Software Engineering

= Herstellung und Entwicklung von Software, der Organisation und Modellierung der zugehörigen Datenstrukturen und dem Betrieb von Softwaresystemen.

Prozesse und Disziplinen

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

Unterstützungsprozesse
- Projektmanagement
- Qualitätsmanagement
- Risikomanagement

Projekt Prozess Modelle

Software Engineering

Zweck: Um Analyse- und Designentwürfe zu diskutieren, abstimmen und zu dokumentieren bzw. zu kommunizieren. In der objektorientierten Softwareentwicklung werden **statische Modelle** (Klassen und Assoziationen) und **dynamische Modelle** (Abläufe und Verhalten) erstellt.

Code und Fix

Codierung oder Korrektur im Wechsel mit Ad-hoc-Tests
Vorteil: Liefert schnell Ergebnisse
Nachteil:
- Schlecht planbar (Funktionalität, Zeit, Kosten und Qualität)
- Aufwand für Korrekturen unangemessen hoch
- Schlecht wartbare Software und nicht skalierbar

Wasserfall

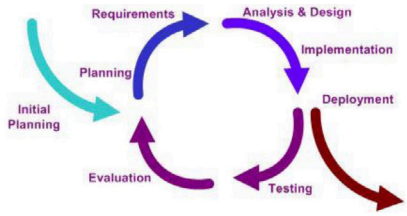
Die Reihenfolge von Aktivitäten/Phasen ist fest definiert.
Annahme: Ziel ist früh bestimmbar und Gelände ist bekannt
Vorteile:
- Hohe Planbarkeit (Funktionalität, Zeit und Kosten).
- Klare Aufteilung der SWE in einzelne Phasen (Analyse, Design, Test,...)

Nachteile:
- Schlechtes Risikomanagement
- Anforderungen sind zu Beginn nie alle bekannt

Inkrementell-Iterativ

Entwicklung in mehreren geplanten und kontrolliert durchgeführten Iterationen schrittweise (inkrementell) entwickelt. Ziele der Iterationen sind Risiko-getrieben Annahmen: Ziel und Gelände sind am Anfang noch unklar.
Vorteile
- Flexibles Modell bei unklaren Anforderungen/Zielen.
- Gutes Risikomanagement (Mitarbeiter und Technologie).

Nachteil: Braucht eine Involvierung des Kunden über die ganze Projektdauer.



- Anforderungsanalyse
- Design
- Implementation
- Testing

Agile Softwareentwicklung

Basiert auf interativ-inkrementellen Prozessmodell. Fokus auf gut dokumentierten und getesteten Code. Risiken werden so lange nicht reduziert, bis Lösung erstmals implementiert wird

Strategie zur Prozesskontrolle
Definierte Prozesskontrolle (Plan-driven)
- Planung wird am Anfang durchgeführt, dann Prozess gesteuert
- Geeignet für gut planbare Problemstellungen

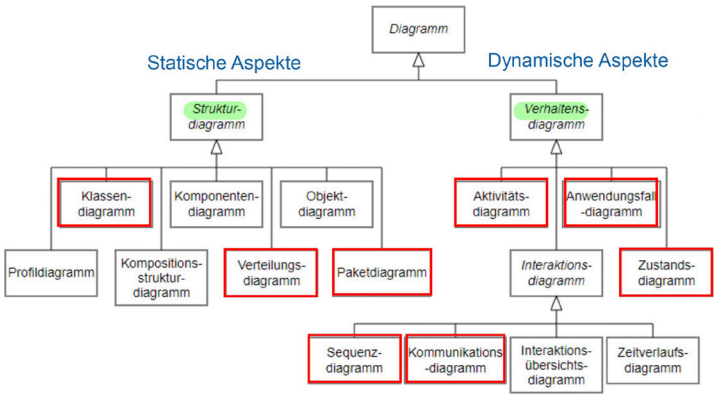
Empirische Prozesskontrolle (Agil)
- Rollende Planung, nur Grobplanung am Anfang
- Prozess wird fortlaufend überwacht
- Geeignet für komplexe Problemstellungen (unbekannte Anforderungen)

Modellierungsartefakte

Software Engineering

Ziel der Anforderungsanalyse:
Wie findet man die **wichtigen Anforderungen** an ein Softwareprodukt?

Artefakte der Software-Entwicklung sind Modelle:

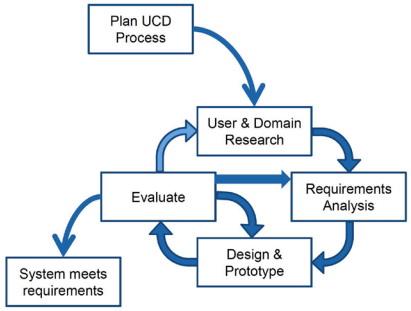


Artefakte werden in der Anforderungsanalyse
- Systemabgrenzung und Systemkontextdiagramm
- Use-Case-Modell und UI-Sketches
- Qualitätsanforderungen und Randbedingungen
- Domänenmodell

User Centered Design

Berücksichtigt die Bedürfnisse, Wünsche, Einschränkungen der Benutzer.

Process / Vorgehen



Usability

= Gebrauchstauglichkeit: Wie gut kann ich ein Softwareprodukt gebrauchen, um meine Aufgabe zu erledigen / mein Ziel zu erreichen?

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

3 wichtige Ziele
- Effektivität
- Effizienz
- Zufriedenheit

Usability-Engineering

= Software-Ergonomie. Ziel: SW-Produkte entwickeln, die effektiv, effizient und zufriedenstellend sind.

Usability, User Experience(UX) und Customer Experience

Usability: Wie einfach kann eine Applikation benutzt werden (Gebrauchstauglichkeit).

UX: Wie fühlt sich die App an = Usability + **Desirability**

Customer Experience: Gesamteindruck der App (Marke)
= Usability + Desirability **+ Brand Experience**

7 Usability Anforderungen

- **Aufgabenangemessenheit** (Minimale Anzahl Schritte für eine Aufgabe)
- **Lernförderlichkeit** (Tooltips, Onboarding, Tutorials)
- **Individualisierbarkeit** (Sprache, Kultur, Barrierefreiheit)
- **Erwartungskonformität** (bzgl. Design, Interaktion, Struktur, Funktionalität)
- **Selbstbeschreibungsfähigkeit** (Affordanzen)
- **Steuerbarkeit**
- **Fehlertoleranz**

Wichtigsten Artefakte

Personas, Usage-Szenarien Kontextszenario, Storyboard, Interaktionsprototypen (Wireframes), Stakeholder Map, Geschäftsprozessmodell

User und Domain Research

User Centered Design

Ziele

– **Wer** sind die Benutzer?
– **Was** ist ihre Arbeit und **was** brauchen sie, um ihre Ziele zu erreichen?
– **Wie** sieht ihre (Arbeits-)Umgebung aus?

Methoden

Contextual Inquiry: Experte beobachtet User, Audio und Video-Aufnahmen Gut, um Szenarien zu diskutieren; **Interviews**, **Beobachtung**, **Fokusgruppen**, **Umfragen**, **Nutzungsauswertung**, **Desktop Research**

Artefakte

Personas, Usage-Szenarien, Domänenmodell,
Stakeholder Map, Geschäftsmodell, UI-Skizzen/Designs/Wireframes

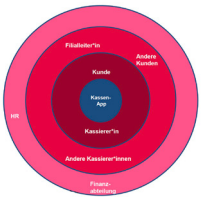
Personas: Eine fiktive Person. Repräsentiert eine bestimmte Benutzergruppe. Mit gleichem Verhalten, Bedürfnissen, Interessen. In einer bestimmten Rolle. Wichtig, um Design-Entscheide zu diskutieren und kommunizieren. Bsp:

Persona: Silvia
– 28-jährig, Medizin-Studentin, wohnt noch zu Hause in Wohlen
– Jobbt nebenbei seit mehreren Jahren als Servierperson im Gasthof Linden
– Quote: «Ich möchte die Gäste effizient bedienen, ohne hektisch zu wirken»

Usage-Szenarien: Kurze Geschichte, wie ein Benutzer die Software in einer konkreten Situation benützt, um eine bestimmte Aufgabe zu erledigen. Beschreiben die **aktuelle** Situation → **Kontextszenario:** Beschreiben die **zukünftige** gewünschte Situation. Wird in der Anforderungsanalyse des UCD (Use-Case-Diagramm) verwendet.

Service Blueprint / Geschäftsprozessmodell
Darstellung der logischen Schritte eines (Service-) Kunden, des Service-Providers sowie weiterer beteiligter Partner. zeigt auch, wo der Kunde mit der Service-Provi-der interagiert und über welche Kanäle.

Stakeholder Map
Zeigt die wichtigsten Stakeholders im Umfeld der Problemdomäne. (App im Zentrum)



Anforderungsanalyse

User Centered Design

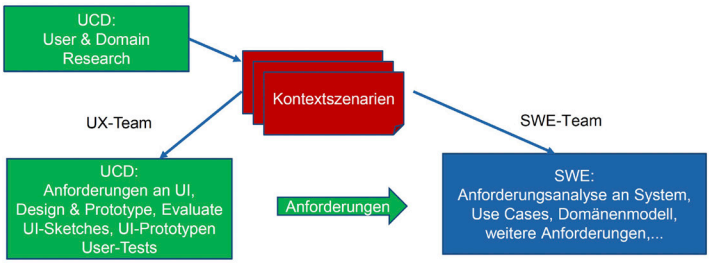
Ziele

Systemabgrenzung und Systemkontextdiagramm.
User-Anforderungen an das zu entwickelnde System ableiten:
– **Wann, wie** und **warum** interagiert der Benutzer mit dem System
– **Was** sind die Anforderungen an die Interaktion aus Benutzersicht

Artefakte

Kontextszenario, UI-Skizzen, Use Cases,
Domänenmodell, FURPS User-Story, Storyboard

Kontextszenario: Interaktionsschritte des Benutzers mit dem zukünftigen System, aus Benutzersicht, optimaler Ablauf, positiv formuliert, Prosa. Keine konkrete UI-Lösungskonzepte. Kontext für spätere Use Cases.



Use Cases

User Centered Design > Anforderungsanalyse

Was wird **wie** erledigt. Textuelle Beschreibung einer konkreten Interaktion eines bestimmten Benutzers mit dem zukünftigen System, aus Sicht des Akteurs. Use Cases müssen mindestens ein **Trigger** bzw. eine Interkation (etwas wird im System ausgelöst) beinhalten. UCs bilden die **Grundlage** für die weiteren Diagramme. UC's dokumentieren funktionale Anforderung.

Akteure

Primärakteur (Primary Actor), Unterstützender Akteur (Supporting Actor), Offstage-Akteur (Offstage Actor, interagieren nicht direkt mit dem System).

3 Ausprägungen

Brief (Kurz)
Titel + 1 Absatz.
Beschreibt Erfolgsszenario (keine Varianten oder Problemfälle).

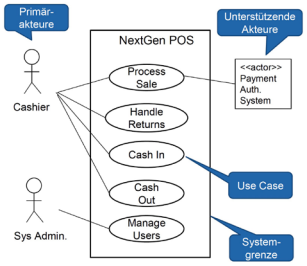
Casual (Informell)
Titel + informelle Beschreibung in ein bis mehreren Absätzen.
Beschreibt auch wichtige Varianten.

Fully dressed (Ausführlich)
Titel + alle Schritte und Varianten werden im Detail beschrieben
Enthalten weitere Informationen zu Vorbedingungen, Erfolgsgarantien, etc.

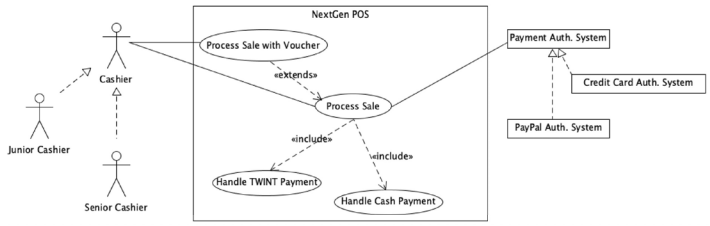
Titel: Verb + Objekt (z.B. «Kasse eröffnen»). Beschreiben das Ziel des Akteurs

Use-Case-Diagramme (Anwendungsfalldiagramm)

Zeigt: Systemabgrenzung, Akteure und Liste der Anwendungsfälle



Mit zusätzlichen Beziehungen (extends, includes)



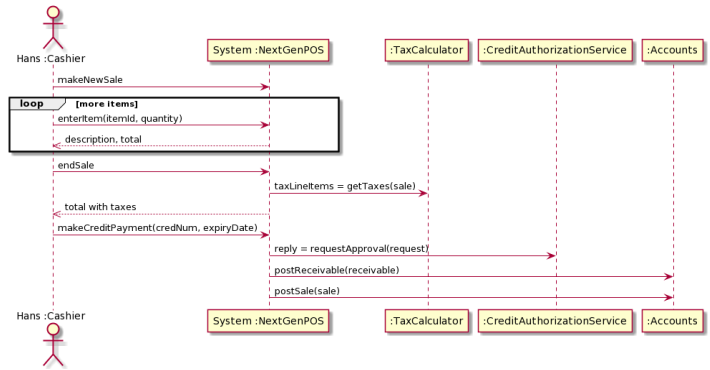
Use-Case Realisierung

Klärt, wie aus verschiedenen UML Artefakten Code abgeleitet wird.
UML-Artefakte: SSD, Domänenmodell, Systemoperationen → **Vorgehen** S.4

Systemsequenzdiagramm (SSD)

User Centered Design > Anforderungsanalyse

Zeigt: Interaktionen der Akteure mit dem System → Inputs und Outputs.
Ziel: Wichtigste Systemoperationen identifizieren



Operation Contract (Beispiel)

Vertrag: enterItem
Operation: `enterItem(itemID: IdemID, quantity: integer)`
Querverweis: UC Teilbestellung erfassen
Vorbedingung: Eine Teilbestellung ist aktiv für diesen Tisch
Nachbedingungen:
– Eine Bestellpositions-Instanz `bpi` **ist** erstellt
– `bpi` **ist** mit aktueller Teilbestellung für aktiven Tisch verknüpft
– `bpi.quantity` **ist** auf quantity gesetzt

Weitere (nicht-funktionale) Anforderungen

User Centered Design > Anforderungsanalyse

User Story

Wer, was und warum. In einem Satz. (Ich als Admin möchte xyz weil...)

FURPS+

Checkliste

– **F**unctionality (Funktionalität): Features, Fähigkeiten, etc.
– **U**sability (Gebrauchstauglichkeit): Accessibility, Affordanz, etc.
– **R**eliability (Zuverlässigkeit): Fehlerrate, Datensicherheit, etc.
– **P**erformance (Performanz): Reaktionszeiten, Durchsatz, Verfügbarkeit, etc.
– **S**upportability (Anpassungsfähigkeit): Wartbarkeit, Internationalisierung, ...
– **+**: Implementation, Interface, Operations, Packaging, Legal

Kriterien sollten möglichst **messbar** sein. Z.B. Zeitdauer für ein Check-In.

Glossar

Definiert die Begriffe, die in Projekt und im Produkt verwendet werden.

Domänenmodell

User Centered Design > Anforderungsanalyse

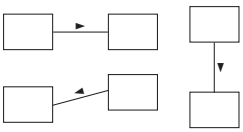
Vereinfachtes UML Klassendiagramm. Visualisiert den **Fachbereich.**
Eine Software sollte möglichst so wie die Fachdomäne strukturiert.

Konzepte anstelle Klassen. Begriff in **Einzahl!**
Konzept soll ein **Fachgebiet** abbilden. Substantiv = Konzept, Attribut

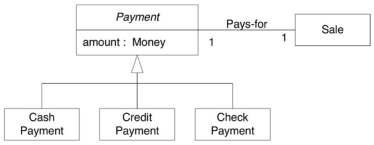
Nur analysieren, **keine Lösungen** entwerfen.

Das Konzepte können **Rollen** oder **Zustände** modellieren.

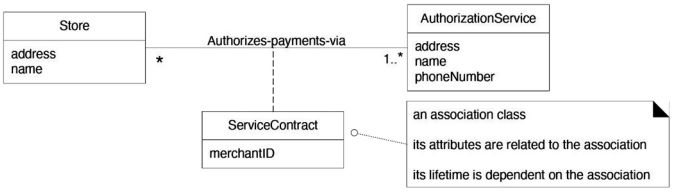
Leserichtung
(wenn nichts angegeben)



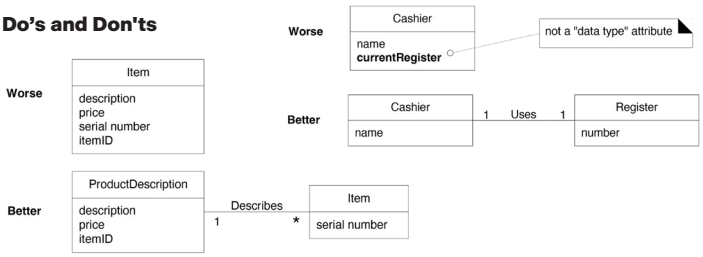
Generalisierung / Spezialisierung



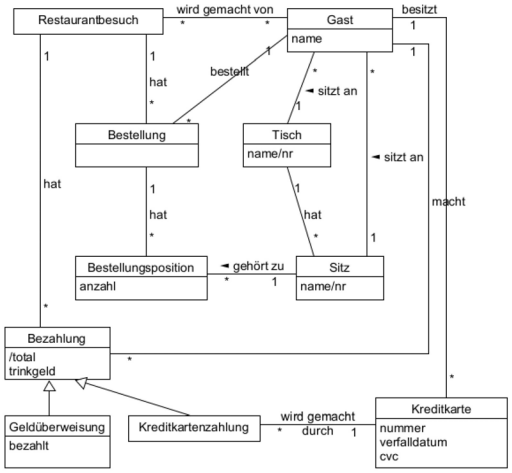
Assoziationsklassen



Do's and Don'ts



Beispiel

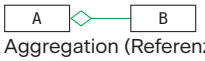


UML-Notation

Generelle Assoziation

Generelle Abhängigkeit (Dependency)

Komposition:
B ist Teil von A
B lebt nur solange A lebt.
A kontrolliert B.



Aggregation (Referenzierung):
B wird von A referenziert.
A und B haben eigene Lebensdauer.
A und B sind unabhängig.

A hat B

```
class A {
  B b = new B();
}
```

A referenziert B

```
class A {
  B b;
  void method(B b) {
    this.b = b
  }
}
```

Generalisierung / Spezialisierung
(Inheritance)

A ist B bzw. **A erbt von B**

```
class A extends B
```

Realisation
(Implementation)

A ist B

```
class A implements B
```

Softwarearchitektur und Design

Ziel: Eine **logische Architektur** aus den Anforderungen abzuleiten.

Eine logische Software-Architektur wird mit einem
UML-Paketdiagramm dargestellt.

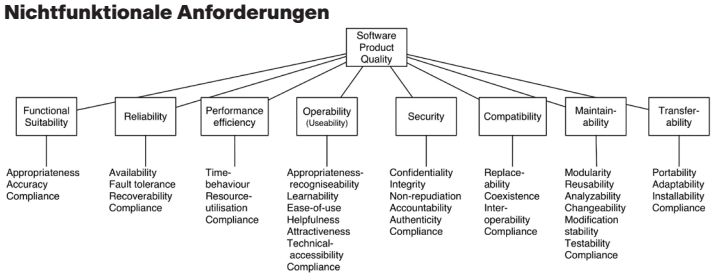
Eine gute Architektur
– unterstützt das **Refactoring**
– ist Voraussetzung für eine erfolgreiche Softwareentwicklung
– muss **heutige und zukünftige Anforderungen** erfüllen können
– muss Anforderungen erfüllen können

Softwarearchitektur ist:
– **Aufteilung** des Gesamtsystems in (möglichst) unabhängige **Teilsysteme** (Bausteine) samt deren Schnittstellen (ermöglicht unabhängige Entwickl.)
– **Verantwortlichkeiten** der Teilsysteme und ihre Abhängigkeiten
– Einsatz einer Basis-Technologie (Programmiersprachen, Frameworks)

1. **Business Analyse/Modelling**: Vision, Stakeholder, Domänenmodell, funktionale und nicht-funktionale Anforderunge, UCs, etc.
2. **Logische Architektur:** Paketdiagramm, Desigmodell
3. **Entwicklung:** Klassen-, Interaktionsdiagramm, Programmierung, Tests

Artefakte: UML-Paketdiagrammen und Architekturpatterns

Nichtfunktionale Anforderungen

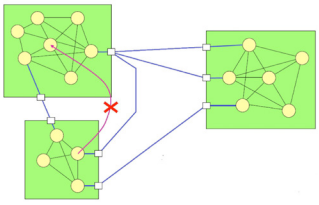


Akzeptanzkriterium: Jede Anforderung muss so formuliert sein, dass sie **gemessen** werden kann.

Moularität

Kapselung: Informationen über die **Implementierung** verbergen. Nur Informationen über das Was einer Klasse (was sie leistet) nach aussen sichtbar machen, nicht aber das Wie (ihre Realisierung). **Ziel: Hohe** Kapselung.

Kopplung: Grad der **Abhängigkeit** zwischen Klassen. In einer eng gekoppelten Klassenstruktur kann eine Änderung an einer Klasse viele Änderungen an anderen Klassen nach sich ziehen.
Ziel: Lose Kopplung



Kohäsion: Anzahl und Vielfalt der **Aufgaben,** für die eine Klasse zuständig ist.
Eine Programmeinheit sollte für genau eine in sich geschlossene Aufgabe zuständig sein. **Ziel: Hoher** Grad an Kohäsion.

Kohäsion und Kopplung beeinflussen sich gegenseitig.

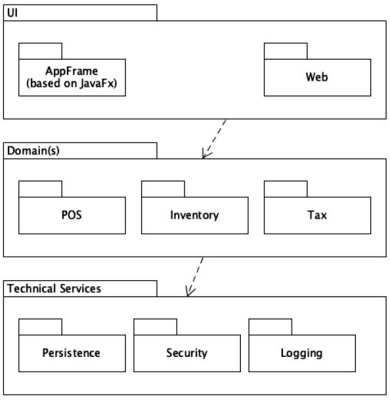
Architekturbeschreibungen

Sind in verschiedene Sichten (Views) aufgeteilt. N+1 View-Model:

- **Logical View:** Welche **Funktionalität** bietet das System an?
  Pakete, Frameworks, Klassen, Interfaces
  Systemsequenz- , Interaktions, Klassen- und Zustandsdiagramme
- **Process View:** Welche **Prozesse** laufen wo und wie ab im System?
  Prozesse, Threads, Performance, Stabilität
  Klassen- und Interaktionsdiagramme
- **Development / Implementation View:** Umsetzung der logischer Struktur.
  Source Code, Executables
  Paket- und Komponentendiagramme
- **Physical / Deployment View:** Auf welche **Infrastruktur** wird ein System ausgeliefert. Netzwerke, Protokolle, Deploymendiagramm

Logische Architektur (UML-Paketdiagramm)

Zeigt die logische Strukturierung.
Abhängigkeiten zwischen Paketen sind klar geregelt.



Architekturpatterns

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

Layered Pattern (Schichtenkonzept):
- Höherer Schichten rufen Funktionalität in unteren Schichten direkt auf.
- Untere Schicht benachrichtigt obere Schicht über Ereignis (Observer).

Pipe-Filter-Pattern: Verarbeitung von Datenströmen (z.B. Java Streams)

Broker: Ein Vermittler vermittelt die Kommunikation zwischen einem Client und dem entsprechenden Subsystem

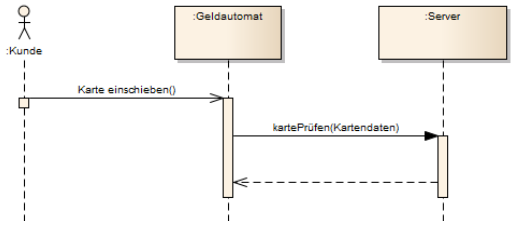
Model View Controller:
- Model: Daten und Logik
- View: Informationsanzeige
- Controller: Verarbeitung der Benutzereingabe

Design-Modelle

Objekt-Entwurf mittles:
- **Statische:** Klassendiagramm
- **Dynamische:** Interaktions-, Zustands-, Sequenz und -Aktivitätsdiagramm

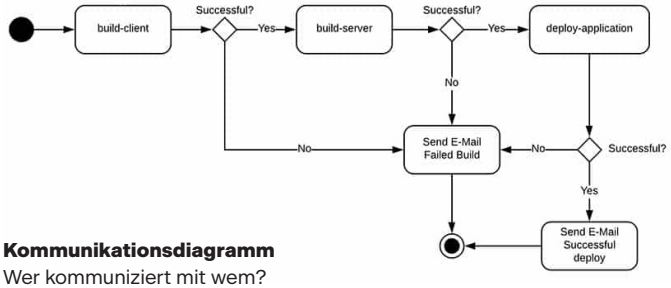
Interaktionsdiagramm

Spezifiziert, auf welche Weise Nachrichten und Daten zwischen Interaktionspartnern ausgetauscht werden. Zeitlich gestaffelt.



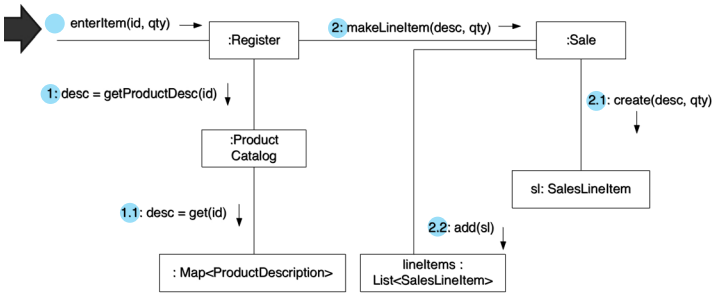
Aktivitätsdiagramm

Wie läuft ein bestimmter Prozess oder ein Algorithmus ab?



Kommunikationsdiagramm

Wer kommuniziert mit wem?



Responsibility-Driven Designs (RDD)

Für den Entwurf von Klassen. Wer hate welche Verantwortlichkeiten?
Verantwortlichkeiten werden durch **Attribute** und **Methoden** implementiert.
Kann auf jeder Ebene (Klasse, Komponente, Schicht) angewendet werden.

Ausprägungen:
- **Doing:** Algorithmen, Code
- **Knowing:** Daten, Attribute

GRASP

Neun Prinzipien (Patterns)

- **Information Expert:** Verantwortlichkeit einer Klasse zuweisen, die bereits über die erforderlichen Informationen verfügt.
- **Creator:** Wer erzeugt neue Instanz einer Klasse? Z.B. Factory
- **Controller:** Wird ein Vermittler zum delegieren benötigt?
  Der Fassaden-Controller übernimmt als erste Instanz vom UI die Ausführung der Systemoperationen.
- **Low Coupling (Abhängigkeit)**
- **High Cohesion (Zuständigkeit):** Eine Klasse sollte möglichst kohäsiv sein.
- **Polymorphism:** Klassen mit ähnlichem (polymorphen) Verhalten.
- **Pure Fabrication:** Um eine hohe Kohäsion, eine geringe Kopplung oder eine bessere Wiederverwendbarkeit zu realisieren.
- **Indirection (Observer):** Vermeiden von direkter Kopplung zwischen Objekten.
- **Protected Variations (Proxy):** Veränderungen und Instabilitäten einer EInheit sollen keinen Einfluss auf andere Elemente haben.

Ziel von Objektorientierten Designs: Klassen mit klaren Verantwortlichkeiten und Kollaborationen (Schnittstellen) zu entwerfen.

SOLID

Single-responsibility Principle: Jede Klasse nur eine einzige Verantwortung haben solle → Hoher Grad an **Kohäsion**

Open-closed Principle: Eine Einheiten sollte **Erweiterungen** möglich machen sollen (dafür offen sein), aber ohne dabei ihr **Verhalten** zu ändern.

Liskov Substitution Principle (Ersetzbarkeitsprinzip): Jede Unterklasse oder abgeleitete Klasse sollte durch ihre Basis- oder Elternklasse ersetzbar sein → implements, extends

Interface Segregation Principle: Ein Client sollte niemals gezwungen werden, eine Schnittstelle zu implementieren, die er nicht benötigt
→ grosse Interfaces aufteilen

Dependency Inversion Principle: Reduktion der **Kopplung** von Modulen. High- und Low-Level Einheiten sollten beide von Abstraktionen abhängen Eliminiert Abhängigkeiten durch und Hinzufügen von Interfaces.

Vorgehen Use-Case Realisierung

1. Controller Klasse definieren
2. Klassen, welche gebraucht werden, identifizieren
3. Verantwortlichkeiten der Klassen unter Berücksichtigung der GRASP Kriterien festlegen.
4. Kommunikationsdiagramm: Kommunikation zwischen den Objekten klären

Creational Design Patterns

Simple Factory (Creator)

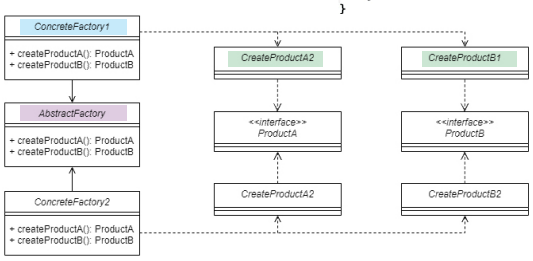
Problem: Das Erzeugen eines neuen Objekts ist aufwändig. **Lösung:** Das Erzeugen des neuen Objekts wird an eine Helfer-Klasse ausgelagert.

Abstract Factory

Problem: Erzeugung verschiedener, inhaltlicher zusammengehöreigen Objekte **(Familien von verwandten Objekten), ohne die konkrete Klasse zu kennen. Lösung:** Es wird eine Factory-Schnittstelle/abstrakten Klasse definiert. Die konkreten Factories erzeugen die verwandten Objekte.

```
// Product families
public class MacOSButton implements Button {
    public void draw() { //...
}

public class MacOSCheckbox implements Checkbox {
    public void draw() { //...
}

// Abstract Factory
public interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

GUIFactory gui = GUI.getFactory();
Button button = gui.createButton();
butotn.draw();
```

```
// Concrete Factory
public class MacOSGUIFactory implements GUIFactory {
    public Button createButton() {
        return new MacOSButton();
    }

    public Checkbox createCheckbox() {
        return new MacOSCheckbox();
    }
}

public class GUI {
    public static getFactory(String osName) {
        return switch (osName) {
            case "mac" -> new MacOSGUIFactory();
            case "windows" -> new WindowsGUIFactory();
            default -> new DefaultGUI();
        }
    }
}
```

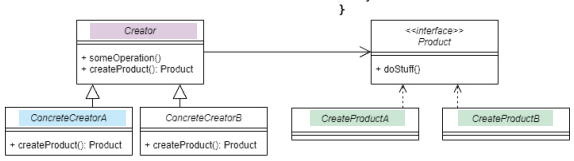


Factory Method

Eingetlich das gleiche wie Abstract Factory. **Tamplate Method** für Factories.
Problem: Eine wiederverwendbare abstrakte Klasse Creator erzeugt ein Product-Objekte. Das Product Objekt muss aber noch spezialisiert werden.
Lösung: Ein konkreter Creator erzeugt das richtige Product-Objekte.
→ Abstract Factory ist komplexer, ist aber flexibler und erweiterbar.

```
// Product families
public class Motorcycle implements Vehicle {
    public void build() { //...
}

public class Car implements Vehicle {
    public void build() { //...
}

// Client
VehicleFactory factory = new MotorcycleFactory();
Vehicle vehicle = factory.create();
```

```
// Abstract Creator (Template Method)
public abstract class VehicleFactory {
    protected abstract Vehicle createVehicle();

    Vehicle create() {
        Vehicle vehicle = createVehicle();
        vehicle.build();
        return vehicle;
    }
}

// Concrete Factory (Concrete Creator)
public class MotorcycleFactory extends VehicleFactory {
    protected Vehicle createVehicle() {
        return new Motorcycle();
    }
}
```



Builder

Problem: Eine Klasse benötigt viele Paramter im Konstruktor.
Lösung: Eine Helfer-Klasse, die es ermöglicht, solche komplexe Objekte Schritt für Schritt zu konstruieren.

```
public record Account (
  long id,
  String name,
  String iban,
  // many more ...
) {}
```

```
Account account = new AccountBuilder(id, iban)
  .setName(name)
  .setOtherParameters(param)
  .build();
```

```
public class AccountBuilder {
  private Account account;

  public AccountBuilder(long id, String iban) {
    account = new Account();
    account.setId(id)
    account.setIban(iban);
    account.setName(DEFAULT_NAME);
  }

  public AccountBuilder setName(String name) {
    account.setName(name);
    return this;
  }

  public Account build() {
    return account;
  }
}
```

Singleton

Problem: Man benötigt eigentlich nur genau eine Instanz einer Klasse.
Lösung: Eine Klasse mit einer statischen Methode, die immer dasselbe Objekt zurückliefert.

```
class Singleton {
    private static Singleton self = null;
    private Singleton() {}
    public synchronized static Singleton getInstance() {
        if (self == null) {
            self = new Singleton();
        }
        return self;
    }
}
```

```
// Meyer's Singleton
class Singleton {
    private static final Singleton self = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return self;
    }
}
```

Structural Design Patterns

Adapter (Wrapper)

Problem: Eine inkompatible Klasse (Adaptee) mit fehlendem Interface soll eingesetzt werden, darf aber nicht verändert werden.
Lösung: Eine Wrapper-Klasse implementiert das gewünschte Interface und ruft die Zielfunktion auf.

```
// Client
void clientCode(Printer printer) {
    printer.print();
}

// Target
class Printer {
    void print();
}

// Adaptee
class LegacyPrinter {
    public void printDocument() {
        // ...
    }
}
```

```
// Adapter
class LegacyPrinterAdapter implements Printer {
    private LegacyPrinter legacyPrinter;

    public LegacyPrinterAdapter(LegacyPrinter printer) {
        this.legacyPrinter = legacyPrinter;
    }

    @Override public void print() {
        legacyPrinter.printDocument();
    }
}
```

Dependency Injection

Problem: Eine Klasse braucht eine Referenz auf ein anderes Objekt.
Lösung: Anstelle, dass die Klasse abhängige Objekte selber erzeugt, wird dieses Objekt von aussen gesetzt (Objektübergabe als Parameter).
→ Aggregation (Referenzierung)

Problem: Wenn die übergebene Klasse ändert, muss unter anderem der Code angepasst werden **→ Lösung: Dependency Inversion Principle:** Das übergebene Objekt implementiert ein Interface.

Composite

Problem: Wenn eine hierarchische Baumstruktur besteht.
Lösung: Alle Klassen implementieren dasselbe Interface. Eine überliegnende Klasse leiter den Aufruf an die untere weiter. Oft ist eine hierarchische, rekursive Struktur vom Fachgebiet her gegeben, deren Elemente in bestimmten Verarbeitungen als Gesamtheit behandelt werden sollen. Alle Methoden im Composite delegieren einfach auf die Methoden der enthaltenen Elemente.

```
public interface Shape {
    void draw(Graphic g);
}

public class Circle implements Shape {
    private Position2D pos;
    private float rad;

    @Override public void draw(Graphic g) {
        g.circle(pos.x, pos.y, rad);
    }
}

public class CompositeShape implements Shape {
    private List<Shape> shapes;

    public void addShape(Shape shape) {
        shapes.add(shape);
    }

    @Override public void draw(Graphic g) {
        for (Shape shape : shapes) {
            shape.draw(g);
        }
    }
}
```

```
public class GraphicLibrary {
    private CompositeShape compositeShape;

    public void setup() {
        compositeShape = new CompositeShape();
        compositeShape.add(new Circle(1, 2, 3));
        compositeShape.add(new Rectange(4, 5, 6, 7));
    }

    public void draw() {
        compositeShape.draw();
    }
}
```

Facade

Problem: Ein ziemlich kompliziertes Subsystem mit vielen Klassen liegt vor. Wie kann seine Verwendung vereinfach werden?
Lösung: Eine Facade Klasse wird definiert, welche eine vereinfachte Schnittstelle zum Subsystem anbietet und die meisten Anwendunge abdeckt.

Decorator (Addon)

Problem: Ein Objekt soll mit zusätzlichen Funktionen versehen werden.
Lösung: Der Decorator (Wrapper), hat das (has-a) ursprüngliche Objekt und selbst dieselbe Schnittstelle hat wie dieses Objekt. Der Decorator kann nun jeden Methodenaufruf entweder selber bearbeiten oder weiterleiten
→ Realisation (Implementation) + Aggreation (Referenzierung)

```
public interface Shape {
    void draw();
}

class Rect implements Shape {
    @Override public void draw(){
        // ...
    }
}
```

```
public class FancyShape implements Shape {
    private Shape shape;

    public FancyShape(Shape shape) {
        this.shape = shape;
    }

    @Override public void draw() {
        // add fancy styles...
        shape.draw();
    }
}

Shape rect = new Rect();
Shape fancyRect = new FancyShape(rect);
fancyRect.draw();
```

Proxy

Problem: Ein Objekt ist nicht oder noch nicht im selben Adressraum verfügbar.
Lösung: Ein Stellvertreter/Platzhalter Objekt (Stub/Proxy) mit demselben Interface wird anstelle des richtigen Objekts verwendet. Das «Proxy» Objekt leitet (delegates) alle Methodenaufrufe zum richtigen Objekt weiter.

Behavioural Design Patterns

Strategy

Problem: Ein Algorithmus soll einfach austauschbar sein
Lösung: Ein Interface für diese Klasse definieren, dass von alternativen Algorithmen implementiert werden muss.

State

Strategy Pattern innerhalb eine Klasse.

Problem: Das Verhalten eines Objekt ist abhängig von seinem inneren Zustand.
Lösung: Das Objekt hat ein Zustandsobjekt. Je nach State wieder dieses Zustandsobjekt einer anderen Strategy zugewiesen. Für jeden Zustand ist eine Strategy definiert.

Ermöglicht es einem Objekt, sein Verhalten zu ändern, wenn sich sein interner Zustand ändert. Das Objekt ändert scheinbar seine Klasse.

Observer

Synonyme: Observer (Listener)/Observable, Subscriber/Publisher

Problem: Ein Objekt (Observable) soll andere benachrichtigen, ohne dass es den genauen Typ der Empfänger (Observers/Listener) kennt.
Lösung: Der Observer übergibt dem Observable eine Callback Methode (genau genommen ein Objekt mit einem @FunctionalInterface). One-to-Many-Beziehung.

Visitor

Ähnlich Observer Pattern.

Problem: Eine externe Klassen muss interne Daten einer anderen Klasse zugreifen. Diese Klass soll um Verantwortlichkeiten erweitert werden, aber ohne dass neue Methoden hinzukommen.
Lösung: Diese Klassen wird mit einem Visitable Interface erweitert. Alle weiteren neuen Verantwortlichkeiten werden dann mit spezifischen Visitor-Klassen realisiert. Die Verarbeitung wird diese Visitor-Klasse ausgelagert.

```
interface Item {
    public int accept(ItemVisitor visitor);
}

class Book implements Item {
    private int price;

    @Override
    public int accept(ItemVisitor visitor)  {
        return visitor.visit(this);
    }
}

interface ItemVisitor {
    int visit(Book book);
    int visit(Fruit fruit);
}

class ItemVisitorImpl implements ItemVisitor {
    @Override
    public int visit(Book book) {
        return book.getPrice();
    }
}

Item[] items = new Item[]{ book, fruit };
ItemVisitor visitor = new ItemVisitorImpl();

int sum = 0;
for(ItemElement item : items) {
    sum += item.accept(visitor);
}
```

```
class MyVisitor<T> implements Visitor<T> {
    StringBuilder output;

    MyVisitor() {
        output = new StringBuilder();
    }

    // Called for each element in the tree
    @Override
    public void visit(T s) {
        output.append(s);
    }

    public String toString() {
        return output.toString();
    }
}

Visitor<String> visitor = new MyVisitor<>();
tree.traversal().inorder(visitor);
String result = visitor.toString();
```

Chain of Responsibility

Problem: Für eine Anfrage gibt es potentiell mehrere Handler, aber von vornherein ist es nicht möglich, den richtigen Handler herauszufinden.
Lösung: Die Handler werden hintereinander miteinander verketteten. Jeder Handler entscheidet, ob er die Anfrage , bevor er sie an den nächsten weitergibt, bearbeiten möchte.

Template Method

Eine **Abstrakte Klasse** definiert das Grundgerüst eines Algorithmus. Die Unterklassen implementiert bestimmte Schritte.

```
abstract class Sketch {
  public Sketch() { setup(); }
  public void setup() {}
  public void draw() {}
  public void update() {}
  private void run() {
    while(true) {
      draw();
      update();
    }
  }
}
```

```
class MySketch extends Sketch {
  double x;
  void setup () { x = 0.0; }
  void update() { x += 1; }
  void draw() {
    circle(x, height / 2, 5);
  }
}
```

Memento (Snapshot)

Ermöglicht das Speichern und Wiederherstellen des vorherigen Zustands eines Objekts (do-undo). **Problem:** Eine zusätzliche immutable wrapper Klasse wird benötigt → Solution: Data Oriented Programming

```
public class Document {
  private String content;

  // ...

  public DocumentMemento createMemento() {
    return new DocumentMemento(content);
  }

  public void restoreFromMemento(DocumentMemento memento) {
    this.content = memento.getSavedContent();
  }
}

// Snapshot
public record DocumentMemento (String content) {}
```

```
// Caretaker
public class DokumentHistory {
  private List<DocumentMemento> mementos = new ArrayList<>();

  public void addMemento(DocumentMemento memento) {
    this.mementos.add(memento);
  }

  public DocumentMemento getMemento(int index) {
      return this.mementos.get(index);
  }
}

document.write("Some content");
history.addMemento(document.createMemento());
document.write("More content");
history.addMemento(document.createMemento());
document.restoreFromMemento(history.getMemento(1));
```

Command

Problem: Aktionen müssen für einen späteren Gebrauch gespeichert werden und/oder Unterstützung für ein Undo anbieten.
Lösung: Ein Invoker ruft Commands auf, weiss aber nichts vom unterliegenden Receiver. Diese Commands rufen dann Methoden des Receivers auf.

```
public interface Command {
    void execute();
}

// Concrete Command
public class TurnOnCommand implements Command {
    private Device device;

    public void execute() {
        device.turnOn();
    }
}

// Client
Device tv = new TV();
Command turnOnTVCommand = new TurnOnCommand(tv);
RemoteControl remote = new RemoteControl();
remote.setCommand(turnOnTVCommand);
remote.pressButton();
```

```
public interface Device {
    void turnOn();
}

// Concrete Receiver
class TV implements Device {
    public void turnOn() { //...
}

// Invoker (Strategy Pattern)
public class RemoteControl {
    private Command command;

    public void setCommand(Command command) {
        this.command = command;
    }

    public void pressButton() {
        command.execute();
    }
}
```

Implementation, Refactoring und Testing

Implementierungsstrategien

Code Driven Development: Zuerst die Klasse implementieren
Test Driven Development: Zuerst Tests für Einheiten schreiben
Behavior Driven Development: Tests aus Benutzersicht beschreiben

Code Smells

- Duplizierter Code
- Lange Methoden
- Klassen mit vielen Instanzvariablen
- Klassen mit sehr viel Code
- Auffällig ähnliche Unterklassen
- Keine Interfaces, nur Klassen
- Hohe Kopplung zwischen Klassen

Testing

1. **Unit-Test (Modultest):** Eine einzlene Einheit (Klasse, Methode, etc.) wird getestet.
2. **Regressionstest:** Automatische Wiederholung von Tests. Stellt sicher, dass nach dem Refactoring der Code immer noch funktioniert
3. **Integrationstest:** Keine Mockups, es werden die richtigen referenzierten Klassen eingesetzt.
4. **Systemtest (Akzeptanztest):** Die gesamte Anwendungslogik wird getestet (Black-Box Test).

Black Box Testing: Die interne Implementierung des zu testenden Objekts is unbekannt. Gegenteil ist White-Box-Testing.

Marshalling

= Serialisierung / Deserialisierung (Unmarshalling):
Umwandeln von Daten für die Übermittlung an andere Prozesse.