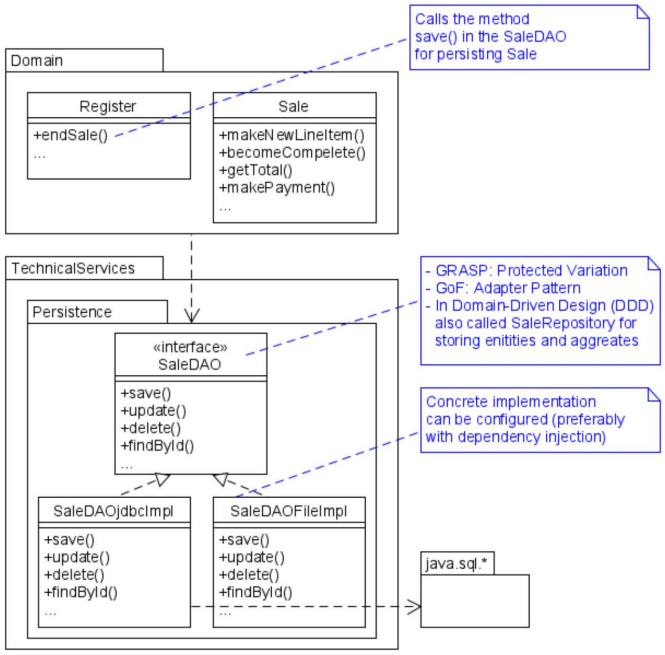
- Active Record (Anti Pattern): Jede Entität ist selber dafür zuständig
- Data Access Object (DAO): Abstrahiert und kapselt den Zugriff auf die Datenquelle
- O/R Data Mapper: Separate Klasse für das Mapping oder Einsatz eines ORM
- Zugriffscode auf Datenbank ist in der Domänenklasse
- Wrapper für eine Zeile einer Datenbanktabelle
- Spiegelt die Datenbankstruktur
- Enthält Daten und Verhalten
- Fachlichkeit und Technik alles in einer Klasse (GRASP: Information Expert?)
- Schlechte Testbarkeit der Domänenlogik ohne Datenbank



- Schlechte Wartbarkeit und Erweiterbarkeit (No separation of concerns!)

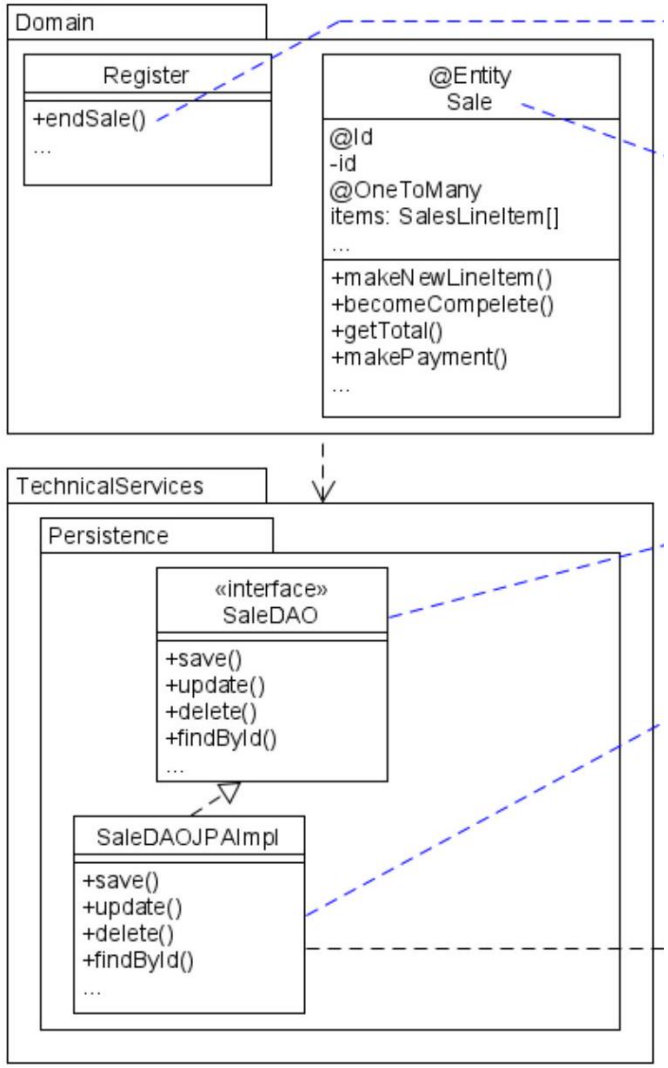
O/R-Mapping von Hand mit Hilfe eines Data Access Objects (DAO)

- Trennung von Fachlichkeit und Technik (Domänenklasse hat hohe Kohäsion)
- Gute Testbarkeit und Mocking der Persistenz
- Bevorzugtes Design ohne Einsatz eines O/RMappers



Verwendung eines O/R-Mappers (JPA mit Hibernate/Eclipselink o.ä. Framework)

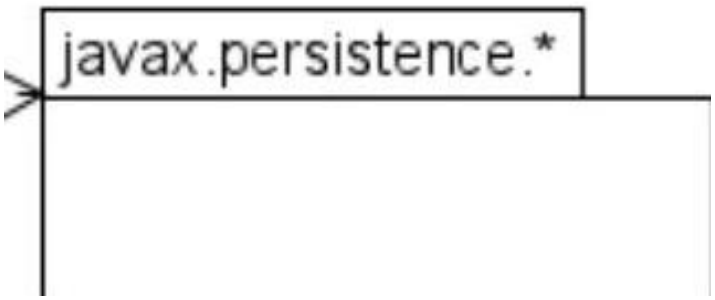
- Viel weniger Aufwand bzw. Code und standardisierte Schnittstelle
- Trennung von Fachlichkeit und Technik (Domänenklasse hat hohe Kohäsion)
- Gute Testbarkeit und Mocking der Persistenz
- DAO ist auch beim Einsatz von JPA empfehlenswert (Trennung von Fachlichkeit und Technik) - JPA könnte aber auch ohne DAO verwendet werden



Calls the method save() in the SaleDAO for persisting Sale

`Same design pattems apply as for`

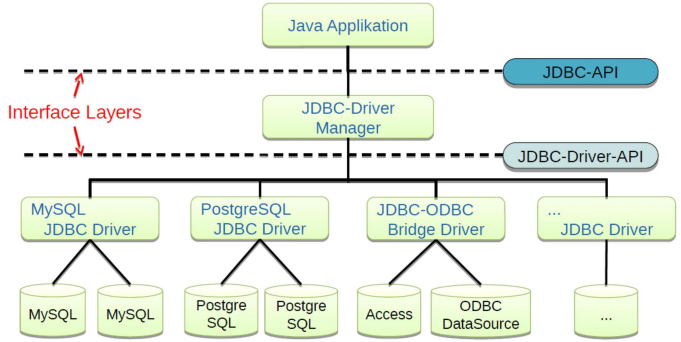
the option with handcrafted DAOs.
For implementing the DAO
the Java Persistence API the Java Persistence API (JPA) and a service provider (e.g. Hibernate) provide an orR mapping and the basic by using the annotations and a configuration file (persistence xml)For queries a Java Persistence Query Language (JPQL) is available.



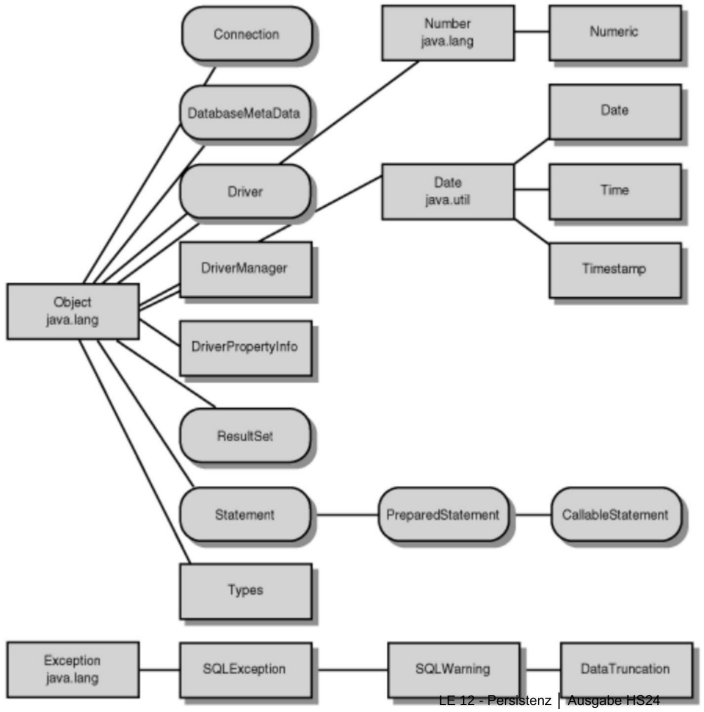
1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapper mit DAO
5. O/R-Mapper mit JPA
6. Wrap-up und Ausblick

Was genau ist JDBC?

- JDBC = Java Data Base Connectivity
- Standardisierte Schnittstelle, um auf relationale Datenbanken mit Hilfe von SQL zuzugreifen
- Cross-Plattform und DB-independent
- JDBC-API ist Teil der Java-Plattfrom seit JDK1.1 (1997)
- Die aktuelle Version ist 4.2



JDBC-API: Interfaces and Classes



Anwendung von JDBC

Basisanweisungen:
1. Install and load JDBC driver
2. Connect to SQL database
3. Execute SQL statements
4. Process query results
5. Commit or Rollback DB updates
6. Close Connection to database

```
import java.sql.*;
public class DbTest {
    public static void main(String[] args)
        throws ClassNotFoundException, SQLException {
        Connection con = DriverManager.getConnection(
            "jdbc:postgresql://test.zhaw.ch/testdb",
            "user", "password");
        Statement st = con.createStatement();
        ResultSet rs = st.executeQuery(
            "SELECT * FROM test ORDER BY name");
        while (rs.next()) {
            System.out.println(
                "Column 1 contains '" +
                rs.getString(2) +"'");
        }
        con.close();
    }
}
```

1. Einführung in Persistenz 2. Design-Optionen für Persistenz 3. Persistenz mit JDBC 4. O/R-Mapping mit DAO 5. O/R-Mapping mit JPA 6. Wrap-up und Ausblick

O/R-Mapping Pattern

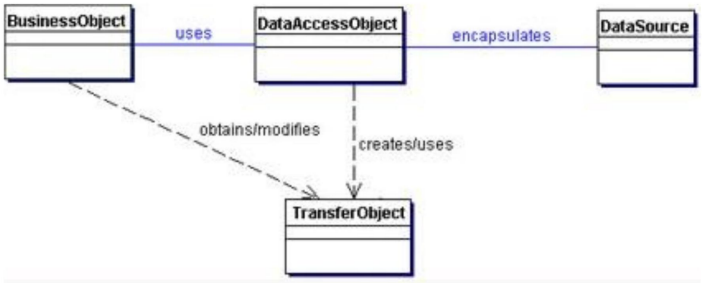
Es sollen beide Varianten des O/R-Mapper Patterns anhand eines praktischen Beispiels betrachtet werden: - DAO (Data Acess Object) ohne ein ORM (Object Relational Mapper) - Umsetzung von DAO mit Hilfe von JPA (Java Persistence API)

DAO - Data Access Object Pattern

- Das Artikel-Objekt repräsentiert das Domain-Model-Objekt. - Die Verbindung zur Datenbank wird durch das DAO sichergestellt. - Enthält die üblichen CRUD-Methoden wie create, read, update und delete. - Kann auch Methoden enthalten wie findAll, findByName, findByID um eine Kollektion von Daten aus der Datenbank abzufragen.

DAO - Data Access Object Pattern

School of



Sun Developer Network - Core J2EE Patterns
http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

Das TransferObject aka. DTO kann zusätzlich für den Transport der Daten in einem verteilten System verwendet werden.

Beispiel Article und ArticleDAO

School of Engineering

Business Object

```
public class Article {
private long id;
private String name;
private float price;
public long getId(){
return id;
}
public void setId(long id){
this id = id
};
...
}


\section*{Data Access Object (DAO)}

//Interface to be implemented by all ArticleDAOs
public interface ArticleDAO {
public void insert(Article item);
public void update(Article item);
public void delete(Article item);
```

public Article findById(int id);
public Collection<Article> findAll();
public Collection<Article> findByName (String name);
public Collection<Article> findByPrice (float price);
}

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapping mit DAO
5. O/R-Mapping mit JPA
6. Wrap-up und Ausblick

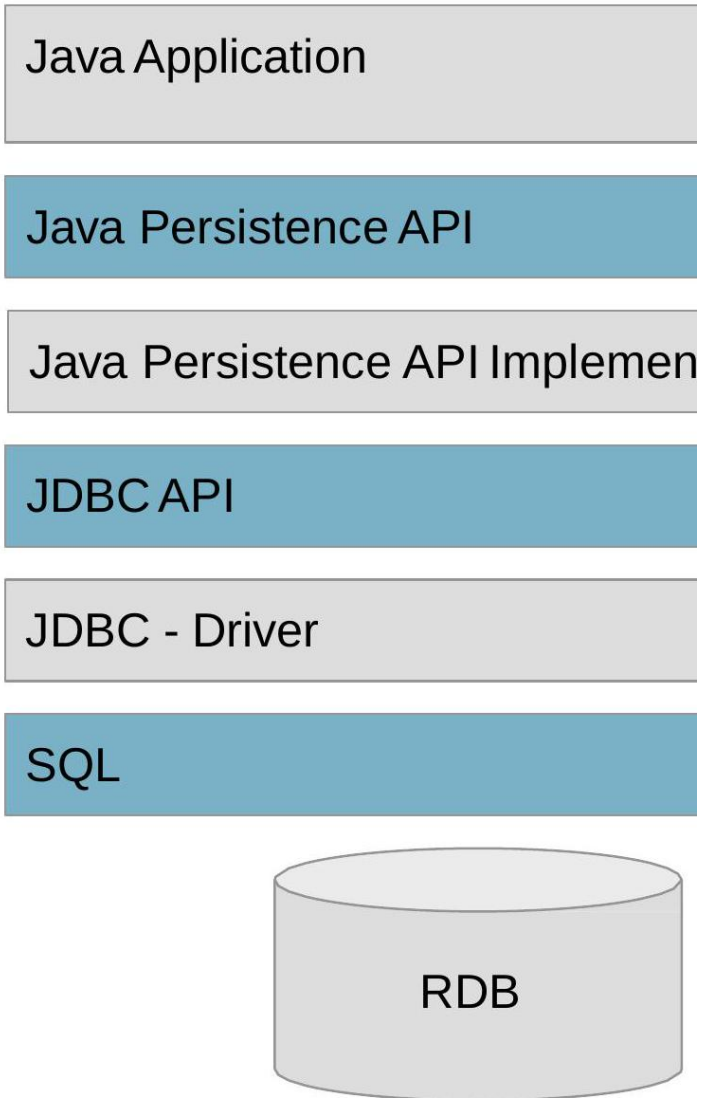
Versprechen von automatischem O/R-Mapping

- Die Applikation wird von der DB entkoppelt - Applikationsentwickler muss kein SQL beherrschen. - Das relationale Modell der Datenbank hat keinen Einfluss auf das OO-Design. - Automatische Persistenz - Automatisierte Abbildung der Objekte in die relationalen Strukturen. - Die Applikationsentwickler muss sich nicht um die «low-level»-Details kümmern. - Transparente Persistenz / Persistence Ignorance - Die Klassen des Domain-Models wissen nicht, dass sie persistiert und geladen werden können und haben keine Abhängigkeit zur Persistenz-Infrastruktur. - JPA ist ein Java Standard für O/R-Mapping - Verschiedene Implementationen, Hibernate vermutlich die bekannteste

JPA (Java Persistence API) Überblick

- Es folgt eine kurze, unvollständige Auflistung der wichtigsten Konzepte von JPA. - Starke Entkopplung der Anwendungslogik von der (relationalen) Datenbank. - Die Domänenklassen sind ganz normale Java Klassen (POJO) - Ausser Annotationen enthalten Sie keinen JPA spezifischen Code. - Referenzen - Werden entweder mit der referenzierenden Klasse (eager loading) oder erst, wenn die Referenz gebraucht wird (lazy loading), geladen. - Referenzen können direkt traversiert werden, JPA erledigt das Laden des referenzierten Objekts im Hintergrund. - Transaktionshandling und das Absetzen von Queries müssen über JPA spezifische Klassen abgewickelt werden. - EntityManagerFactory, EntityManager, EntityTransaction

Technologie-Stack



Java 5+
JPA Spezifikation EclipseLink (TopLink), Hibernate, OpenJPA etc.
JDBC 4.0 Herstellerspezifisch SQL (und Dialekte)

Entity Metadata

- Kennzeichnung mit Annotation @Entity oder Mapping mit XML - Klasse kann Basisklasse oder abgeleitet sein - Klasse kann abstrakt oder konkret sein - Serialisierbarkeit ist bezüglich Persistenz nicht erforderlich

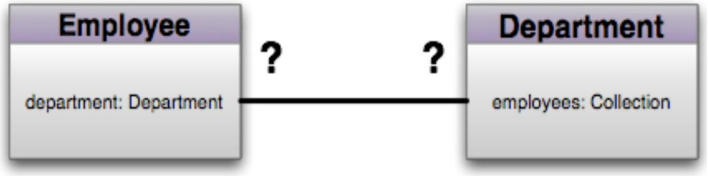
Beispiel Entity

School of - Minimale Anforderung an eine Entity
@Entity
public class Employee {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
private long id;
private String name;
private String lastName; }

- Primärschlüssel können in Zusammenarbeit mit der Datenbank gener

@Entity public class Employee {
@Id
@GeneratedValue(strategy = GenerationType.IDENTITY)
public Integer id;
}
public class Employee {
@TableGenerator(name = Ëmp_Gen", table = ÏD_GEN", pkColumnName = "GEN_NAME",
valueColumnName = "GEN_VAL")
@Id
@GeneratedValue(strategy = GenerationType.TABLE, generator = Ëmp_Gen")
private int id; }

Parent-Child Beziehung



- Mapping des Klassenmodells auf das DB-Schema mittels JPA: Metadata ist erforderlich. - Je nach Klassenmodell wird entweder eine many-to-one Beziehung oder eine one-to-many Beziehung gemappt. - Falls beide Richtungen gemappt werden sollen, so muss definiert werden, dass für beide derselbe Foreign-Key zugrunde liegt.

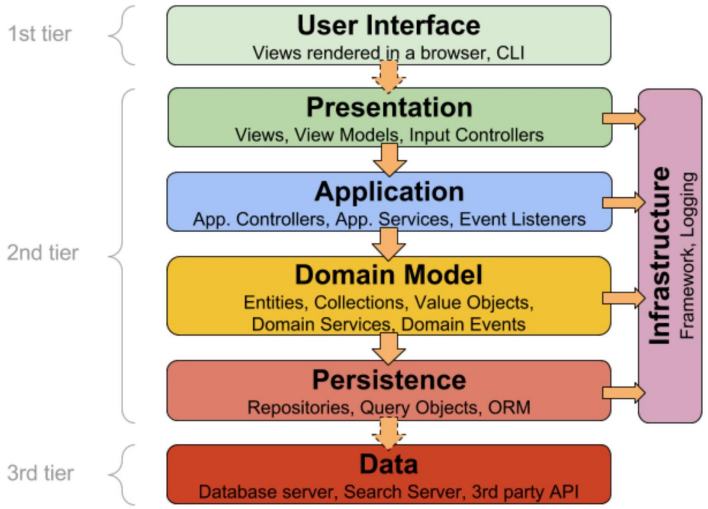
Employee
id
name
salary
department id

Ausblick Design Pattern Repository

- Ein System mit einem komplexen Domänen-Model profitiert wie vorher beschrieben von einer Data-Mapper-Schicht (mit JPA und DAOs), um die Details des Datenbankzugriffcodes zu isolieren. - Eine zusätzliche Abstraktionsschicht oberhalb des Data-Mappers kann helfen um die Konstruktion von Datenbank-Abfragen (Queries) an einem Ort zu konzentrieren. - Diese zusätzliche Schicht wird um so wichtiger je mehr Domänen-Klassen vorhanden sind, die viele Zugriffe auf die Datenbank vornehmen. - Die zusätzliche Schicht wird als Repository bezeichnet - Das Konzept stammt aus Domain Driven Design, DDD (Eric Evans). - Wird in den Wahlpflichtmodulen ASE1/2 anhand von Spring Data behandelt.

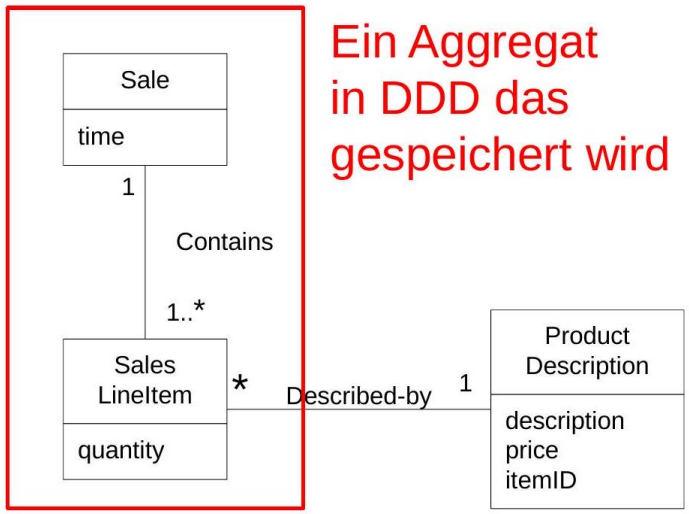
Design Pattern Repository: Schichtenmodell

- 3-Tier Architecture - Persistenz kann mittels Repositories umgesetzt werden



Idee und Beispiel Repository Pattern

School of Engineering InIT Institut für angewandte Informationstechnologie - Eine Repository vermittelt zwischen Domänen- und Data-Mapping Schicht
9 0/** * The GRASP controller for the use case process sale.
11 */
\emptysetpublic class ProcessSaleHandler {
} private ProductDescriptionRepository catalog;
private SaleRepository saleRepository;
private Sale currentSale;
public ProcessSaleHandler(ProductDescriptionRepository catalog, SaleRepository saleRepository) {
} public void makeNewSale() {
} public void enterItem(String id, int quantity) {
} public Money getTotalOfSale() {
} @Transactional
public void endSale() {
assert(currentSale != null && !currentSale.isComplete());
this.currentSale.becomeComplete()
this.saleRepository.save(currentSale);
}
public Money getTotalWithTaxesOfSale() {
} public void makePayment() {



/**
* Repository for Sale
* An implementation of CRUD and common search methods
* is automatically generated by Spring Data.
*/
6 @Repository
\emptysetpublic interface SaleRepository extends CrudRepository<Sale, String>
public List<Sale> findOrderByDateTime();
public List<Sale> findByDateTime(final LocalDateTime dateTime);
L}

1. Einführung in Persistenz
2. Design-Optionen für Persistenz
3. Persistenz mit JDBC
4. O/R-Mapping mit DAO
5. O/R-Mapping mit JPA
6. Wrap-up und Ausblick

- Viele Applikationen verlangen, dass Daten dauerhaft gesichert werden müssen nach dem Programmende.
- Bei kleineren Applikationen kann diese Persistenz auch selber ausprogrammiert werden.
- Dabei sollte aber das Design Pattern Data Access Object (DAO) oder Repository angewendet werden.
- Für grössere Applikationen werden heute sogenannte O/R-Mapper eingesetzt.
- Java bietet mit dem Java Persistence API (JPA) eine standardisierte Schnittstelle für das O/R-Mapping, für die es viele Provider gibt.

Framework Design

Um was geht es?

- Ein Framework ist ein Programmiergerüst, das dem Anwendungsprogramm einen Rahmen gibt und wiederverwendbare Funktionalität zur Verfügung stellt.
- Es bietet gezielt Orte an, wo es erweitert oder angepasst werden kann.
- In Frameworks kommen gewisse Design Patterns zum Einsatz.
- Frameworks werden heutzutage sehr häufig eingesetzt.

Lernziele LE 13 - Framework Design

- Sie sind in der Lage:
- die Eigenschaften von Frameworks zu nennen,
- Design Patterns im Einsatz von Frameworks anzuwenden,
- Prinzipen von modernen Frameworks zu verstehen,
- die Auswahl und den Gebrauch von Frameworks kritisch einzuschätzen.

1. Einleitung und Definition

2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

Framework Charakterisierung

- Leider gibt es keine allgemein akzeptierte exakte Definition eines Frameworks und der Begriff wird für viele Programmbibliotheken eingesetzt.
- Für unsere Zwecke möchten wir den Begriff folgendermassen abgrenzen:
- Ein Framework enthält keinen applikationsspezifischen Code.
- Ein Framework gibt aber den Rahmen («Frame») des anwendungsspezifischen Codes vor.
- Die Klassen eines Frameworks arbeiten eng zusammen, dies im Gegensatz zu einer reinen Klassenbibliothek wie z.B. die Java Collection Klassen.
- Ein Framework muss für den Einsatz gezielt erweitert und/oder angepasst werden.
- Applikations-Container wie z.B. Spring Framework oder Java EE (neu Jakarta EE) schliessen wir ebenfalls ein.
- Die Entwicklung eines neuen Frameworks ist eine aufwändige Angelegenheit.
- Wiederverwendbare Software (und dazu gehören natürlich Frameworks) sollte ein höheres Level im Bereich Zuverlässigkeit besitzen, was ebenfalls mit zusätzlichem Aufwand verbundenen ist.
- Erweiterbare Software (und dazu gehören natürlich Frameworks) erfordert eine tiefergehende Analyse darüber, welche Teile erweiterbar sein sollen, was zu einem höheren Architektur- und Designaufwand führt.
- Eigentlich sprechen alle diese Punkte dagegen, selber ein Framework zu entwickeln. Weshalb wird dies trotzdem behandelt?

Framework Einsatz und Entwicklung erweiterbarer Software

- Alle Design Patterns, die wir heute behandeln, können für die Entwicklung erweiterbarer Software eingesetzt werden.
- Dies muss nicht zwingend ein Framework sein, das auf GitHub publiziert wird, sondern es kann auch einfach eine Komponente sein, die in mehreren eigenen Anwendungen in verschiedenen Kontexten eingesetzt wird.
- Das Wissen um den Aufbau von Frameworks hilft auch, deren Einsatz, aber auch deren Grenzen zu verstehen.

Kritische Bemerkungen zu Frameworks

- Frameworks tendieren dazu, im Laufe der Zeit immer mehr Funktionalität zu «sammeln».
- Was auf den ersten Blick positiv scheint, kann im zweiten Blick zu inkonsistentem Design und funktionalen Überschneidungen führen, die den Einsatz immer mehr erschweren.
- Der Einsatz eines Frameworks sollte gut überlegt werden.
- Einerseits erfordert dies gute Kenntnisse des Frameworks, andererseits ist nach der «Verheiratung» der Anwendung mit dem Framework eine «Scheidung» nur noch schwierig und mit hohem Aufwand möglich.
- Allenfalls sollte das Framework nur über eigene Schnittstellen verwendet werden (keine direkte Abhängigkeit), was aber unter Umständen

die Nützlichkeit des Einsatzes in Frage stellt.

1. Einleitung und Definition
2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

Recap: Aufbau Design Patterns

- Beschreibungsschema
- Name
- Beschreibung Problem
- Beschreibung Lösung
- Hinweise für Anwendung
- Beispiele

Recap: Anwendung von Design Patterns

- Design Patterns sind ein wertvolles Werkzeug, um bewährte Lösungen für wiederkehrende Probleme schnell zu finden.
- Sie helfen, im Team effizient über Lösungsmöglichkeiten zu kommunizieren.
- Ihre Anwendung stellt aber immer einen Trade-Off zwischen Flexibilität und Komplexität dar.
- Es ist keineswegs so, dass ein Programm automatisch besser wird, wenn mehr Patterns angewendet werden.

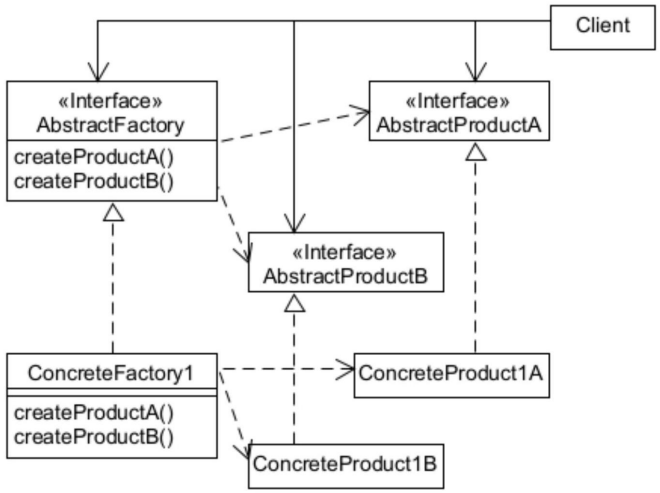
Design Patterns

- Abstract Factory
- Factory Method
- Command
- Template Method

Abstract Factory: Problem und Lösung

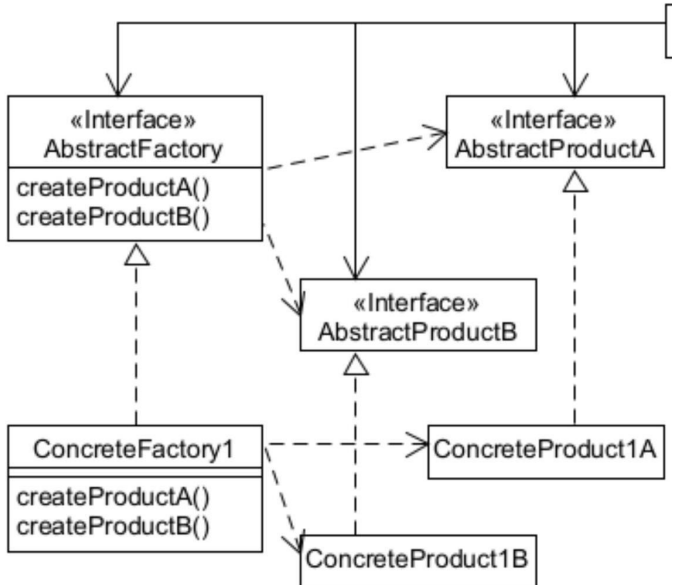
- Problem

- Die Erzeugung verschiedener, inhaltlich zusammengehörender Objekte («Product»), ohne aber die konkreten Klassen zu kennen, damit diese austauschbar sind.
- Lösung
- Eine AbstractFactory und abstrakte «Products» definieren.
- Die AbstractFactory hat für jedes «Product» eine eigene «create» Methode.
- Eine konkrete Factory davon ableiten, die dann konkrete «Products» erzeugt.



Abstract Factory: Hinweise

- Hinweise
- Eigentlich «nur» eine Verallgemeinerung einer «SimpleFactory».
- Die verschiedenen Produkte hängen inhaltlich miteinander zusammen, zum Beispiel verschiedene Teile der anzusteuernden Hardware.



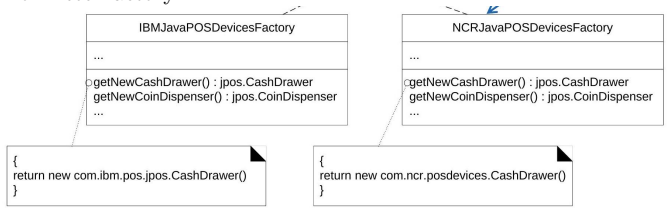
Abstract Factory: Beispiel Point Of Sale (POS) Terminal

School of Engineering

- Die elektronische Kasse muss Hardware wie z.B. die Notenschublade oder den Münzspender ansteuern.
- Typischerweise kommen die einzelnen Komponenten vom selben Hersteller.
- Pro Hersteller gibt es eine konkrete Implementation von IJavaPOSDevicesFactory.

```
this is the Abstract
Factory--an interface fo
creating a family of
related objects
```

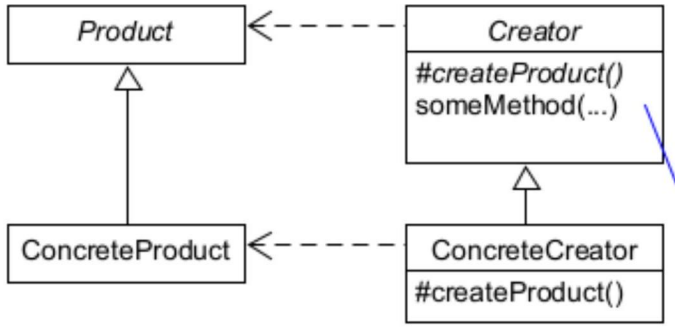
Abstract Factory ↙
Konkrete Factory



com.ncr.posdevices.CashDrawer
isDrawerOpened()

Factory Method: Problem und Lösung

- Problem
- Eine (wiederverwendbare) Klasse Creator hat die Verantwortlichkeit, eine Instanz der Klasse Product zu erzeugen. Es ist aber klar, dass Product noch spezialisiert werden muss.
- Lösung
- Eine abstrakte Methode in der Klasse Creator definieren, die als Resultat Product zurückliefert.
- Konkrete Klassen von Creator können dann die

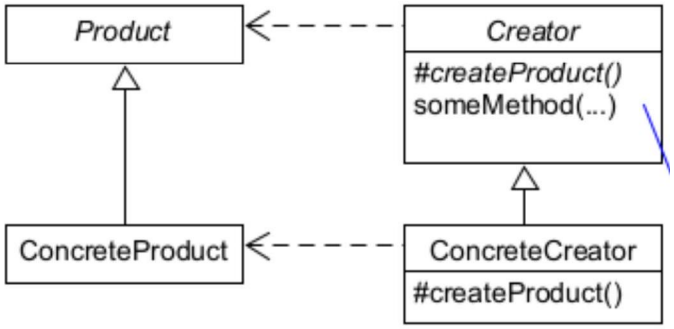


richtige Subklasse von Product erzeugen.

Factory Method: Hinweise

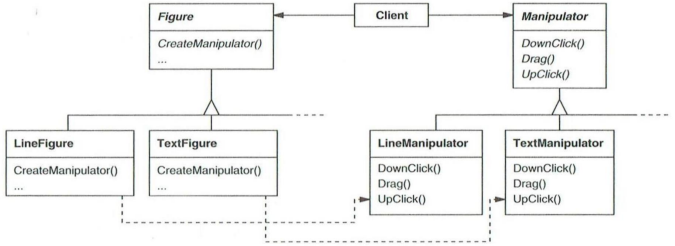
- Hinweise
- Es ist durchaus erlaubt, dass bereits Creator und Produkt konkret sind und somit eine Basisfunktionalität zur Verfügung stellen.
- Es gibt parallele Vererbungshierarchien mit Creator wie auch Product an der Spitze.

- Kann auch als Variante des Design Patterns «Template Method» interpretiert werden.



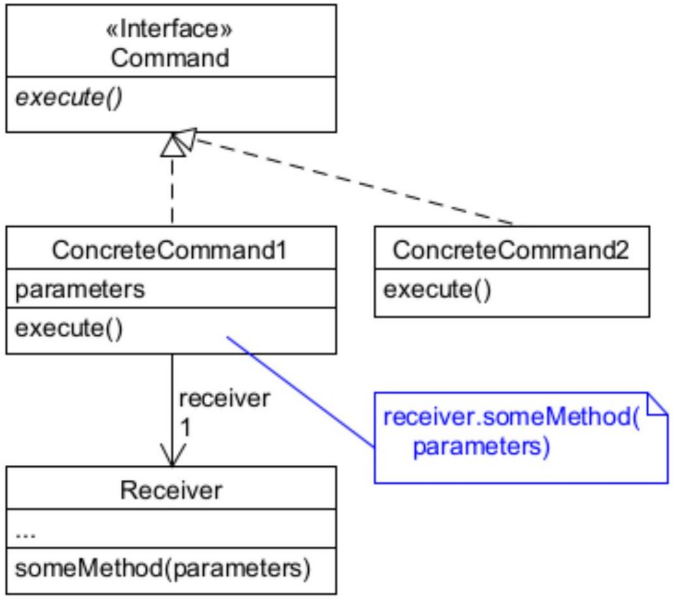
Factory Method: Beispiel GoF (2/2)

- Das Zeichenprogramm («Client») besitzt eine KlassenHierarchie von Figuren.
- Um Figuren übers UI verändern zu können, gibt es eine abstrakte Manipulator Klasse.
- Jede konkrete Figur hat nun die Aufgabe, seine Manipulator Klasse zu instanziieren.



Command: Problem und Lösung

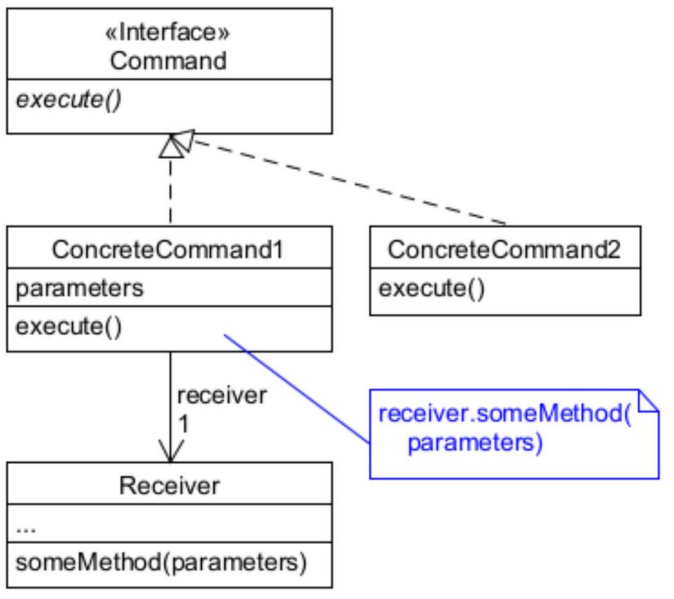
- Problem
- Aktionen müssen für einen späteren Gebrauch gespeichert werden und dabei können sie noch allenfalls priorisiert oder protokolliert werden und/oder Unterstützung für ein Undo anbieten.
- Lösung
- Ein Interface wird definiert, das nur die Auslösung der Aktion erlaubt.
- Implementationen dieses Interface überschreiben die Methode zur Auslösung der Aktion.
- Meistens bedeutet die Aktion, dass eine Methode auf



einem anderen Objekt aufgerufen wird.
- Dazu muss die Aktion die Parameter dieser Methode zwischenspeichern.

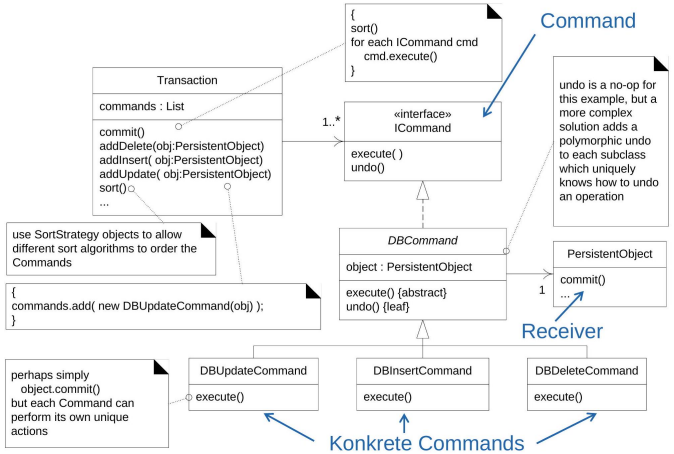
Command: Hinweise

- Hinweise
- Erstellung der Aktion und das Auslösen liegen zeitlich auseinander.
- Bevor Aktionen ausgelöst werden, können sie bei Bedarf noch sortiert oder priorisiert werden. Denken Sie dabei an eine Datenbank.
- Der Receiver muss nicht zwingend über eine Assoziation sichtbar sein. Es ist auch ein Lookup über z.B. einen Namen denkbar.
- Falls eine Rückabwicklung der Aktion («Undo») notwendig ist, kann die entsprechende Methode direkt in der Aktion eingefügt werden oder es gibt eine separate Aktion dafür.



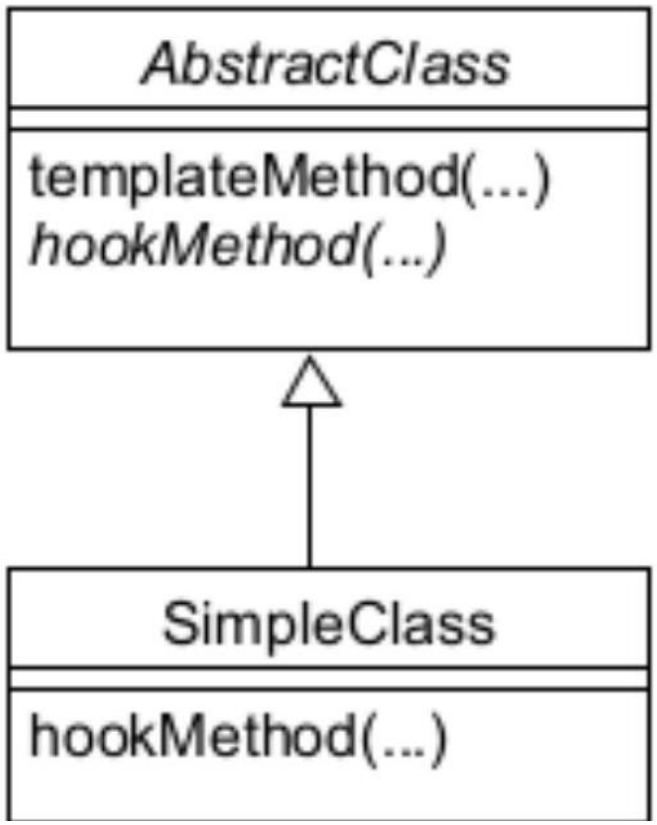
Command: Beispiel Point Of Sale Terminal

- Eine Transaktion eines Persistenz-Frameworks setzt sich aus den Aktionen für jedes veränderte Objekt zusammen.
- Aktionen sind update, insert und delete.
- Eine undo Methode ist ebenfalls vorhanden.



Template Method: Problem und Lösung

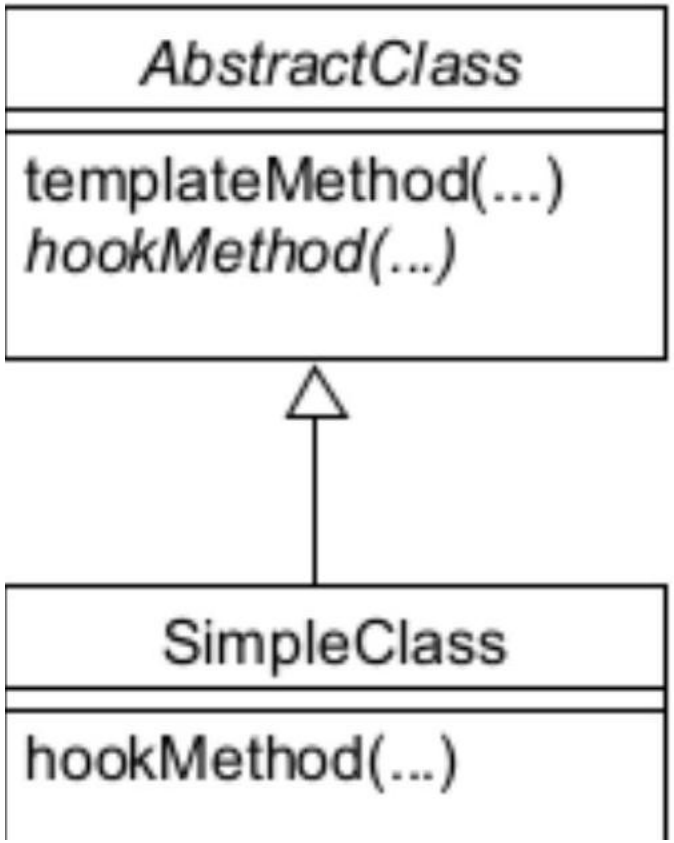
- Problem
- Ein Ablauf/Algorithmus soll so entworfen werden, dass er in gewissen Punkten angepasst werden kann.
- Lösung
- In einer abstrakten Klasse wird eine Template Method hinzugefügt, die diesen Ablauf/Algorithmus implementiert.
- Die Template Method ist fertig geschrieben, ruft aber noch abstrakte Methoden («hookMethod») auf.
- Diese Methoden dienen als Variations- resp. Erweiterungspunkte



und mit ihrer Implementation kann der Ablauf/Algorithmus auf den aktuellen Kontext angepasst werden.

Template Method: Hinweise

- Hinweise
- Die «hookMethod» kann entweder rein abstrakt sein oder bereits eine Standard-Implementation enthalten.
- Eine Factory Method kann in diesem Zusammenhang ebenfalls als «hookMethod» interpretiert werden.
- Es ist nicht einfach, im Voraus alle Orte zu identifizieren, wo Anpassungen notwendig sein müssen.
- Verwandtschaft mit einer Strategy. Eine Strategy benutzt Delegation, um einen ganzen Algorithmus zu variieren, während



Template Method Vererbung benutzt, um einen Teil des Algorithmus zu variieren.

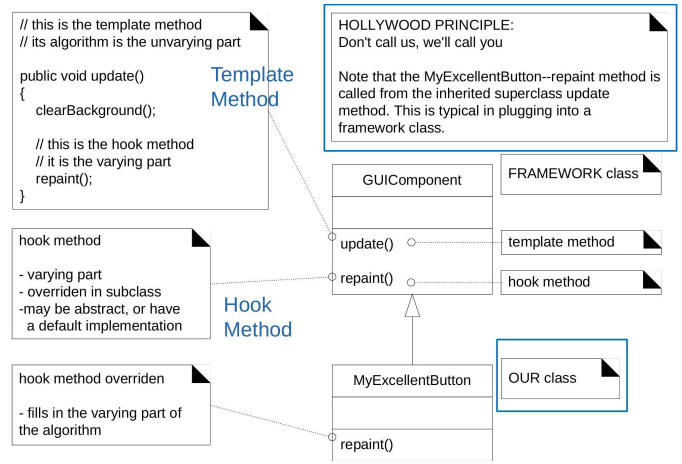
- Hollywood Prinzip: «Don't call us, we call you». Der eigene Code wird von fremdem Code aufgerufen (oder: der Code des Frameworks ruft den Code der Umsetzung auf).

Template Method: Beispiel Larman GUI Framework

School of Engineering
InIT Institut für angewandte Informationstechnologie

- Ein GUI Framework stellt Komponenten zur Verfügung.

- Die Basisklasse GUIComponent stellt die Template Method update() zur Verfügung, die repaint() aufruft.
- Die Methode repaint() muss dann von unserer Klasse überschrieben werden.



1. Einleitung und Definition
2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

Einleitung Persistenz-Framework im Buch von Larman

- Framework für Speicherung von Objekten (siehe [1] Kap. 38).
- Primäres Ziel: Prinzipien des Framework Designs zeigen.
- Sekundäres Ziel: Problemstellungen von Persistenz-Frameworks und mögliche Lösungsansätze zeigen.
- Was fehlt?
- Eigentliche RDB-Zugriffe. Im Buch werden verschiedene Lösungen skizziert.
- Eigentliche Behandlung von Collections und Assoziationen. Im Buch wird dafür die Verwendung vom Design Pattern «Virtual Proxy» erwähnt.
- Abfragen (Queries) werden gar nicht behandelt. Da ja beliebige Speichertechnologien unterstützt werden sollen, ist dies aber auch nicht verwunderlich.
- Der vollständige Programmcode.

Themen Persistenz-Framework von Larman

- Persistenz-Fassade
- Mapping auf RDB
- Mapper für jede Klasse
- Objekt-Identifikation
- Verfeinerung Mapper
- Zustandsverwaltung bezüglich Transaktionen
- Proxy für Lazy Loading von referenzierten Objekten

1. Einleitung und Definition
2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

Moderne Framework Patterns

- Die bewährten Design Patterns finden nach wie vor ihre Anwendung im Framework Design.
- In den letzten Jahren wurden aber noch weitere Mechanismen populär.
- Dependency Injection, meistens gesteuert über Annotationen

- Convention over Configuration: Nur durch das Einhalten von (Namens-)Konventionen wird das Framework aktiv und macht das Gewünschte.
- Implementation von Interfaces basierend auf den Methoden des Interfaces (z.B. Spring Data Repository-Interfaces). Der Methodenname spezifiziert sozusagen seine Implementation, allenfalls noch ergänzt mit Annotationen
- Ist ein Standard Java-Sprachelement ab Java 5 (z.B. @override).
- Können selber deklariert werden.
- Werden «normalen» Sprachelementen hinzugefügt
- Vorteil: Wenn beim Laden einer annotierten Klasse die Annotations-Klasse nicht gefunden wird, gibt es keine Fehlermeldung, sondern die Annotation wird stillschweigend entfernt.
- Anders gesagt fügen Annotationen keine harte Abhängigkeit hinzu und sind somit geeignet, die Domänenlogik frei von ungewünschten (technischen) Abhängigkeiten zu halten.

Steuerung über Annotationen

- Annotationen per se haben ja keine Funktionalität. Es braucht «jemand», der die Annotationen liest und dann Aktionen ausführt.
- Auswertung von Annotationen:
- Beim Starten der Anwendung wird das Framework ebenfalls gestartet.
- Das Framework sucht die Anwendungsklassen auf dem Klassenpfad ab, untersucht allfällige Annotationen und führt die gewünschten Aktionen aus.
- Mögliche Aktionen des Frameworks:
- Dependency Injection von Framework Objekten in Anwendungsobjekte (über Constructor oder Set-Methode).
- Automatisches Implementieren von Interfaces.
- Hinzufügen von Funktionalität zu Anwendungsklassen.
- Achtung: Dieser Vorgang kann zu unerwünschten Verzögerungen beim Start führen.

Java Mechanismen für das Hinzufügen von Funktionalität

- 2 Zeitpunkte
- Während (respektive am Schluss) der Kompilierung über einen AnnotationProcessor.
- Beim Starten einer Anwendung können Anwendungsklassen beim Laden (über einen FrameworkClassloader) noch verändert werden.
- Was wird verändert
- Quellcode hinzufügen.
- Bytecode hinzufügen und bestehenden abändern.
- Für das Implementieren von Interfaces kann java.lang.reflect.Proxy eingesetzt werden.
- Wer verändert
- AnnotationProcessor kann Quellcode und Bytecode hinzufügen.
- Beim Starten einer Anwendung kann Byte Code verändert und hinzugefügt, sowie die Proxy Klasse angewendet werden.

Agenda

1. Einleitung und Definition
2. Design Patterns in Frameworks
3. Fallstudie Persistenz-Framework
4. Moderne Framework Patterns
5. Wrap-up und Ausblick

- Gerade Frameworks müssen sorgfältig mit bewährten Design Patterns entworfen werden.
- Traditionelle Framework Patterns sind die Template Methode und die Factory Method, die es erlauben, dass in Framework Klassen ein Algorithmus realisiert wird, der aber in anwendungsspezifischen, abgeleiteten Klassen noch an den aktuellen Kontext angepasst werden kann.
- Das Command Pattern erlaubt es, dass das Framework anwendungsspezifischen Code aufrufen kann, ohne dass das Framework angepasst werden muss.
- AbstractFactory dient dazu, die Erzeugung einer Familie verwandter Objekte zu ermöglichen.
- Larman hat die Grundzüge eines Persistenz-Frameworks in seinem Buch entworfen, das didaktischen Zwecken dient und den Entwurf eines Frameworks an einem umfangreicheren Beispiel demonstriert.
- Moderne Frameworks setzen auf die Steuerung durch Annotationen, vor allem für Dependency Injection.
- In der nächsten Lerneinheit werden wir:
- den ganzen Stoff SWEN1 kurz repetieren und
- eine alte Semesterendprüfung (SEP) gemeinsam lösen.

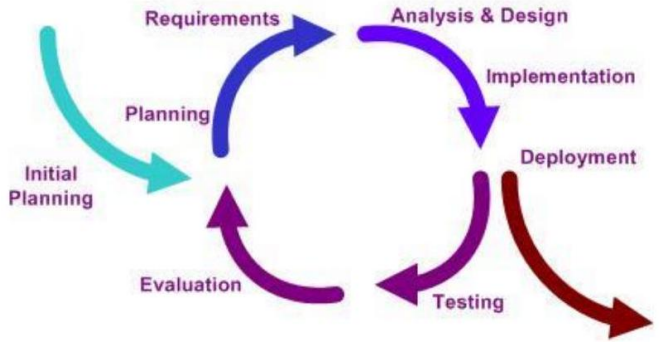
wrapup

Angewendeter iterativ-inkrementeller Softwareentwicklungsprozess in SWEN1/PM3

- Der Softwareentwicklungsprozess wurde so angepasst (engl. tailoring), dass die wesentlichen Artefakte in einem Softwareprojekt im Kontext eingeführt werden können.
- Die Software wird in Iterationen entwickelt (2 Wochen Rhythmus).
- Jede Iteration hat ein Ziel und wird nach Abschluss reviewed.
- Es gibt drei Meilensteine, die im Projektverlauf ein besonderes Ereignis darstellen bzw. den Abschluss einer Phase: Projektskizze (M1), Lösungsarchitektur (M2) und Beta-Release (M3)
- In jeder Iteration werden Anforderungen, Analyse & Design, Implementation und Testing gemacht (Software entsteht in Inkrementen).
- Der angewendete Softwareentwicklungsprozess und das Projektmanagement eines iterativ-inkrementellen Projektes wird in PM3 noch detaillierter erklärt.

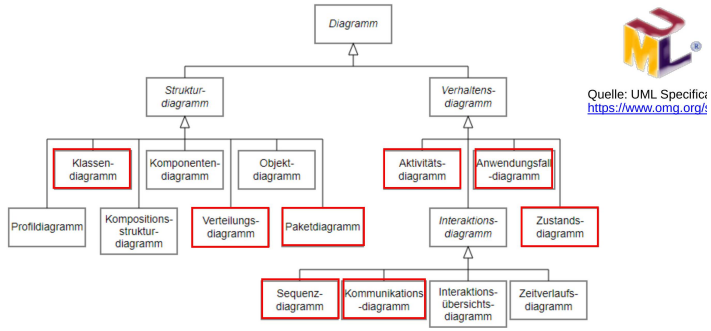
Wesentliche Resultate bzw. Artefakte

- Anforderungsanalyse
- Funktionale Anforderungen mit Use Cases
- Qualitätsanforderungen und Randbedingungen
- Domänenmodell
- Design
- Softwarearchitektur
- Use Case Realisierung (statische und dynamische Modelle)
- Implementation
- Quellcode (inkl. Javadoc)
- Testing
- Unit-Tests



- Integrations- und Systemtests

Modellierung und Modelle mit der UML



Quelle: UML Specifica
https://www.omg.org/s

Quelle: UML Specification, https://www.omg.org/spec/UML/
☐ für die Modellierung in SWEN1 relevant

Gebrauch der UML (nach Martin Fowler)

- UML as a Sketch

- Informelle und unvollständige Diagramme (z.T. von Hand gezeichnet), um schwierige Teile des Problems oder der Lösung zu verstehen und zu kommunizieren
- Die agile Community bevorzugt diese Anwendungsart von UML
- UML as a Blueprint
- Relativ detaillierte Analyse und Design-Diagramme für Code-Generierung oder um existierenden Code besser zu verstehen
- Klassische UML-Tools für ein Forward- und Reverse-Engineering (Roundtrip)

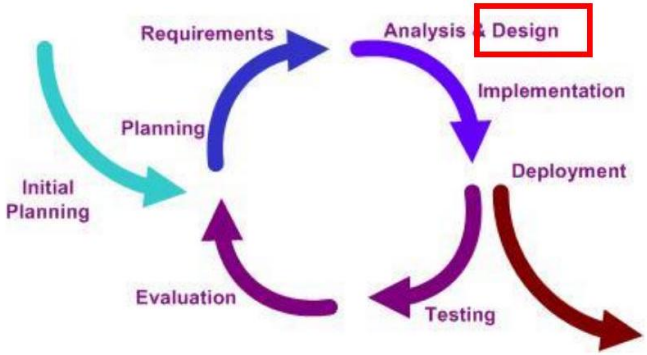
- UML as a Programming Language
- Komplete, ausführbare Spezifikation eines Software-Systems in UML
- MDA-Tools zur Modellierung und Generierung

Überblick Anforderungen & Analyse

- User Research (Personas und Szenarien, Contextual Inquiry)
- Sketching und Protoyping
- Ableiten und Modellieren von Use Cases (dt. Anwendungsfälle)
- Detaillierung der Use Case (UML-Use-Case-Diagramm, Use-CaseSpezifikationen, UI-Sketching
- Qualitätsanforderungen und Randbedingungen erheben und festhalten.
- Modellierung der Fachlichkeit und Begriffe des Anwenders in einem Domänenmodell (konzeptuelles UML-Klassendiagramm)
- Bei der objektorientierten Analyse (OOA) liegt die Betonung darauf, die Objekte - oder Konzepte in dem Problembereich zu finden und zu beschreiben!

Überblick Design

- Design und Modellierung einer für die Problemstellung geeigneten Softwarearchitektur (UML-Paketdiagramm, UML-Verteilungsdiagramm)
- Use-Case-Realisierung und Klassendesign mit Verantwortlichkeiten (UML-Klassendiagramm, UML-Sequenzdiagramm, UMLKommunikationsdiagramm, UML-Zustandsdiagramm, UML-

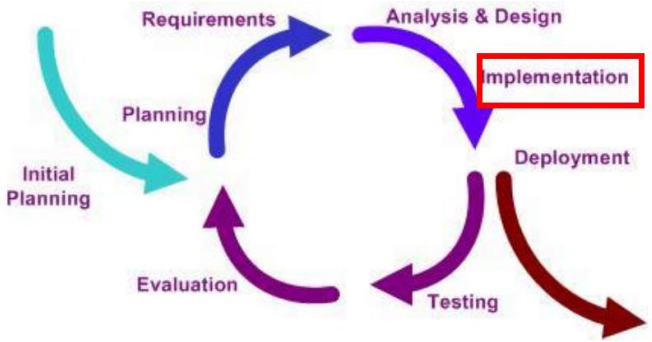


Aktivitätsdiagramm)

- Entwurf mit bewährten Design Patterns
- Beim objektorientierten Design (OOD) liegt die Betonung darauf, geeignete Softwareobjekte und ihr Zusammenwirken (engl. collaboration) zu definieren, um die Anforderungen zu erfüllen!

Überblick Implementation

- Umsetzung des Designs in Code der entsprechenden (objektorientierten) Programmiersprache
- Verwendung von geeigneten Algorithmen und Datenstrukturen zur Implementierung des Designs
- Code Smells sofort bei deren Aufdeckung verbessern (Refactoring)



- Laufende Dokumentation des Quellcodes (nach Clean CodePrinzipien)

Überblick Testing

- Laufendes Design und Implementierung von Unit-Tests
- Planung, Design und Durchführung von weiteren Tests auf den Teststufen Integration und System je nach Problemstellung
- Dokumentation des Testkonzepts und der Tests