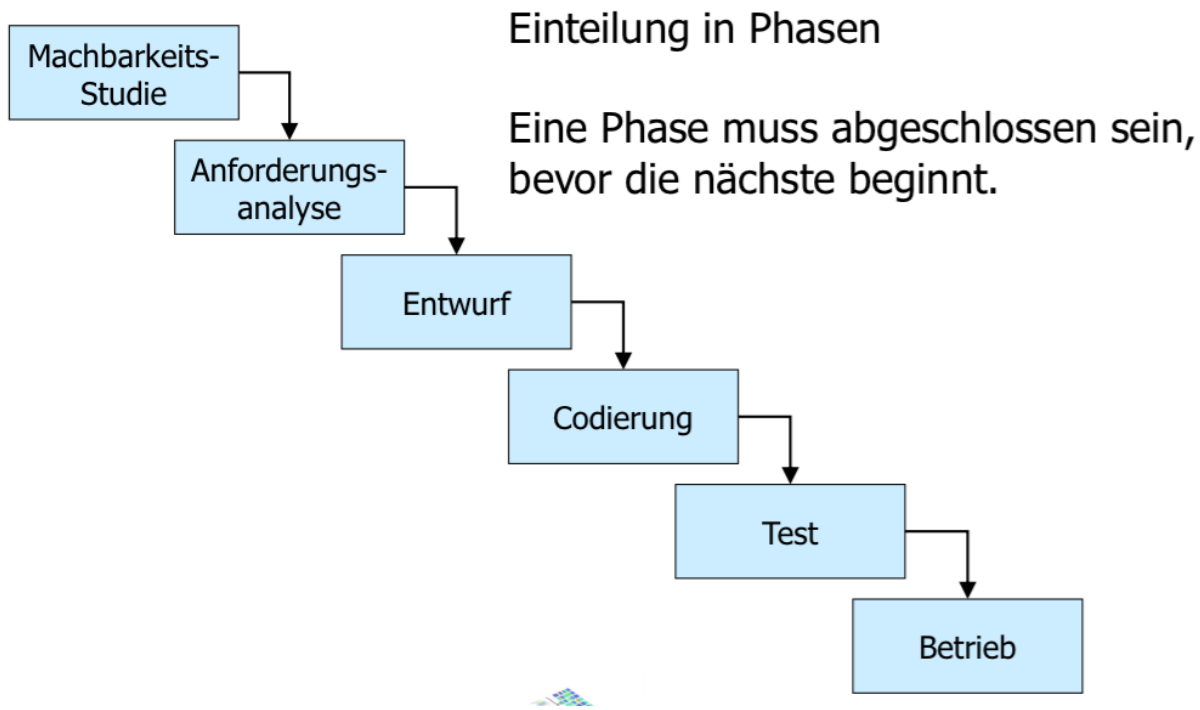
Softwaretechnik Zusammenfassung

Prozess-Modelle:

Bestimmung der Rahmenbedingungen für Softwareentwicklung:

- Phasen, die den Ablauf beschreiben
- Teilprodukte und ihre Anforderungen, die erreicht werden müssen

Wasserfallmodell:



Machbarkeitsstudie:

- Lösungsvorschläge, erste Kostenkalkulationen

Anforderungsanalyse:

- Lastenheft (spezifiziert, was Software leisten soll)

Entwurf:

- Architektur (Objekte/Klassen)
- Meist Top-Down (Zerlegung in immer kleinere Komponenten)

Codierung:

- Implementierung der Software
- Implementierungsbericht (Abweichung vom Entwurf/Zeitplanung)

Test:

- Modultest, Integrationstest, Systemtest (einzeln, mehrere, alle)

Varianten:

- Mit Rückkopplung: Jede Phase kann die vorherige beeinflussen

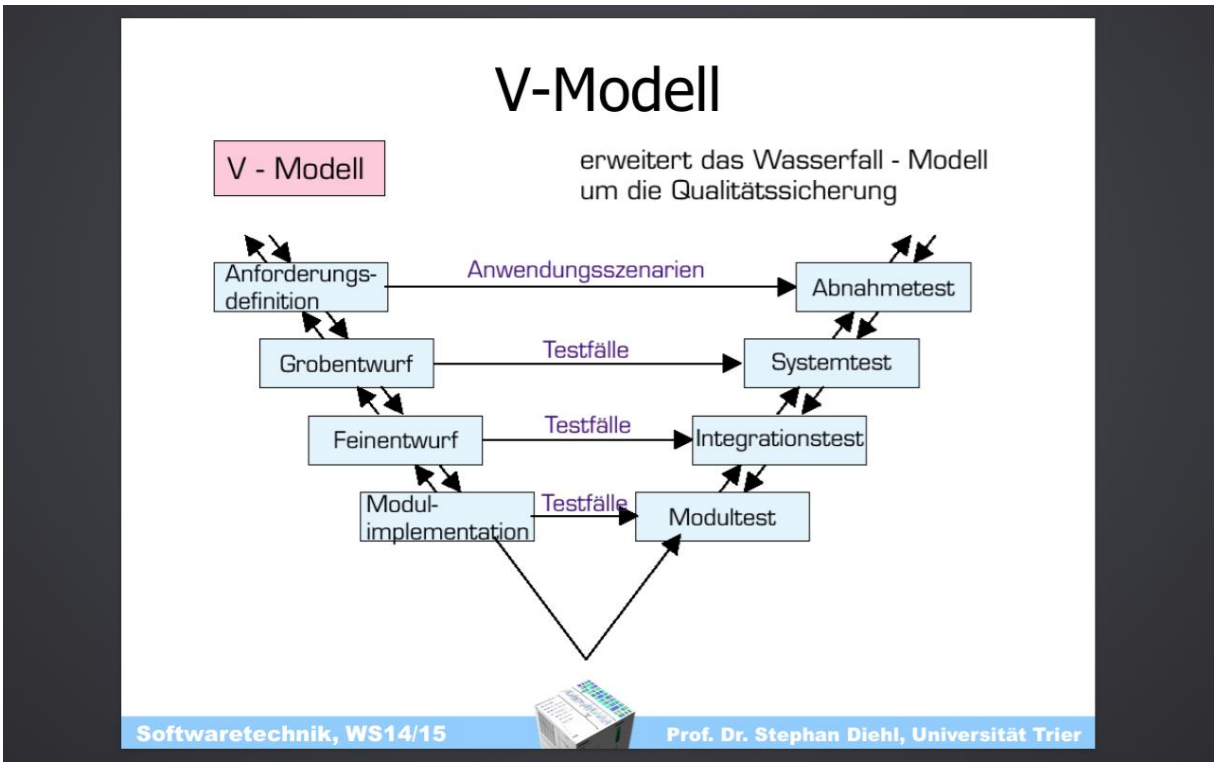
Nachteile:

- Kosten schwer am Anfang zu schätzen
- Pflichtenheft gibt nur unzureichend Information an Kunden
- Anforderungen zu früh gesetzt

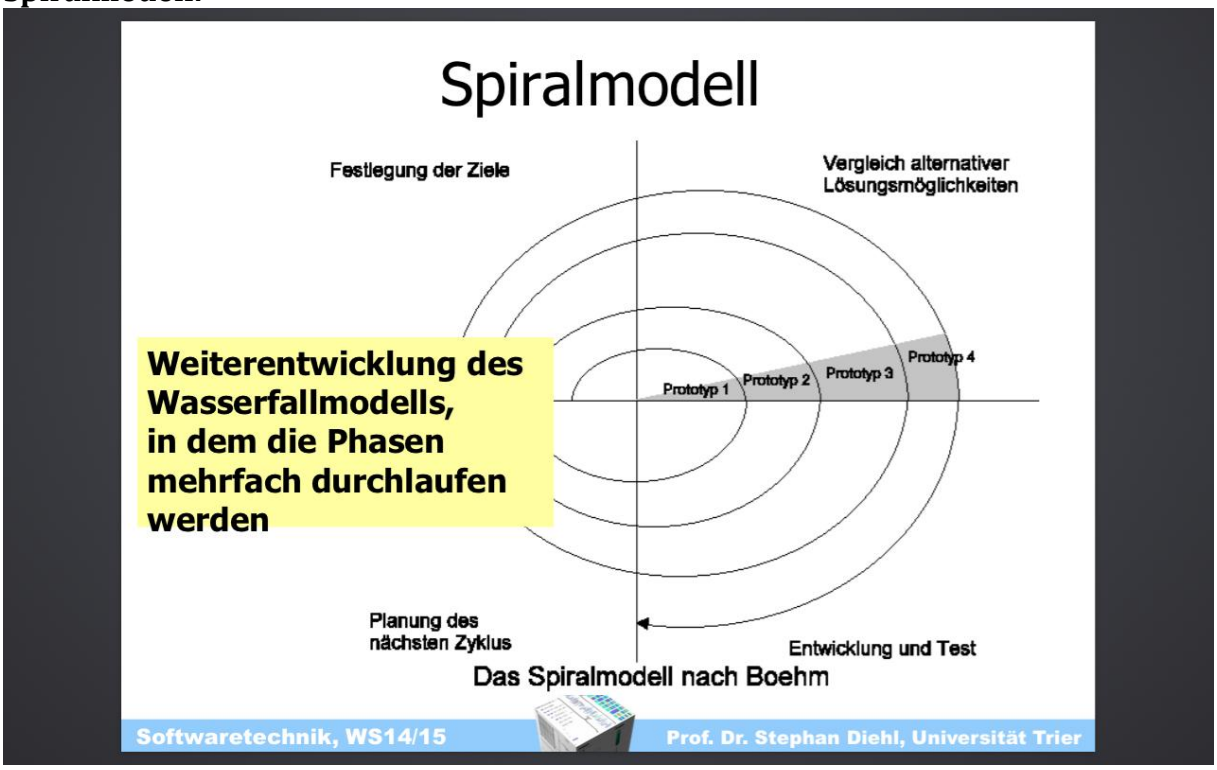
Evolutionäre SW-Entwicklung:

Erfassung der Kernanforderungen
Implementierung und Auslieferung des Kerns
Erfassen weiterer spezifischer Anforderungen

V-Modell:



Spiralmodell:



OO-Modell:
Wiederverwertung von Objekten
OOA: Analysemuster
OOD: Entwicklungsmuster
OOP: Klassenbibliotheken oder eigene Klassen

UI-Entwicklung:

Paper Prototypes:
Skizzieren des UI auf Papier
Storyboard (Zusammenhänge der UI)

Anforderungsanalyse:

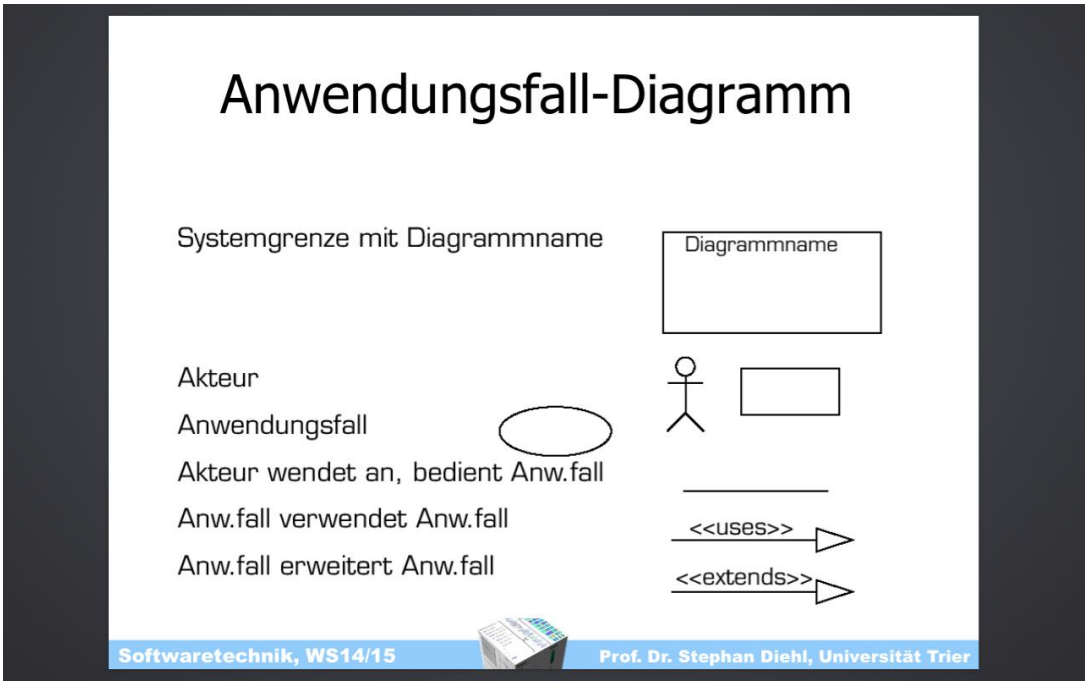
Anforderungen:
Funktional, Nicht-Funktional
Muss, soll, kann Anforderungen

Pflichtenheft:
1. Zielbestimmung (Muss-/Soll-/Kann-Kriterien)
2. Produkt-Einsatz (Anwendungsbereich, Zielpublikum, Betriebsbedingungen)
3. Produkt-Umgebung (Software, Hardware)
4. Produkt-Funktionen (Auflistung der Funktionen)
5. Produkt-Daten
6. Produkt-Leistungen (Laufzeitanforderungen)
7. Benutzungsoberfläche
8. Qualitäts-Zielbestimmung
9. Testszenarien
10. Entwicklungs-Umgebung
11. Ergänzungen

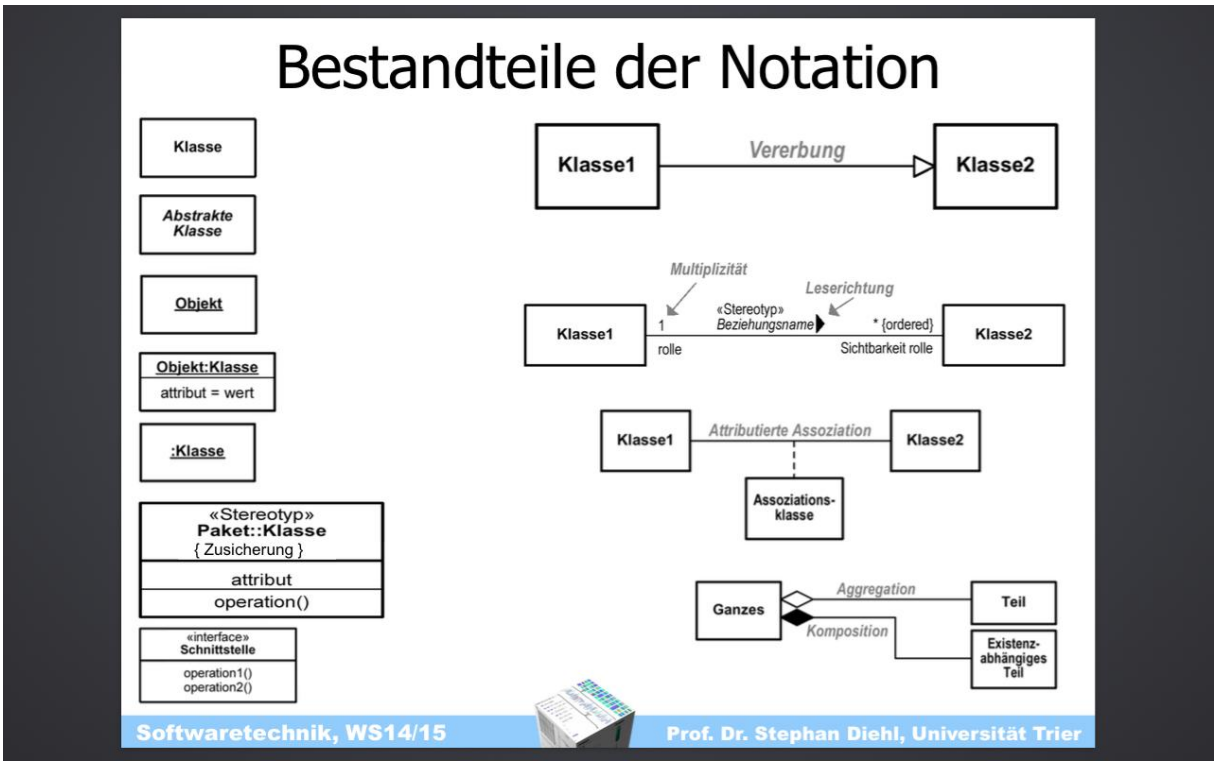
Diagramme:

UML:
Diagrammtypen: Use-Case, Klassendiagramm, Implementierungsdiagramm,
Komponentendiagramm, Einsatzdiagramm, Verhaltensdiagramm

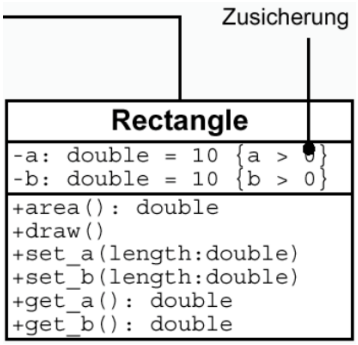
Anwendungsfall (Use Case)
Akteur: Etwas das sich verhalten kann, Benutzer, System, Organisation
Menschen, Sensoren, andere Software
Anwendungsfall: Beschreibt eine Interaktion mit dem System
Szenario: Spezifische Folge von Interaktionen



Klassendiagramm:



Vererbung wird mit Pfeilen dargestellt



Konformität = ein Objekt einer Unterklasse ist in jeder Beziehung als Objekt der Oberklasse einsetzbar. Klassenhierarchien sollten so konstruiert werden, dass Konformität stets sichergestellt ist.

Assoziation = Eine Assoziation beschreibt eine Relation zwischen Klassen, d.h. die gemeinsame Semantik und Struktur einer Menge von Objektbeziehungen. Es werden gerichtete Assoziationen und bidirektionale Assoziationen unterschieden.

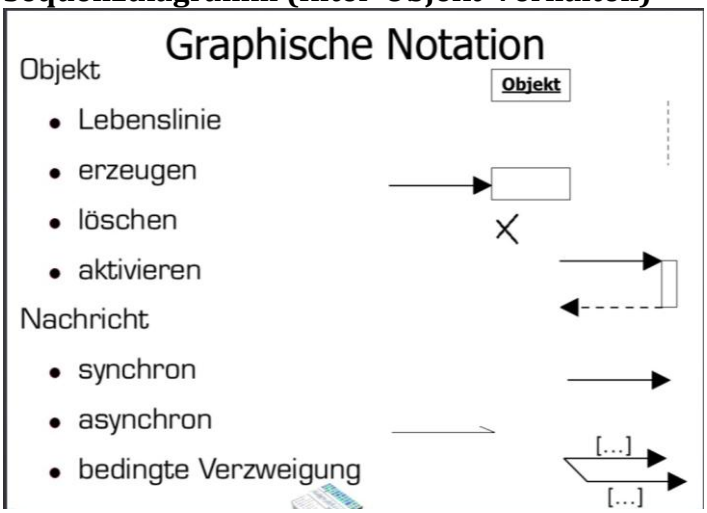
Eine Aggregation ist eine Assoziation, bei der die beteiligten Klassen keine gleichwertige Beziehung führen, sondern eine Ganze-Teile-Hierarchie darstellen.
Eine Komposition ist eine strenge Form der Aggregation, bei der die Teile vom Ganzen existenzabhängig sind.

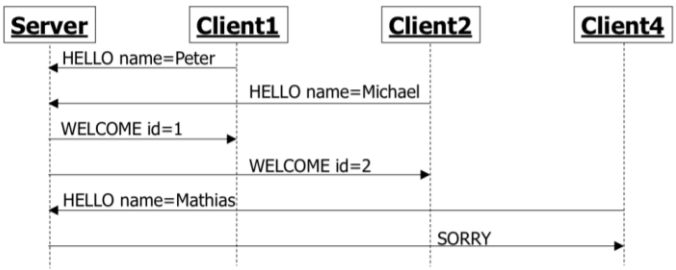
CRC-Karten (class responsibility collaborators)

Vorderseite:
- Verhalten, Operationen, dynamische Aspekte
- Klassenname, Oberklassen, Unterklassen, Verantwortlichkeiten, Helfer
- Verantwortlichkeiten (Was macht die Klasse, was verwaltet sie?)
- Helfer (mit den Verantwortlichkeiten verbundene Klassen)

Rückseite:
- Struktur, statische Aspekte

Sequenzdiagramm (Inter-Objekt-Verhalten)





Fallunterscheidungen werden durch einen Kasten signalisiert, es gibt folgende Fallunterscheidungen (Art und Bedingung stehen oben links im Kasten):
„opt“: Der Inhalt wird nur dann ausgeführt, wenn die Bedingung erfüllt ist.
„alt“: If-Then-Else-Konstrukt, durch horizontale gestrichelte Linien aufgeteilt in Bereiche mit eigenen Bedingungen oder else.
„loop“: Wiederhole solange Bedingung gilt

Zustandsdiagramm (Inter-Objekt-Verhalten):
Zustände, Übergänge, Ereignisse

Übergänge können Bedingungslos (Transition), oder bedingt (bewachte Transition) sein. Bedingungen werden in [] geschrieben.
Aktivitäten innerhalb eines Zustands:
„entry/aktion“ - Wird beim Eingang in diesen Zustand ausgeführt
„exit/aktion“ - Wird beim Austritt aus diesem Zustand ausgeführt
„do/aktion“ - Wird ausgeführt, Parameter möglich
„event/aktion“ - Wird ausgeführt, wenn event getriggert wird.

Auflösen von Mehrfachvererbung in Java:

```
class A {
 private int a;
 int getA() { return a; }
}

class B {
 private int b;
 int getB() { return b; }
}

class C extends A,B {
 private int c;
 int getC() { return c; }
}
```

Nicht zulässig in Java

```
class A {
 private int a;
 int getA() { return a; }
}

interface B {  int getB(); }

class B1 implements B {
 private int b;
 int getB() { return b; }
}

class C extends A implements B {
 private int c;
 private B myB = new B1();
 int getC() { return c; }
 int getB() { return myB.getB(); }
}
```

Entwurfsmuster:

Werden aufgeteilt in Creational Patterns (Factory), Structural Patterns (Composition von Klassen), Behavorial Patterns

Singleton:

Von der Klasse soll es nur ein Objekt geben. Die Klasse besitzt ein privates Attribut *instance*, welches mit public Methode *getInstance()* abgerufen werden kann.

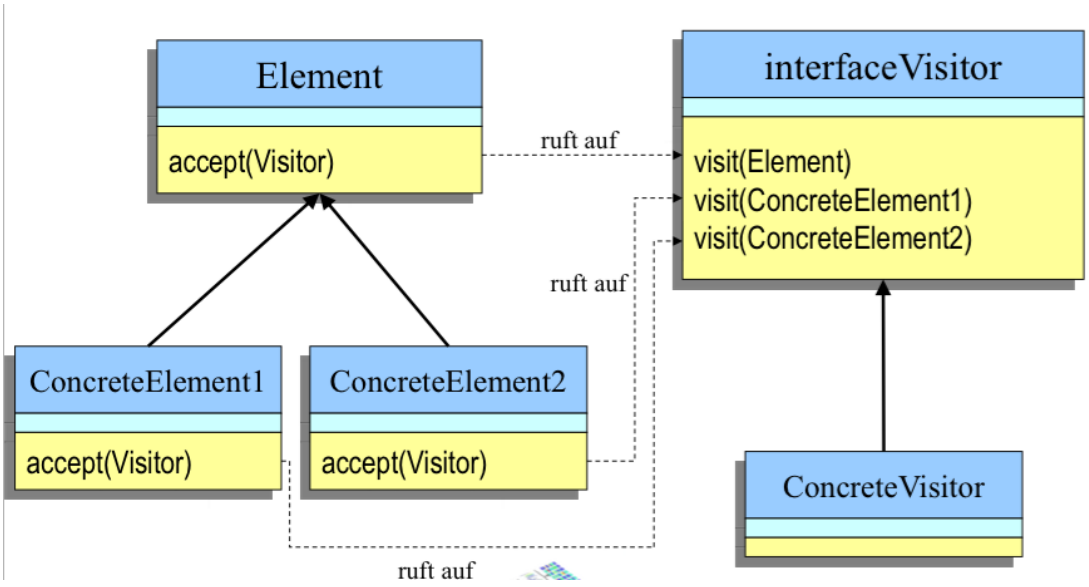
Factory:

Es existiert eine statische Methode in der Objektklasse oder in einer Factory-Klasse. Die Methode generiert ein neues Objekt und liefert es zurück.

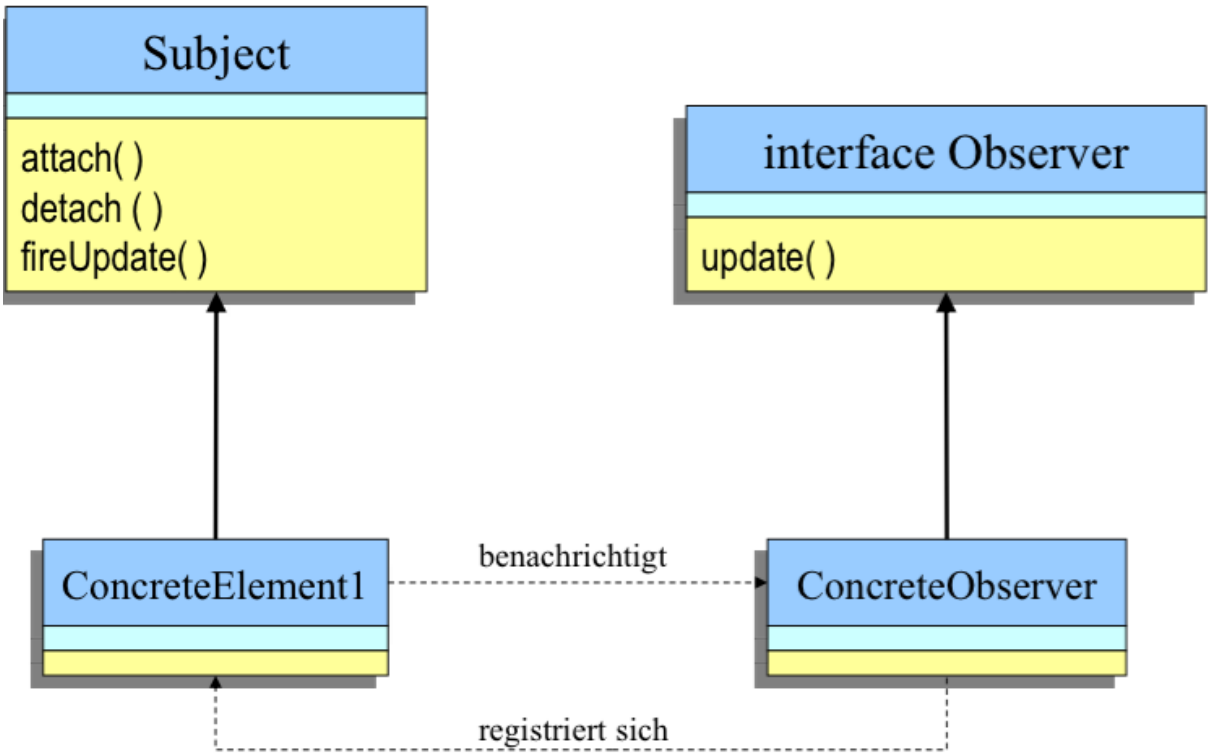
Composite:

Verschachtelung von Elementen (z.B. Menü, Untermenü)

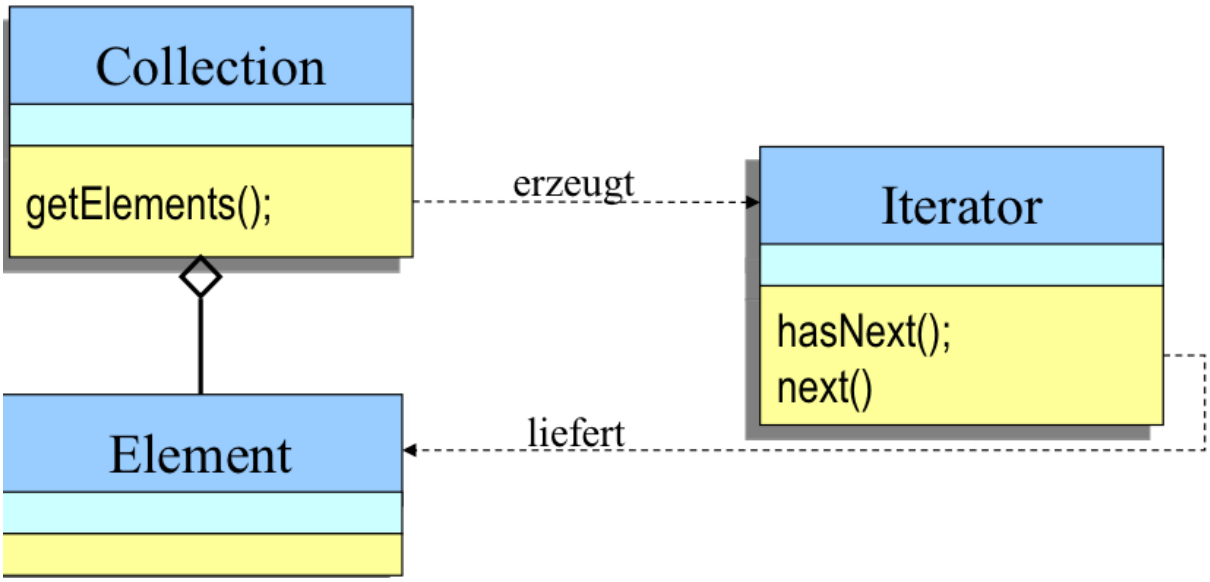
Visitor:



Observer:



Iterator:



Strategy:
Eine Strategy-Klasse gibt als Interface eine Methode o.Ä. vor.

MVC:
Model: Anwendungsobjekt
View: Darstellung des Modells (auch mehrfach)
Controller: Benutzereingaben/Reaktionen
Gut für observer: Werden Daten im Modell verändert, wird der View aktualisiert.
Gut für composite: Bsp. Menüs
Strategy für einheitliche Kommunikation für Views

Qualitätssicherung

Quantitative Qualitätssicherung:
Metriken, die die Qualitätssteigerung messbar machen.
Analytische Qualitätssicherung:
Syntaxtesting, Software-Tests
Konstruktive Qualitätssicherung:
Frühzeitig bei der Entwicklung Fehler vermeiden.

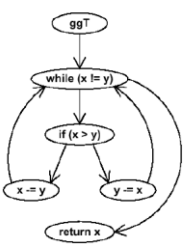
Funktionale Testverfahren:
Blackbox-Testing (Testen anhand von erwarteten Ein-/Ausgaben mit Grenzwerten)
Strukturtests:
Testen des Aufbaus eines Programms
Pfadtesten:
Anweisungsüberdeckung (Statement coverage):
Jede Anweisung mindestens einmal ausgeführt
Zweigüberdeckung (Branch coverage):
Jede Kante im Kontrollflussgraphen mindestens einmal durchlaufen
Pfadüberdeckung (Path coverage):
Jeder Pfad im Kontrollflussgraphen mindestens einmal durchlaufen
Angabe von Eingaben, die die Bedingungen erfüllen.
Regressionstesten:
Nach jeder Änderungen werden alle Tests durchlaufen, um Fehler zu vermeiden

Softwaremetriken

Prozess-Metriken:
Zeitverbrauch, Ressourcenverbrauch
Produkt-Metriken:
Umfangsmetriken, Stilmetriken, Vererbungshierarchien, Methodenebene, Klassenebene
Umfangsmetriken:
Lines of Code etc.
Halstead-Metriken:
Zähle Operatoren (int, (), {}, assert ...)
Zähle Operanden (Variablen, Konstanten)
n1=#Unterschiedlicher Operatoren
n2=#Unterschiedliche Operanden
N1=#verwendete Operatoren
N2=#verwendete Operanden
D (difficulty) = $\frac{n1}{2} * \frac{N2}{n2}$ V (Volumen) = $N * \log_2(n1 + n2)$
McCabe-Metriken:

McCabe-Metrik

Ansatz: *Strukturelle Komplexität* berücksichtigen



Quelle: *Kontrollflußgraph* G
Ziel: *Zyklomatische Komplexität* $V(G)$

$V(G) = e - n + 2p$
e: Anzahl der Kanten (hier: 7)
n: Anzahl der Knoten (6)
p: Anzahl der Komponenten (1)
Hier: $V(G) = 3$

Ist $p = 1$, so ist $V(G) = \pi + 1$
mit π: Anzahl der *Bedingungen* (2)