

Domain Modell

- Objektorientierte Zerlegung eines Domäne (Fachbereich) in konzept. Klassen
- Eigenarten
 - Verbindungen mit Mengen und Bezeichnungen
- keine Lösungselemente
- keine JVR Klasse
 - keine Operationen / Verantwortlichkeiten

Vorgehen

- Konzeptionelle Klassen definieren
- Wiederverwertung bestehender Modelle
 - Analysemuster einsetzen
 - Kategorienliste einsetzen
- UML Diagramm zeichnen
- Klassendiagramm ohne Operationen
 - nur ungerichtete Assoziationen

Assoziationen und Attribute hinzufügen

Substantive herausziehen

aus UC's herausziehen

Mehrdeutigkeit der Sprache beachten

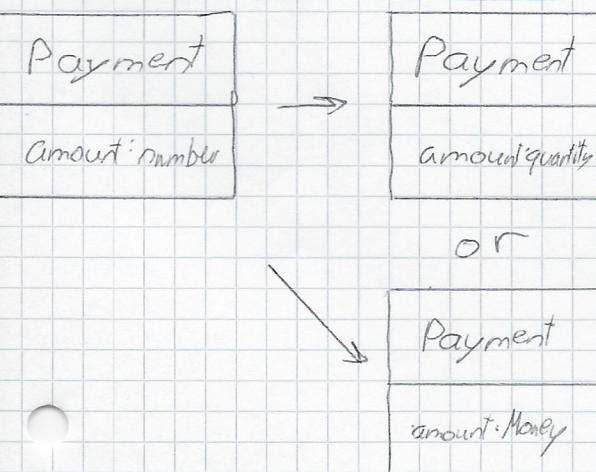
Berichtsobjekte (Quittung) nur aufführen, wenn speziell aufgeführt

Beschreibungsklassen herausfinden

Attribute

- einfache Datentypen
- primitiv
- zusammengesetzt

Mengen und Einheiten



Motivation

- erleichterter Einstieg im Problembereich
erster Schritt Richtung JVR Klassen des Designs
low representational gap bei OO Modell

Kategorienliste auszug

- Geschäftstransaktionen ^{BSP}
- als Ganzes (Sales)
 - Transaktionsposition (SalesLineItem)
 - verbundenes Produkt (Item)
 - Registrierung Transaktionen (Register)
 - Rollen von Beteiligten (Cashier)
 - Ort (Store)

Beschreibungen von Dingen

Ereignisse mit Zeit/Ort

Physische Objekte

Kataloge

Kontainer

Dinge in Container

andere Systeme

Assoziationen

Verbindungen zwischen Klassen
ungerichtet

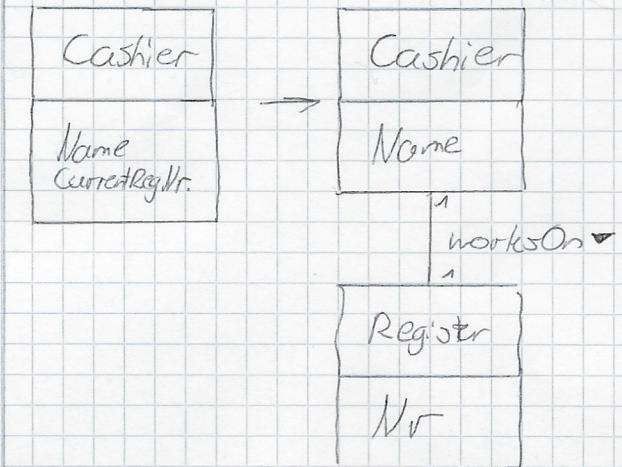
nicht zu viele Eintragen

Richtlinien

keine komplexen Attribute, sondern eigene Klassen

Mengen und Einheiten korrekt modellieren

Attribute nicht als Fremdschlüssel def.



Elaboration Iteration 1

Was wurde gemacht

UC's und Akteure benannt

kurzer Anforderungsworkshop

wichtigste Qualitätsmerkmale definiert

Vision, Risiko Liste

Planung 1. Iteration

Nicht in dieser Iteration

Wegwerfcode

keine Tests

nv länger als einige Monate

Was wird gemacht

Entwicklung Risikoreichste Komponenten

Mehrheit Anforderungen stabilisiert

Schätzung Aufwand Gesamtprojekt

Artefakte

Domain Model

Design Model

Software Architektur Dokument
-Probleme und deren Lösungen

Data Model
-DB Schema

UI Prototyp

Aufbau

Meistens rel. kurz 2-4 Iterationen

am Schluss einsatzfähiger Code

Feedback aus Tests und von Anwendern einbauen

Anforderung in Workshop aufnehmen

Objektorientiertes Design

vorher abgeschlossen

Anforderungsanalyse

- Anforderungsworkshop
- UC's
- UC Diagramm
- Vision
- non funkt Anforderung
- Domain Rules
- SSD
- Operation Contracts
- Domain Model

OOO Erfolgsfaktoren

Entwurfsmuster anwenden
wo sinnvoll und nicht überfließen

Prozess

1-2 Tage pro Iteration

Hauptteil in Elaboration, v.a.
Architektur

Verfeinerung in Construction Phase

Vorgehen

Logische Architektur festlegen
(Wo ist welches Modul, Abhängigkeiten)

Systemoperationen realisieren
- Basis SSD

- Hilfe durch Operation Contracts
- Nachbedingungen
- Domänenmodell inspirationen für Klassendiagramm

UML Klassendiagrammzeichnen Klassen Interaktionsdiagramm

Was wird gemacht

Architekturentwurf

Design-Klassendiagramm

Interaktionsdiagramm für ausgewählte Systemoperationen (Aufbrechen Blackbox)

Zusammenhänge

UC's \Rightarrow Anforderungen für einzelne Systemoperationen

Operation Contracts \Rightarrow genaue Anforderungen für einzelne Systemoperationen

Domänenmodell \Rightarrow inspiriert Design-Klassendiagramm

Code \Rightarrow Bauplan für mehr Code

Bedeutung

Zentral für Wartbarkeit und Qualität der Software

- wie schnell Fehler finden
- wie viel vom System bei Änderungen betroffen
- wie schnell kann sich jemand einarbeiten

Basis für Implementation

Logische Architektur

Gesamtheit der wichtigsten Design Entscheide

- Basis technologien z.B Java

- besondere Massnahmen um Anforderungen zu erfüllen z.B redundante Daten speicherung

Teilsysteme und ihre Abhängigkeiten

Top Level View, das grosse Ganze

Konzept

Alles in einem Modul gehört zusammen

wenig Kopplung nach außen

$x \rightarrow y$ x benutzt y

Schichten Entwurfsmuster

Zerlegung Gesamt system in Schichten

- je weiter unten desto allgemeiner
- je höher desto spezifischer

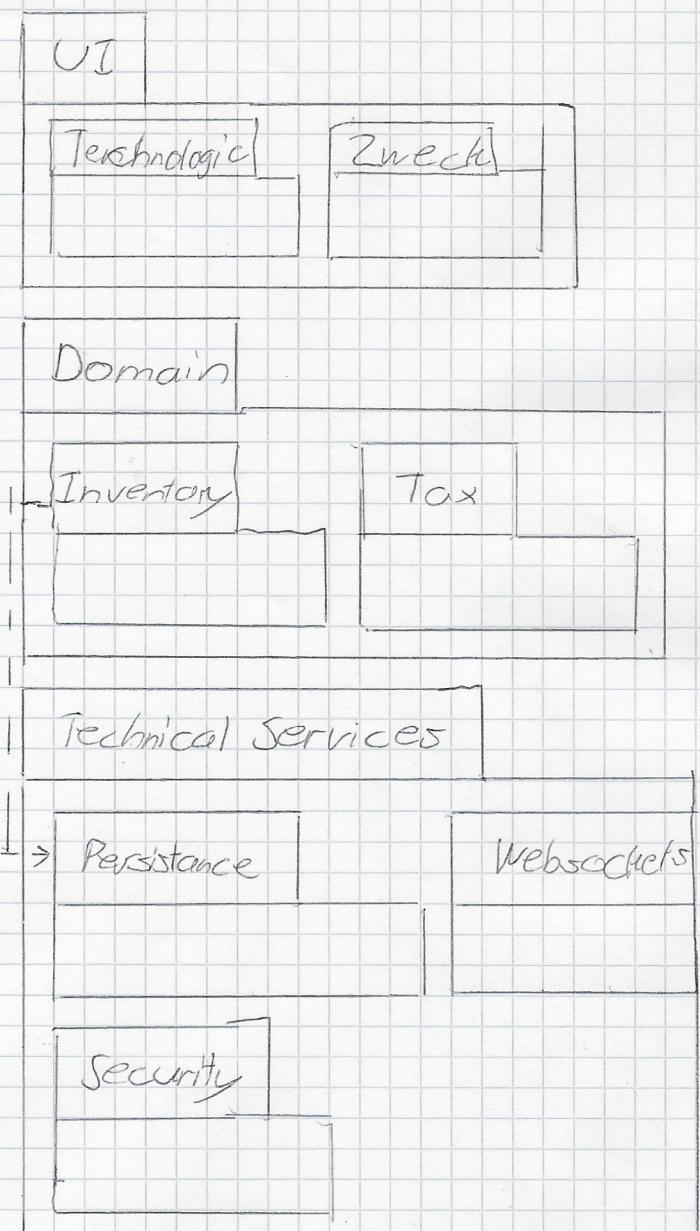
z.B. überst GUI

- Kopplung nur von oben nach unten

Pakete umsetzen

com.myCompany.myproject.Schicht Subsidiert

Example



UML Sequenzdiagramm

Zeitachsen dominant, von oben nach unten

Nachrichtenaustausch von links nach rechts

Möglichkeiten für

- Wiederholungen $\{ \text{loop } [\text{bed}] \}$
do loop until bed=true

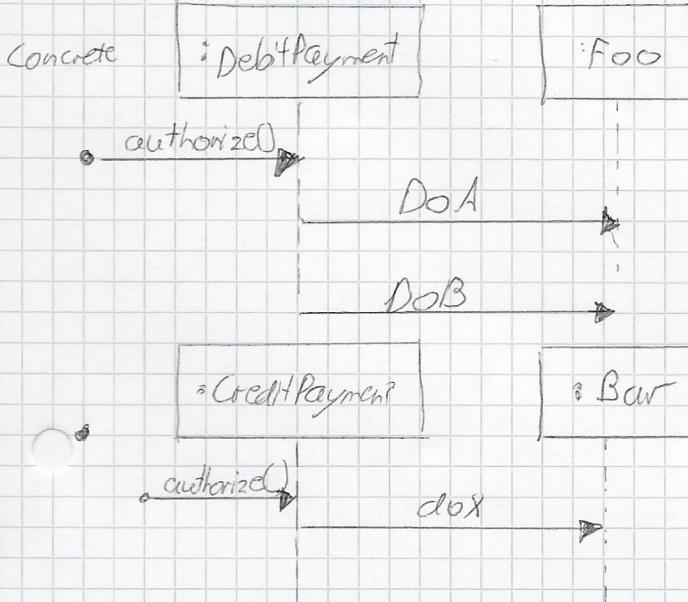
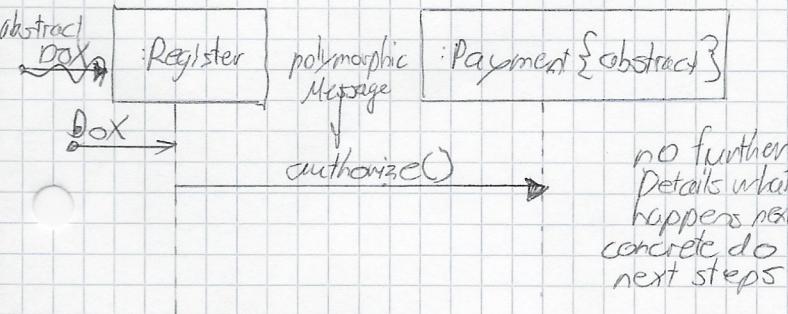
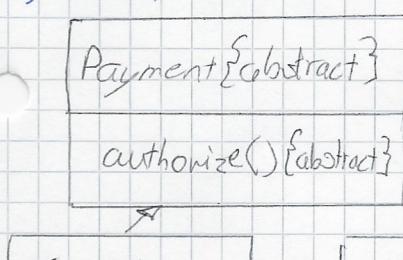
- Bedingung opt [color=red]
if endif
alt [color=green]
else

Hierarchisches Diagramm

→ = synchron

--- = asynchron

Polymorphismus



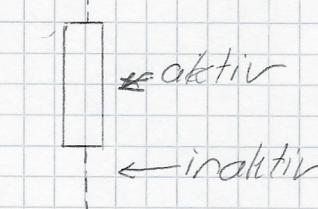
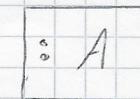
Notation

— request

--- response

► synchron

> asynchron

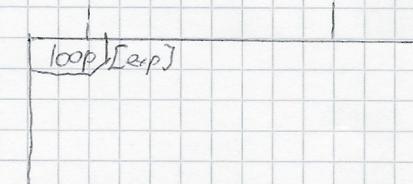


● → Trigger message

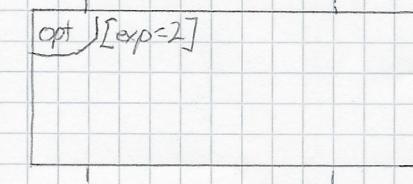
message without param

message with param (p1, p2)

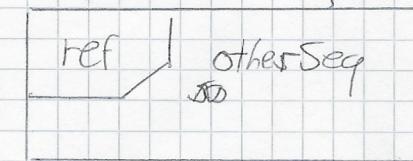
← returnvalue¹, RV2



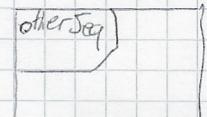
loop this block
until exp=true



if exp=2 do
block



references other
sequencediagram
with otherSeq name



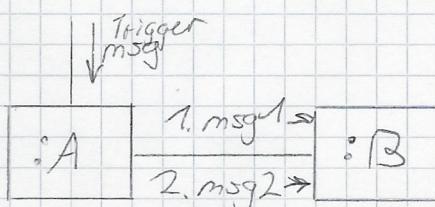
UML Kommunikationsdiagramm

Zeitlicher Ablauf über Nummern der Nachrichten

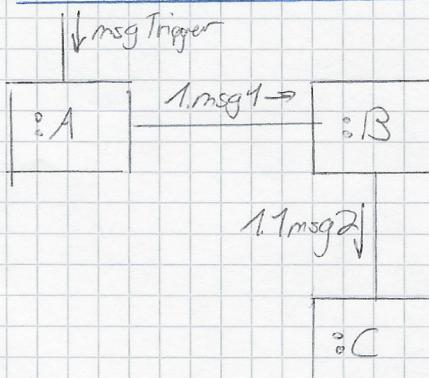
Platzsparender als Sequenzdiagramm

geeigneter für Handzeichnungen

Nachrichten

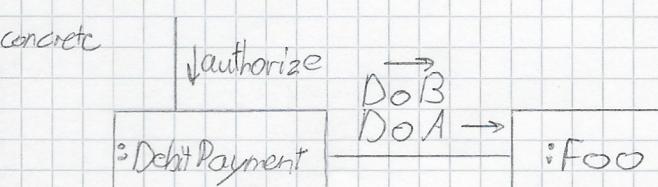
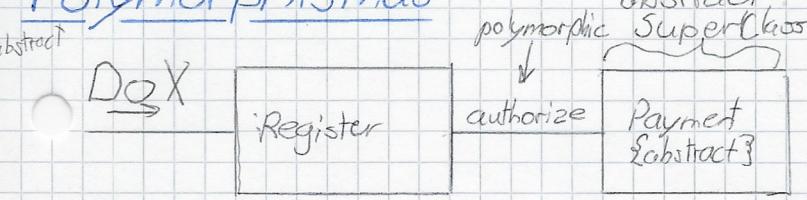


Nachrichten verschachtelt

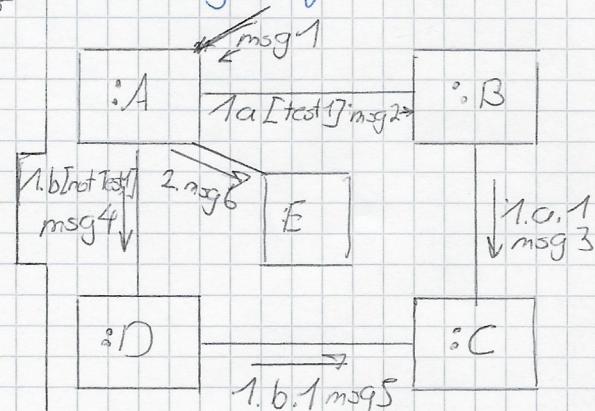


Nachricht 1.1 wurde von Nachricht 1 ausgelöst, deshalb verschachtelte Nummerierung

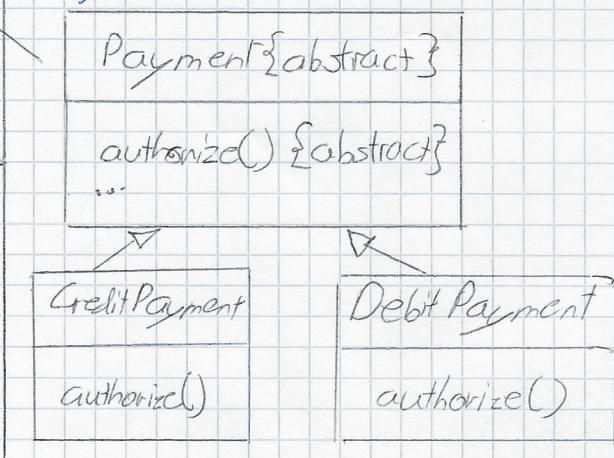
Polymorphismus



Bedingung



Polymorphismus II



UML Klassendiagramm

Basis: Domänenmodell

Zusätzlich Operationen / Methoden

nur gerichtete Assoziationen

UML Notationen

A → B A extends B

A - -> B A implements B

A - - -> B A hat Abhängigkeit zu B
z.B Variable, Parameter
sehr generell

A → B Ich kann von A nach B
navigieren z.B via Variable

A —♦ B Aggregation A verschwindet
wenn B verschwindet

A → B Aggregation: A bleibt auch
wenn B verschwindet

+ public

- private

protected

~ package

method/class {abstract} abstrakte Methode/Klasse
- myAttribute int = 5 {readOnly} final
- Map<String, ArrayList> String[] Collection

Attribute : Type

method(param.) : ReturnType

<<Interface>> myInterface

method(paramName: Type) : ReturnType

method() or return (paramName: Type)

<<constructor>> Classname(paramType)

Method throw () {exception IOException}

finalMethod() {leaf}

method() //assumed public

attribute //assumed private

A →¹ B // A has 1 variable
order

with name order type of

Type B

.. there are more, but not shown

Grasp II PV

Protected Variation Patterns

Veränderung auf in instabilem Teil von Systemen soll möglichst wenig Auswirkung auf stabilem Teil haben

~~sket~~ Solution: Stabiler Interface dazwischen schalten

z.B. bei Teilen deren Implementierung sich wahrscheinlich noch ändert

z.B. virtuelle Maschinen / Betriebssysteme

Variationspunkt: Falls Veränderungen vorprogrammiert → unbedingt PV einbauen
Veränderungen sind sicher

Entwicklungszeitpunkt

- Veränderung nicht sich, aber wahrscheinlich
- nicht in Anforderung enthalten

nicht alles schützen

Je später in der Entwicklung, desto weniger wird sich verändern
zu viel PV → zuviel Komplexität, zu teuer

GRASP

→ genereller als Got

methodischer Ansatz für OO Entwurf

Verantwortlichkeiten an SW Klassen zuweisen

Verantwortlichkeit

Doing (Algorithmen, Code)

- Aktionen anderer Objekte anstoßen / kontrollieren
- selber was tun

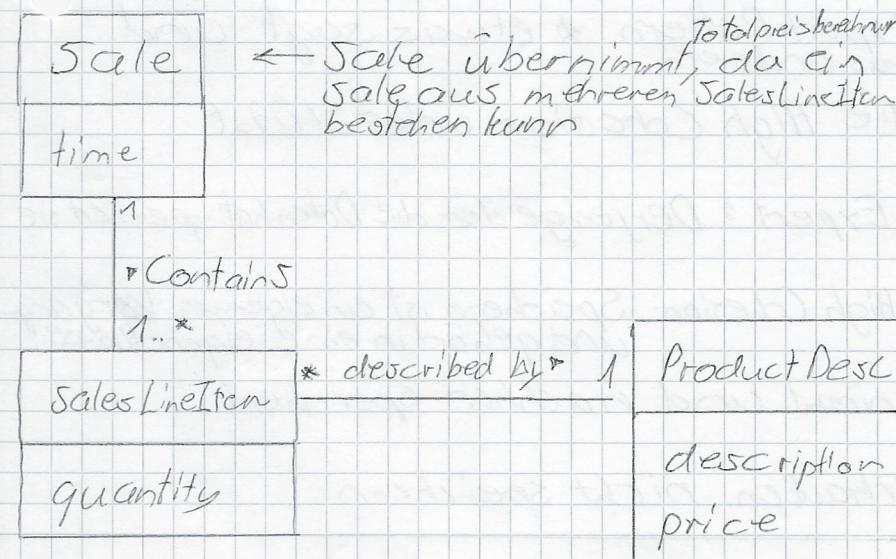
Knowing (Daten, Attribute)

- eigene private Daten
- verwandte Objekte kennen
- abgeleitete Dinge kennen

Information Expert Pattern

Wer Infos hat um Aktion auszuführen
übernimmt Verantwortlichkeit

low Coupling / high Cohesion erfordert andere Lösung (Separate Klasse)



Creator Pattern

Wer soll Instanz erstellen

Kandidaten

Behälterklasse (widerspricht jedoch o. g. Grasp Prinzip)
Klassen mit enger Zusammenarbeit
Klasse die Initialisierungsdaten kennt

Alternative: Factory Klasse

Unterstützt geringe Kopplung

Aufbau eines Musters

Name

Beschreibung Problem

Beschreibung der Lösung

Kontext für Anwendung

Was ist ein Objekt

Kombination Daten + zugehörigen Code

Vermeiden

reine Datenklassen

SoleHandler oder SoleManager

Do it yourself Pattern

Eine SW Klasse macht das was die Domänenklasse macht

Bring Objects to life

SW Klassen bekommen Code

SW Klassen können selbstständig komplexe Aufgaben lösen

Low Coupling Pattern

Änderungen möglichst lokal

möglichst geringe Kopplung zwischen Klassen

Qualitätskriterium für OOD



High Cohesion

Objekte fokussiert und verständlich behalten

möglichst 1 Klasse - 1 Aufgabe

klare, eindeutige und schmale Verantwortlichkeit

Controller Pattern

Welches Objekt realisiert in erster Instanz die Systemoperationen?

- Fassadencontroller: Root Object, System übergeordnetes System
- UController: Pro NC eine künstliche Klasse

Controller: Oberstes Objekt der Domänenlogik an Schnittstelle zur nächst höheren Schicht

UI hat keine Domänenlogik und bearbeitet Systemereignisse nicht selber sollte im Entwurf als erstes beantwortet werden

Polymorphismus

- Verletzt Low Coupling ermöglicht dafür sehr komplexe Dinge einfach darzustellen

- Viele if's / switch cases Indiz für Polymorphismus
Interface als Superklasse bietet Flexibilität

Pure Fabrication Pattern

Wer verantwortlich wenn Expert Pattern * etwas sagt und Low Coupling / High Cohesion etwas anderes.

Lösung: künstliche Klasse die High Cohesion unterstützt

z.B Speicherung von Daten: Expert: Derjenige der die Daten hat speichert sie

High Cohesion: Speichern ist ein eigener Vorgang und gehört in eine eigene Klasse

→ Klasse welche Daten annimmt und nachher speichert

oder: sale soll nur verkaufen nicht speichern

Indirection Pattern

Verantwortung zuweisen, dass direkte Kopplung vermieden wird

Vermittler-Objekt dazwischenschalten

- EventListener
- Adapter

z.B. Websocket Handler zwischen UI und Domänenlogik

Gang of four Gof

Adapter pattern

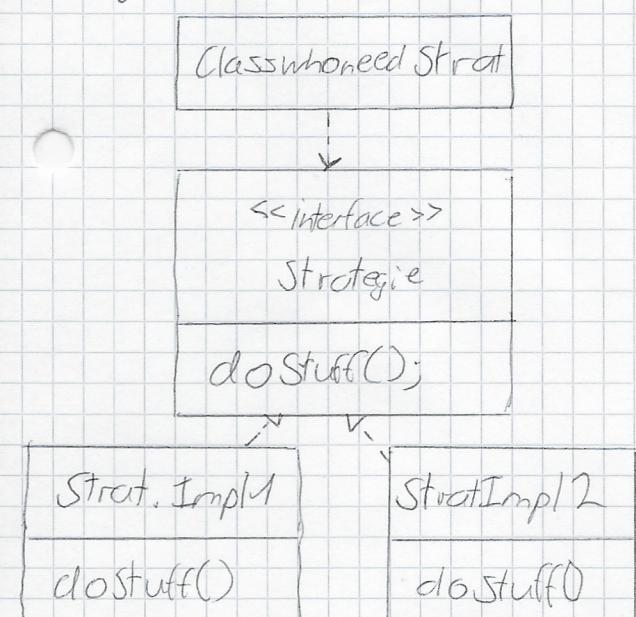
Problem: Verbinden zweier inkompatibler Schnittstellen

<code><<Interface>></code>	optimiert für eigene Anwendung
Interface	
Adapter	
inkompatible Schnittstelle	Mittelbare Schnittstelle

Pattern aus Grasp: Indirection
 Polymorphismus
 Pure Fabrication
 High Cohesion
 Low Coupling
 Protected Variation

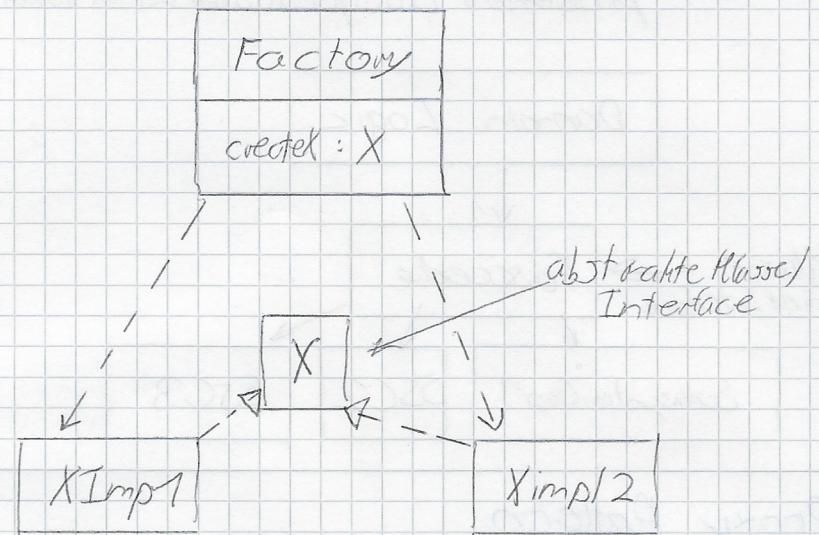
Strategie Pattern

Problem: Einfache und schnelle an unterschiedliche aber verwandte Algorithmen



Factory Pattern

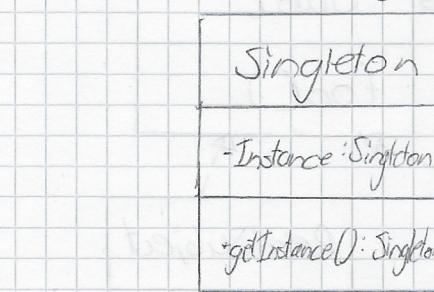
Problem: Wer erstellt Objekt



Zugriff auf Factory z.B. durch Singleton

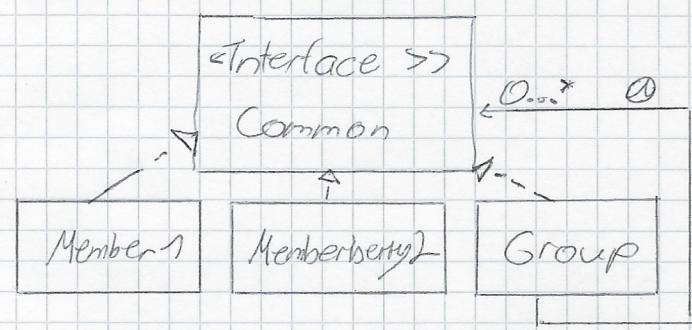
Singleton Pattern

Problem: Brauchen eine Instanz einer Klasse welche global sichtbar sein soll



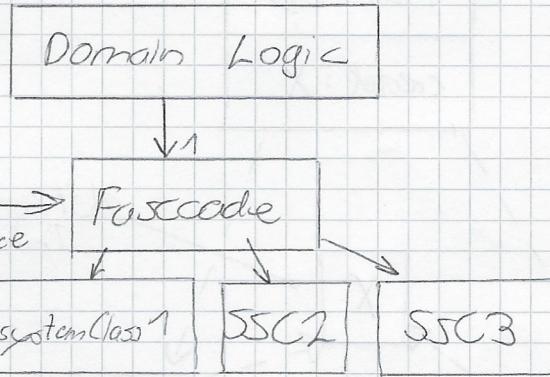
Composite Pattern

Problem: Gruppe von Objekten soll gleich behandelt werden wie einzelnes Objekt



Fascade Pattern

Problem: 1 Interface benötigt damit Subsysteme problemlos ausgetauscht werden können



Observer Pattern

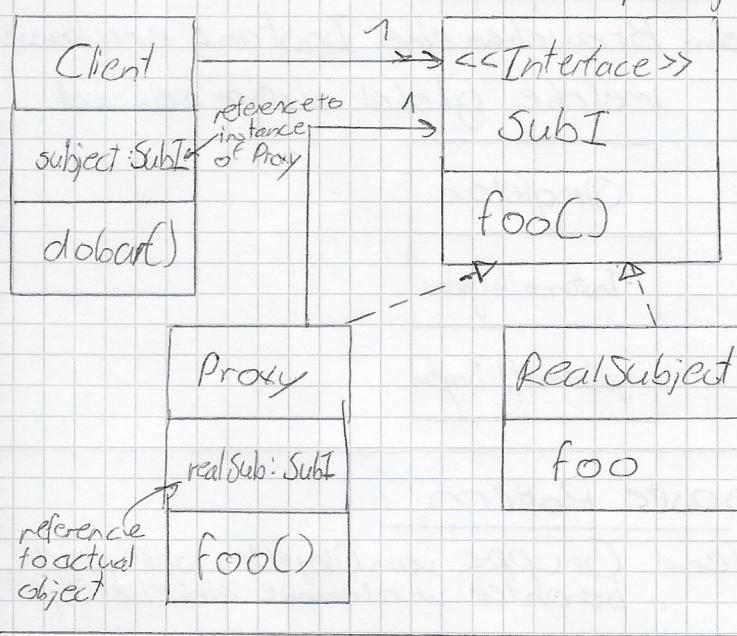
Problem: Versch. Objekte (Observer) sollen über Zustandsänderung eines Objekts (Observable) informiert werden ohne direkte Kopplung



Proxy Pattern

Problem: Ansprechen von externem Dienst, falls vorhanden, sonst lokal arbeiten

ähnlich wie Adapter ohne Anpassungsfähigkeit



Aktivitätsdiagramm

sequentielle und parallele Abläufe

Darstellung von Workflows / Datenflows / Algorithmen

no direct time axis from top to bottom

may go up, down and sideways

Notationen

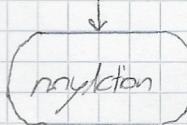
● start of activity



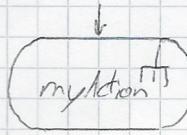
○ End of activity

Partition 1 | Partition 2

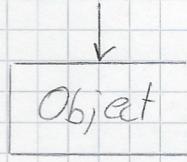
Show different involved
parties



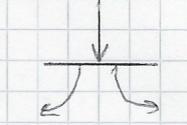
Action, does something,
transitions upon its completion



Action consists of a separate
activity flow displayed elsewhere



Object is used or produced by
actions. Allows modeling of
~~modell~~ data/object flows



Forke, one incoming, multiple parallel
outgoing transitions / object flows



Wait until all incoming flows
are done, then do outgoing

Richtlinien

Aktionsträger auf denselben
Abstraktionslevels anwenden
⇒ Sub-aktivitätsdiagramme verwenden
falls notwendig

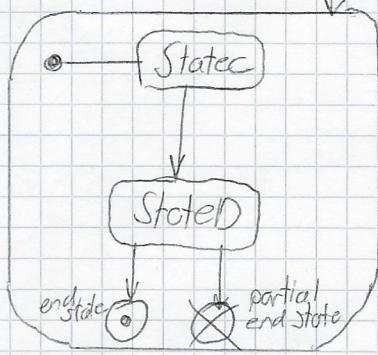
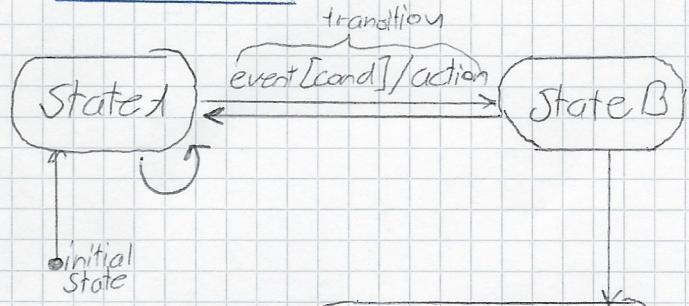
in UP: Business Modeling

Klassent Sequenz → Aktivitätsdiagramm

Zustandsdiagramm (Lifecycle)

Zeigt Zustände und wie sich diese ändern durch externe Einflüsse und welchen Einfluss diese Zustände nach aussen haben

Notation



Anwendung

keine Geschäftsprozesse
Maschinensteuerung
Embedded Systems
Transaktionen

UC Realisierung

Realisierung der 1. Elaborationsiteration

Designentscheid klären und verständlich machen

in Zentrum Entwurfstüchtigkeit, nicht UML

Architektur: Domänen schicht

Command Query Separation Principle CQS

Pattern für Methode

Eine Methode soll entweder eine Aktion ausführen (Command) oder eine Anfrage durchführen

Nötwendigkeit für weitere Systemoperationen

Start-up UC

Implizit oder explizit

Sollte erst am Schluss entworfen werden

abhängig von OS und Programm sprache

Begriff: Initial Domain Object

Vorgehen

zuerst Controller Klasse bestimmen

Teilverantwortlichkeit ableiten und Zusammenarbeit organisieren

Patterns

Creator / Information Expert

Low Coupling and High Cohesion prüfen

Anbindung UI - Domänenlogik

Initialisierung 2 Varianten

- zuerst initiales Domänenobjekt, dieses wird dem UI übergeben

- zuerst UI, dies holt Domänenobjekt aus einer bekannten Quelle

Synchrone Aufrufe von UI her

- primär über Controller

- sekundär über andere Objekte der Domänenlogik

Async Mitteilung: Spezielle Patterns

Initial Domain Object

erzeugt und initialisiert die restlichen Domänenobjekte (soweit nötig)

Meist Wurzel eines Containers.

Supplementary Specs (non UML)

21/1

- FURPS
- Functionality
 - Usability
 - Reliability
 - Performance
 - Supportability

Implementation Restrictions

Interfaces

Operations

Packaging

Standards

Documentation

Ausschreiben von Berichten
(Inhalt/Layout)

Einschränkungen bez.

- Entwicklungsbedingungen
- verwendete Software
- Drittprodukte

Req Business Rules

keine direkten Anforderungen, haben aber Einfluss darauf

Gesetze / Geschäftspolitik

Aufbau

- ID, Name
- Beschreibung der Regel
- vorhersehbare Änderungen
- Quelle / Ursache

Req Funktionalen Anforderungen

gehören meist zu UC's

manchmal nicht für UC's wenn keine Userinteraktion vorliegt

Req Vision

Zus von Projektziele für

- neue Mitarbeiter
- Geschäftsfertigung

Allg Ziele der Stakeholder

Systemeigenschaften / Beschreibung auf Verpackung

Req Glossar

Liste relevanter Begriffe

Aufbau

- Begriff
- Def. + Infos
- weitere Erklärungen
- Format (Typ, Länge, Einheit)
- Validierungsregeln
- Alias (Homonym)
- Beziehung zu anderen Elementen

Req SSD

zeigt Input/Output von System

zeigt Interaktion der Akteure

UML Notation

basierend auf einem UC

Parameter + Return können im Glossar beschrieben werden

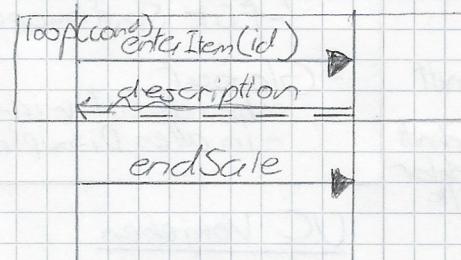
Aktor gibt Befehl an System (Blackbox), System gibt Antwort

→ send

← -- return (fakultativ falls kein return value)

: Actor

: System



Absicht der Requests
betonen enterItem()
not scan()

- in Elaboration Phase