Inhalt

Anforderungsanalyse

Usability

User Experience:

1. UX starts by being **useful**
2. Functionality, people **must be able to use it**
3. The way **it looks and feels** must be **pleasing**
4. This helps create an overall **brand experience**

Usability = Wie einfach kann eine SW-Applikation benutzt werden

User Experience = Usability + Desirability

Customer Experience = Usability + Desirability + Brand Experience

Usability Engineering

Usability Definition: **Effektivität, Effizienz und Zufriedenheit** mit der die adressierten Benutzer ihre Ziele erreichen in ihren spezifischen Kontexten.

Ziel: Software entwickeln, welche die drei oben genannten Ziele erfüllt

- 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

Wichtigste Anforderungsbereiche bezüglich Usability

Aufgabenangemessenheit:

- Minimale Anzahl Schritte für eine Aufgabe
- Nur Informationen, die wichtig sind für eine Aufgabe
- Kontextabhängige Hilfe
- Minimal Anzahl Benutzereingaben

Selbstbeschreibungsfähigkeit:

- Benutzer ausreichend Informieren
- Begriffe des Benutzers verwenden
- Affordanzen

Kontrolle:

- Interaktion sollte der Benutzer steuern können
- Alle Benutzeraktionen sollten rückgängig gemacht werden können
- Aktionen müssen jederzeit abgebrochen werden können

Erwartungskonformität

- Bezüglich
  - Design
  - Interaktion
  - Struktur
  - Komplexität
  - Funktionalität
- Konsistenz
  - Terminologie
  - Verhalten
  - Informationsdarstellung

Fehlertoleranz

- Benutzerfehler möglichst vermeiden

    - Klar kommunizieren
    - Validieren
    - Nicht während Tippen
    - Benutzer helfen
    - Einfache Korrektur
    - Kein Datenverlust

Individualisierbarkeit

- System sollte anpassbar sein bezüglich
    - Know-How, Erfahrung des Benutzers
    - Sprache
    - Kultur
    - Benutzer mit Einschränkungen
- Beispiele
    - Für Experten
        - Tastaturkürzel
        - Automatisierung
    - Für Anfänger
        - Tooltips
        - Wizards
        - …

Lernförderlichkeit

- System sollte Benutzer Informationen über unterliegende Konzepte und Regeln anbieten
    - Um sein mentales Modell anzugleichen
    - Nur auf Verlangen des Users
    - User sollte einfach/häufige/wichtige Tasks ohne Vorkenntnisse erledigen können
    - System sollte es ermöglichen, komplexere Konzepte bei der Verwendung zu erlernen

User-Centered Design

Berücksichtigt die Bedürfnisse, Wünsche, Einschränkungen der Benutzer in jeder Phase des Design-Prozesses

User & Domain Research

Ziele bezüglich User

- **Wer** sind die Benutzer
- **Was** ist ihre Arbeit, ihre Aufgaben, Ziele ?
- **Wie** sieht ihre Arbeits- Umgebung aus?
- **Was brauchen sie,** um Ihre Ziele zu erreichen?
- Welche **Sprache** sprechen sie, welche Begriffe verwenden sie ?
- Welche **Normen** sind wichtig für sie (organisatorisch, kulturell, sozial)
- **Pain** Points in ihrer Arbeit (Brüche, Workarounds)

Hauptziele bezüglich Domäne:

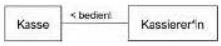
- Business der Firma verstehen
- Domäne verstehen
    - Sprache
    - Wichtigste Konzepte

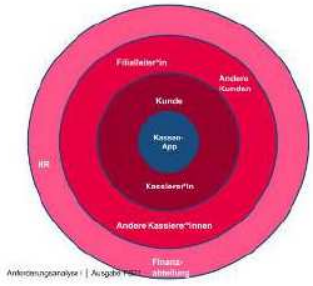
- Prozesse

Methoden

- Contextual Inquiry
- Interviews
- Beobachtungen
- Fokusgruppen
- Umfragen
- Nutzungsauswertung
- Desktop Research

Wichtige Artefakte:

- Personas
  - Fiktive Person, repräsentiert bestimmte Benutzergruppe
- Usage-Szenarien
  - Kurze Geschichte wie eine Persona ein Produkt in einer konkreten Situation benutzt, um eine bestimmte Aufgabe zu erledigen
  - Kontext → **Zukünftige gewünschte Situation**
  - Usage → beschreibt a**ktuelle** Situation
- Mentales Modell
  - 

- Domänenmodell
- Stakeholder Map
  - Zeigt die wichtigsten Stakeholder im Umfeld der Problemdomäne Bsp:
  - 

- Service Blueprint / Geschäftsprozessmodell

Anwendungsfälle (Use Cases)

Anforderungen

- Sind nie im Vorneherein vollständig bekannt.
- **Müssen** zusammen mit den Benutzern und anderen Stakeholdern **erarbeitet werden.**
  - Sie haben häufig **implizite** Anforderungen nicht explizite
  - Explizite Anforderungen sollten hinterfragt werden
- Können kaum je zu Beginn vollständig erhoben werden, sondern **entwickeln sich im Verlaufe des Projekts**
  - Problematisch bei nicht iterativem Prozesse.

Use Cases

Use Cases bilden in iterativen SWE-Prozessen eine **zentrale Rolle**

- Funktionale Anforderungen werden hauptsächlich mit UCs dokumentiert.
    - Mittels UCs können Anforderungen einfach mit dem Kunden diskutiert werden → Kommunikation
- UCs sind ein wichtiger Teil der iterativen Projektplanung
    - Projekt wird entlang von UCs geplant
- UC-Realisierungen bestimmen die Lösungsarchitektur und treiben das Lösungsdesign
- UCs werden für funktionale Systemtests eingesetzt
- UCs bilden die Basis für Benutzerhandbücher

Use Cases:

- Actor → Externe Person die mit System (SuD) interagiert während einem Anwendungsfall
- Externe Systeme, Organisationen, Maschinen können auch Akteure sein
- Auch Zeit kann Akteur sein

3 Arten von Akteuren:

- Primärakteure
    - Initiiert einen Anwendungsfall, um sein Teilziel zu erreichen
    - Erhält den Hauptnutzen des Anwendungsfalls
    - Beispiel Kasse: Kassier
- Unterstützende Akteur
    - Hilft dem SuD bei der Bearbeitung eines Anwendungsfalls
    - Beispiel Kasse: externer Dienstleister wie Zahlungsdienst für Kreditkarten
- Offstage-Akteur
    - Weitere Stakeholder, die nicht direkt mit dem System interagierten
    - Beispiel Kasse: Steurbehörde

3 Ausprägungen von Anwendungsfällen:

- Kurz
    - Titel + 1 Absatz
    - Beschreibt Standardablauf (keine Varianten, Problemfälle)
- Informell (Casual UC)
    - Titel + informelle Beschreibung in ein bis mehrere Absätze
    - Beschreibt auch wichtige Varianten
- Vollständig (Fully dressed UC)
    - Titel + alle Schritte und Varianten werden im Detail beschrieben
    - Enthalten weitere Informationen zu Vorbedingungen, Erfolgsgarantien, etc.

Weiters:

- Umfang eines guten Use Case
    - Muss einen konkreten Nutzen für den Akteur erzeugen
    - Eine Handlung die eine Person an einem Ort zu einer Zeit mit dem System ausführt
    - Sollte mehr als eine einzelne Interaktion umfassen
- Titel eines UC
    - Aktiv formulieren
        - Verb! + evtl. Objekt vorangestellt (z.B. Kasse eröffnen)
    - Sollte Ziel des Akteurs beschreiben
- Boss-Test

    - Dafuq=
- EBP-Test (Elementary Business Proc.)
    - Eine Aufgabe, die von einer Person an einem Ort zu einer bestimmten Zeit ausgeführt wird, als Reaktion auf ein Business Event.
- Size Test
    - Mehr als eine einzelne Interaktion
    - Fully dressed meist mehrere Seiten lang

Finden von Anwendungsfällen:

- Schritt 1: Systemgrenzen definieren
- Schritt 2: Primärakteure identifizieren
- Schritt 3: Jobs (Ziele, Aufgaben) der Primärakteure identifizieren

Include und extends:

- Include
    - Enthalten im haupt UC, aber keine eigenständige UC
    - Keine Verbindung zu Akteuren
- Extends
    - Eigenständiger UC, der eine Erweiterung eines anderen darstellt, und
    - Ursprünglicher UC nicht verändert werden soll
    - Sonst besser als Erweiterung im UC Text einfügen

Anwendungsfälle Brief, Casual und Fully dressed UC → LE03 Präsentation Folien ab Folie 25

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

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

Fully-dressed UC

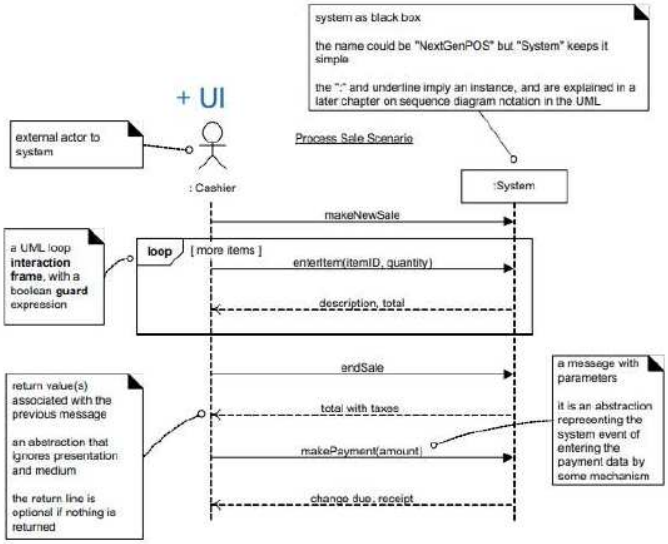
Detaillierte Beschreibung des Ablaufs mit allen Alternativszenarien (Ab Folie 27)

- Wann
    - Ende der Inception- und v.a. in Elaboration-Phase (Anforderungsdisziplin)
    - Nachdem die meisten UCs identifiziert und kurz beschrieben worden sind
    - Die **wichtigsten UCs (10%),** die die Architektur bestimmen, werden im Detail ausformuliert

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

Systemsequenzdiagramm

- 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



- Links ist Primärakteur aufgeführt

    - Initiiert die Systemoperation (via UI)
- Mitte das System (:System)
    - Muss die Systemoperation zur Verfügung stellen
- Rechts
    - Sekundärakteure, falls nötig

Systemoperation (Siehe Bild oben)

Formal wie ein Methodenaufruf

- Treffender Name, der die Absicht des Akteurs repräsentiert
- Evlt. Mit Parametern
    - Informationen, die für die Ausführung der Systemoperation nötig sind, aber noch nicht im System vorhanden sind
    - Details zu den Parametern sollten im Glossar erläutert werden
- Durchgezogener Pfeil für Methodenaufruf
- Rückgabewert
    - Kann fehlen, falls unwichtig
    - **Kein Methodenaufruf**, sondern indirekter Update des UI (deshalb gestrichelte Linie)

Um Systemoperationen zu finden, Szenario, des UCs Schritt für Schritt durchgehen → Für jeden Schritt des Akteurs überlegen, welche Systemoperationen nötig sind

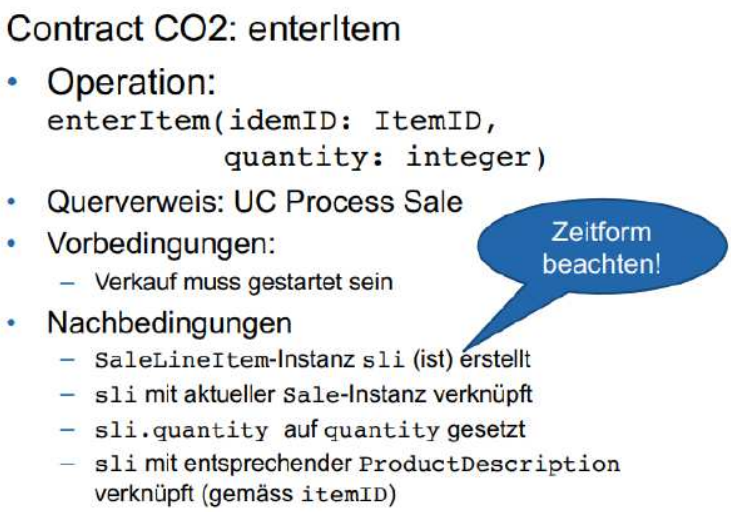
Falls Info, welche benötigt werden, um Operation auszuführen im System nicht vorhanden sind, als Parameter mitgeben.

Operation Contract

Eine (System) Operation kann mit einem Vertrag noch genauer spezifiziert werden

- Name plus Parameterliste
- Vorbedingungen
- Nachbedingug
    - Basiert auf Domändenmodell

Beispiel:



Wann soll ich Operation Contracts verwenden?

- Nur wenn aus einem Anwendungsfall nicht klar wird, was Systemoperationen genau machen muss
    - Meist nur bei sehr komplizierten Operationen und oder
    - Wenn Entwicklung der Systemoperation ausgelagert wird
- Erst gegen Ende des Meilensteins Lösungsarchitektur oder kurz vor Start des Designs der Systemoperation

Systemoperationen definieren die Schnittstelle (API) des Systems

Weitere Anforderungen

Vor allem nicht-funktionale Anforderungen siehe FURPS+

FURPS+

Checkliste für zusätzliche Anforderungen

- Functionality
    - Features, Fähigkeiten, Sicherheit
- Usability
    - Usability-Anforderungen
    - Accessibilty
- Reliability
    - Fehlerrate, Wiederanlauffähigkeit, Vorhersagbarkeit, Datensicherung
- Performance
    - Reaktionszeiten, Durchsatz, Genauigkeit, Verfügbarkeit, Ressourceneinsatz
- Supportability
    - Anpassungsfähigkeit, Wartbarkeit, Internationalisierung, Konfigurierbarkeit
- +
    - Implementation
    - Interface
    - Operations
    - Packaging
    - Legal

Glossar

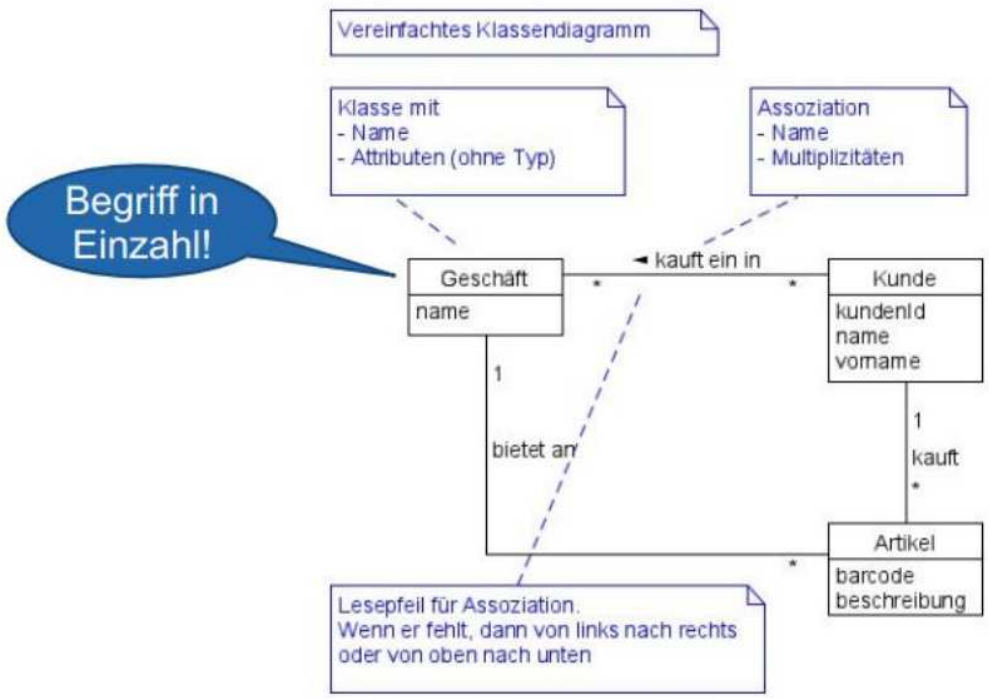
Einfaches Glossar zur Definition von Begriffen, die in diesem Projekt und im SW-Produkt verwendet werden.

Data Dictionary definiert zusätzliche Datenformate, Wertebereiche, Validierungsregeln

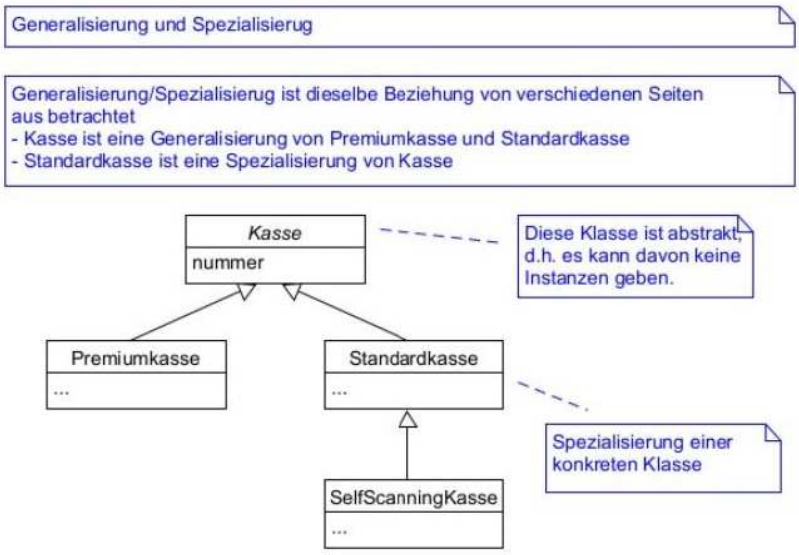
Domänenmodellierung

- Das Domänenmodell wird als **UML Klassendiagramm in einer vereinfachten Form** gezeichnet
- **Konzepte** werden als Klassen modelliert
- Eigenschaften von Konzepten werden mit **Attributen** modelliert. Die Typangabe entfällt üblicherweise
- **Assoziationen** werden verwendet, um Beziehungen zwischen Konzepten zu modellieren. Dabei beschreibt der Name der Assoziation die Beziehung und an beiden Enden werden **Multiplizitäten** angeschrieben

Beispiel:



- Mit Generalisierung / Spezialisierung kann die Assoziation noch genauer definiert werden:



Vorgehen

- Zuerst werden die **Konzepte** identifiziert
  - Eigenes oder fremdes Fachwissen und Erfahrung verwenden
  - Substantive aus Anwendungsfällen herausziehen
  - Kategorienliste verwenden
- Konzepte mit **Attributen** versehen
  - Fachwissen
- Konzepte in **Verbindung** zueinander setzen
  - Fachwissen
  - Kategorienliste anwenden
- Dabei Auftraggeber und/oder Fachexperte beiziehen
- Vorgehensweise eines Kartografen anwenden

Substantive aus Anwendungsfällen herausziehen

- Überprüfe jedes **Substantiv** von jeder Anforderung/Anwendungsfall, ob es ein relevantes Konzept des Fachgebiets beschreibt
- Beachte dabei die Mehrdeutigkeit der natürlichen Sprache
- Beispiel «Handle Sale» Use Case:
  - **Customer** arrives at POS **checkout** with **goods** and/or **services** to purchase
  - **Cashier** starts a new **sale**
  - Cashier enters **item identifier**
- Nicht alle **Substantive** sind **Konzepte,** manche sind auch Attribute oder gehören nicht zum Fachgebiet

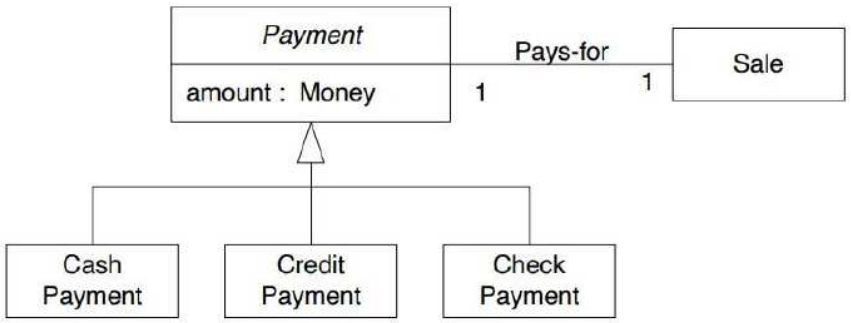
Analysemuster

Beschreibungsklassen:

- Ein Artikel ist ein physischer Gegenstand oder eine Dienstleistung, die ein Kunde kaufen kann
- Ein Geschäft hat typischerweise mehrere Artikel vom selben typ in den Verkaufsregalen
- Ein Artikel hat zumindest die Attribute Beschreibung, Preis, Serie Nummer und einen Code, der als Barcode auf der Verpackung aufgedruckt wird
- **Attribute, die für alle Artikel eines Typs gleich sind, werden in eine eigene Klasse herausgezogen**

Generalisierung und Spezialisierung:

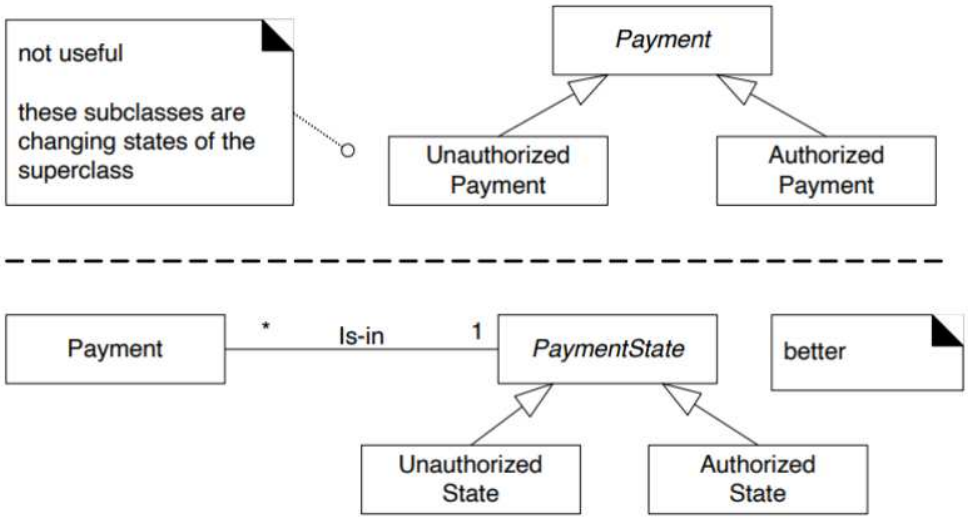
- Assoziationen und Attribute der generalisierten Klasse werden an die spezialisierten Klassen wetiergegeben



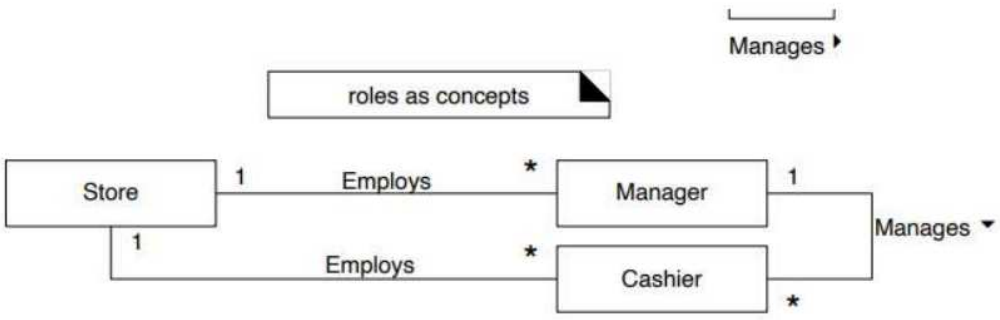
Zustände im Domänenmodell

- Verschiedene konkrete und abstrakte Konzepte haben verschiedene **Zustände**, in denen sie sich befinden
- Lösung → Eigene Hierarchie für die Zustände definieren

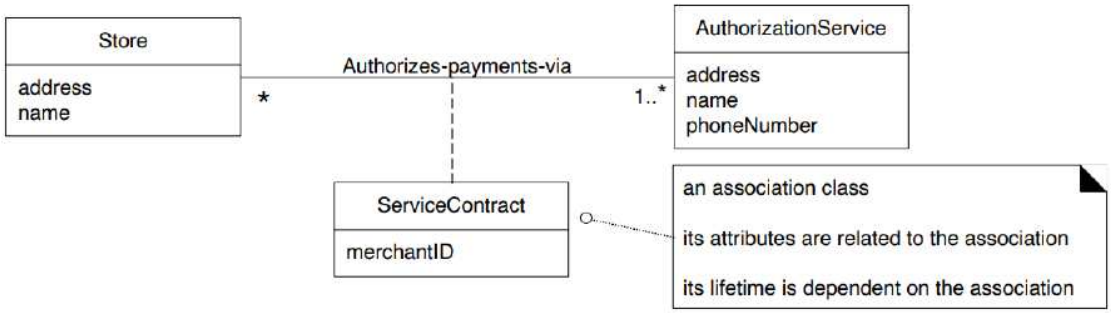
Beispiel:



Rollen als Konzepte:



Assoziationsklassen:



Masseinheiten:

- Gerade numerische Angaben sind oft mit einer Masseinheit verbunden
  - Preis, Gewicht, Volumen, Geschwindigkeit
  - Ohne Masseinheit kann die angegebene Zahl nicht korrekt interpretiert werden
- Häufig macht es Sinn, diese Masseinheit im DM explizit als Konzept zu modellieren
  - Money, Weight, Volume
- Eine entsprechende SW-Klasse kann später in der Umsetzung noch weitere hilfreiche Methoden aufnehmen
- Amount: Number → Schlecht stattdessen → Amount: Money

Zeitintervalle:

- Attribute von Konzepten sind meistens ziemlich stabil (z.B. der Name einer Person), andere Attribute werden jedoch häufig geändert
- Ist es wichtig, den Verlauf der Änderungen nachzuvollziehen und zukünftige Änderungen zu planen, muss das Attribut mit einem **Gültigkeitsintervall** versehen werden.

Softwarearchitektur und Design

- Die Architektur muss heutige und zukünftige Anforderungen erfüllen können und Weiterentwicklungen der Software und seiner Umgebung ermöglichen
- Zentrale Aufgabe der **Architekturanalyse**
  - **Analyse** der funktionalen und insbesondere nichtfunktionalen Anforderungen im Hinblick auf die Konsequenzen der Architektur
  - Unter Berücksichtigung der Randbedingungen und ihrer zukünftigen Veränderungen
  - Dabei müssen **Qualität** und **Stabilität der Anforderungen selbst** überprüft werden
    - Lücken in den Anforderungen müssen aufgedeckt werden
    - Gerade bei **nichtfunktionalen Anforderungen,** da diese häufig als selbstverständlich gesehen werden

Bausteine und Schnittstellen

Architektur beeinflusst den gesamten Systemlebenszyklus
↳ Analyse, Entwurf, Implementierung
↳ Betrieb und Weiterentwicklung
↳ Entwicklung und Betriebsorganisation

Baustein:

- Paket
- Komponente
- Library
- Kann aus weiteren Bausteinen aufgebaut sein

Hat mindestens eine Schnittstelle

Eine gute Architektur unterstützt das Refactoring
↳ & weitere Entwicklung mit zukünftigen Technologien

- Systemschnittstelle (externe Schnittstelle)
- Systeminterne Schnittstelle
    - Z. Bsp. Schicht
- Benutzerschnittstelle

Schnittstellen

Ein Modul bietet Schnittstellen an

- Sogenannte **exportierte** Schnittstellen
- Definieren angebotene Funktionalität
- Sind im Sinne eines Vertrags garantiert
- Einzige Information, die von aussen bekannt sein muss, um Modul zu verwenden
- Modul kann intern beliebig verändert werden, solange Schnittstellen gleich bleiben

Importierte Schnittstellen

- Verwendet ein Modul andere Module, so importiert sie deren Schnittstellen
- Einzige Kopplung zwischen den Modulen

Prinzip einer modularen Struktur ist:

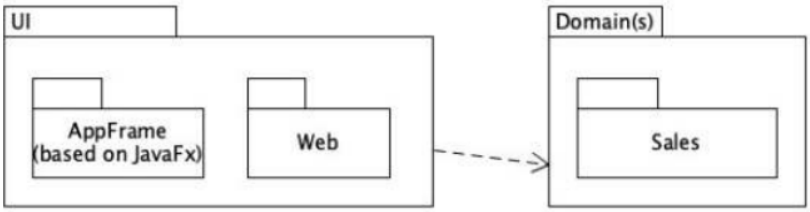
- Zwischen den Modulen möglichst schwache Kopplung und Kommunikation nur über Schnittstellen
- Innerhalb eines Moduls
    - Alle Funktionalitäten und Daten, die benötigt werden
    - Von aussen nicht sichtbar
    - Meist starker Zusammenhang

Kohäsion: Ein Mass für die Stärke des inneren Zusammenhangs

Kopplung: Ein Mass für die Abhängigkeit zwischen zwei Modulen

Ziel: Hohe Kohäsion und geringe Kopplungl

UML Paketdiagramme



UI ist abhängig von Domains

Architekturpatterns

Ab Foile 48 Softwarearchitektur und Design 1

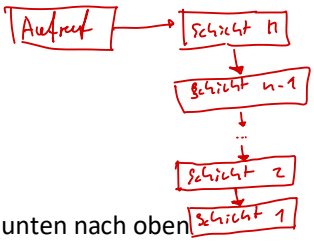
47

Layered Pattern

- Zerlegung des Gesamtsystems in Schichten
- Je weiter unten, desto allgemeiner
- Je höher, desto anwendungs-spezifischer

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

• Aufrufszenario
↳ höhere Schichten rufen Funktionalität in unteren Schichten direkt auf



- Zuoberst ist das Benutzerinterface
- **Kopplung nur von oben nach unten, NIE** von unten nach oben

Aufrufszenario:

- Höhere Schichten rufen Funktionalität in unteren Schichten direkt auf
- Untere Schicht benachrichtigt obere Schicht über Ereignis (Observer) → **Kein direkter Aufruf**

Client-Server TCP/IP

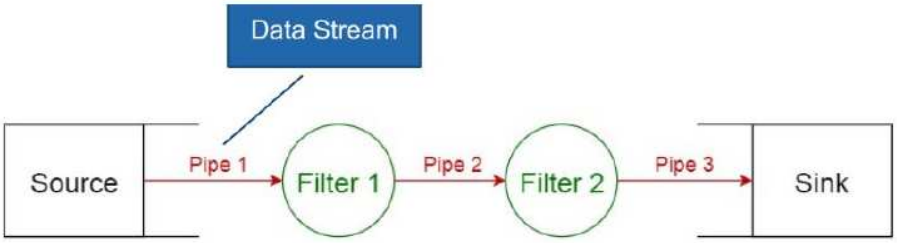
- Ein Server und mehrere Clients
- Ein Server stellt einen oder mehrere Services zur Verfügung
- Der Client macht eine Anfrage(Request) zum Server
- Der Server sendet eine Antwort (Response) zurück

Master-Slave Pattern

- Der Master verteilt die Aufgaben auf mehrere Slaves
- Die Slaves führen die Berechnung aus und senden das Ergebnis zum Master
- Der Master berechnet das Endergebnis

Pipe-Filter-Pattern

- Das Pattern kommt bei der Verarbeitung von **Datenströmen** zum Einsatz (Linux Pipe, RxJS Observable Streams, Java Streams,…)
- Jeder Verarbeitungsschritt wird durch einen **Operator** wie Filter, Mapper, etc. umgesetzt



Broker Pattern

- Das Pattern wird eingesetzt, um verteilte Systeme mit **entkoppelten Subsystemen** zu koordinieren.
- Der Broker (Vermittler) vermittelt die Kommunikation zwischen einen Client und dem entsprechenden Subsystem
- Bsp.: Message Broker

Event-Bus-Pattern

- Der Pattern umfasst vier Hauptkomponenten:
  EventSource, EventListener, Channel und Event bus
- Die Event Sources publizieren Meldungen zu einem bestimmten Channel auf dem Event Bus
- EventListeners
  - Melden sich für bestimmte Events an
  - Werden informiert, sobald sich entsprechende Meldungen auf dem Kanal befinden

Model-View-Controller

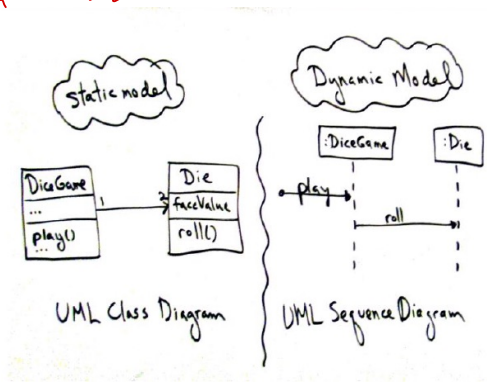
- Eine interaktive Anwendung wird in drei Komponenten aufgeteilt
  - **Model**: Daten und Logik
  - **View**: Informationsanzeige

    - **Controller**: Verarbeitung der Benutzereingabe
- Bewirkt Entkopplung von UI und Logik
- Erlaubt Austauschbarkeit der UIs
- Alternativen:
    - MVVM
    - MVP

Relevante UML Diagramme für das Design

Folien → LE 06

- Klassendiagramm
- Paketdiagramm
- Aktivitätsdiagramm
- Paketdiagramm
- Sequenzdiagramm
- Kommunikationsdiagramm
- Zustandsdiagramm



Design-Modelle

Statische Modelle:

- Statische Modelle, wie beispielweise das UML-Klassendiagramm, unterstützen den Entwurf von Paketen, Klassennamen, Attributen und Methodensignaturen

Dynamische Modelle:

- Dynamische Modelle, wie beispielsweise UML-Interaktionsdiagramme, unterstützen den Entwurf der Logik, des Verhaltens des Codes und der Methodenkörper

Statische und dynamische Modelle ergänzen sich und werden **parallel erstellt**

Klassendiagramm

- Das Diagramm beantwortet die zentrale Frage:
  Aus welchen Klassen besteht mein System und wie sind sie miteinander verknüpft?
- Es beschreibt die statische Struktur des zu entwerfenden oder abzubildenden Systems
    - Welche Klassen und Objekte existieren im System
    - Welche Attribute, Operationen und Beziehungen haben sie untereinander
    - Es enthält alle relevanten Strukturzusammenhänge und Datentypen
- Es bildet die Brücke zu den dynamischen Diagrammen
- Das UML-Klassendiagramm kann für mehrere Zwecke verwendet werden:
    - Aus einer konzeptuellen Perspektive wird ein Domänenmodell mit einem vereinfachten UML-Klassendiagramm modelliert
    - Im Design wird das UML-Klassendiagramm als Design-Klassendiagramm (DCD) mit zusätzlichen Notationselementen verwendet (Lösungsdomäne)

Interaktionsdiagramm

- Ein Interaktionsdiagramm spezifiziert, auf welche Weise Nachrichten und Daten zwischen Interaktionspartnern ausgetauscht werden
- Es gibt zwei Arten von UML-Interaktionsdiagrammen:
    - Sequenzdiagramm
    - Kommunikationsdiagramm
- Interaktionsdiagramme werden eingesetzt, um die Kollaborationen bzw. den Informationsaustausch zwischen Objekten zu modellieren (Dynamik)

- Sequenzdiagramme bieten eine reichhaltigere Notation
- Kommunikationsdiagramme eignen sich gut für Handskizzen

Sequenzdiagramm

- Das Diagramm beantwortet die zentrale Frage:
  Wer tauscht mit wem welche Informationen in welcher Reihenfolge aus?
- Es stellt den zeitlichen Ablauf des Informationsaustausches zwischen Kommunikationspartnern dar
- Es sind Schachtelung und Flusssteuerung (Bedingungen, Schleifen, Verzweigungen) möglich
- Das UML-Sequenzdiagramm kann in mehreren Perspektiven verwendet werden
  - In der Analyse werden mit einem System-Sequenzdiagramm (SSD) die Input-/Output-Ereignisse (Systemoperationen mit Rückgabeantworten) für ein Szenario eines Use Cases modelliert
  - Im Design wird das Sequenzdiagramm verwendet, um die Interaktion bzw. Kollaboration zwischen Objekten zur Realisierung eines konkreten Use-Case-Szenarios zu modellieren

Kommunikationsdiagramm

- Das Diagramm beantwortet die zentrale Frage:
  Wer kommuniziert mit wem? Wer «arbeitet» im System zusammen?
- Es stellt ebenfalls den Informationsaustausch zwischen Kommunikationspartnern dar
- Der Überblick steht im Vordergrund
- Es sind nicht alle Elemente aus dem Sequenzdiagramm unterstützt

Zustandsdiagramm

- Das Diagramm beantwortet die zentrale Frage:
  Welche Zustände kann ein Objekt, eine Schnittstelle, ein Use Case, ...bei welchen Ereignissen annehmen?
- Präzise Abbildung eines Zustandsmodells (endlicher Automat) mit Zuständen, Ereignissen, Nebenläufigkeiten, Bedingungen, Ein- und Austrittsaktionen
- Zustände können wieder aus Zuständen bestehen (Schachtelung möglich)
- Das Zustandsdiagramm wird vor allem in der Modellierung von Echtzeitsystemen, Steuerungen und Protokollen verwendet

Aktivitätsdiagramm

- Das Diagramm beantwortet die zentrale Frage:
  Wie läuft ein bestimmter Prozess oder ein Algorithmus ab?
- Es kann eine sehr detaillierte Visualisierung von Abläufen mit Bedingungen, Schleifen und Verzweigungen modelliert werden
- Es sind Parallelisierung und Synchronisation von Aktionen möglich

GRASP

- GRASP steht für General Responsibility Assignment Software Patterns und bezeichnet eine Menge von **grundlegenden Prinzipen bzw. Pattern,** mit denen die Zuständigkeit bestimmter Klassen objektorientierter Systeme festgelegt wird
- Sie beschreiben allgemein welche **Klassen und Objekte wofür zuständig** sein sollten (Verantwortlichkeiten und Kollaborationen)

Prinzipien und Pattern

- Ein Prinzip ist in diesem Kontext ein Grundsatz oder Postulat, das zu einem guten objektorientierten Design führen soll
- Ein Pattern ist ein benanntes Problem-Lösungspaar, das in neuen Kontexten angewendet werden kann, das Ratschläge enthält, wie es in neuen Situationen angewendet wird, und seine Kompromisse, Implementierungen, Varianten beschreibt
- Beschreibungsschema:
    - Name
    - Problembeschreibung
    - Lösungsbeschreibung
    - Kontext für Anwendung, Kompormisse, Variationen
- Die Idee dabei ist, mit dieser «Pattern-Language» gutes, wiederverwendbares Design zu kodifizieren

Die neun GRASP Prinzipien bzw. Patterns: General Responsibility Assignment Software Patterns

- Information Expert
- Creator
- Controller
- Low Coupling
- High Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Information Expert

- **Problem**: Gibt es ein grundlegendes Prinzip, um Objekten Verantwortlichkeiten zuzuweisen
- **Lösung, Ratschläge:** Weisen Sie die Verantwortlichkeiten einer Klasse zu, die über die erforderlichen Informationen verfügt, um sie zu erfüllen
- Alternativen: Low Coupling oder High Cohesion erfordern andere Lösung, nämlich eine «künstliche» Klasse
- Es sind auch partielle Verantwortlichkeiten möglich
- Folie 46 LE06

Creator

- **Problem:** Wer soll dafür Verantwortlich sein, eine neue Instanz einer Klasse zu erzeugen?
- **Lösung, Ratschläge:** Weisen Sie einer Klasse A die Verantwortlichkeit zu, eine Instanz der Klasse B zu erstellen, wenn eine oder mehrere der folgenden Aussagen wahr ist/sind:
    - A eine Aggregation oder Kompositum von B ist
    - A registriert oder erfasst B-Objekte
    - A arbeitet eng mit B-Objekten zusammen oder hat eine enge Kopplung
    - A verfügt über Initialisierungsdaten für B (d.h A ist Experte bezüglich Erzeugung von B)
- Wenn mehrere Optionen anwendbar sind, sollten Sie eine Klasse A vorziehen, die ein Aggregat oder ein Kompositum ist
- Alternativen: Factory Pattern, Dependency Injection(DI)

Controller

- **Problem**: Welches erste Objekt jenseits der UI-Schicht empfängt und koordiniert (kontrolliert) eine Systemoperation
- Systemoperationen wurden zuerst in der Analyse von System-Sequenzdiagrammen eingeführt. Diese nehmen die Input-Ereignisse unseres Systems entgegen
- **Lösung, Ratschläge:** Weisen Sie Verantwortlichkeit einer Klasse zu, die folgende Bedingungen erfüllt:
    - Variante 1: **Fassaden Controller**
      Sie repräsentiert das «Root-Objekt», System bzw. übergeordnetes System auf dem die Software läuft.
    - Variante 2: **Use Case Controller**
      Pro Use-Case-Szenario eine «künstliche» Klasse, in der die Systemoperation abläuft
- **Wichtig**: Controller macht selber nur wenig und delegiert fast alles!
- Wenn ein Fassaden Controller eine zu geringe Kohäsion aufweist und zu gross wird, ist ein Use Case Controller zu präferieren

Low Coupling

- **Problem:** Wie erreicht man **geringe Abhängigkeit**, wie begrenzt man die Auswirkungen von Änderungen und wie verbessert man die Wiederverwendbarkeit?
    - Kopplung = Mass für Abhängigkeit von anderen Elementen
    - Hohe Kopplung: Element ist von vielen anderen Elementen abhängig
    - Niedrige Kopplung: Element ist nur von wenigen anderen Elementen abhängig
    - Klassen mit hoher Kopplung leiden häufig unter folgenden Problemen:
        - Aufgrund von Änderungen in verbundenen Klassen sind oft lokale Änderungen nötig
        - Schwieriger zu verstehen
        - Schwieriger wiederzuverwenden, weil für ihre Verwendung auch die Klassen vorhanden sein müssen, von denen sie abhängig sind
- **Lösung:**
    - Bei der Zuweisung von Verantwortlichkeiten, darauf achten, dass die Kopplung gering bleibt

High Cohesion

- Kohäsion ist ein Mass für die Verwandschaft und Fokussierung eines Elements
- Hohe Kohäsion: Element erledigt nur wenige Aufgaben, die eng miteinander verwandt sind
- Geringe Kohäsion: Element, das für viele unzusammenhängende Dinge verantwortlich ist
- Klassen mit niedriger Kohäsion leiden häufig unter den folgenden Problemen:
    - Schwierig zu verstehen
    - Schwierig wiederzuverwenden
    - Brüchig und instabil, sind laufend von Änderungen betroffen
- **Lösung, Ratschläge:**
    - Weisen Sie Verantwortlichkeiten so zu, dass die Kohäsion hoch bleibt

Polymorphism

- **Problem**: Wie werden typabhängige Alternativen gehandhabt?
    - Operation weist viele if-then-else bzw. grosse switch-case Anweisungen auf
    - Sie möchten ein bestimmtes Verhalten (z.B. Einsatz eines externen Dienstes) konfigurierbar machen

- **Lösung, Ratschläge:** Weisen Sie das typabhängige Verhalten mit polymorphen Operationen der Klasse zu, dessen Verhalten variiert.
    - Dies ist eine der grundlegenden Ideen in der objektorientierten Programmierung (Generalisierung / Spezialisierung)
    - Achtung: Überprüfen Sie, ob es sich tatsächlich auch um eine «is a» Beziehung zwischen Superklasse und Subklassen handelt
    - L in SOLID sollte eingehalten werden

Pure Fabrication

- **Problem:** Welches Objekt sollte die Verantwortlichkeit haben, wenn Sie nicht gegen High Cohesion und Low Coupling oder andere Ziele verstossen wollen, aber die Lösungen, die beispielweise vom Information Expert vorgeschlagen werden, nicht passen ?
    - Viele Design-Klassen können direkt aus dem Fachbereich (Domänenmodell) abgeleitet werden und erfüllen Low Representational Gap.
    - Aber es gibt auch viele Situationen, wo es Probleme mit einer geringen Kohäsion, einer starken Kopplung und einer geringen Wiederverwendung gibt, wenn die Verantwortlichkeiten der Klasse in der Domänenschicht zugewiesen wird
- **Lösung, Ratschläge:**
    - Weisen Sie einen hoch kohäsiven Satz von Verantwortlichkeiten einer künstlichen Hilfsklasse zu
    - Wird nur erstellt, um eine hohe Kohäsion, eine geringe Kopplung oder eine bessere Wiederverwendbarkeit zu realisieren

Indirection

- **Problem:** Wie soll eine Verantwortlichkeit zugewiesen werden, um eine direkte Kopplung zwischen zwei oder mehr Objekten zu vermeiden? Wie können Objekte entkoppelt werden, so dass die Kopplung geringer und das Wiederverwendungspotential grösser wird?
- **Lösung, Ratschläge:** Weisen Sie die Verantwortlichkeit einem zwischengeschalteten Objekt zu, das zwischen den anderen Komponenten oder Diensten vermittelt, so dass diese nicht direkt gekoppelt sind. (vlg das D in SOLID)
- Der Vermittler schafft eine Indirektion zwischen den anderen Komponenten
- Alternativen: Protected Variations
- Viele GoF Design Patterns wie Adapter, Bridge, Facade, Observer oder Mediator verwenden dieses Prinzip
- Viele Indirections sind Pure Fabrications

Protected Variations → "so quasi" information hiding

- **Problem:** Wie sollen Objekte, Subsysteme und Systeme entworfen werden, sodass Veränderungen und Instabilitäten in diesen Elementen keinen Einfluss auf andere Elemente haben?
- **Lösung, Ratschläge:** Identifizieren Sie die Punkte, an denen Veränderungen und Instabilitäten zu erwarten sind; weisen Sie Verantwortlichkeiten so zu, dass diese Punkte durch ein stabiles Interface eingekapselt werden (O und D in SOLID)
- Es sollten zwischen folgenden Änderungspunkten unterschieden werden
    - Variationspunkt: Veränderungen sind sicher (in Anforderung); Zwingen PV Konzepte einbauen
    - Entwicklungspunkt: Veränderungen sind nicht sicher, werrden aber mit hoher Wahrscheinlichkeit eintreffen, sind nicht in Anforderungen enthalten
- Spekulative Anwendungen sind zu vermeiden, da dies zu unnötiger Komplexität führt

Entwurf mit Design Patterns

Folgende Design Patterns muss man kennen:

- Adapter
- Factory
- Singleton
- Dependency Injection
- Proxy
- Chain of Responsibility

1. Adapter
2. Simple Factory
3. Singleton
4. Dependency Injection
5. Proxy
6. Chain of Responsibility
7. Decorator
8. Observer
9. Strategy
10. Composite
11. State
12. Visitor
13. Facade

Adapter

- Eine Adapter Klasse wird zwischen zwei Klassen eingesetzt, wenn die beiden miteinander nicht kompatibel sind.
- Oft verwendet für externe Dienste, die in die eigene Anwendung integriert werden sollen

Simple Factory

- Das Erzeugen eines neuen Objekts ist **aufwändig**, deshalb eine **eigene** Klasse fürs Erzeugen von Objekten schreiben
- Oft ist die Erzeugung des neuen Objekts von irgendeiner Art von Konfiguration abhängig

Singleton

- Man benötigt von einer Klasse nur eine **einzige** Instanz und diese muss **global** sichtbar sein
- Erstelle eine Klasse mit einer statischen Methode, die immer dasselbe Objekt zurückliefert (statische Methode public deklarieren)

Dependency Injection

- Problem:
  Eine Klasse braucht eine Referenz auf ein anderes Objekt. Dieses Objekt muss ein bestimmtes Interface definieren, je nach Konfiguration aber mit einer anderen Funktionalität
- Lösung:
  Anstelle, dass die Klasse das abhängige Objekt selber erzeugt, wird dieses Objekt von aussen (Injector) gesetzt
- Ersatz für das Factory Pattern
- Direkter Widerspruch zum GRASP Creator Prinzig

Proxy

- Problem:
  Ein Objekt ist nicht oder noch nicht im **selben** Adressraum verfügbar
- Lösung:
  Ein **Stellvertreter Objekt** mit **demselben** Interface wird anstelle des richtigen Objekts verwendet
- Das Proxy Objekt leitet alle Methodenaufrufe zum richtigen Objekt weiter

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 in einer einfach verketteten Liste hintereinandergeschaltet

Jeder Handler entscheidet dann, ob der die Anfrage selber beantworten möchte oder sie an den nächsten Handler weiterleitet

- Als Variante davon leitet jeder Handler die Anfrage an den nächsten Handler weiter, unabhängig davon, ob er sie selber behandelt oder nicht
- Es könnte sein, dass gar kein Handler die Anfrage behandelt

Decorator

- Problem:
  Ein Objekt(nicht eine ganze Klasse) soll mit zusätzlichen Verantwortlichkeiten versehen werden
- Lösung:
  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.
- Strukturell identisch mit Proxy aber hat eine andere Absicht

Observer

- Problem:
  Ein Objekt soll ein anderes Objekt benachrichtigen, ohne dass es den genauen Typ des Empfängers kennt
- Lösung:
  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

Strategy

- Problem:
  Ein Algorithmus soll einfach austauschbar sein
- Lösung
  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

Composite

- Problem:
  Eine Menge von Objekten haben dasselbe Interface und müssen für viele Verantwortlichkeiten als Gesamtheit betrachtet werden
- Lösung:
  Sie definieren ein Composite, das ebenfalls dasselbe Interface implementiert und Methoden an die darin enthaltenen Objekte weiterleitet

State

- Problem:
  Das Verhalten eines Objekts ist **abhängig** von seinem **inneren Zustand**
- Lösung:
  Das Objekt hat ein darin enthaltenes **Zustandsobjekt**
  Alle Methoden, deren Verhalten vom Zustand abhängig sind, werden über das **Zustandsobjekt geführt**

Visitor

- Problem:
  Eine Klassenhierarchie soll um (weniger wichtige) Verantwortlichkeiten erweitert werden, ohne dass viele neue Methoden hinzukommen
- Lösung:
  Die Klassenhierarchie wird mit einer Visitor-Infrastruktur erweitert. Alle weiteren neuen Verantwortlichkeiten werden dann mit spezifischen Visitor-Klassen realisiert

Facade

- Problem:
  Einsatz von kompliziertem Subsystem mit vielen Klassen. Wie kann man Verwendung so vereinfachen, dass alle Team-Mitglieder es korrekt und einfach verwenden können
- Lösung:
  Eine Facade Klasse wird definiert, welche eine vereinfachte Schnittstelle zum Subsystem anbietet und die meisten Anwendungen abdeckt
- Eine Facade **kapselt**, im Gegensatz zum Adapter, ein Subsystem **nicht vollständig** ab.