2018

Zusammenfassung SWEN1

SEP FS 2019

PASCAL BRUNNER

1 Inhaltsverzeichnis

Es konnten keine Einträge für ein Abbildungsverzeichnis gefunden werden.

2 Grundlegendes

Grundsätzlich wird die Methodik des Unified Process behandelt. Heutzutage wird dieses Vorgehen oftmals von agilen Prozessen abgelöst, jedoch kann auch UP für agile Prozesse verwendet werden, sofern man diesen leicht anpasst. UP definiert oftmals kleinere Schritt zum Erfolg einer Software Anwendung. Diese Schritte werden oft als UML dokumentiert

3 Grundsatz Entwicklungsprozesse

Ein Projekt ist ein unternehmerisches Vorhaben, welches ein bestimmtes Ziel mit bestimmten Mitteln (Zeit, Budget, Personal) realisiert werden muss. Oftmals ist es sehr komplex, welches nur einmalig durchgeführt wird.

3.1 Der Software-Entwicklungsprozess

Das schrittweise Vorgehen, um eine textliche Anforderung in ein Software-Produkt zu transformieren, wird als **Software-Entwicklungsprozess** bezeichnet. Wobei ein Prozess möglichst geordnete Schritte zu einem bestimmten Ziel ist.

3.1.1 Prozess-Modell

- Reihenfolge der Arbeitsabläufe (Phasen)
- Jeweils durchzuführende Aktivitäten
- Definition der Teilprodukte (Dokumente, Prototypen, etc.)
- Abnahmekriterien für die Teilprodukte
- Verantwortlichkeiten, notwendige Mitarbeiterqualifikationen
- Standards, Richtlinien, Methoden

4 Klassische Prozess-Modelle

4.1 Code and Fix

Das Grundmodell des Programmierens ist es, zuerst ein Programm zu schreiben und anschliessend Fehler zu finden und diese zu beheben. Dies birgt jedoch den Nachteil, dass Fehlerbehebungen (oftmals konzeptionelle Fehler) eine Veränderung der Struktur bedeutet. Die Software ist fehlerhaft, da diese nicht systematisch getestet wurde. Des Weiteren wurde oft festgestellt, dass die Anforderungen der Kunden nicht erfüllt werden konnte, da diese nicht eine aktive Rolle hatten.

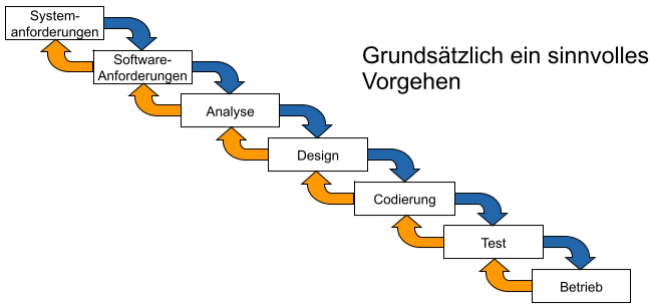
4.2 Genereller Lösungsansatz

Durch die Einführung von verschiedenen Phasen, wird das Softwareprodukt zeitlich strukturiert. Die Phasen laufen nacheinander durch:

- Analyse (der Anforderungen)
- Design (Entwurf)
- Implementation
- Test
- Einführung

4.3 Wasserfallmodell

Das Wasserfallmodell ist das klassische Prozess-Modell. Die Phasen werden in der abgebildeten Reihenfolge durchlaufen. Die nächste Phase wird erst begonnen, wenn die vorherige abgeschlossen ist. Für einen erfolgreichen Abschluss der Phase gilt auch die entsprechenden Dokumente aufbereitet zu haben. Nach dem Top-Down-Prinzip wird das Projekt realisiert von abstrakt (System) zum Konkreten (Code), wobei der Kunde nur bei der Erstellung des Pflichtenhefts beteiligt ist.



Das Wasserfallmodell ist nicht grundsätzlich schlecht, jedoch werden wichtige Fragen, bspw. Architektur, Schnittstellen von Subsystemen oder die Kundenakzeptanz, erst spät geklärt. Gerade für Kleinprojekte oder für die Einführung einer Software XYZ kann dieses Modell immer noch sehr erfolgreich sein.

Abbildung 1 Wasserfallmodell

4.3.1 Grundprobleme

Der Auftraggeber kann am Anfang noch nicht genau sagen, was er alles möchte. Erst wenn er erste Resultate sieht, kommen ihm die unterschiedlichen Anforderungen in den Sinn.

4.4 V-Modell

Das V-Modell ist eine Erweiterung des Wasserfallmodells und wurde durch zwei Aspekte erweitert:

- Verifikation (wird das Produkt korrekt entwickelt?)
- Validation (wird das richtige Produkt entwickelt?)

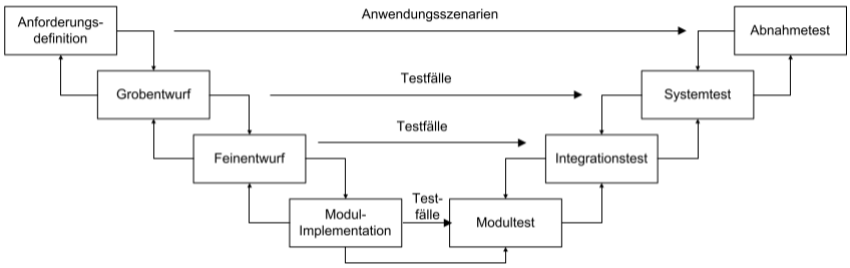


Abbildung 2 V-Modell

Bei der Anforderungsdefinition werden die Anwendungszenarien aufgestellt und durch einen Abnahmetest validiert. Des Weiteren werden beim Grob- & Feinentwurf bereits erste Testfälle entwickelt, welche für die Verifikation bei System- bzw. Integrationstest verwendet werden. Für jedes Modul wird zudem die Modultest entwickelt. Für jede Phase wird im Detail beschrieben, welche Aktivitäten von wem und welche Produkte (Dokumente) in welcher Form zu erstellen sind.

4.4.1 Vorteile

- Integrierte Methode zur Darstellung der Entwicklung des Systems, der Qualitätssicherung, des Konfigurationsmanagements und des Projektmanagements
- Generisches Vorgehensmodell
- Gut geeignet für grosse Projekte, vor allem eingebettete Systeme

4.4.2 Nachteile

- Weniger geeignet für Anwendungsbereiche und klein bzw. mittlere Systeme
- Grosser Overhead an Dokumentation
- Nur mit geeignetem CASE-Unterstützung handhabbar
- Auftraggeber meist nicht in der Lage korrekt zur formulieren
- Auftraggeber nur am Anfang / Ende mit einbezogen
- Machbarkeit muss gegeben sein

4.5 Prototypen-Modell

Als Antwort auf das Grundproblem der Wasserfallmodelle wurde das Prototypen-Modell entwickelt. Es beinhaltet dieselben Phasen, jedoch wird für jede Phase ein Softwareprototyp des Systems erstellt.

4.5.1 Softwareprototyp

- Zeigt ausgewählte Eigenschaften des Endproduktes
- Wird verwendet, um relevante Anforderungen oder Entwicklungsriskien zu klären
- Diskussionsbasis beim Kunden
- Sammeln von praktischen Erfahrungen

Man unterscheidet zwischen vier Arten von Prototypen:

- Demonstrationsprototyp (Vorbereitungsphase)
    - Akquise
    - Schnell aufgebaut (rapid prototyping)
    - Nach Gebrauch weggeworfen
- Prototyp im engeren Sinn (Analysephase)
    - Parallel zur Modellierung
    - Spezifische Aspekte veranschaulichen
    - Provisorisches, lauffähiges System

- Labormuster (Design-Phase)
    - Beantwortung konstruktionsbezogenen Fragen
    - Demonstriert technische Umsetzbarkeit
    - Modelliert meistens die Architektur oder Funktionalität
- Pilotsystem (Implementationsphase)
    - Implementiert den Kern des Systems
    - Übergang von Pilotsystem zum endgültigen Produkt ist fliessend

Mit Prototypen lassen sich normalerweise zwei Sichtweisen des Systems realisieren:

- Horizontaler Prototyp
    - Nur bestimmte Ebene (GUI, Datenbank, Middleware)
    - Ebene wird möglichst vollständig implementiert
- Vertikaler Prototyp
    - Ausgewählte Teiler über alle Ebenen
    - Bspw. für Echtzeitverhalten

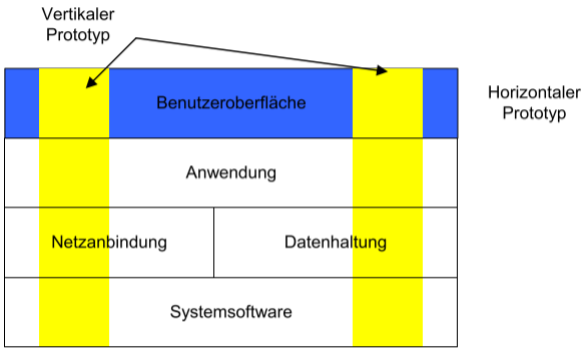


Abbildung 3 vertikaler vs. horizontaler Prototyp

4.5.2 Vorteile

- Reduktion Entwicklungsrisikien
- Sinnvolle Integration ins Prozess-Modell
- Starke Einbindung des Kunden möglich

4.5.3 Nachteile

- Hoher Entwicklungsaufwand
- Ersetzt nicht Dokumentation
- Beschränkungen / Grenzen oftmals nicht klar ersichtlich

4.6 Evolutionär / inkrementelles Modell

Grundidee ist, das Gesamtsystem in mehreren Zyklen unter Einbezug des Kunden aufzubauen. Der grobe Ablauf sieht wie folgt aus:

- Anforderungen an System werden möglichst vollständig erfasst
- Kern- und Mussanforderungen werden entworfen und implementiert (Architektur)
- Auftraggeber erhält Nullversion zum Ausprobieren
- In der nächsten Entwicklungsstufe werden die Erfahrungen und Änderungswünsche des Kunden berücksichtigt
- Auftraggeber erhält neue Version mit zusätzlichen Funktionen zum Testen
- Usw.

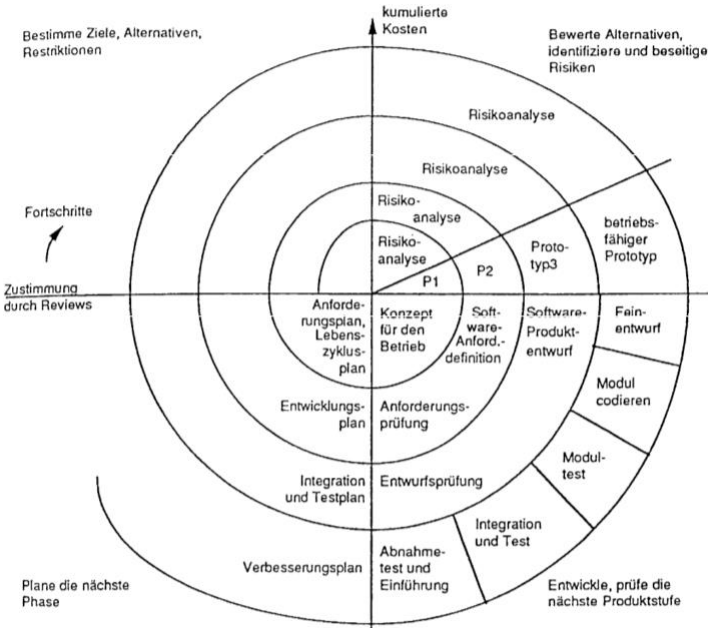
4.6.1 Vorteile

- Auftraggeber erhält regelmässig ein lauffähiges System mit dem aktuellen Entwicklungsstand
- Änderungswünsche können rechtzeitig berücksichtigt werden
- Auswirkungen können frühzeitig untersucht werden

4.6.2 Nachteile

- Ständig neue Anforderungen können den Entwicklungsprozess beliebig verlängern

4.7 Spiralmodell



Das Spiralmodell ist ein Metamodell, da es andere Prozessmodelle beinhaltet. Die Entwicklung eines Spiralmodell verläuft in mehreren Zyklen. Jeder Zyklus weist 4 Phasen auf:

- Phase 1
  - Festlegung der Ziele des Teilprodukts (Leistung, Funktionalität, etc)
  - Mögliche Alternativen und ihre Randbedingungen
- Phase 2
  - Evaluierung der Alternativen im Hinblick auf die Ziele
  - Identifzieren und Überwinden von allfälligen Risikien (bspw. Prototyp, Befragung, Simulationen)
- Phase 3
  - Anforderungsanalyse / Entwurf / Implementation und Verifikation des Produktes der nächsten Generation
  - In Abhängig der verbleibenden Risiken wird das eigentliche Prozessmodell festgelegt
- Phase 4
  - Planung des nächsten Zyklus einschliesslich Ressourcen
  - Aufteilung in (unabhängige) Komponenten
  - Überprüfung (Review) Phase 1-3 durch betroffene Personengruppe
  - Einverständnis über den nächsten Zyklus herstellen (commitment)

Die Abfolge der Zyklen wird als Spirale dargestellt (Projektbeginn in der Mitte). Je grösser die Fläche der Spirale, desto grösser die kumulierten Kosten des Projektes. Am Anfang wird versucht die Spirale möglichst eng zu halten → minimale Kosten, um die wichtigsten Risiken in den Griff zu bekommen. Das Spiralmodell ist ein Risiko-getriebenes Modell, oberstes Ziel ist es die Risikominimierung.

5 Moderne Entwicklungsprozessmodell

Da die Software Komplexität stetig gestiegen ist, wollte man mit modernen Enwicklungsprozessmodellen die Schwächen der klassischen Modellen überwinden, des Weiteren wurde es an die OO-Programmierung angepasst und die Wiederverwendung wurde besser gefördert. Dabei wird dies durch verschiedene Charakteristiken getrieben:

- Anwendungsfall getrieben
- Architektur zentriert
- Iterativ-inkrementell
- Komponentenbasiert

Das bekannsteste Entwicklungsprozessmodell ist der sogenannte Unified Software Development Process (UP)

5.1 Unified Process

Ist ein iteratives und Anwendungsfall-orientiertes Vorgehen. Die Architektur steht im Zentrum, da zuerst die Architektur erstellt wird und diese dann als Fundament für weiteres Vorgehen gilt. Dies führt dazu, dass Risikien früh minimiert werden.

Der UP sieht vor, dass er für unterschiedliche Problemstellungen angepasst (tailoring) werden kann.

5.1.1 Merkmale

UP ist komponentenbasiert, d.h. das Gesamtsystem wird aus einzelne Komponenten zusammengebaut. Ein Komponent ist dabei ein Teil des System, der

- Unabhängig von anderen Teilen entwickelt werden kann
- Wiederverwendet werden kann
- Leicht ausgewechselt werden kann
- Klare Schnittstelle nach aussen besitzt (Interfaces)
- Selbst entwickelt oder eingekauft werden kann

5.1.2 Anwendungsfälle / Anwendungsfall getrieben

Ein Anwendungsfall beschreibt in Prosa und einfach verständlich eine typische Interaktion eines Benutzers (Akteurs) mit dem System. Es gibt eine Sequenz von Aktionen, die zu einem beobachtbaren Resultat für den Akteur führt.

Akteur

Ist ein Benutzer des Systems in einer bestimmten Rolle. Derselbe Benutzer kann das gleiche System auf verschiedene Arten benutzen, wobei jeweils seine Rolle ändert.

Nutzen

Ist ein zentrales Element in UP. Ein Anwendungsfall beschreibt die funktionale Anforderung an das System. Die Gesamtheit aller Anwendungsfälle beschreibt die gesamte Funktionalität des Systems. Wichtigster Aspekt ist jedoch, dass die Anwendungsfälle die Grundlage für den ganzen Entwicklungsprozess bilden. Aus den Anwendungsfällen werden Entwurfs- und Implementationsmodelle entwickelt, die die Anwendungsfälle realisieren. Beim Testen in den verschiedenen Phasen, wird überprüft ob die Modelle die Anwendungsfälle abdecken.

Aufbau

- Vorspann: Name, Ebene (Anwenderziel oder Subfunktion), Primärakteur, Stakeholder und Interessen

- Standardablauf
    - Beschreibt den Informationsaustausch zwischen Akteur und System als neutrale Drittperson
    - Aktive Formulierung mit dem Akteur / System als Subjekt
    - Ping-Pong zwischen Akteur und System
- Erweiterungen
    - Weniger häufige Abläufe
    - Beziehen sich auf konkreten Schritt im Standardablauf
- Nachspann: Spezielle Anforderungen, Technikvariationen, Häufigkeit und Verschiedenes

Vorgehen

1. Systemgrenzen definieren
2. Primärakteure identifizieren
3. Ziele der Primärakteure identifizieren

    Nicht nach Ablauf, sondern nach Ziele fragen → öffnet Blick für neue Lösungen

    Zielhirarchie

        Oberziel (bspw. Interessanter Beruf)

        Unterziel (bspw. Studium erfolgreich absolvieren)

        Mittel zum Ziel (bspw. Modul besuchen)
4. Use Case schreiben

Varianten

- Kurz («brief»)
    - Standardablauf → 1 Abschnitt
- Informell («casual»)
    - Standardablauf → 1 Abschnitt
    - Wichtigsten Ausnahmen und Fehler als weitere Abschnitte
- Voll ausgearbeitet («fully dressed»)
    - Siehe Abschnitt Aufbau

5.1.3 Systemgrenzen

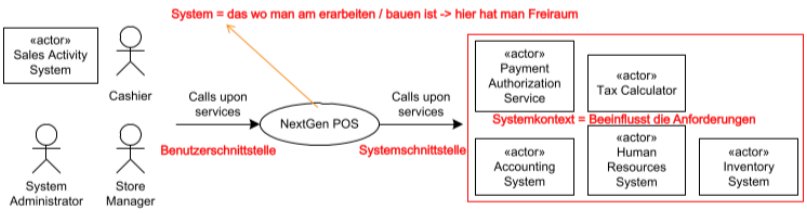


Abbildung 4 Skizze Systemgrenzen

Die Systemgrenzen werden dadurch definiert, dass die externen Akteure und Systeme aufgelistet werden.

-

5.1.4 Architektur-zentriert

Das zweite wichtige Element ist die Systemarchitektur. Sie wird parallel zu den Anwendungsfällen entwickelt, beide beeinflussen sich gegenseitig. Während der Entwicklungsprozess werden sowohl Systemarchitektur wie auch die Anwendungsfälle zunehmend verfeinert. Die Architektur dokumentiert die wichtigsten Entscheidungen über

- Organisationsstruktur des Softwaresystems
- Wichtigsten Strukturelemente, deren Zusammenspiel und Schnittstelle

- Aufteilung in Subkomponenten
- Architekturstil

Die Architektur wird durch verschiedene Sichten beschrieben.

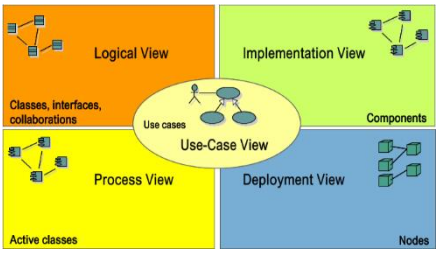


Abbildung 5 Sichten der Architektur

- Logische Sicht (Logical View)
  - Beschreibt gewünschte Funktionalität im System
  - Abstraktion des Design-Modells
  - Zeigt wichtigsten Klassen, Subsysteme und Packages
- Prozess Sicht (Process View)
  - Beschreibt welche Tasks, Prozesse, Threads zur Laufzeit existieren und wie sie zusammenspielen
- Implementation Sicht (Implementation View)
  - Beschreibt das fertige Softwareprodukt und wie Teile verwaltet werden
  - Welche zusätzlichen Dateien mit Daten, Konfiguration sind notwendig
- Verteilung Sicht (Deployment View)
  - Beschreibt wie das System ausgeliefert, installiert und die verschiedenen Knoten verteilt wird
- Anwendungsfall Sicht (Use-Case View)
  - Beschreibt die wichtigsten Anwendungsfälle und -szenarien die Architektur bestimmen

Bei diesen Architektursichten werden nur die wichtigsten statischen und dynamischen Aspekte berücksichtigt. Die Architektur muss die Realisierung von allen (jetzigen und zukünftigen) Anwendungsfälle erlauben. Die Realisierung der Anwendungsfälle muss sich auf die gewählte Architektur abstützen.

Vorgehen beim Entwickeln einer Architektur

Zu Beginn wird die Architektur nur in groben Zügen entworfen. Anschliessend werden die Hauptfunktionen des Systems im Detail spezifiziert und in Form von Subsystem, Klassen und Komponenten realisiert. Die Realisierung von Anwendungsfällen führt zu einer Verfeinerung der Architektur. Dieser Prozess wird wiederholt bis die Architektur stabil ist.

5.1.5 Iterativ und inkrementell

Die Softwareentwicklung dauert meistens mehrere Monate / wenige Jahre, deshalb muss das Vorgehen in kleinere Einheiten, sogenannte Iterationen unterteilt werden.

Iteration

Eine Iteration ist eine Art Miniprojekt, das in einem *Inkrement* des Systems resultiert. Diese Iterationen müssen kontrolliert ablaufen. Eine Iteration baut auf dem Zustand der Artefakten aus der vorhergehenden Iteration auf. Dabei durchläuft eine Iteration jeweils folgende Phasen:

- Anforderungen
- Analyse — Was soll in dieser Iteration erreicht werden (Ziel)
- Design — Wie sollen die Ziele erreicht werden (Lösungsentwurf)
- Implementation — Umsetzung des Lösungsentwurfs
- Test — In wie weit sind die Ziele erreicht worden

Wird das Ziel einer Iteration erreicht, beginnt die nächste Iteration. Wenn nicht, muss die Iteration mit einem neuen Ansatz wiederholt werden. Das Resultat einer Iteration sollte ein ausführbares

System sein, das die ausgewählten Anwendungsfälle realisiert (Art Prototyp). Das Inkrement muss nicht immer additiv sein, sondern kann auch ein bestehendes besser ersetzen.

Bei unvorhergesehen muss allenfalls eine zusätzliche Iteration eingefügt oder die Reihenfolge muss angepasst werden.

Vorteile

- Kosten- und Terminrisiko wird reduziert
- Zielgerichtete Entwicklung
- Ändernde Bedürfnisse können zielgerichtet eingeflossen werden

5.1.6 Lebenslauf eines Softwaresystems

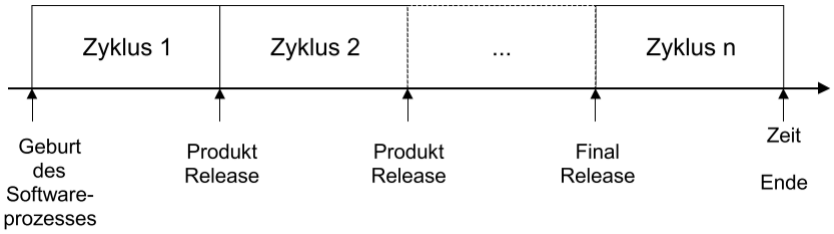


Abbildung 6 Ein Lebenszyklus

Der Lebenslauf eines Softwaresystem umfasst mehrere Lebenszyklen. Jeder Lebenszyklus endet mit einem Produkt Release an den Kunden. Typischerweise besteht der Produkt Release aus

- Source Code
- Handbücher
- Artefakten (verschiedene Modelle – siehe nächstes Kapitel)

5.1.7 Modelle

Die verschiedenen Modelle stellen eine vollständige Sicht des Gesamtsystem nach einem bestimmten Gesichtspunkt dar.

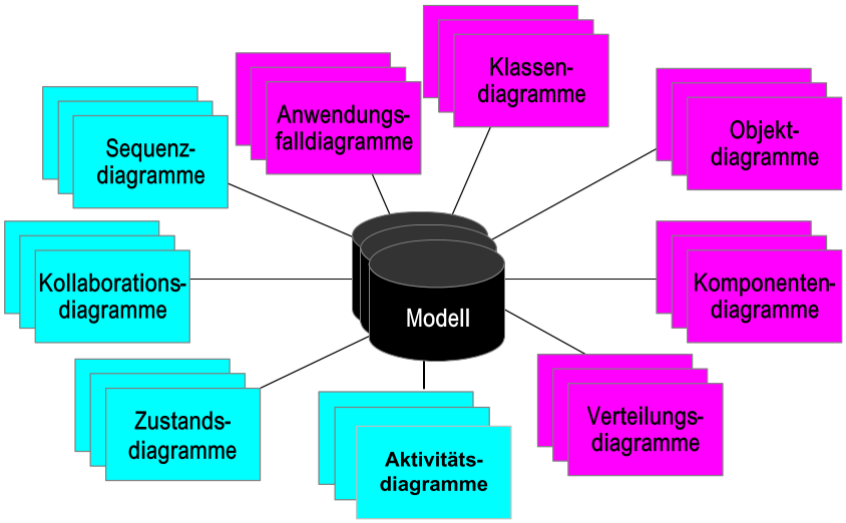


Abbildung 7 Die wichtigsten Modelle des UP

Domänenmodell (Domain Model)

Beschreibt den Problembereich des Systems in Form von Domänenklassen. Domänenklassen sind typischerweise Geschäftsobjekte (Rechnung, Konto, Vertrag etc) realse Objekte, Konzepte oder Ereignisse (Ankunft, Abfahrt, Überweisung)

Geschäftsmodell (Business Object Model)

Beschreibt die (ganzen) Geschäftsprozesse eines Unternehmens. Bspw. in Form von Anwendungsfälle und ist eine Erweiterung zum Domänenmodell.

Anwendungsfallmodell (Use Case Model)

Es enthält alle Anwendungsfälle des Systems und beschreibt die Aussenansicht des Systems. Die Beschreibung ist in Sprache des Kunden.

Analysemodell (Analysis Model, optional)

Beschreibt die konzeptionelle Innenansicht des Systems. Dient den Entwicklern, um die Anforderungen und Anwendungsfälle besser verstehen zu können. Es beinhaltet keine Implementierungsdetails. Die Beschreibung ist in der Sprache des Entwicklers

Designmodell (Design Model)

Beschreibt die konkrete Innenansicht des Systems und dienen als direkte Vorlage für die eigentliche Implementierung.

Verteilungsmodell (Deployment Model)

Beschreibt die Systemknoten (Rechner, Prozessor) und ihre Verbindungen. Des Weiteren beschreibt es die Verteilung der Komponenten auf die Systemknoten.

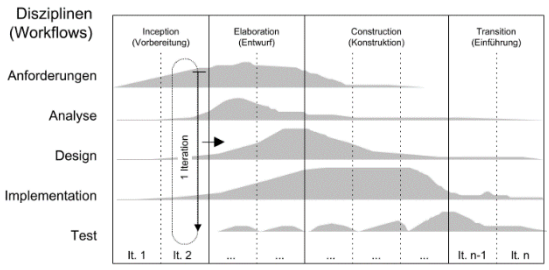
Implementationsmodell (Implementation Model)

Beschreibt das implementierte System mit all seinen Subsystemen und wie die Elemente des Designmodell implementiert werden.

Testmodell (Test Model)

Beschriebt wie die ausführbaren Komponenten (builds) in Integrations- und Systemtests getestet werden. Das Testmodell besteht aus Testplan, Testfällen, Testprozeduren, Testkomponenten und Testbericht.

5.1.8 Phasen eines Lebenszyklus



Jede Phase besteht aus einer oder mehreren Iterationen und endet mit einem Meilenstein. Der Meilenstein ermöglicht es einem Management, entsprechende Entscheidungen zu treffen und ist

definiert durch das Vorhanden sein von bestimmten Artefakten.

Abbildung 8 Verlauf eines Lebenszyklus im UP

Die vier Phasen werden in Iterationen aufgeteilt:

- Vorbereitungsphase (Inception)
  1-2 Iterationen
- Entwurfsphase (Elaboration)
  2-4 Iterationen
- Konstruktionsphase (Construction)
  2-7 Iterationen
- Einführungsphase (Transition)
  1-2 Iterationen

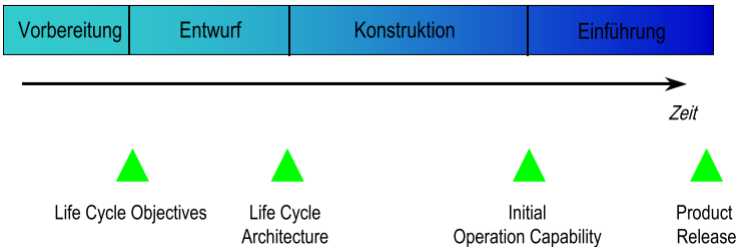


Abbildung 9 Meilenstein pro Phase

Ablauf

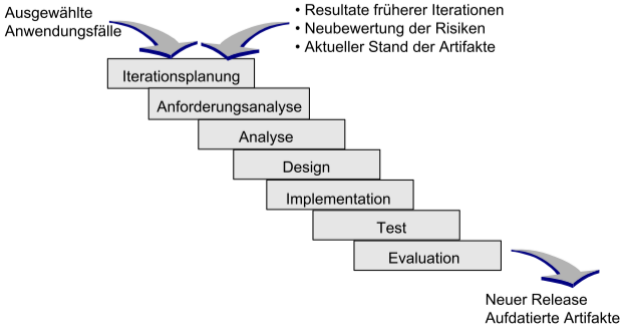


Abbildung 10 Ablauf einer Iteration

Die Iterationslänge sollte im Vorhinein geklärt werden und während des gesamten Projektes nicht verändert werden. Sollte eine Aufgabe nicht innerhalb der Iteration realisierbar sein, so sollte diese auf eine spätere Iteration erfolgen. In jeder Iteration werden die Arbeiten an fünf Disziplinen (Anforderung, Analyse, Design, Implementation, Test) erledigt und entsprechend die Artefakten angepasst. Die Arbeiten an den Disziplinen laufend grundsätzlich sequenziell ab, können sich aber auch überlappen.

5.1.9 Die 4-Phasen

Inception / Vorbereitung

Es gilt eine Entscheidungsgrundlage über den Start des Projektes vorzubereiten. Dafür wird das Ziel und der Rahmen des Projektes abgesteckt und ein erster Architekturentwurf erstellt. Des Weiteren werden die kritischen Risiken identifiziert und minimiert. Damit diese Ziele erreicht werden können werden unteranderem die Ressourcen (Budget, Time) grob eingeschätzt und geplant, die Anwendungsfälle im Detail ausgearbeitet und einen ersten Business Case entworfen.

Resultat: Meilenstein «Life Cycle Objectives» (Lebenszyklusziele)

Elaboration / Entwurf

Die Entwurfsphase dient zur Erarbeitung der systematischen Erfassung von funktionalen und nichtfunktionalen Anforderungen. Dafür wurde die Grundarchitektur festgelegt und einen Projektplan vereinbart. Des Weiteren wurde die Risiken so reduziert, dass sie während der Umsetzung bewältigt werden können. Ca. 80% der Anwendungsfälle des Systems wurden identifiziert, priorisiert und strukturiert. Die wichtigsten 10% wurden im Detail ausformuliert und evtl. bereits realisiert, jedoch sicher wurde die Architektur bestimmt.

Resultat: Meilenstein «Life Cycle Architecture» (Lebenszyklusarchitektur)

Artefakte: Domain Model (Domäne BA), Design Model (Domäne Architektur), Software Architektur Dokument (beschreibt wichtisgte Schlüsselprobleme der Architetkur und Entscheidungen)

Ziel: ca. 50 % der Anforderungen stehen und sind stabilisiert, Hauptrisiken werden verringert Architekturstehen, erste UseCases stehen, Entscheidnachvollziehbarkeit wieso wurde was gemacht?

Construction / Umsetzung

Das gesamte System wird in mehreren Iterationen, durch die Realisierung der Anwendungsfälle, implementiert. Während der Implementation sollte die Architektur nur noch minimal geändert werden.

Resultat: Meilenstein «Initial Operation Capability» (Erste Funktionstüchtigkeit)

Transition / Einführung

Während der Einführungsphase werden die letzten Änderungen zur Zufriedenheit des Auftraggebers realisiert. Hierfür wird ein Beta-Release ausgeliefert, bei welchem im Anschluss Feedback gesammelt wird. Des Weiteren werden die Artefakten fertiggestellt und es gibt ein Review des Business Plans.

Resultat: Meilenstein «Product Release»

5.1.10 Die sechs Disziplinen im Detail

Anforderungen (Requirements)

Alle Anforderungen an das System werden aufgestellt und die Umgebung wird genau beschrieben. Anforderungen müssen so beschrieben werden, dass sie jeweils überprüfbar sind. Dies ist vor allem bei den nicht-funktionalen Anforderung eine gewisse Herausforderung.

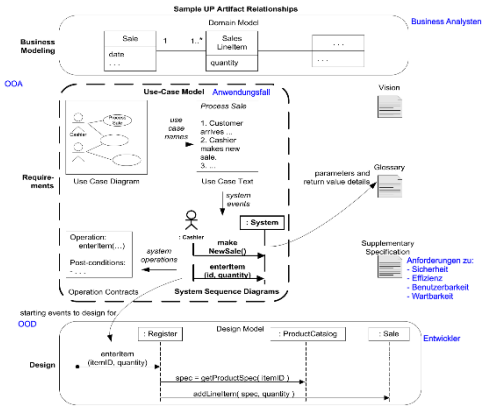


Abbildung 11 Zusammenspiel zwischen den einzelnen Rollen und Artefakten

Artefakten:

- Geschäfts- und Domänenmodell
- Anwendungsfallmodell
- Glossar
- Feature-Liste
- Liste mit den zusätzlichen Anforderungen
- Entwürfe und Prototypen der Benutzerinterfaces

Erfolgsfaktoren:

- Anforderungen mit dem Kunden aufnehmen und ihn in die Verantwortung nehmen
- Probleme analysieren
- Noch nicht in Lösungen denken
- Positive Einstellung gegenüber Änderungen der Anforderungen entwickeln

➔ Ein grosser Teil der Fehlschläge in der Software-Entwicklung basieren auf mangelhaften Anforderungen

Analyse

Die Anforderungen werden auf diversen Ebenen analysiert.

Artefakten:

- Analysemodell
- Analyse der Realisierung der Anwendungsfälle
- Architektursicht des Analysemodells

Design

Das System wird inkl. funktionale & nicht-funktionale Anforderungen entworfen.

Artefakten:

- Designmodell
- Design der Realisierung der Anwendungsfälle
- Schnittstellen (Interfaces)
- Architektursicht des Designmodells
- Einführungsmodell und Architektursicht

Implementation

In dieser Disziplin wird das System schrittweise aufgebaut. Jede Iteration fügt neue Funktionalitäten zum bestehenden System hinzu und resultiert in ein lauffähiges System (build)

Artefakten:

- Implementationsmodell
- Ausführbare Subsystem und Komponenten
- Schnittstellen (implementierte und verwendete)
- Architektursicht des Implementationsmodells
- Integrationsplan

Test

Integrations- und Systemtest werden für jede Iteration geplant, entwickelt, durchgeführt und ausgewertet.

Artefakten:

- Testmodell mit Testfällen, Testprozeduren und Testkomponenten
- Testplan
- Defekte
- Testevaluation

Zusätzliche Disziplinen

- Projektmanagement
- Entwicklungsumgebung (Environment)
- Konfigurations- und Änderungsmanagement (Configuration- and Change-Management)
- Verteilung (Deployment)

5.2 Vorteile moderne Entwicklungsprozesse

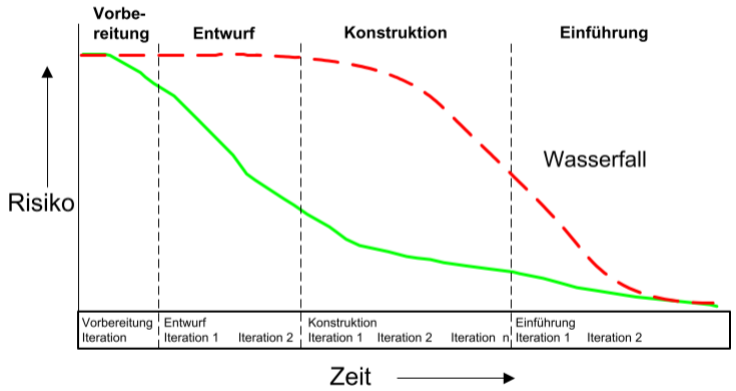


Abbildung 12 Risikoprofil bei moderne vs. klassiche Entwicklungsprozesse

Durch moderne Entwicklungsprozesse wird das Risikoprofil stark verbessert, da laufend die Risiken angepasst und angegangen werden können. Jede Iteration liefert nicht nur neue Artefakten, sondern auch ein lauffähiges System. Durch den iterativen Prozess, lasst sich der Auftraggeber besser einbinden und die neuen Anforderungen, Erkenntnisse und Risiken können laufend berücksichtigt werden. Des Weiteren können Entwickler sich besser auf kurzfristige Ziele fokussieren.

6 Projektmanagement

6.1 Projektplanung

Die Idee des Softwareprojektes inkl. Abklärungen, Analyse und erste Anwendungsfälle werden im Rahmen einer Projektskizze festgehalten.

6.1.1 Arbeitspakete

Damit die Arbeiten sinnvoll geplant und überwacht werden können, werden Arbeitspakete geschnürt. Ein Arbeitspaket beinhaltet, das erwartete Ergebnis, Termine, Aufwandschätzung und aufgelaufene Kosten. Die Arbeitspakete bilden die Grundlage für die Terminplanung, sowie für die Überwachung des Budgets während des Projektfortschritts.

Die Aufteilung kann in verschiedenen Ansätzen erfolgen:

- Subsysteme / Komponenten
- Entwicklungsprozesse
- Organisationeinheit
- Geographisch

6.1.2 Projektstrukturplan

Der Projektstrukturplan ist das Resultat der Aufteilung in Arbeitspakete. Der Projektstrukturplan ist ein kritischer Faktor für den Projekterfolg. Es muss darauf geachtet werden, dass er nicht Designentscheidungen vorwegnimmt. Es gibt zusammen mit dem Budget einen sehr guten Aufschluss über den Entwicklungsansatz, die Prioritäten und Risiken aus Sicht des Projektleiters.

Klassischer Projektstrukturplan

Der klassische Ansatz entsteht durch die Aufteilung des Systems in Subsysteme und Komponenten, welches folgende Probleme mit sich bringt:

- Arbeitspakete werden aufgrund des Produktdesigns definiert, welche zu diesem Zeitpunkt jedoch noch gar nicht steht
- Aufteilung ist sehr projektspezifisch
- Projektübergreifende Vergleiche sind sehr schwierig bis unmöglich

Evolutionärer Projektstrukturplan

Der evolutionäre Ansatz strukturiert die Arbeitspakete anhand des Entwicklungsprozesses, dies kann die zu erwartenden Anpassungen im Projektplan besser berücksichtigen. Dies führt dazu, dass Planungsgenauigkeit zusammen mit dem Entwicklungsprozess sukzessiv entwickelt. Dieser Projektstruktur gilt als Startpunkt für die konkrete Planung und wird laufend angepasst.

6.1.3 Software-Entwicklungsplan

Der Software-Entwicklungsplan ist der eigentliche Projektplan, wo der Entwicklungsprozess instanziiert wird. Dieser Plan muss die Vertragsvereinbarungen und ev. An interne Standards halten. Der Software-Entwicklungsplan legt die Details zum gesamten Prozess fest:

- Welche Disziplinen werden jeweils durchgeführt
- Welche Artefakten erstellt werden müssen
- Zeitplan
    - Phasenplan
        - Grobe Zeitplanung in Anfangsphase des Projektes
        - Start, Ende, Phasen

        - Pro Phase
            - Anzahl Iterationen
            - Welche Artefakten werden in dieser Phase wie weit bearbeitet
            - Meilensteine
    - Iterationsplan
        - Detaillierte Planung für eine Iteration
        - Wird am Ende einer Iteration für die nächste Iteration erstellt

6.2 Risikomanagement

Für die Projektplanung ist es wichtig, die möglichen Risiken stets im Auge zu behalten und möglichst früh anzugehen. Das Risikomanagement ist gemäss ISO 31'000 wie folgt definiert:

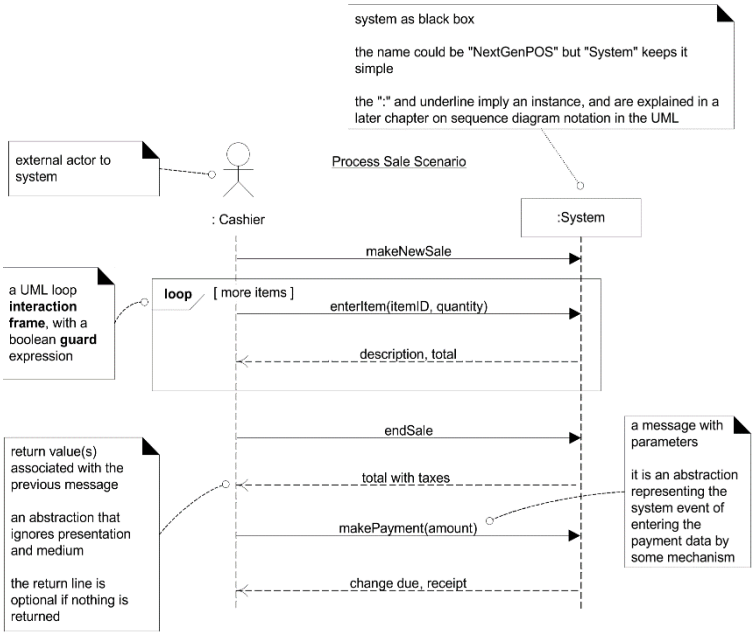
- Risk Assessment
    - Risk Identification
        - Was sind Riskien und Ursache
    - Risk Analysis
        - Ursache analysieren
        - Abhängigkeiten
        - Konsequenzen
        - Eintrittswahrscheinlichkeit
        - Risikomatrix
    - Risk Evalutation
        - Priorität der Behandlung eines Risikos
- Risk Treatment
    - Risiko bewusst angehen
    - Einkalkulieren
    - Vermeiden
    - Ursache eliminieren
    - Wahrscheinlichkeit vermindert
    - Impact vermindern
    - Risiko aufteilen
- Risikomatrix
    - Einstufung der Risikos anhand Konsequenzen und Eintrittswahrscheinlichkeit (Einteilung in 3 oder 5 Stufen)

Likelihood	Consequences				
	Insignificant	Minor	Moderate	Major	Severe
Almost certain	M	H	H	E	E
Likely	M	M	H	H	E
Possible	L	M	M	H	E
Unlikely	L	M	M	M	H
Rare	L	L	M	M	H

Abbildung 13 Beispiel Risikomatrix

6.3 Artefakte

6.3.1 System Sequenz Diagram (für PSIT3 optional)



Widerspiegelt den «Klickaufwand» zwischen Benutzer und System. Ist vor allem ein Dokument, welches als Grundlage für eine Diskussion zur Benutzerakzeptanz gilt.

Die **Motivation** eines solchen Dokumentes ist, dass man versteht welche Ereignisse von einem Akteur auf das System einwirken. Jedes Systemereignis wird von einer Systemoperation behandelt. Des Weiteren kann ein SSD auch die Interaktion zwischen mehreren Systemen aufzeigen.

Abbildung 14 System Sequenz Diagram

Das SSD hilft dazu, eine klare Schnittstelle zwischen System, UI und Domain zu erstellen. So wird die Logik nicht im UI abgebildet.

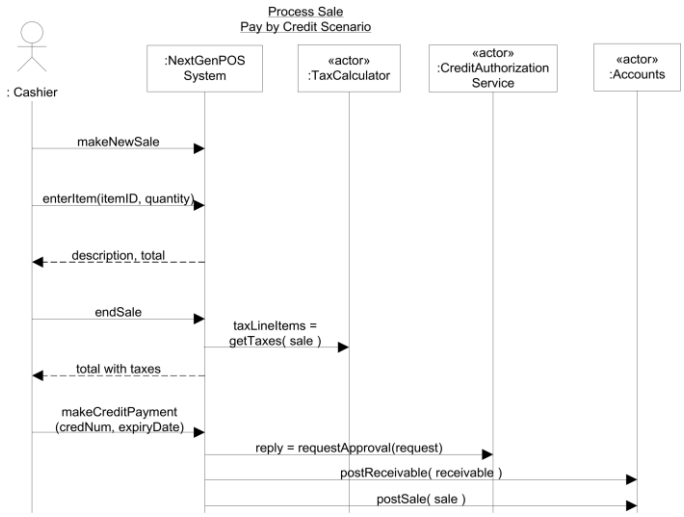


Abbildung 15 SSD für Interaktion zwischen Systeme

6.3.2 Domänenmodell

Es wird nur der Fachbereich dargestellt und die Software Entwicklung soll nicht im Vordergrund stehen. Es sind keine Lösungselemente im Domänenmodell vorzufinden. Das Domänenmodell hilft für neue Mitarbeitenden schneller in das Thema einzulesen.

Domänenmodel (DM) ist die Analyse und das DCD ist für die Entwicklung

6.3.3 Sequenzdiagramm

Bei einem Sequenzdiagramm ist die Zeitachse relevant (man liest von oben nach unten), wobei der Nachrichtenaustausch von links nach rechts erfolgt. Gute Einsatzmöglichkeit von Sequenzdiagramme sind Wiederholungen, Bedingungen, sowie hierarchische Diagramme

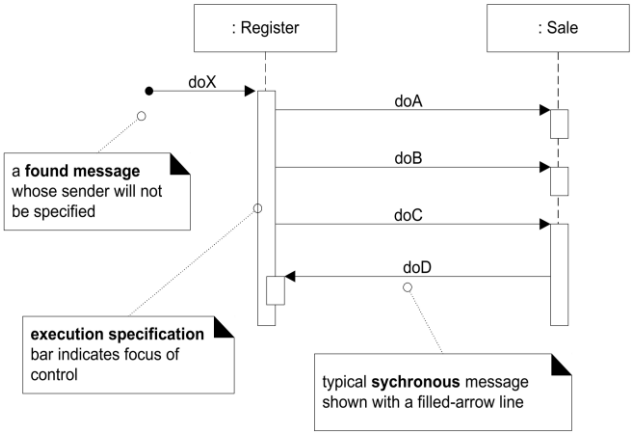


Abbildung 16 Beispiel-Sequenzdiagramm

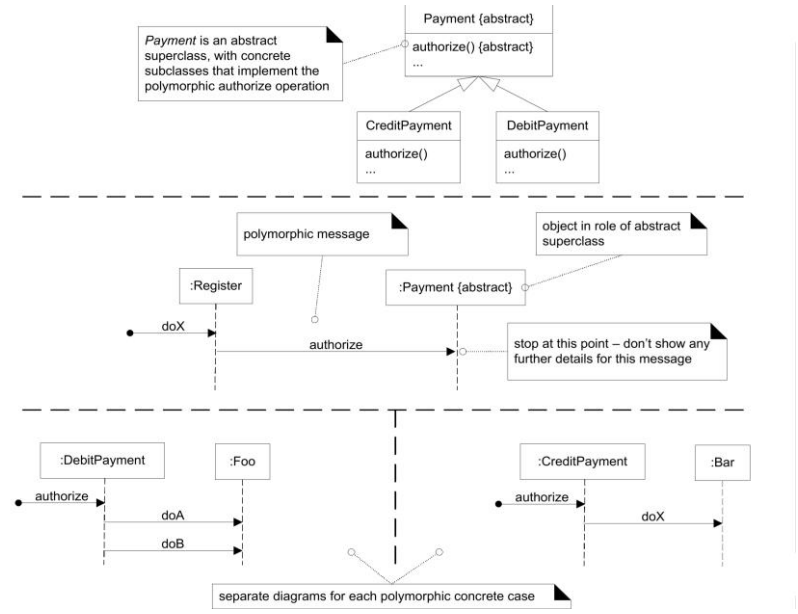


Abbildung 17 Sequenzdiagramm für ein polymorphes Verhalten

Polymorph: *Beispiel* Eine Nachricht kann je nach User / Mensch unterschiedlich aufgefasst werden.

6.3.4 Kommunikationsdiagramm

Stellt semantisch das gleiche dar wie das Sequenzdiagramm. Bei einem Kommunikationsdiagramm ist die Nummerierung der Nachrichten für den zeitlichen Ablauf relevant (Verschachtelung möglich). Im Vergleich zu einem Sequenzdiagramm ist es platzsparender. Vor allem für Handzeichnungen geeignet.

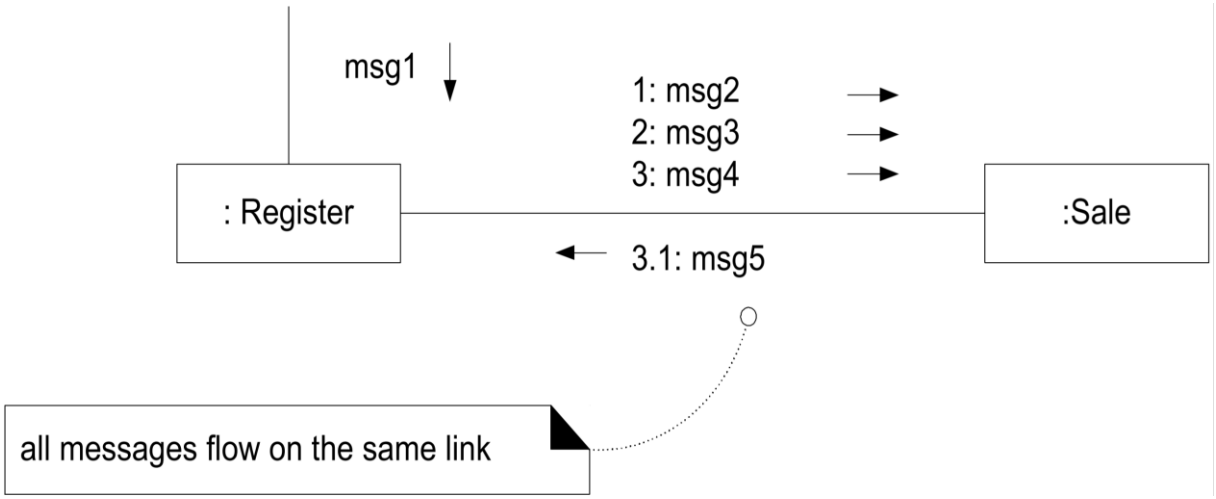


Abbildung 18 Beispiel Kommunikationsdiagramm (Nachricht)

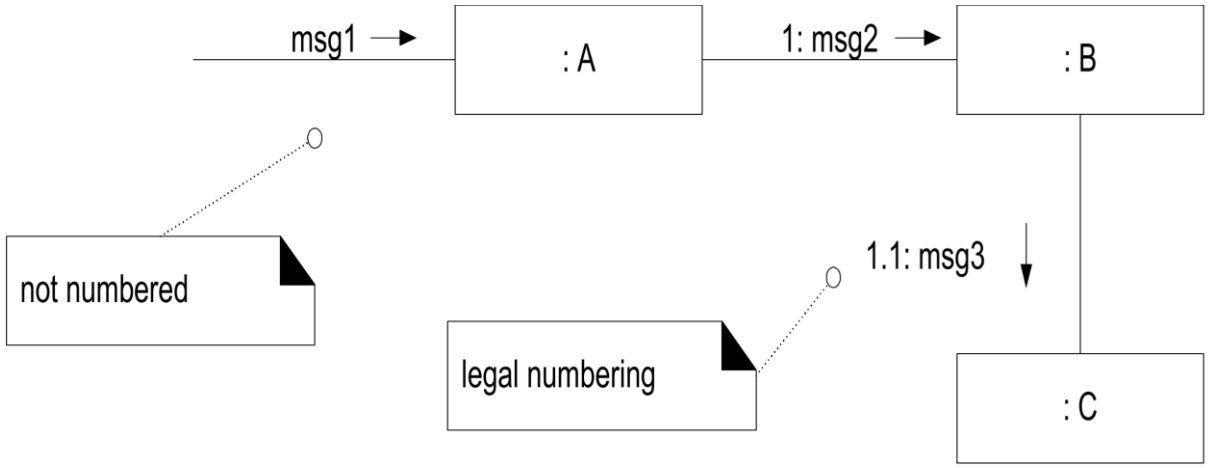


Abbildung 19 Beispiel Kommunikationsdiagramm (verschachtelte Nummerierung)

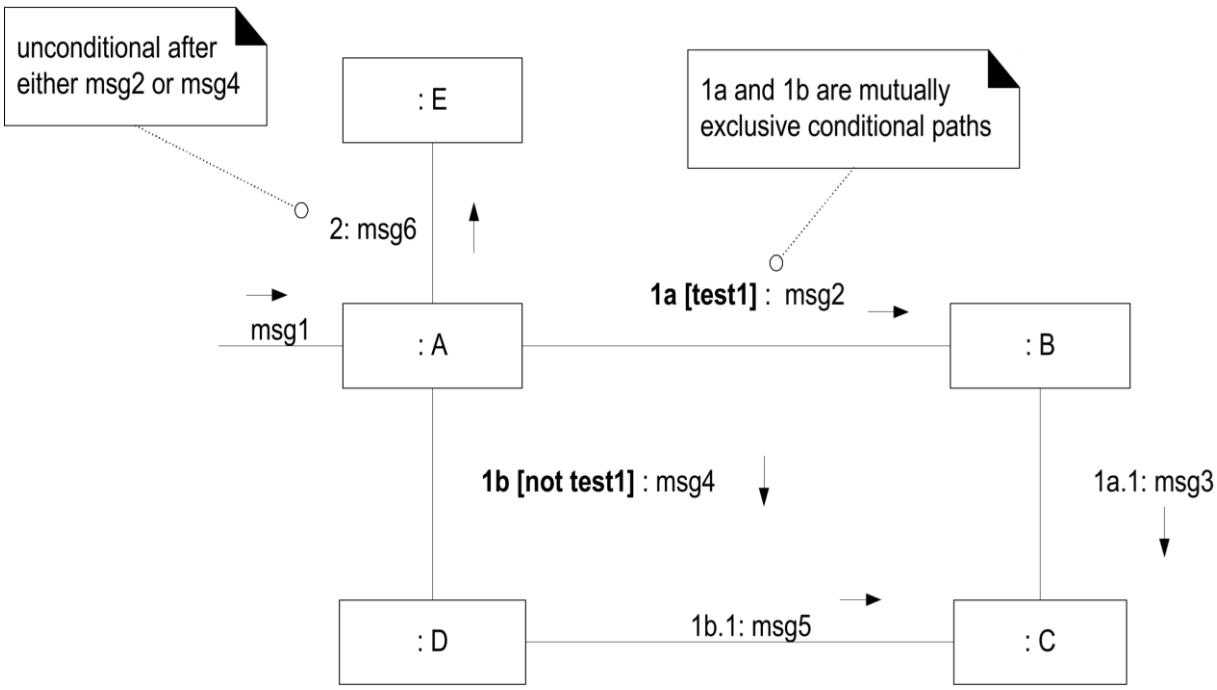


Abbildung 20 Beispiel Kommunikationsdiagramm (Bedingung)

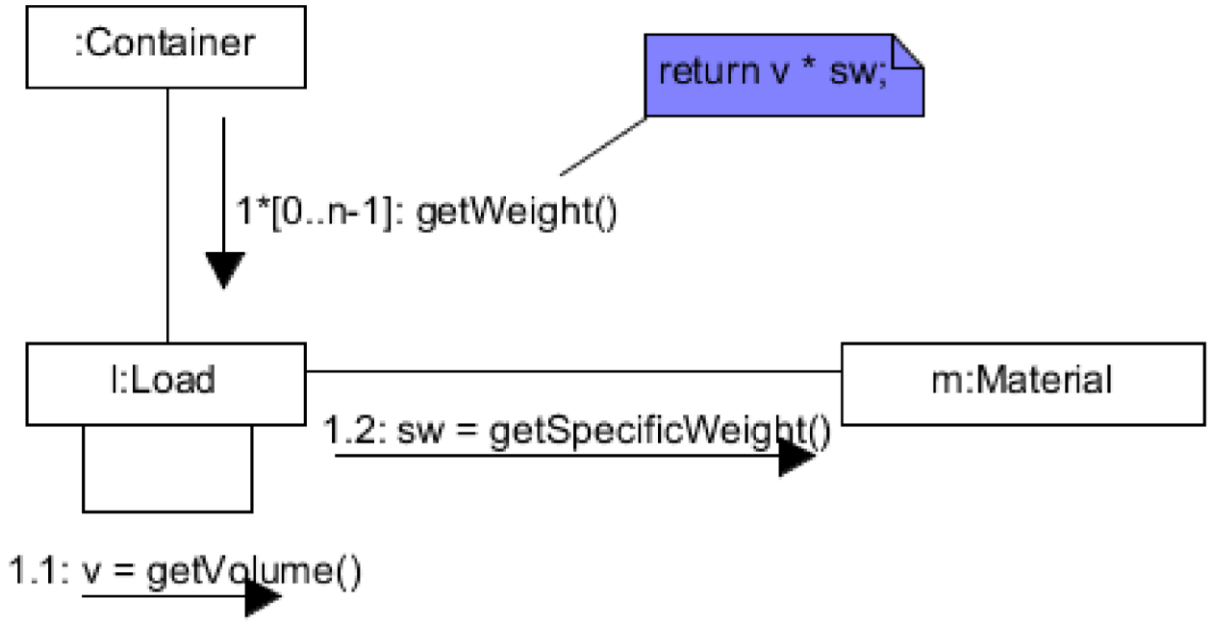


Abbildung 21 Beispiel Kommunikationsdiagramm (Schlaufe)

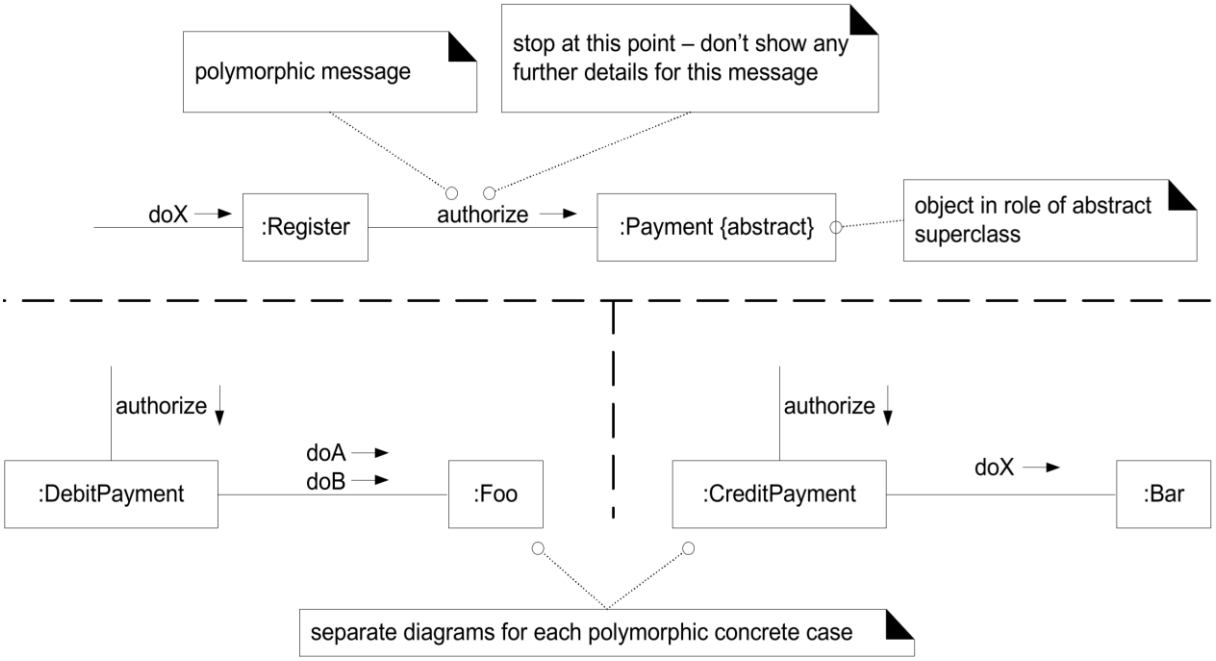
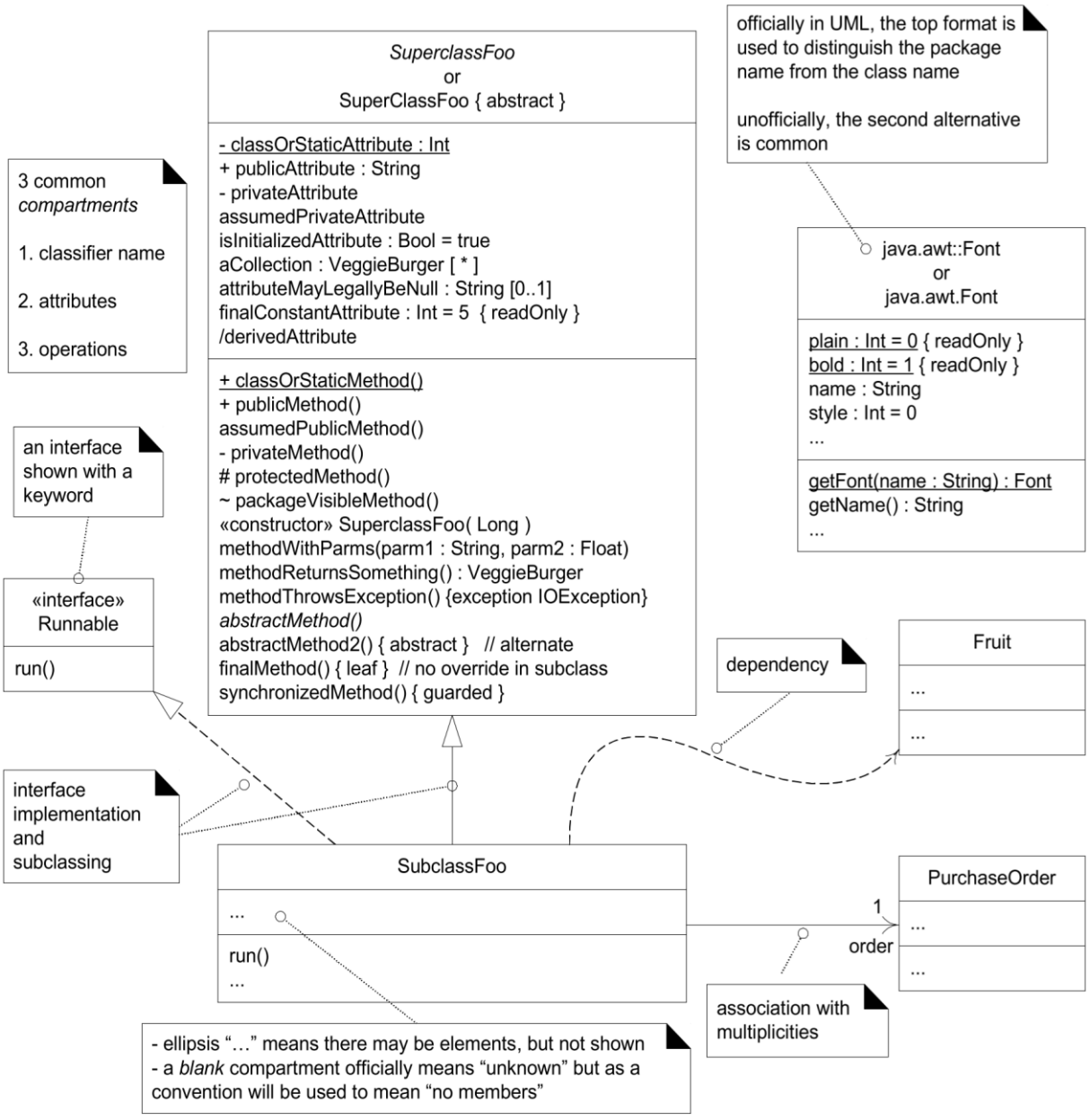


Abbildung 22 Beispiel Kommunikationsidiagramm (polymorph)

6.3.5 Klassendiagramm

Composite: «besteht aus» ein Beispiel dafür wäre ZHAW zu Departement. Die ZHAW braucht ein Department und das Department braucht die ZHAW.



6.4 GRASP Entwurfsmuster

- G = General
- R = Responsability
- A = Assignment
- S = Software
- P = Pattern

Grundidee: Dabei geht es um ein Responsability Driven Design. Ein gesamtes Projekt kann nicht von einer Person allein organisiert werden. Aus diesem Grund werden die Aufgaben auf unterschiedliche Personen aufgeteilt und man organisiert sich, wie der Nachrichtenaustausch stattfinden soll. Bei den Personen gibt es zwei Ausprägungen

1. Doing → Die machen etwas bspw. Algorithmen oder Code

2. Knowing → Er kennt wie man es machen muss, bzw. kennt jemand der weiss wie man es machen muss

GRASP wird eher nach Prinzipien geführt

- Information Expert (1)
    - Wichtigstes Prinzip der Zuteilung von Verantwortlichkeiten
    - «Der weiss über dieses Thema alles»
    - Er braucht sämtliche Informationen, damit er die Aufgabe erfüllen kann
    - Niedrige Kopplung -> man fragt jeweils den Informations Experten
    - Partielle Verantwortlichkeiten sind möglich
- Low Coupling (2)
    - Geringe Kopplung ist anzustreben
    - Es braucht jedoch eine minimale Kopplung, damit die Objekte miteinander kommunizieren
    - Qualitätskriterium für OO-Design
- High Cohesion (3)
    - Man sol lauf eine möglichst hohe innere Kohäsion achten
        - Modularer Entwurf
        - Ergänzt sich gut mit low coupling
    - Klare, eindeutige und schmale Verantwortlichkeit
- Controller (4)
    - Fassaden Controller
        - Root-Objekt, System, übergeordnetes System
    - Use-Case Controller
        - Pro Anwendungsfalle eine künstliche Klasse
- Creator (5)
    - Befasst sich mit der Frage wer eine neue Instanz einer Klasse zu erstellen
    - Sogenannte Factory Klasse ist eine Klasse mit mehreren Methoden, welche ein Objekt zurückliefert
    - Unterstützt geringe Kopplung
    - New, Factory, IoC als Möglichkeiten
- Polymorphismus (6)
    - Referenz zu Refactoring
    - Einsatz eines Interfaces als Superklasse für mehr Flexibilität
    - Grundkonzept von OO
- Pure Fabrication (7)
    - Künstliche Hilfsklasse, die High Cohesion unterstützt
    - Reine Erfindung eines SW-Entwicklers
    - Führt zu funktionsorientierem Entwurf
- Indirection (8)
    - Verantwortung so zuweisen, dass eine direkte Kopplung vermieden wird
    - Vermittler-Objekt dazwischenschalten
    - Geringere Kopplung
- Protected Variation (9)
    - Stabiles Interface dazwischenschalten
    - Sehr wichtig
    - Grundlegend
    - Basis für effiziente iterative Entwicklung und Wartung

7 Design Code

Sonar = Code Analyse

CQS = Command Query Sequence → Abfrage die etwas tut oder abfragt, aber nicht beides

Voraussetzung für Refactoring → man hat Unit-Tests mit Test-Coverage von grösser als 2/3

8 Testen

9 Gang of Four (GOF)

9.1 Adapter Pattern

- Structural Pattern
- Anpassung inkompatibler Schnittstellen (Interfaces)

Das Adapter Pattern bietet eine Brücke (Adapter) zwischen zwei inkompatiblen Interfaces. Dabei wird eine Klasse erstellt, welche für die Verbindung aller Funktionalitäten von verschiedenen unabhängigen Interfaces ist.

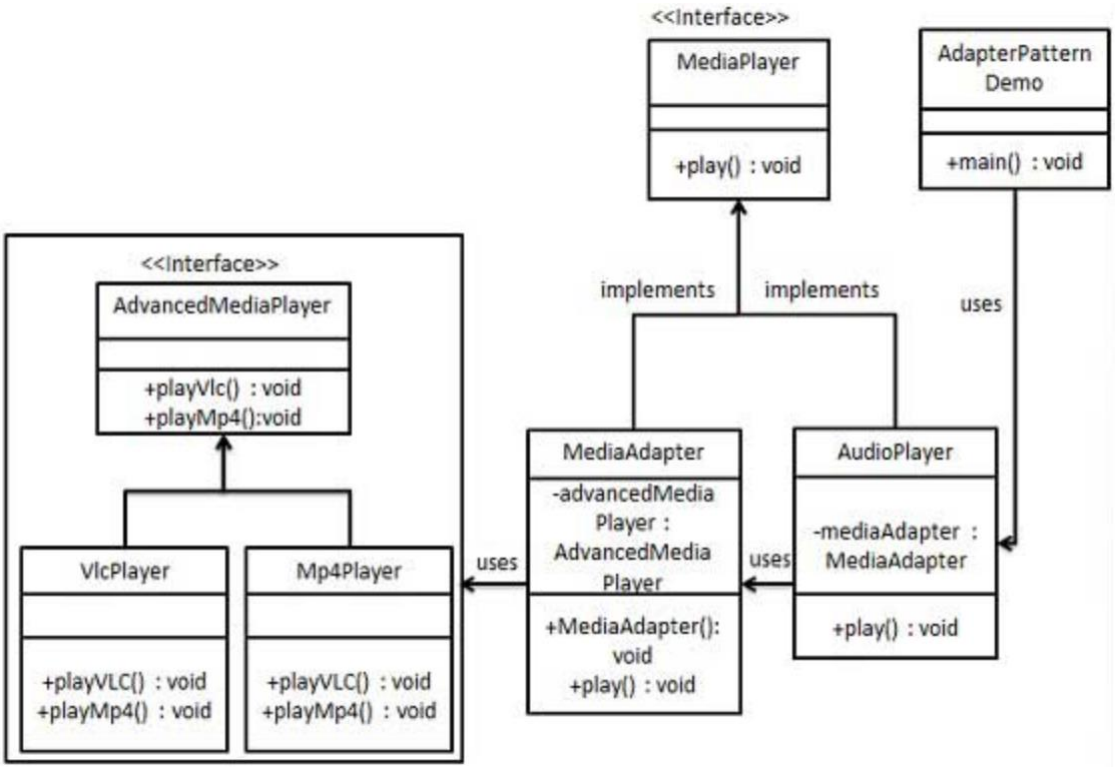


Abbildung 23 UML Adapter Pattern

9.2 Factory

- Creational Pattern
- Instanz der Factory Klasse erzeugt Instanzen einer anderen Klasse

Das Factory-Pattern ist eines der meist verwendeten Design Patterns, es ist eine geeignete Arte Objekte zu erzeugen. Ein neues Objekt wird erzeugt, ohne dass die dahinterstehende Logik bekannt ist.

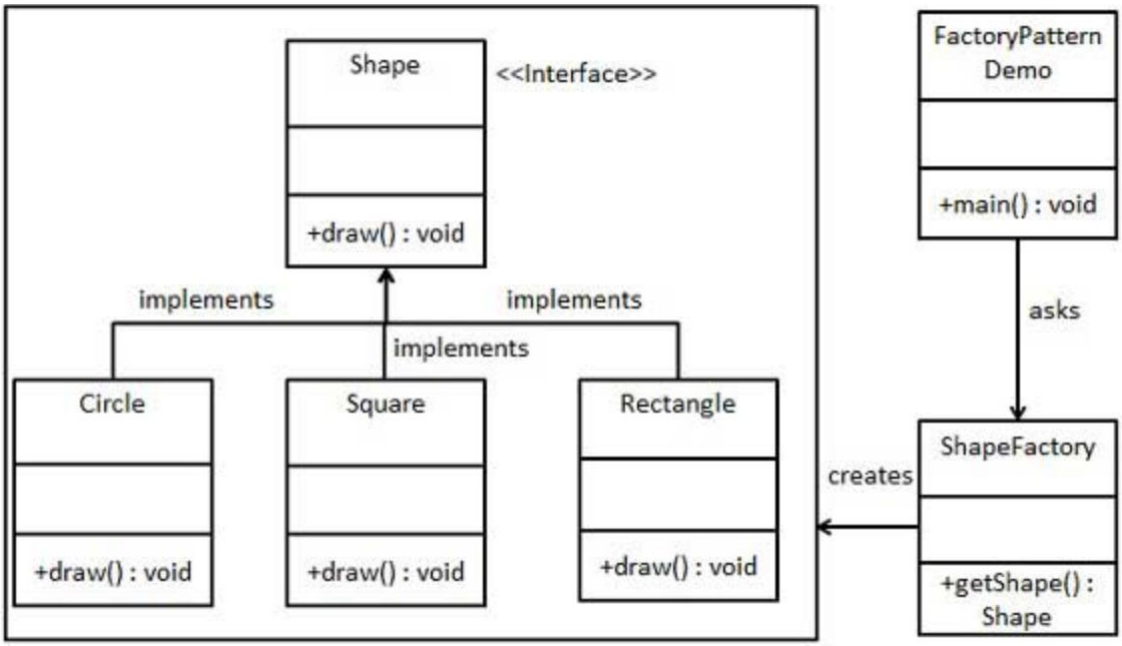


Abbildung 24 UML FactoryPattern

9.3 Singleton

- Creational Pattern
- Nur genau 1 Instanz einer Klasse

Das Singleton-Pattern gehört zu den einfachsten Patterns und stellt sicher, dass es nur ein Objekt dieser Klasse gibt. Dies wird sichergestellt in dem der Constructor auf private gestellt wird und es eine Methode getInstance() gibt

```
//make the constructor private so that this class cannot be
//instantiated
private SingleObject(){}

//Get the only object available
public static SingleObject getInstance(){
    return instance;
}

public void showMessage(){
    System.out.println("Hello World!");
}
```

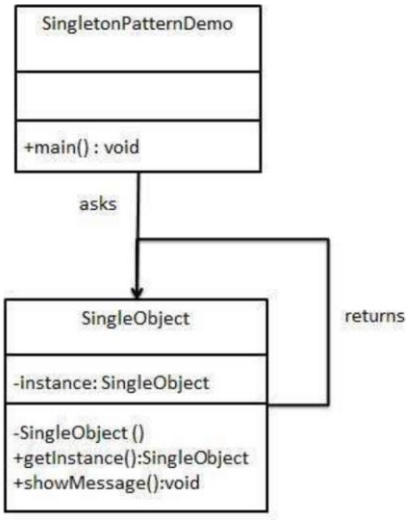


Abbildung 25 UML SingletonPattern

9.4 Strategy Pattern

- Behavior Pattern
- Polymorphismus auf Algorithmen angewendet

Das Strategy Pattern ist eine Möglichkeit das Verhalten oder der Algorithmus einer Klasse während der Runtime zu verändern. Wir erstellen verschiedene Objekte welche verschiedenen Strategien repräsentieren, sowie ein Context-Object welches das Verhalten nach den Strategy Objects verändert. Das Strategy Object kann der Algorithmus eines Context Object ausführen

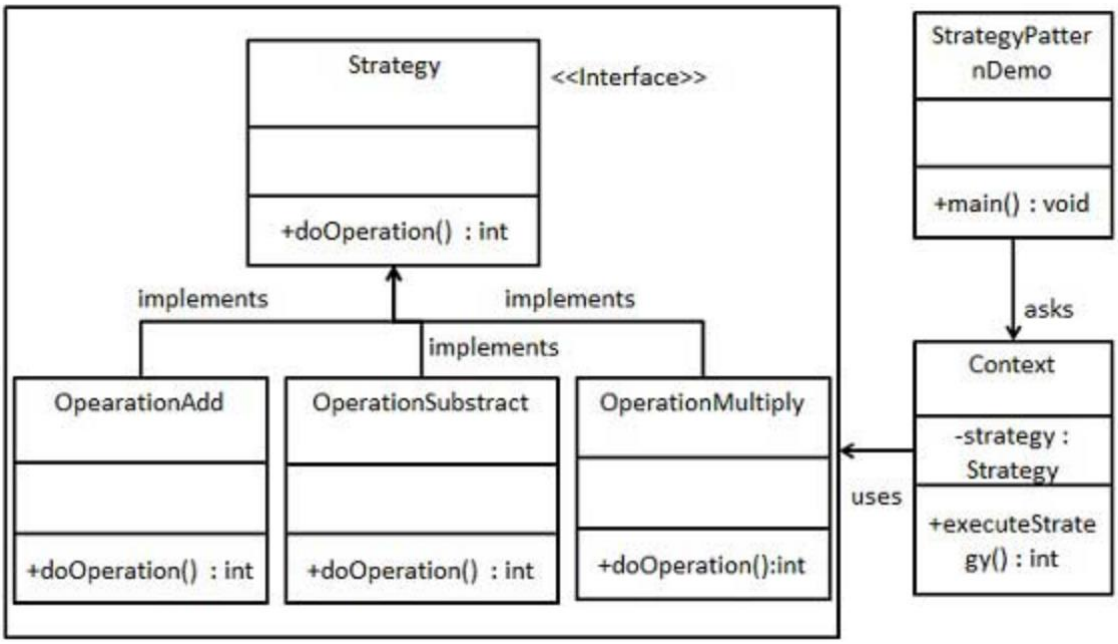


Abbildung 26 StrategyPattern

9.5 Composite

- Structural Pattern
- Eine Gruppe von Objekten gleich wie ein einzelnes behandeln

Das Composite Pattern verfolgt das gleiche Ziel wie das Singleton-Pattern, jedoch für eine Gruppe von Objekten. Es lässt sich eine Hirarchie (Baumstruktur) abbilden, bspw. Organigram von Mitarbeitenden.

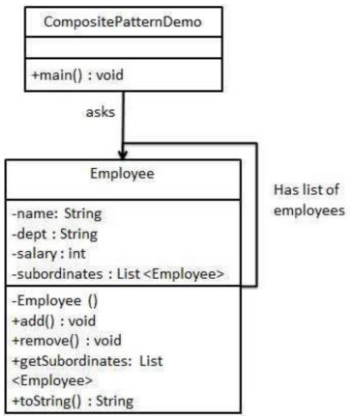


Abbildung 27 CompositePattern

9.6 Facade

- Structural Pattern
- Eine Klasse «spielt» Schnittstelle für ein ganzes Subsystem

Das Facade Pattern versteckt die Komplexität eines Systems und stellt ein Interface zur Verfügung. Dieses Pattern beinhaltet eine einzige Klasse, die vereinfachte Methoden bereitstellt, die vom Client benötigt werden, und Aufrufe an Methoden bestehender Systemklassen delegiert.

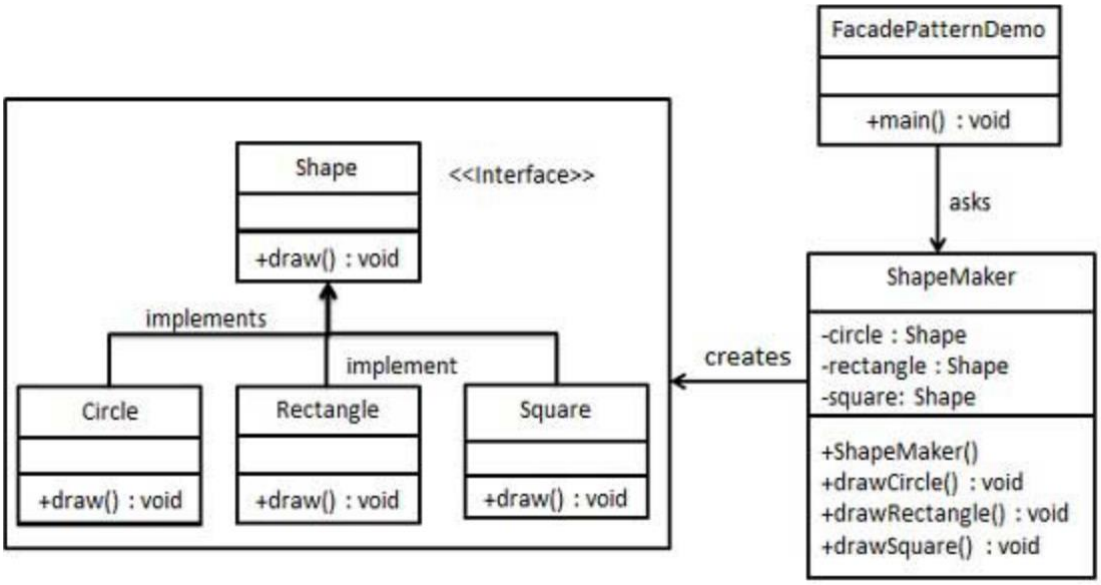


Abbildung 28 FacadePattern

9.7 Observer

- Behavioral Pattern
- Kopplung über Observer Interface, keine direkte Kopplung

Das Observer Pattern wird eingesetzt, wenn es eine one-to-many Beziehung zwischen den Objekten gibt. Bspw. wenn ein Objekt geändert wird, dass alle anderen auch benachrichtigt werden.

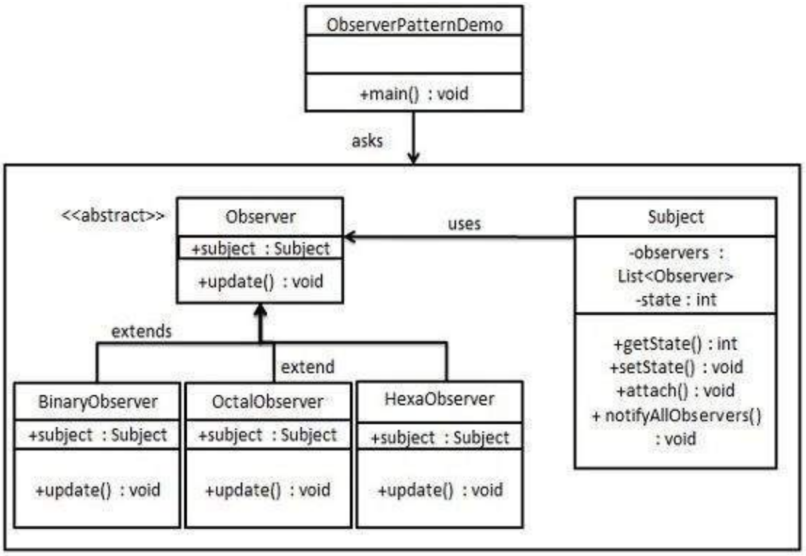


Abbildung 29 ObserverPattern

9.8 Abstract Factory Pattern

- Creational Pattern
- Factory of factories

In diesem Pattern ist ein Interface für die Erstellung der zugehörigen Objekten ohne genaue Spezifizierung der Klasse zuständig.

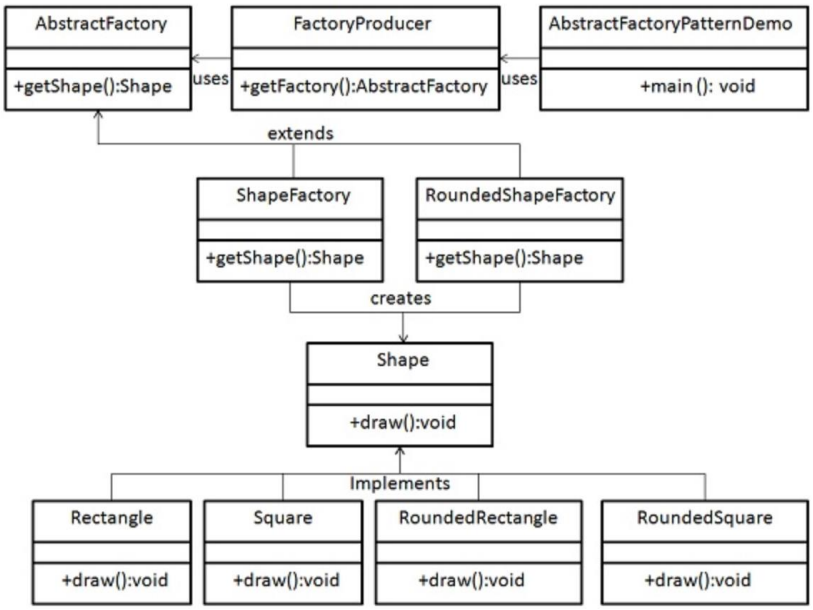


Abbildung 30 UML Abstract Factory Pattern

9.9 Builder Pattern

- Creational Pattern
- Das Builder-Pattern erstellt ein komplexes Objekt aus einfachen Objekten und mit einem schrittweisen Ansatz

Das Builder Pattern erstellt das Objekt Step by Step, der Build ist jedoch unabhängig der anderen Objekte

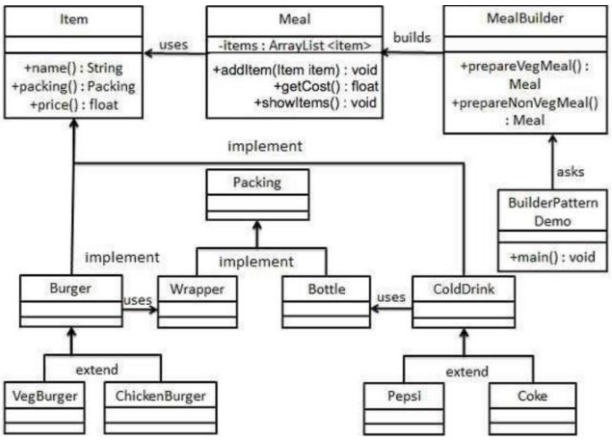


Abbildung 31 UML Builder Pattern

9.10 Prototype Pattern

- Creational Pattern
- Das Prototyp Pattern erstellt eine Kopie eines Objektes

Dieses Muster beinhaltet die Implementierung einer Prototyp-Schnittstelle, die sagt, dass ein Klon des aktuellen Objekts erstellt werden soll. Dieses Muster wird verwendet, wenn die Erstellung eines Objekts direkt kostspielig ist. Beispielsweise soll nach einer kostspieligen Datenbankoperation ein Objekt angelegt werden. Wir können das Objekt zwischenspeichern, seinen Klon bei der nächsten Anforderung zurückgeben und die Datenbank bei Bedarf aktualisieren, wodurch sich die Datenbank-Aufrufe reduzieren.

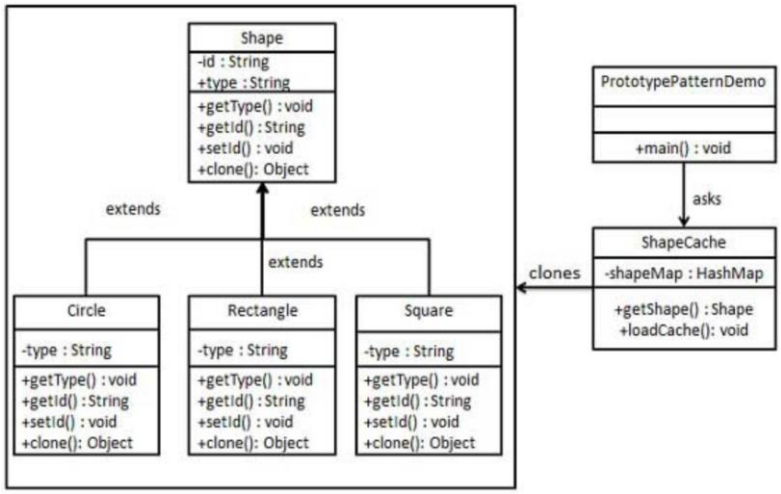


Abbildung 32 UML Prototype Pattern

9.11 Bridge Pattern

- Structual Pattern
- Das Bridge Pattern wird verwendet, wenn es darum geht, eine Abstraktion von ihrer Implementierung zu entkoppeln, so dass die beiden unabhängig voneinander variieren können. Dieses Muster

Dieses Muster beinhaltet ein Interface, das als Brücke fungiert und die Funktionalität konkreter Klassen unabhängig von Interface-Implementierungsklassen macht. Beide Arten von Klassen können strukturell verändert werden, ohne sich gegenseitig zu beeinflussen.

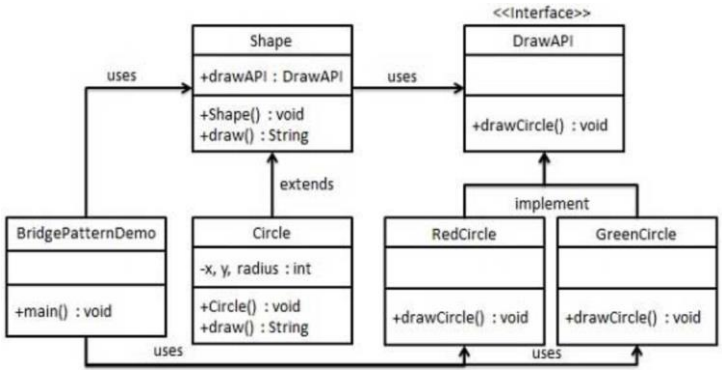


Abbildung 33 UML Bridge Pattern

9.12 Decorator Pattern

- Structual Pattern
- Das Dekorator-Pattern ermöglicht es einem Benutzer, einem bestehenden Objekt neue Funktionen hinzuzufügen, ohne dessen Struktur zu verändern.

Dieses Muster erzeugt eine Dekoratorklasse, die die ursprüngliche Klasse umhüllt und zusätzliche Funktionalität bietet, ohne die Signatur der Klassenmethoden zu beeinträchtigen.

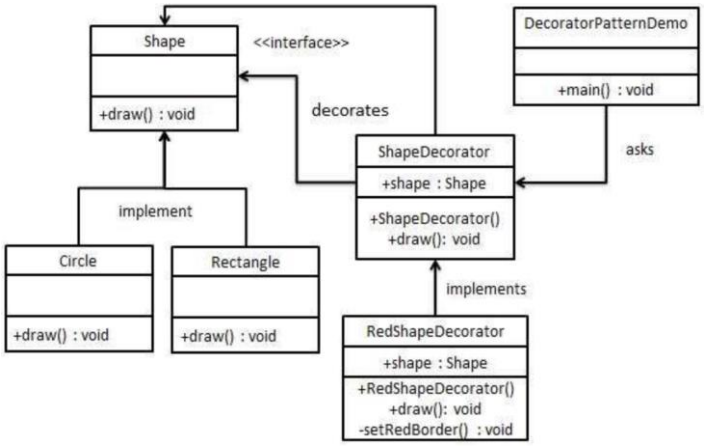


Abbildung 34 UML Decorator Pattern

9.13 Flyweight Pattern

- Structual Pattern
- Das Flyweight-Pattern wird hauptsächlich verwendet, um die Anzahl der erzeugten Objekte zu reduzieren und den Speicherbedarf zu verringern und die Leistung zu steigern.

Flyweight Pattern versucht, bereits existierende ähnliche Objekte durch Speichern wiederzuverwenden und erstellt ein neues Objekt, wenn kein passendes Objekt gefunden wird.

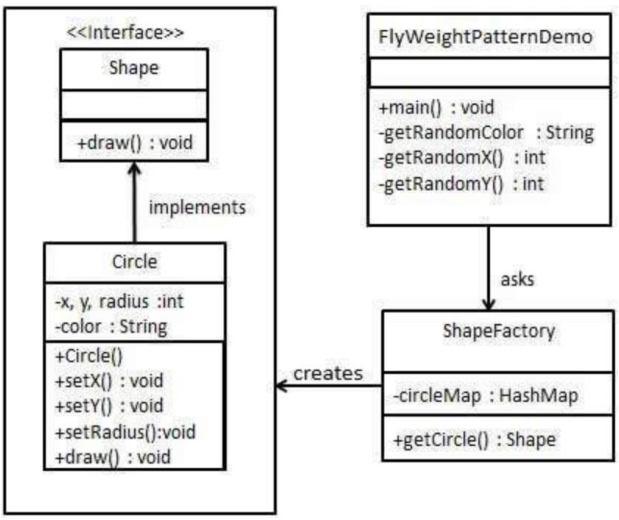


Abbildung 35 UML Flyweight Pattern

9.14 Proxy Pattern

- Structual Pattern
- Eine Klasse repräsentiert die Funktionalität einer anderen Klasse

Im Proxy-Pattern erstellen wir ein Objekt mit Originalobjekt, um seine Funktionalität mit der Aussenwelt zu verbinden.

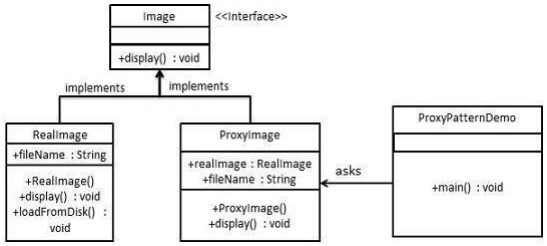


Abbildung 36 UML Proxy Pattern

9.15 Chain of Responsibility Pattern

- Behavioral Pattern
- Dieses Pattern entkapselt Sender und Empfänger

In diesem Pattern enthält normalerweise jeder Empfänger eine Referenz auf einen anderen Empfänger. Wenn ein Objekt den Auftrag nicht bearbeiten kann, gibt es ihn an den nächsten Empfänger weiter und so weiter.

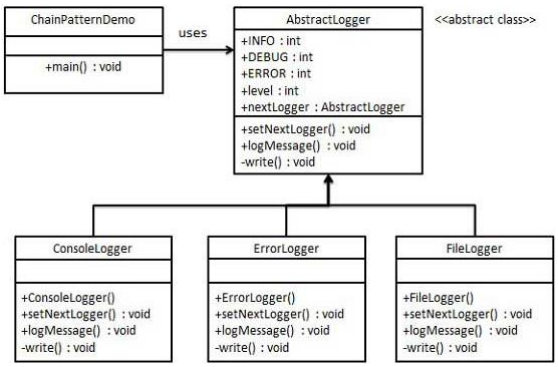


Abbildung 37 UML Chain of Responsibility Pattern

9.16 Command Pattern

- Behavioral Pattern
- Eine Anforderung wird unter einem Objekt als Befehl umgebrochen und an das Aufrufobjekt übergeben.

Invoker object sucht nach dem entsprechenden Objekt, das diesen Befehl verarbeiten kann und übergibt den Befehl an das entsprechende Objekt, das den Befehl ausführt.

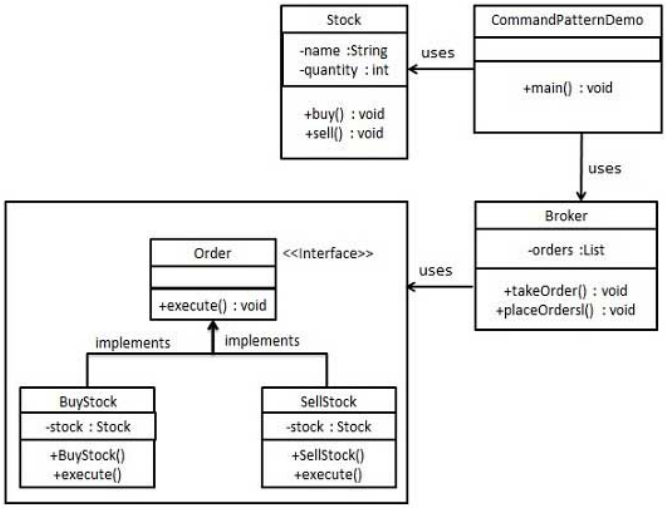


Abbildung 38 UML Command Pattern

9.17 Interpreter Pattern

- Behavioral Pattern
- Bietet eine Möglichkeit Sprachen auf Grammatik oder Ausdruck zu bewerten

Dieses Muster beinhaltet die Implementierung eines Ausdrucksinterfaces, das die Interpretation eines bestimmten Kontextes anweist. Dieses Muster wird in SQL-Parsing, Symbolverarbeitungs-Engine usw. verwendet.

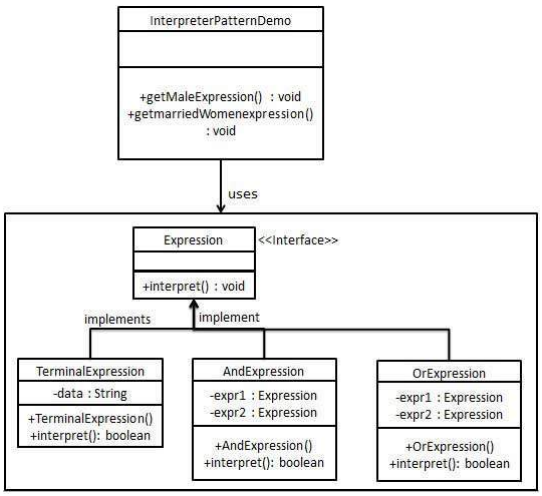


Abbildung 39 UML Interpreter Pattern

9.18 Iterator Pattern

- Behavioral Pattern
- Dieses Muster wird verwendet, um eine Möglichkeit zu erhalten, sequentiell auf die Elemente eines Sammelobjekts zuzugreifen, ohne dass man seine zugrunde liegende Darstellung kennen muss.

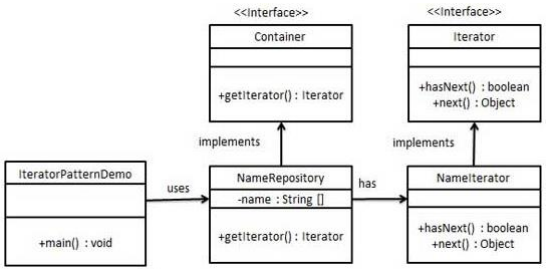


Abbildung 40 UML Iterator Pattern

9.19 Mediator Pattern

- Behavioral Pattern
- Dieses Pattern wird verwendet, um die Komplexität der Kommunikation zwischen einzelnen Klassen zu minimieren

Dieses Muster stellt eine Mediatorklasse bereit, die normalerweise die gesamte Kommunikation zwischen verschiedenen Klassen übernimmt und eine einfache Wartung des Codes durch lose Kopplung unterstützt.

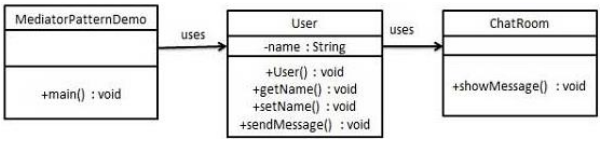


Abbildung 41 UML Mediator Pattern

9.20 Memento Pattern

- Behavioral Pattern
- Das Memento-Pattern wird verwendet, um den Zustand eines Objekts in einen früheren Zustand zurückzusetzen.

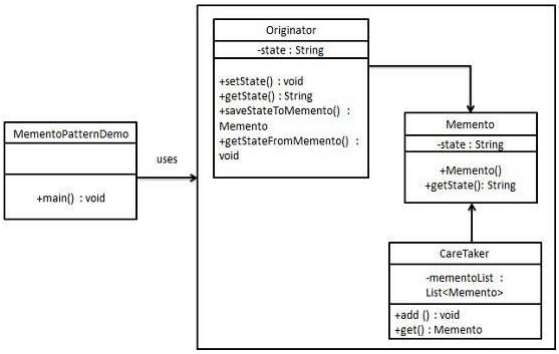


Abbildung 42 UML Memento-Pattern

9.21 State Pattern

- Behavioral Pattern
- Im State-Pattern ändert sich das Verhalten einer Klasse basierend auf ihrem Zustand.

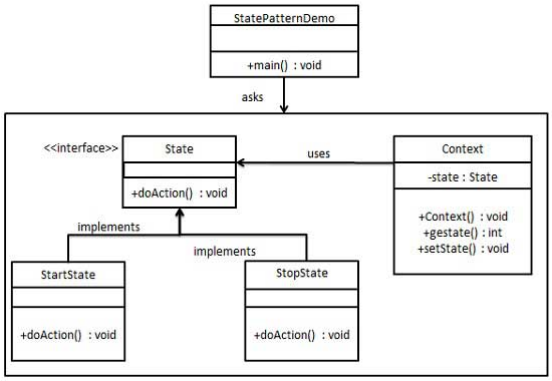


Abbildung 43 UML State Pattern

9.22 Template Pattern

- Behavioral Pattern
- Im Template-Pattern stellt eine abstrakte Klasse definierte Wege / Vorlagen zur Ausführung ihrer Methoden zur Verfügung.

Seine Unterklassen können die Methodenimplementierung nach Bedarf übersteuern, aber der Aufruf muss auf die gleiche Weise erfolgen wie durch eine abstrakte Klasse definiert.

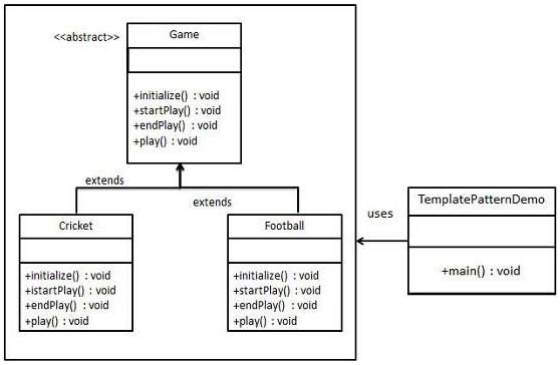


Abbildung 44 UML Template Pattern

9.23 Visitor Pattern

- Behavioral Pattern
- Im Visitor-Pattern verwenden wir eine Besucherklasse, die den Ausführungsalgorithmus einer Elementklasse ändert.

Auf diese Weise kann der Ausführungsalgorithmus des Elements je nach Besucher variieren.

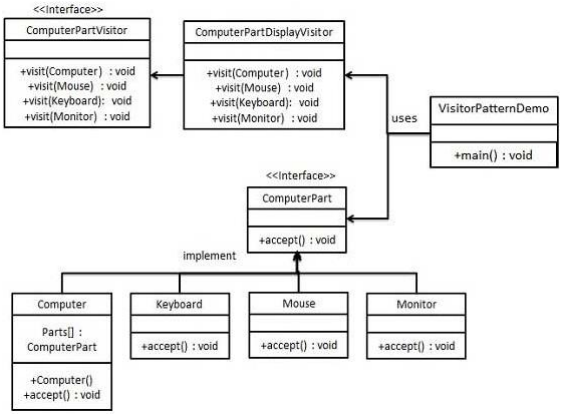


Abbildung 45 UML Visitor Pattern

10 Glossar

Fachbegriff	Beschreibung
CASE	Computer-aided Software Engineering
Polymorph	Vielseitig, verschiedengestaltig