Was sind Requirements?

- Bedingung oder Eigenschaft, die ein System benötigt,
  - um ein Problem zu lösen

    oder
  - um ein Ziel zu erreichen

    oder
  - um einem Vertrag, Standard oder Ähnlichem zu genügen

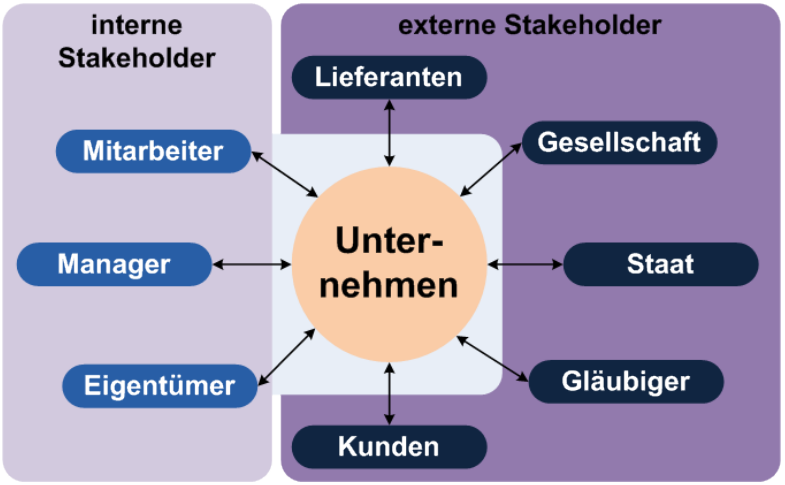
- [Pohl. Requirements Engineering]

Was sind Requirements?

- Werden an ein Programm gestellt
- Spezifizieren:
  - Eigenschaften
  - Funktionalität
  - Einsatzsszenario
  - Qualität
- Werden von Stakeholdern bestimmt

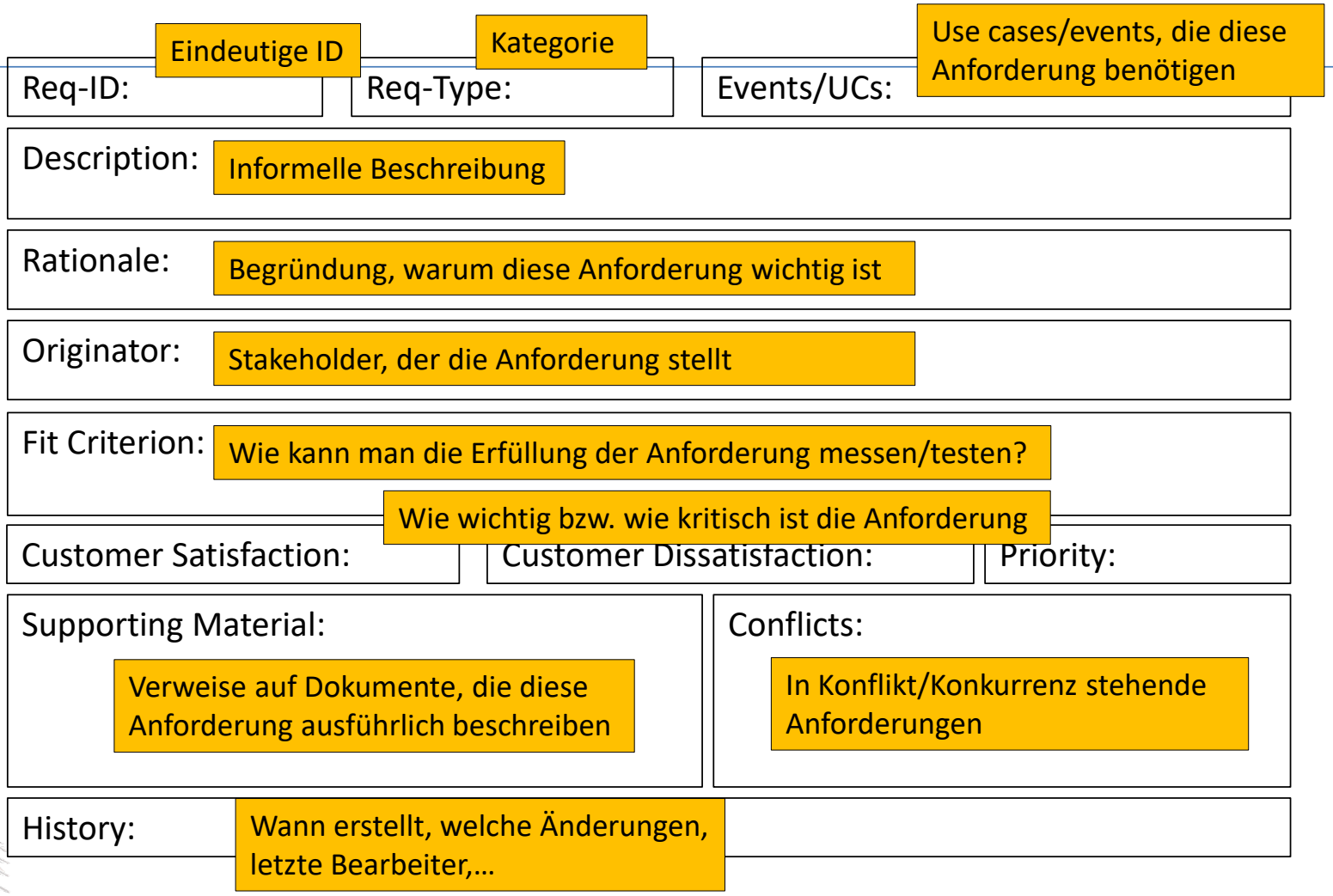
Probleme (3) -- Widersprüche

- Verschiedene Stakeholder können widersprüchliche Anforderungen haben
- Neue Stakeholder mischen sich ein



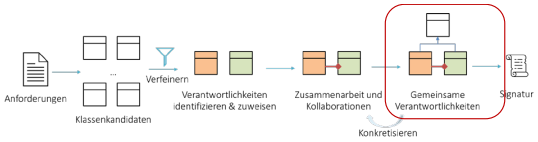
© Grochim
[http://de.wikipedia.org/wiki/Stakeholder]

Volere: Snow Card



Wiederholung: Entwicklungsprozesse

- Herangehensweisen und Prozesse für die Entwicklung von Softwaresystemen
- **Sequentiell:** Strickte Abfolge der Phasen im Lebenszyklus mit Dokumentationen am Ende einer Phase (gut für Projekte mit hohen Sicherheitsanforderungen, klaren und nicht änderbaren Anforderungen)
- **Iterative:** Kurze Entwicklungszyklen, zur Realisierung bestimmter Features in ein auslieferbares Produkt (gut für sich ändernde und unklare Anforderungen, schnelles Feedback, kleinere, agile Teams)
- **Scrum**: Managementframework, welches in festen Sprints Ergebnisse erzielt (gut für neue Produktentwicklung, passt zu agiler Entwicklung; kombinierbar mit Praktiken des XP)
- **Lean**: ähnlich zu Scrum, aber flexibler und darauf ausgelegt, die Arbeit an angefangenen Teilen zu minimieren (gut für Systeme in der Wartung; schnelle Bearbeitung kleinerer Funktionseinheiten)



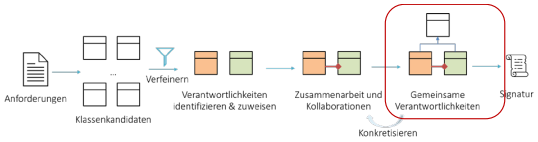
Entwerfen von Guten Hierarchien

Modelliere eine "kind-of" Hierarchie:

- Unterklassen sollten *alle geerbten Verantwortlichkeiten unterstützen*, und eher noch mehr

Schiebe gemeinsame Verantwortlichkeiten so hoch wie möglich:

- Klassen, die *gemeinsame Verantwortlichkeiten teilen* sollten *von einer gemeinsamen abstrakten Superklasse erben*; führe fehlende Superklassen ein



Entwerfen von Guten Hierarchien …

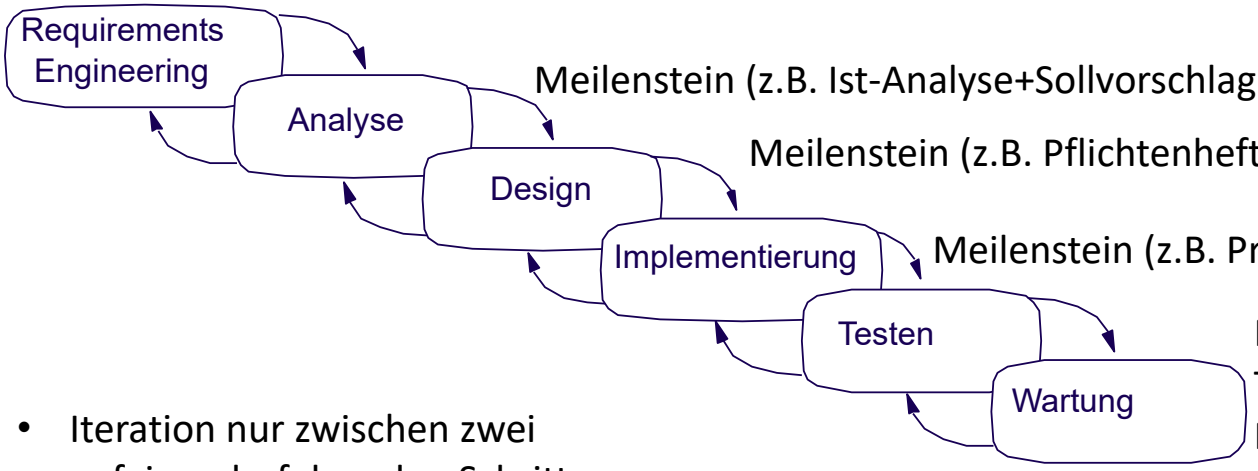
Stelle sicher, dass abstrakte Klassen nicht von konkreten Klassen erben:

- Eliminiere dies durch die Einführung weiterer *gemeinsamer abstrakter Superklassen*: abstrakte Klassen sollten Verantwortlichkeiten in einem implementierungsunabhängigen Weg unterstützen

Eliminiere Klassen, die keine neue Funktionalität hinzufügen:

- Klassen sollten entweder neue Verantwortlichkeiten oder eine bestimmte Implementierung von vererbten Verantwortlichkeiten hinzufügen

Wasserfallmodell



- Iteration nur zwischen zwei aufeinanderfolgenden Schritten
- Abgeschlossener Schritt als sicherer Rückgriff

Wasserfallmodell: Diskussion

- Vorteile:
  - *Dokumentation* nach jeder Phase verfügbar
  - Klare *Trennung* der Phasen und Verantwortlichkeiten
  - Analog zu Ingenieurprojekten (Brückenbau etc.)

- Nachteile:
  - *Starres* Vorgehen (veraltete Dokumentation)
  - Reaktionen auf *geänderte* Anforderungen schwierig
  - Anforderungen, Design, etc. *früh fixiert*, Änderungen nicht vorgesehen (aber Änderungen sind natürlich!)
  - *Späte* Qualitätsprüfung (Baue ich überhaupt das *richtige Produkt?* Erster *Prototyp sehr spät* verfügbar!)
  - Meist unrealistisch in der Praxis

Agile Softwareentwicklung

- Relativ neuer Ansatz (ca. 1999)
- Im Spannungsfeld zwischen:
  - Qualität, Kosten und Zeit
  - Ungenauen Kundenwünschen und instabilen Anforderungen
  - Langen Entwicklungszeiten und überzogenen Terminen
  - Unzureichender Qualität

- Gegenbewegung zu schwergewichtigen, bürokratischen iterativen Prozessen, die oft zu viel Dokumentation erfordern

Manifesto für Agile Software Entwicklung

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Kent Beck
Ward Cunningham
Andrew Hunt
Robert C. Martin
Dave Thomas

Mike Beedle
Martin Fowler
Ron Jeffries
Steve Mellor

Arie van Bennekum
James Grenning
Jon Kern
Ken Schwaber

Alistair Cockburn
Jim Highsmith
Brian Marick
Jeff Sutherland

[Agilemanifesto.org]

Was ist Agile Softwareentwicklung?

- Menge von Softwareentwicklungsmethoden
  - Basierend auf iterativer und inkrementeller Entwicklung
- Meist kleine Gruppen (6-8)
- Kunde ist in Projekt integriert

Schwergewichtige Prozesse	Agile Prozesse
Dokumentenzentriert	Codezentriert
Up-Front Design	Minimale Analyse zu Beginn
Reglementiert	Adaptiv, Prozess wird angepasst
Abarbeitung eines Plans	Ständige Anpassung der Ziele
Lange Releasezyklen	Häufiges Deployment

Die 12 agilen Prinzipien

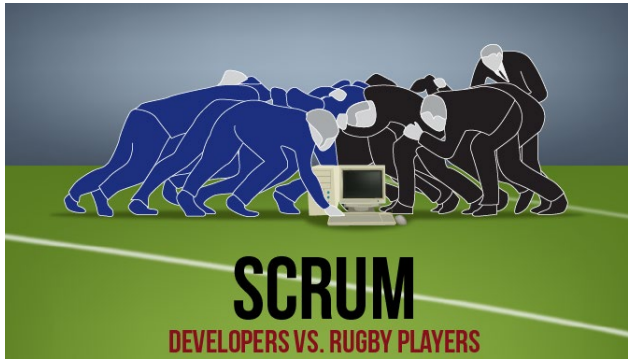
1. Kundinnenzufriedenheit durch frühzeitige und kontinuierliche Auslieferung wertvoller Software!
2. Änderungen begrüßen, selbst wenn sie spät in der Entwicklung kommen. Agile Prozesse nutzen Änderungen zugunsten des Wettbewerbsvorteils des Kunden.
3. Liefere funktionierende Software häufig (zw. wenigen Wochen und Monaten) aus.
4. Tägliche Zusammenarbeit von Kundinnen und Entwicklerinnen.
5. Baue Projekte mit motivierten Mitarbeiterinnen und gib ihnen die Umgebung und Unterstützung, die sie benötigen und vertraue ihnen, dass sie erfolgreich ihre Arbeit beenden.
6. Konversation von Angesicht zu Angesicht ist die effektivste und effizienteste Methode der Kommunikation!

Die 12 agilen Prinzipien

7. Das primäre Maß für Fortschritt ist funktionierende Software.
8. Agile Prozesse bieten Kontinuität in der Entwicklung, so dass Investorinnen, Entwicklerinnen und Anwenderinnen ein beständiges Tempo aufrecht erhalten können.
9. Ständige Aufmerksamkeit gegenüber technisch hervorragender Qualität und gutem Design erhöht Agilität.
10. Einfachheit – die Kunst, unnötige Arbeit zu minimieren – ist essentiell.
11. Die besten Architekturen, Anforderungen und Designs stammen aus sich selbst organisierenden Teams.
12. Regelmäßig evaluiert das Team wie es noch effektiver arbeiten kann und passt sich entsprechend an.

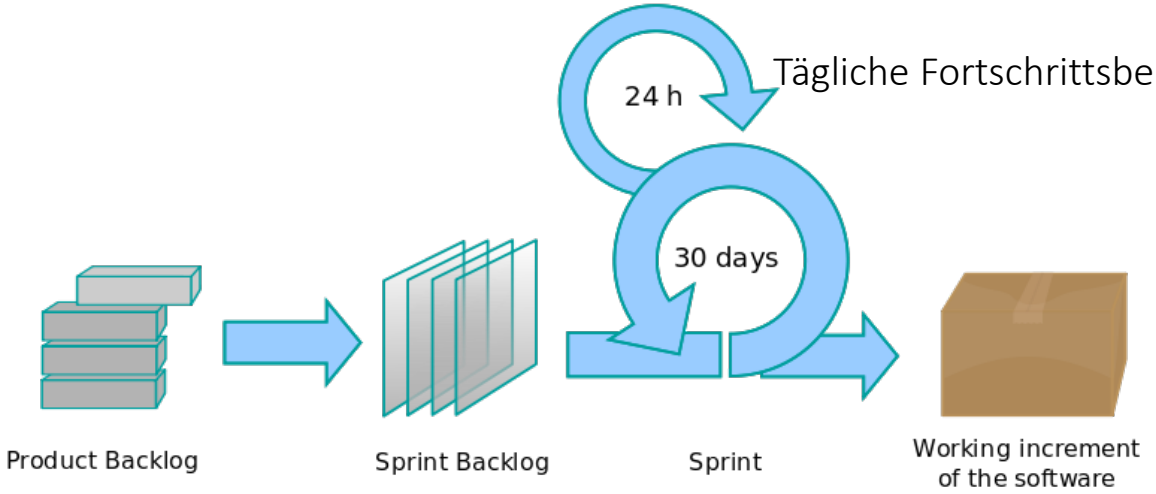
Was ist Scrum?

- SCRUM ist ein *agiles Managementframework,* bei dem Entwicklerteams als eine *Einheit* zusammenarbeiten, um ein gemeinsames Ziel zu erreichen.
- Ziel: Ordnung ins Chaos der Entwicklung zu bringen
  - Flexibel auf Änderungen der Anforderungen durch kurze Entwicklungszyklen reagieren
  - Enthält *keine* Praktiken, die vorschreiben, wie die Software entwickelt werden soll



Leitidee und Ablauf

- Ein Team ist besonders produktiv, wenn es sich mit seinem Produkt identifiziert und auch dafür die Verantwortung trägt.



Dokument über Anforderungen (Liste von Features, Bugs, etc.)

Dokument, welches die Aufgaben listet, die getan werden müssen, um das Feature zu implementieren.

Zeitraum der Realisierung

Laufendes Produkt mit neuem Feature

Prinzipien von Scrum

- Es gibt *keine Projektleiterin*!
  - Team teilt sich selbständig Arbeit auf.
- *Pull-Prinzip*
  - Nur das Team kann entscheiden, wieviel Arbeit in gegebener Zeit geleistet werden kann
- *TimeBox*
  - Es existieren klare zeitliche Grenzen
- Potential *Shippable Code*/Produkt
  - Ergebnis sind immer fertig Produkte

Rollen in Scrum

- Produkt Owner (Die Visionärin, kein Chef!)
  - Pflegt und priorisiert Product Backlog, fachlicher Ansprechpartnerin für Kunden (evtl. bei tägl. SCRUMs dabei)
  - Anforderungsmanangement, Releasemanagement, Kosten und Nutzen Betrachtung
- Das Team (Die Lieferanten)
  - 5-10 Personen meist interdisziplinär
  - Selbst-organisierend, tägl. Meetings
- Scrum Coach*
  - Verantwortung für SCRUM-Prozess
  - Moderiert, vermittelt, optimiert
- Die Kunden (Geldgeber)
- Die Anwender (Benutzen das Produkt am Ende)
- Die Manager (Organisatorische Leitung des Teams)

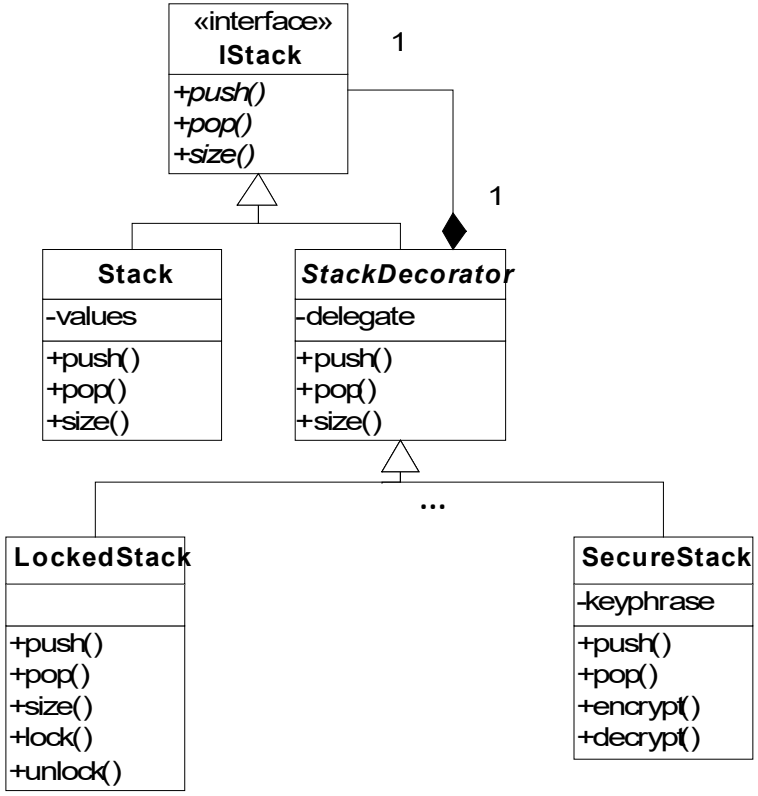
*wird meist als Master, hergeleitet aus Mastery, bezeichnet

Decorator – Structural Pattern I

Beschreibung	Inhalt
Pattern Name und Klassifikation	Decorator – Structural Pattern
Zweck	Fügt dynamisch Funktionalität zu bereits bestehenden Klassen hinzu.
Motivation	Wir benötigen flexible Implementierungen einer Klasse, die je nach Kontext unterschiedlich ausfallen.
Anwendbarkeit	Funktionale Erweiterungen sind optional. Anwendbar, wenn Erweiterungen mittels Vererbung unpraktisch ist.
Konsequenzen	Flexibler als statische Vererbung. Problem der Objektschizophrenie (ein Objekt ist zusammengesetzt aus mehreren Objekten). Viele kleine Objekte.

Decorator – Structural Pattern IV

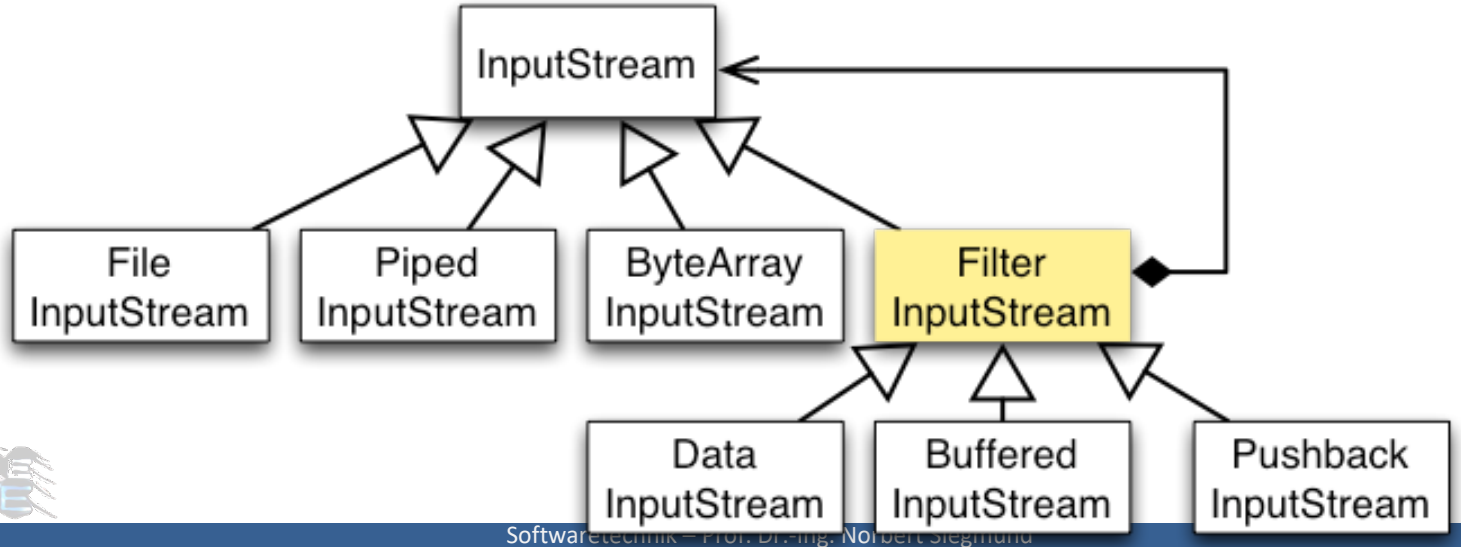
- Struktur des Beispiels



Decorator in Java

- java.io enthält verschiedene Funktionen zur Ein- und Ausgabe:
  - Programme **operieren auf Stream-Objekten** ...
  - **Unabhängig** von der Datenquelle/-ziel und der Art der Daten

```
FileInputStream fis = new FileInputStream(“my.txt");
BufferedInputStream bis = new BufferedInputStream(fis);
GzipInputStream gis = new GzipInputStream(new ObjectInputStream(bis));
```

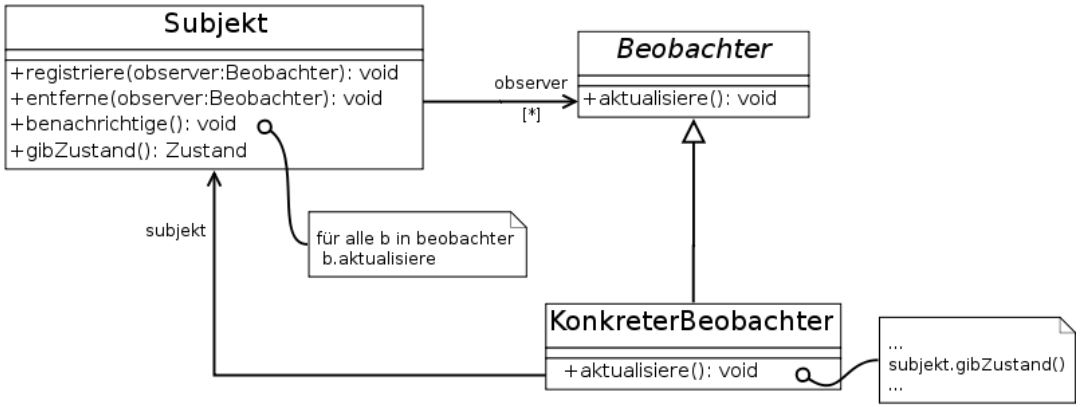


Observer – Behavioral Pattern I

Beschreibung	Inhalt
Pattern Name und Klassifikation	Observer – Behavioral Pattern
Zweck	Objekt verwaltet Liste von abhängigen Objekten und teilt diesen Änderungen mit
Auch bekannt als	Publish-Subscribe
Motivation	Implementiert verteilte Ereignisbehandlung (bei einem Ereignis müssen viele Objekte informiert werden). Schlüsselfunktion beim Model-View-Controler (MVC) Architektur-Pattern
Anwendbarkeit	Wenn eine Änderung an einem Objekt die Änderung an anderen Objekten erfordert und man weiß nicht, wie viele abhängige Objekte es gibt. Wenn ein Objekt andere Objekte benachrichtigen willl, ohne dass es die Anderen kennt.

Observer – Behavioral Pattern IV

- Struktur
- Code




```
interface IObserver {
      public void update(String message);
}
interface ISubject {
      public void registerObserver(Observer observer);
      public void removeObserver(Observer observer);
      public void notifyObservers();
}
```

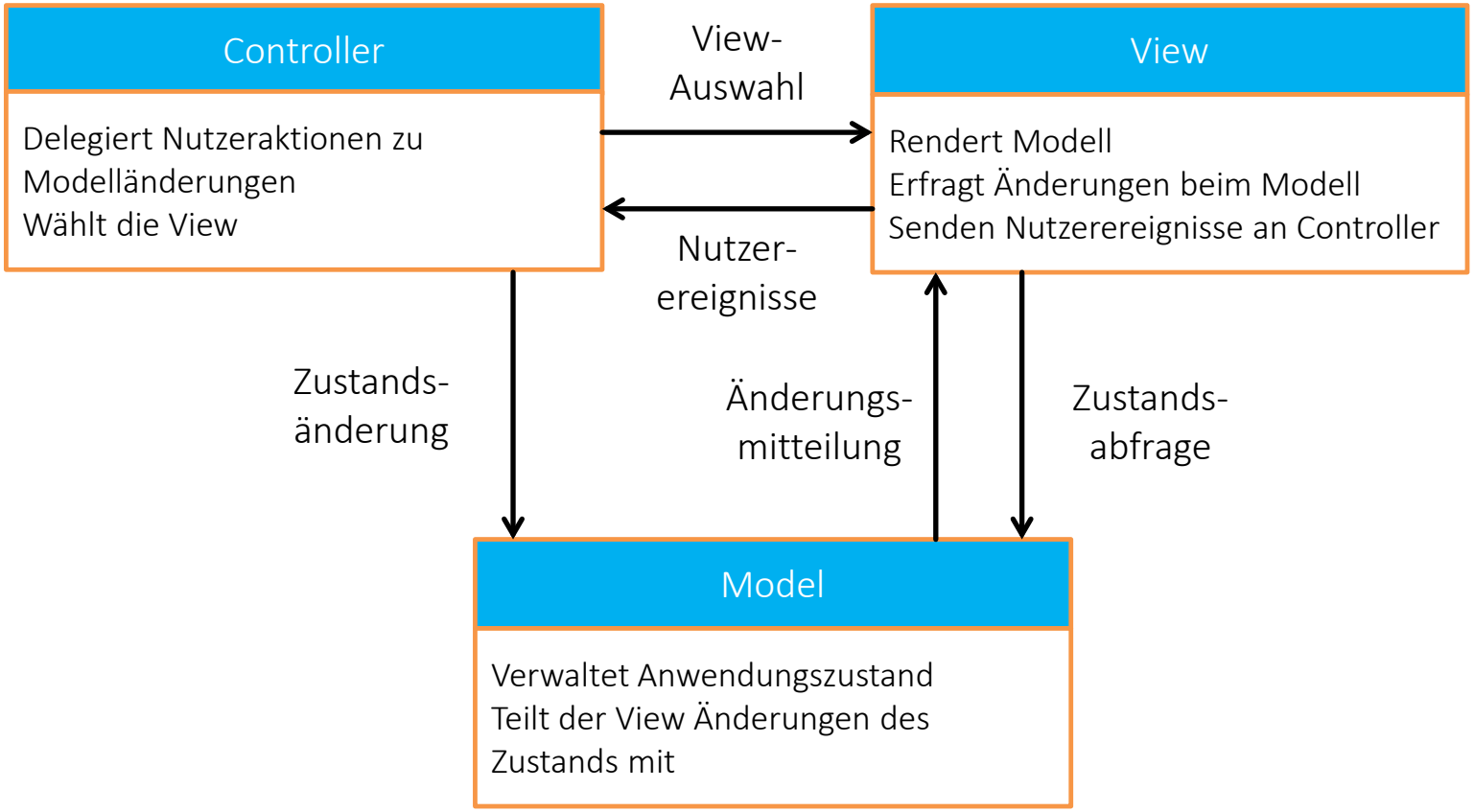
Model-View-Controller (MVC) Architektur

Idee: Separiere *Präsentation* und *Interaktion* von den *Daten* des Systems

- Das System ist strukturiert in drei Komponenten:
  - *Model*: verwaltet Systemdaten und Operationen auf den Daten
  - *View*: Präsentiert die Daten zum Nutzer
  - *Controller*: händelt Nutzerinteraktion; schickt Informationen zur View und zum Model

- Nützlich, wenn es mehrere Wege gibt auf Daten zuzugreifen
- Ermöglicht das Ändern der Daten unabhängig von deren Repräsentation
- Unterstützt unterschiedliche Präsentationen der selben Daten

MVC Übersicht



SOLID: Prinzipien für OO Design

Ziel: Software verständlicher, flexibler und wartbarer zu machen

Single-Responsibility Principle
Open-Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

Prinzipien stellen Basis eines jeden OO-SW Designs dar

Single-Responsibility Principle

- Jedes Modul, Klasse, Funktion eines Programms soll Verantwortung über eine bestimmte Teilfunktionalität haben
- Es sollte nur diese *eine* Verantwortung haben (siehe Responsibility-Driven Design)

- Effekt:
  - Robust gegenüber zukünftigen Änderungen
  - Änderungen sind lokalisiert an einen Ort
  - Einfaches Verständnis über die Verantwortung, wenn limitiert auf eins

Open-Closed Principle

- Software Entitäten (Module, Klassen, Funktionen, etc.) sollten offen für Erweiterungen, aber geschlossen für Modifikationen sein
- Funktionalität kann erweitert werden ohne existierenden Code zu modifizieren

- Effekte:
  - Klasse kann z.B. durch Vererbung erweitert werden ohne vorhandenen Code zu modifizieren
  - Klasse kann von anderen benutzt werden, ohne sie verändern zu müssen (Information Hiding)

Liskov Substitution Principle

- Jedes Objekt S, welches ein Subtyp (Kindklasse) eines Typs T ist, sollte anstelle eines Objekts von Typ T benutzen werden können, ohne das sich das Verhalten des Programmes (Korrektheit, Performance, etc.) verändert
- Strong behavioral subtyping genannt

- Effekte:
  - Garantiert semantische Interoperabilität von Typen in einer Hierarchie

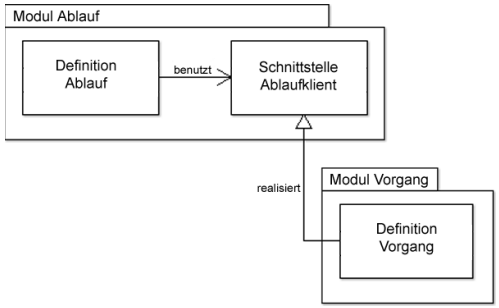
Interface Segregation Principle

- Kein Klient sollte nicht dazu gezwungen werden, abhängig von Methoden zu sein, die man nicht nutzt
- Große Interfaces in mehrere kleinere aufteilen, so dass Klienten nur tatsächlich benutzt Teile implementieren müssen

- Effekte:
  - Komponenten werden in kleinere, leichter zu wartende Systeme zerlegt
  - SW ist einfacher zu refaktorisieren durch kleinere Interfaces
  - Klassen müssen nur die für sie relevanten Methoden kennen
  - Führt zu höherer Kohäsion (siehe GRASP)

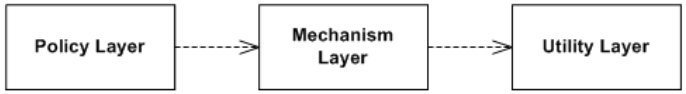
Dependency Inversion Principle (DIP)

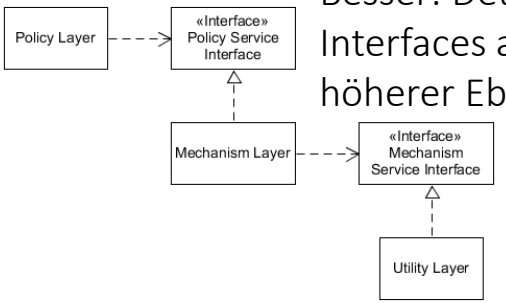
- Module höherer Ebenen sollten nicht von Modulen von unteren Ebenen abhängig sein, stattdessen nur von Abstraktionen
- Abstraktionen sollten nicht von Details abhängen, sondern anders herum



- Effekte:
  - Führt zu verringerte Kopplung von Komponenten
  - Verhindert zyklische Abhängigkeiten zwischen Komponenten

Höhere Ebene ist abhängig von Detailebene





Besser: Details sind durch Interfaces abgekoppelt von höherer Ebene

Exkurs: Cohesion / Kohäsion

Cohesion ist ein Maß, *wie gut Teile einer Komponente "zusammen gehören".*

- Cohesion ist schwach, wenn Elemente nur wegen ihrer Ähnlichkeit ihrer Funktionen zusammengefasst sind (z.B., **`java.lang.Math`**).
- Cohesion ist stark, wenn alle Teile für eine Funktionalität tatsächlich benötigt werden (z.B. **`java.lang.String`**).

Welche Eigenschaft führt zu einer besseren Qualität?

- Starke cohesion *verbessert Wartbarkeit* and Adaptivität durch *die Einschränkung des Ausmaßes von Änderungen* auf eine kleine Anzahl von Komponenten.

Exkurs: Coupling / Kopplung

Coupling ist ein Maß der *Stärke der Verbindungen* zwischen Systemkomponenten.

- Coupling ist eng (tight) zwischen Komponenten, wenn sie stark voneinander abhängig sind (z.B., wenn sehr viel Kommunikationen zw. beiden stattfindet).

- Coupling ist lose (loose), wenn es nur wenige Abhängigkeiten zwischen Komponenten gibt.

Welche Eigenschaft führt zu einer besseren Qualität?

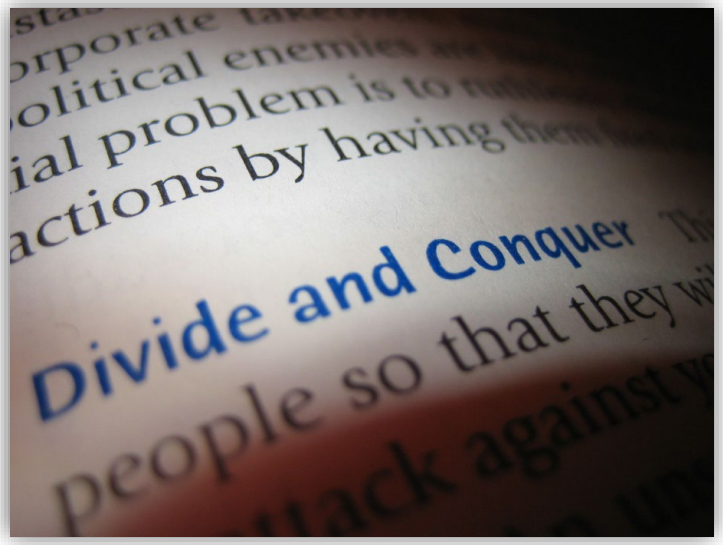
- **Loose coupling** *verbessert Wartbarkeit* und Adaptabilität, da *Änderungen in einer Komponente sich weniger wahrscheinlich auf anderen Komponenten auswirkt.*

Fünf Kriterien für gutes Design

- Modular Decomposability
- Modular Composability
- Modular Understandability
- Modular Continuity
- Modular Protection

Fünf Kriterien: Modular Decomposability

- Problem kann in wenige kleinere, weniger komplexe Sub-Probleme zerlegt werden
- Sub-Probleme sind durch einfache Struktur verbunden
- Sub-Probleme sind unabhängig genug, dass sie einzeln bearbeitet werden können



Fünf Kriterien: Modular Decomposability

- Modular decomposability setzt voraus: Teilung von Arbeit möglich
- Beispiel: Top-Down Design
- Gegenbeispiel: Ein globales Initialisierungsmodul, eine große Main-Methode

Fünf Kriterien: Modular Composability

- Gegenstück zu modular decomposability
- Softwarekomponenten können beliebig kombiniert werden
- Möglicherweise auch in anderer Domäne

- Beispiel: Tischreservierung aus NoMoreWaiting kann auch für das Vormerken von Büchern benutzt werden (gutes Design!)

Fünf Kriterien: Modular Understandability

- Entwickler kann jedes einzelne Modul verstehen, ohne die anderen zu kennen bzw. möglichst wenige andere kennen zu müssen

- Wichtig für Wartung
- Gilt für alle Softwareartefakte, nicht nur Quelltext
- Gegenbeispiel: Sequentielle Abhängigkeit zwischen Modulen

Fünf Kriterien: Modular Continuity

- Kleine Änderung der Problemspezifikation führt zu Änderung in nur einem Modul bzw. möglichst wenigen Modulen

- Beispiel 1: Symbolische Konstanten (im Gegensatz zu Magic Numbers)
- Beispiel 2: Datendarstellung hinter Interface kapseln
- Gegenbeispiel: Magic Numbers, (zu viele) globale Variablen

Fünf Kriterien: Modular Protection

- Abnormales Programmverhalten in einem Modul bleibt in diesem Modul bzw. wird zu möglichst wenigen Modulen propagiert

- Motivation: Große Software wird immer Fehler enthalten
- Beispiel: Defensives Programmieren
- Gegenbeispiel: Nullpointer in einem Modul führt zu Fehler in anderem Modul

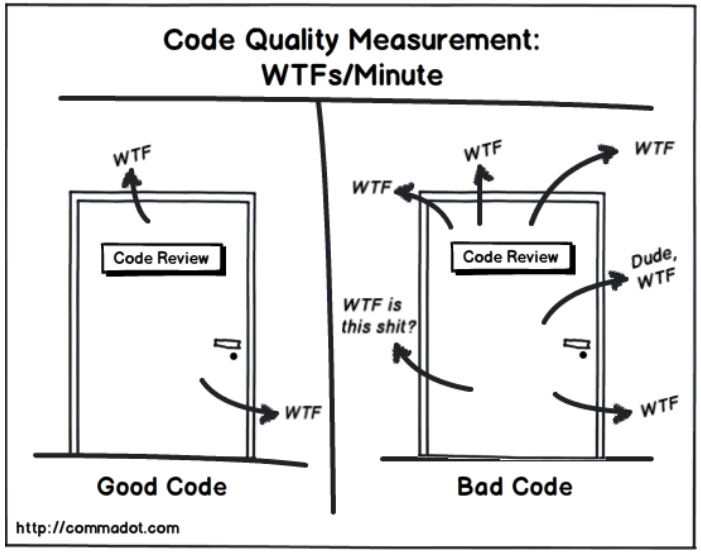
Unit Tests

- Ziel: Individuelle Module (meist auf Klassenebene) und Funktionen werden getestet, um deren korrekte Funktionsweise sicherzustellen

- Typischerweise automatisiert
- Oft durch Entwickler spezifiziert
- Fokus auf eine Funktion/Methode/Modul
- Stubs/Mock-Objekte, wenn dabei andere Module aufgerufen werden
  - Stubs/Mocks emulieren anderen Objekte/Methoden im Programm, welche notwendig sind, um das eigentliche Modul zu testen

Code Reviews

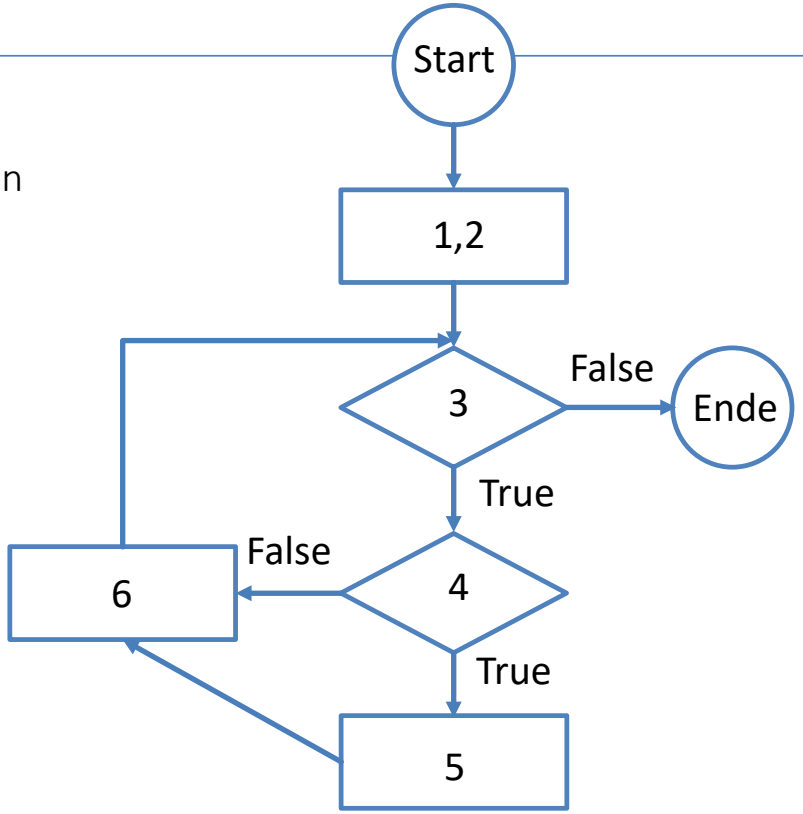
- Eine Familie verschiedener Techniken
  - Pair Programming
  - Walkthroughs
  - Inspections
  - Personal reviews
  - Formal technical reviews
- Review/inspizieren:
  - Zur genauen Begutachtung
  - Mit einem Auge auf Korrektur und Bewertung
- Menschen (peers) sind die Begutachter



Kontrollflussgraph

- G = (V, E) wobei
  - V ist eine Menge von Basisblöcken
  - E ist eine Menge von gerichteten Kontrollflüssen

- Beispiel:




```
1. a = Read(b)
2. c = 0
3. while (a > 1)
4.    If (a^2 > c)
5.        c = c + a
6.    a = a - 2
```

Graph = Menge aller möglichen Ausführungen eines Programmes.
Pfad = Eine konkrete Ausführung eines Programmes.

Verifikation und Validation

Verifikation:

- Bauen wir das das *Produkt richtig*?
  - D.h., entspricht es der Spezifikation?

Validation:

- Bauen wir das *richtige Produkt*?
  - D.h., entspricht es den Nutzererwartungen?