Technische Universität Berlin

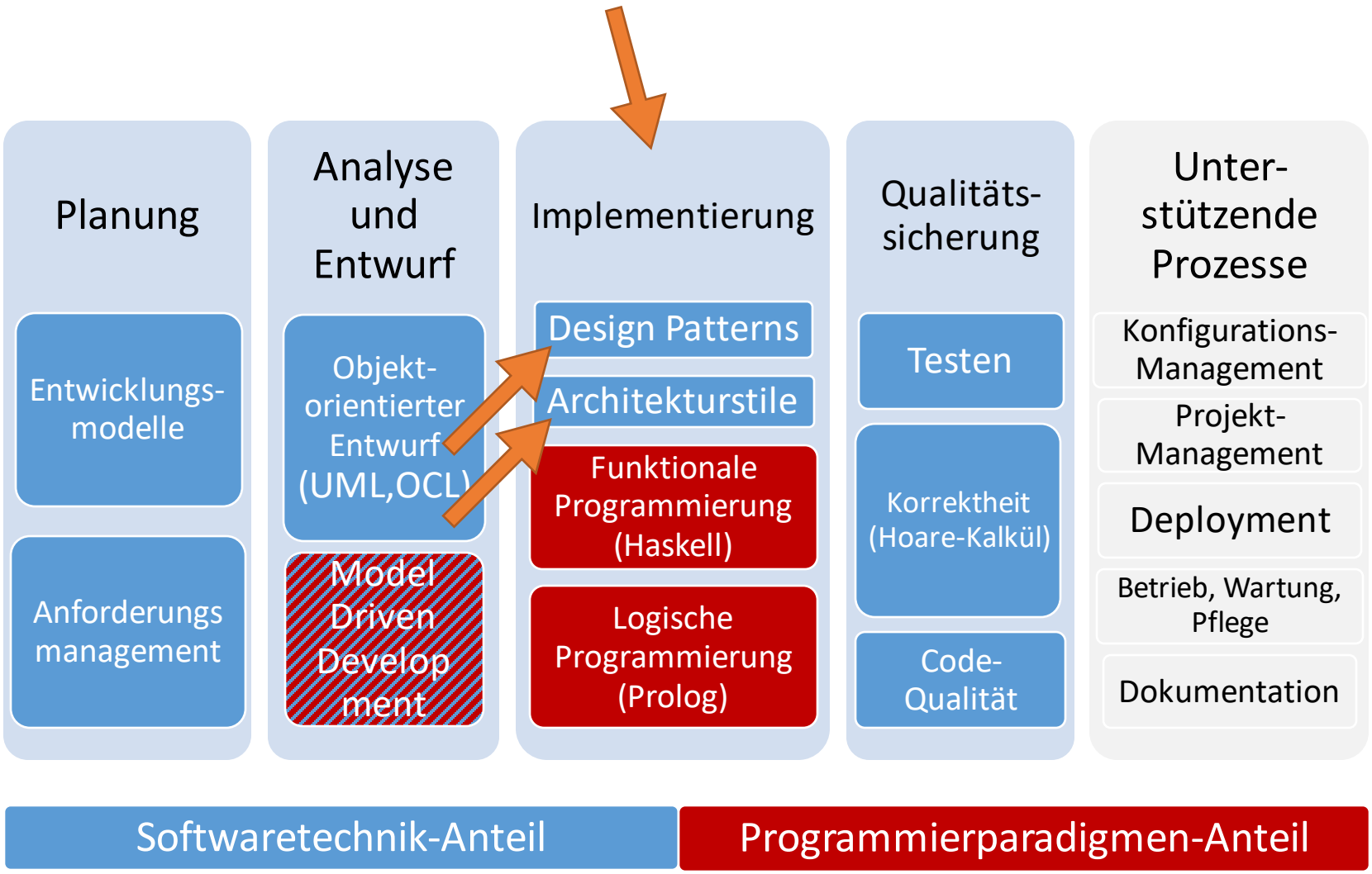
Softwaretechnik und Programmierparadigmen

11 Implementierung

Prof. Dr. Sabine Glesner
Software and Embedded Systems Engineering
Technische Universität Berlin

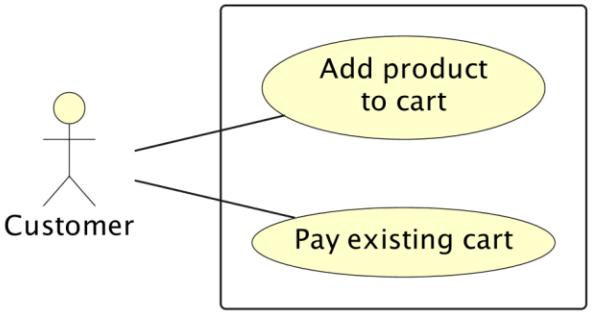
SESE

Diese VL

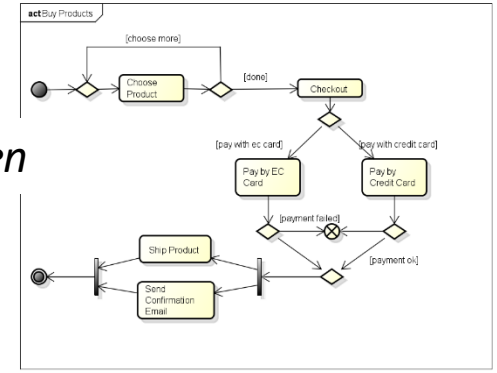

Planung
Entwicklungs-modelle
Anforderungs management
Analyse und Entwurf
Objekt-orientierter Entwurf (UML,OCL)
Model Driven Develop ment
Implementierung
Design Patterns
Architekturstile
Funktionale Programmierung (Haskell)
Logische Programmierung (Prolog)
Qualitäts-sicherung
Testen
Korrektheit (Hoare-Kalkül)
Code-Qualität
Unter-stützende Prozesse
Konfigurations-Management
Projekt-Management
Deployment
Betrieb, Wartung, Pflege
Dokumentation
Softwaretechnik-Anteil
Programmierparadigmen-Anteil

Implementierung

Funktionale und nicht-funktionale Anforderungen sind nun bekannt und **abstrakte Modelle** vom System wurden entworfen



Verhalten

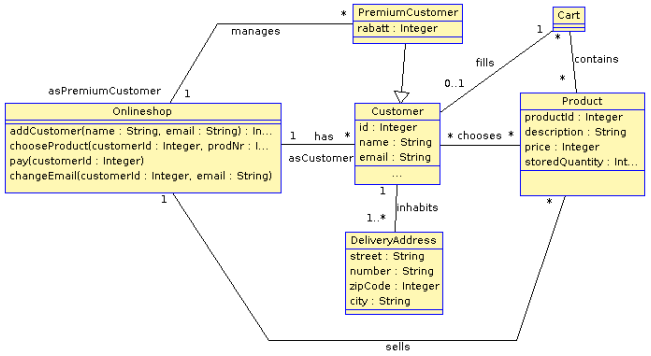


Anforderungen

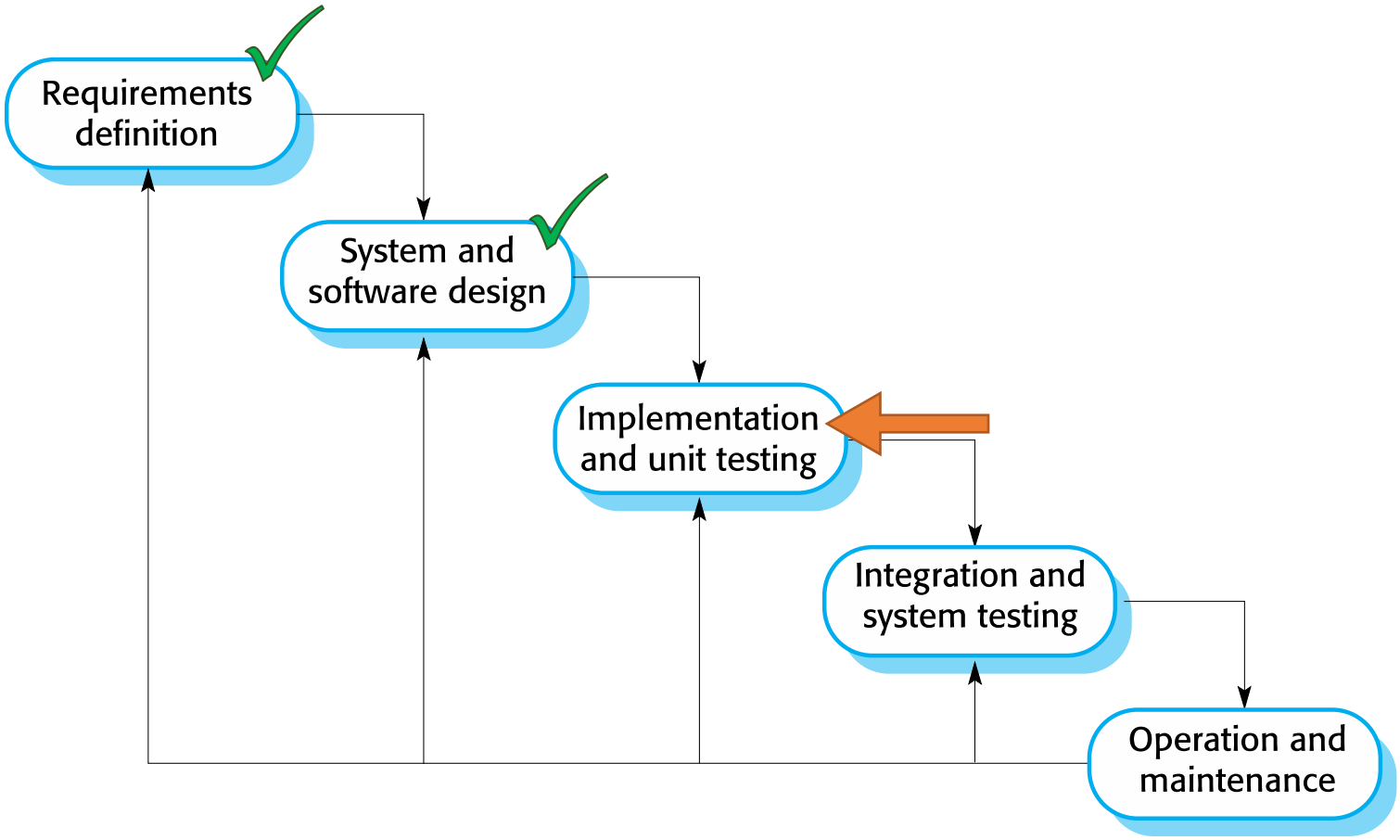
Sie werden gebeten für einen kleines Unternehmen, das Schuhe und Kleidung verkauft, die Verwaltungssoftware eines Online-Shops zu entwickeln. Der Onlineshop soll es dem Kunden ermöglichen, Produkte in einen Warenkorb zu legen und diesen zu bezahlen. Als Bezahlmethoden sind zunächst Bankeinzug und Kreditkartenzahlung vorgesehen. Bevor die Bestellung aufgegeben wird, muss sichergestellt werden, dass die Bezahlung tatsächlich erfolgen kann.

Weiterhin soll das System gleichzeitig auch die Nutzer:innen verwalten. Sowohl Kunden als auch Mitarbeiter sollen registriert werden können. Auf Sicherheit soll entsprechend geachtet werden.
Produkte sollen über das Webinterface auch gesucht werden können. Dabei sollen Rechtschreibfehler toleriert werden und innerhalb von einer akzeptablen Zeit sollen passende Produkte angezeigt werden.
Alle Funktionen sollen von Nicht-Entwicklern ausgiebig getestet werden.

Struktur



Implementierung



Ian Sommerville, Software-Engineering, Chapter 2

Inhalt

Implementierung

- Einführung
- Architekturstile
- Design Patterns

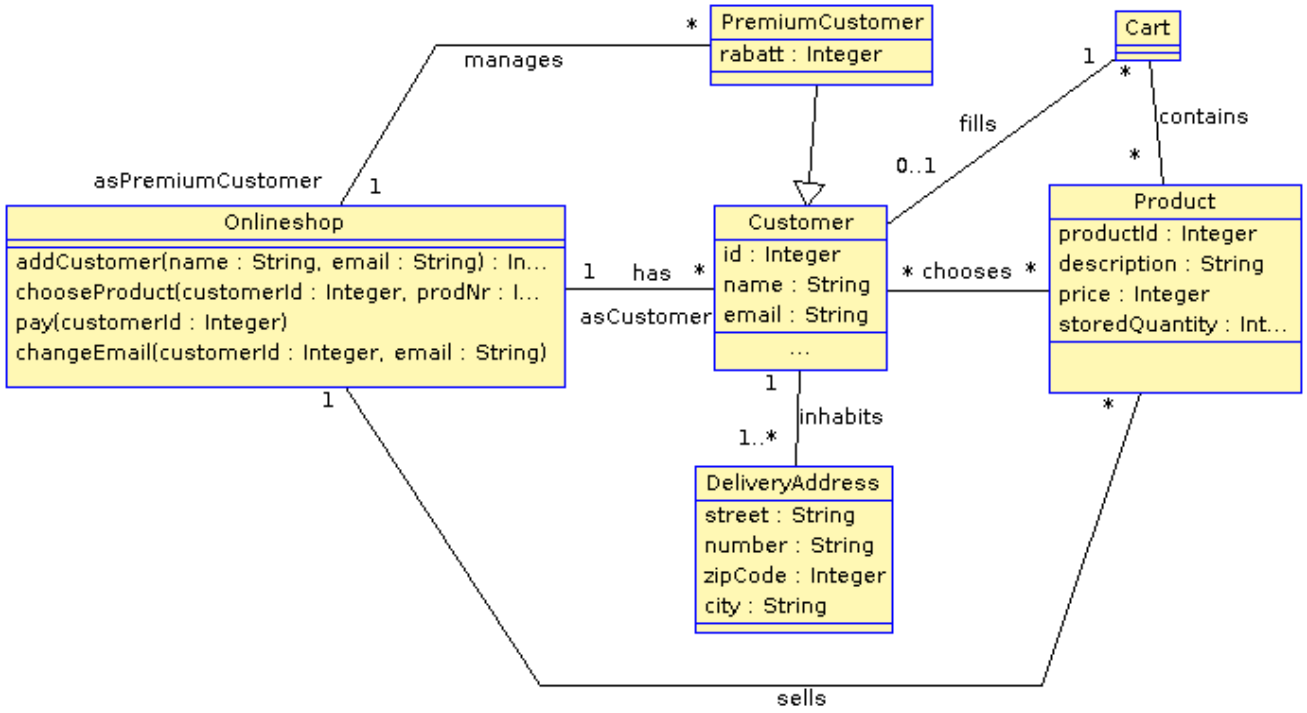
Inhalt

Implementierung

- Einführung
- Architekturstile
- Design Patterns

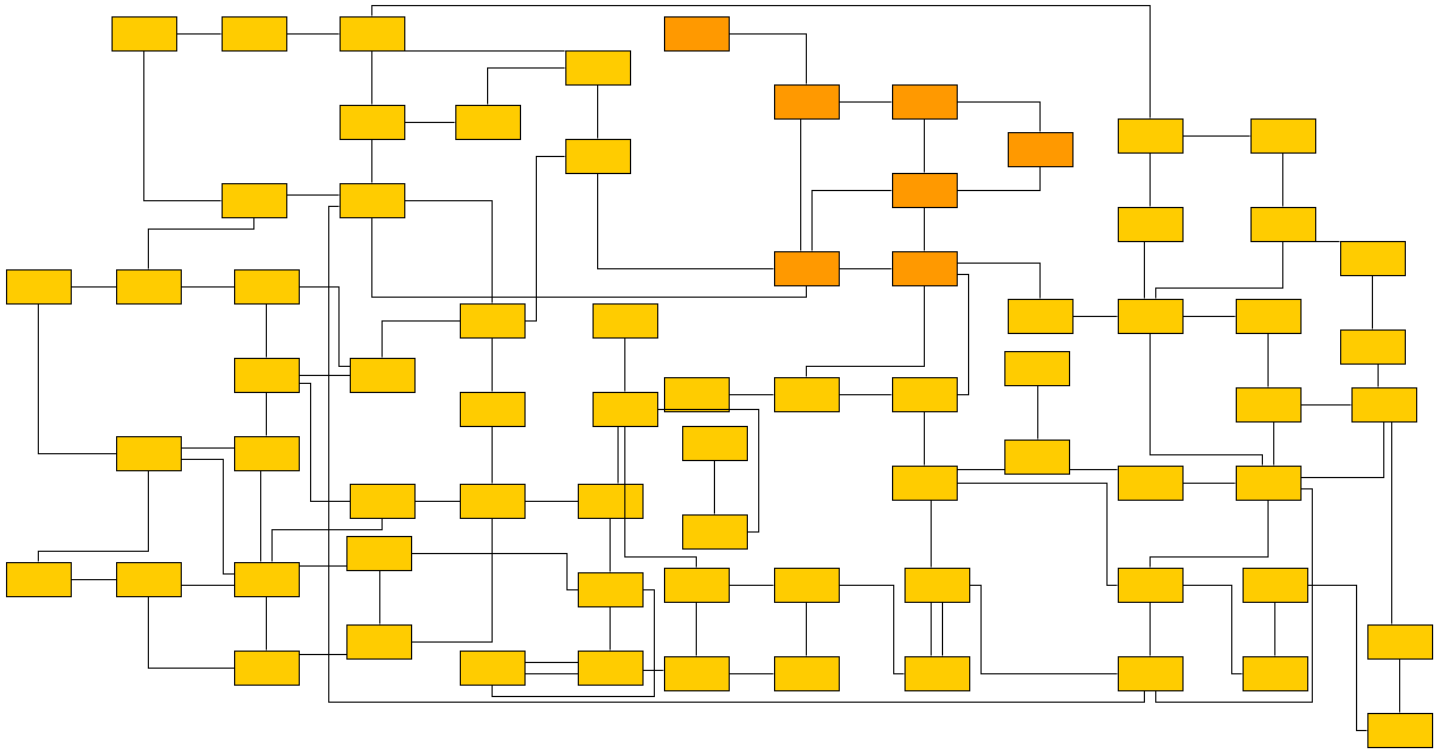
Motivation

Bisher ist die **Struktur unseres Systems** recht übersichtlich…



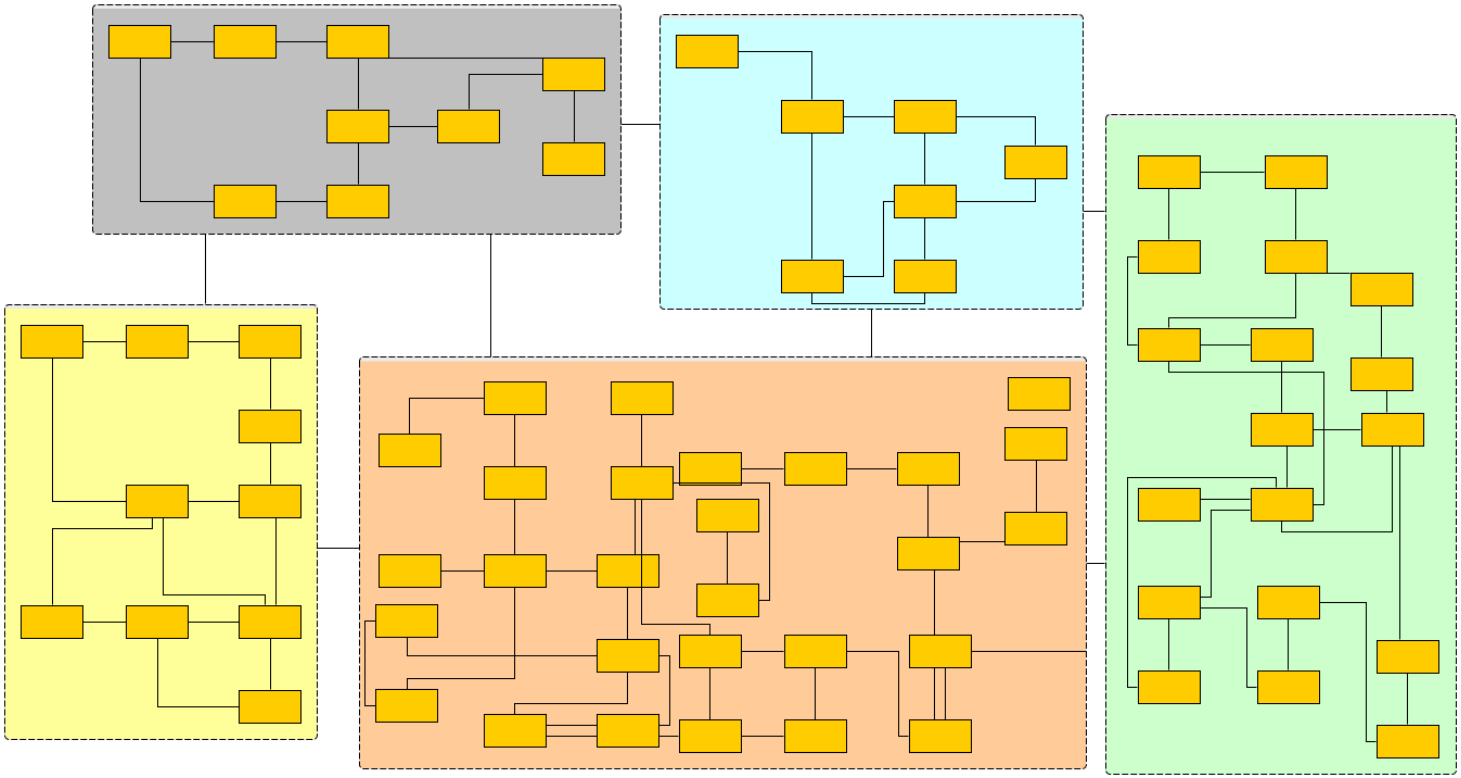
Motivation

…das ist aber nicht immer der Fall



Modularität

Ziel ist eine **sinnvolle Aufteilung** des Gesamtsystems **in Komponenten**



Modularität

Die Aufteilung kann nach verschiedenen Kriterien erfolgen

Separations of Concerns	Trennung nach Anforderungen/Story
Information Hiding	Verschiedene Sichten auf die Daten
Wiederverwendbarkeit	Häufig genutzte Teile/Generelle Teile
Erweiterbarkeit	Vorgesehene Anbindung/Spezialisierung
Wartbarkeit	Übersichtlich/in sich geschlossen

Oder sich aus anderen Entscheidungen ergeben

- Wahl der Plattform
- Wahl der Programmiersprache(n)
- Wahl des GUI-Systems/Frontend
- Wahl der Persistierung/Datenhaltung

Unterstützende Software

Versions-verwaltung

Integration/ Deployment

Problem& Aufgaben-verfolgung

Dokumentation

javadoc

Inhalt

Implementierung

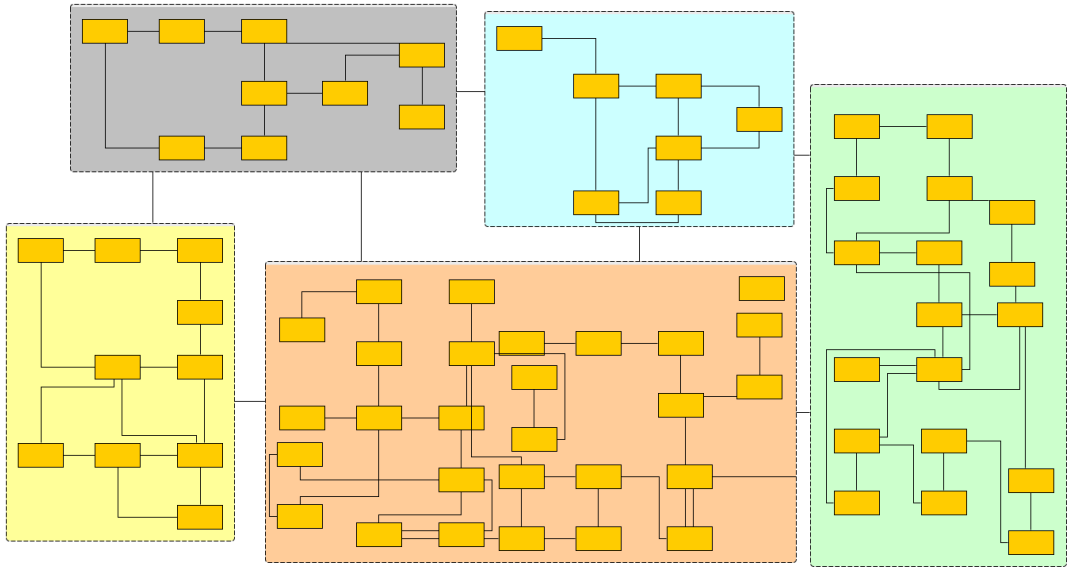
- Einführung
- Architekturstile
- Design Patterns

Architekturstile

Einige **prinzipielle Systemstrukturen** haben sich als häufig angewendetes **Muster** (Architekturstil) etabliert

In dieser VL:

- Model-View-Controller
- Layer-based
- Repository-based
- Pipes-and-Filter
- Event-based
- Interrupt-based
- Client-Server
- Peer-to-Peer
- Service-oriented

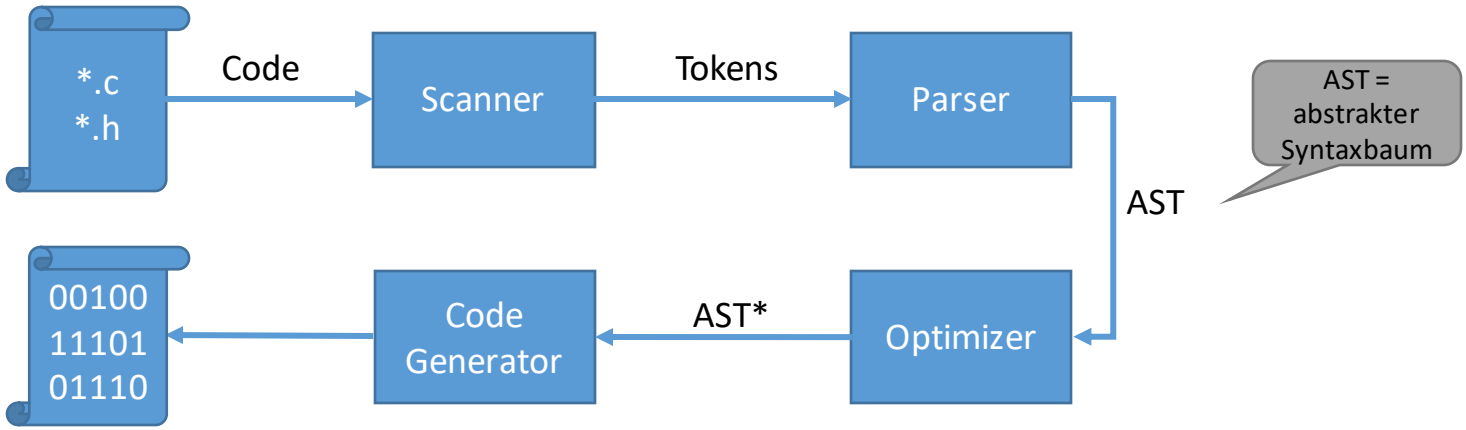


Pipes-and-Filter Architecture

Arbeitsschritte **nur durch Daten verknüpft**

- fließen wie durch ein Rohr von einer Komponente zur nächsten
- typisch für Systeme, die Daten schrittweise **weiterverarbeiten**
- Bearbeitung **sequenziell** und **parallel** möglich

Compiler



Pipes-and-Filter Architecture

Vorteile

- System bildet **Geschäftsprozesse** direkt ab
- übersichtlich
- modular, leicht erweiterbar
- Systemzustand in den Daten gekapselt

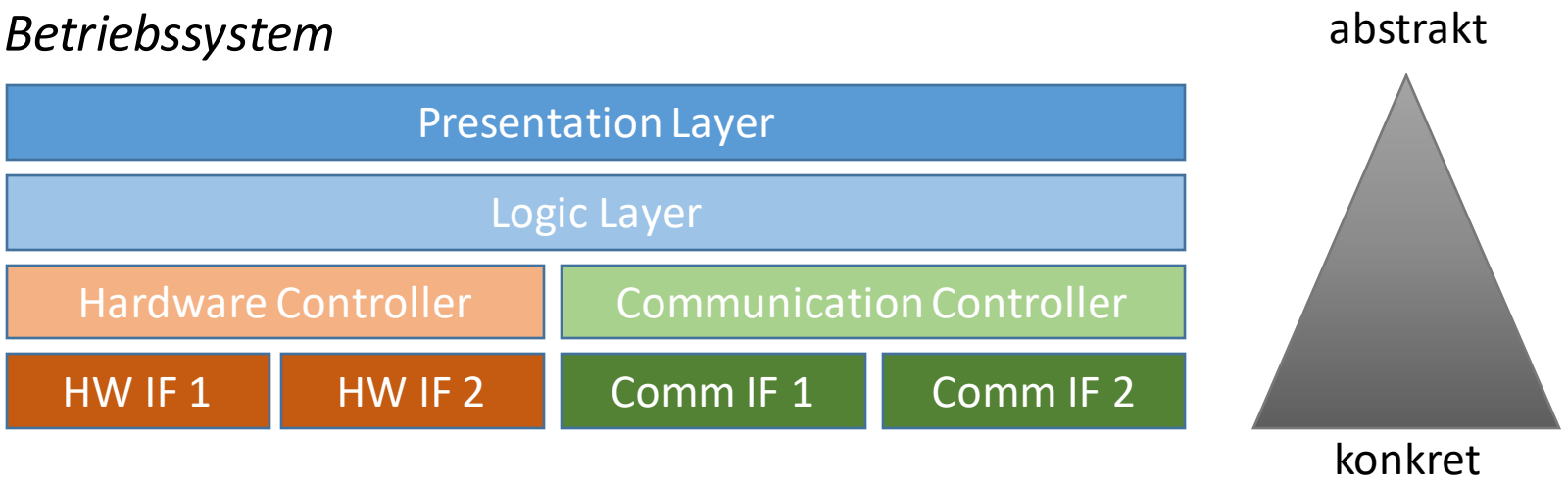
Nachteile

- Daten müssen von jeder Komponente erneut aufbereitet werden
- Vorgegangene Schritte müssen immer vollständig abgeschlossen werden

Layer-based Architecture

Aufteilung in mehrere **Abstraktionsschichten**

- Trennung z.B. von technischen Details und Inhalten
- Schichten bieten darüber liegenden Schichten Dienste an
- Elementare Dienste in unterster Schicht



Layer-based Architecture

TCP/IP-Referenzmodell (Wikipedia)

verschiedene Schichten und Protokolle der Internet-Kommunikation

Klar festgeschriebene Schnittstellen!

<table>
<tr><th>OSI-Schicht</th><th>TCP/IP-Schicht</th><th>Beispiel</th></tr>
<tr><td>Anwendungen (7)</td><td rowspan="4">Anwendungen</td><td rowspan="3">HTTP, UDS, FTP, SMTP, POP, Telnet, OPC UA</td></tr>
<tr><td>Darstellung (6)</td></tr>
<tr><td>Sitzung (5)</td></tr>
<tr><td></td><td>SOCKS</td></tr>
<tr><td>Transport (4)</td><td>Transport</td><td>TCP, UDP, SCTP</td></tr>
<tr><td>Vermittlung (3)</td><td>Internet</td><td>IP (IPv4, IPv6), ICMP (über IP)</td></tr>
<tr><td>Sicherung (2)</td><td rowspan="2">Netzzugang</td><td rowspan="2">Ethernet, Token Bus, Token Ring, FDDI, IPoAC</td></tr>
<tr><td>Bitübertragung (1)</td></tr>
</table>

Layer-based Architecture

Vorteile

- **Abstraktion** von Details der einzelnen Schichten
  Separation of Concerns, Information Hiding
- **Flexibler** Austausch von Schichten möglich
  Wartbarkeit, Erweiterbarkeit

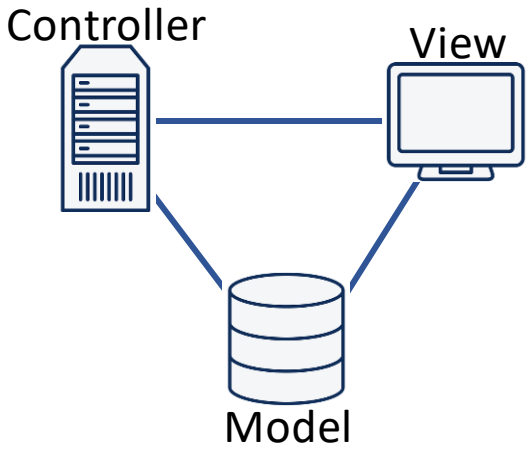
Nachteile

- Trennung kann **Performance-Nachteile** bringen
  Spezialfälle mit effizienteren Lösungen müssen generalisiert werden
- Anfragen/Antworten müssen über mehrere Schichten **weitergeleitet** werden

Model View Controller

Model

- enthält die persistenten **Daten**
- **Geschäftslogik** innerhalb dieser Daten
- **unabhängig** von anderen Einheiten



View (Präsentation)

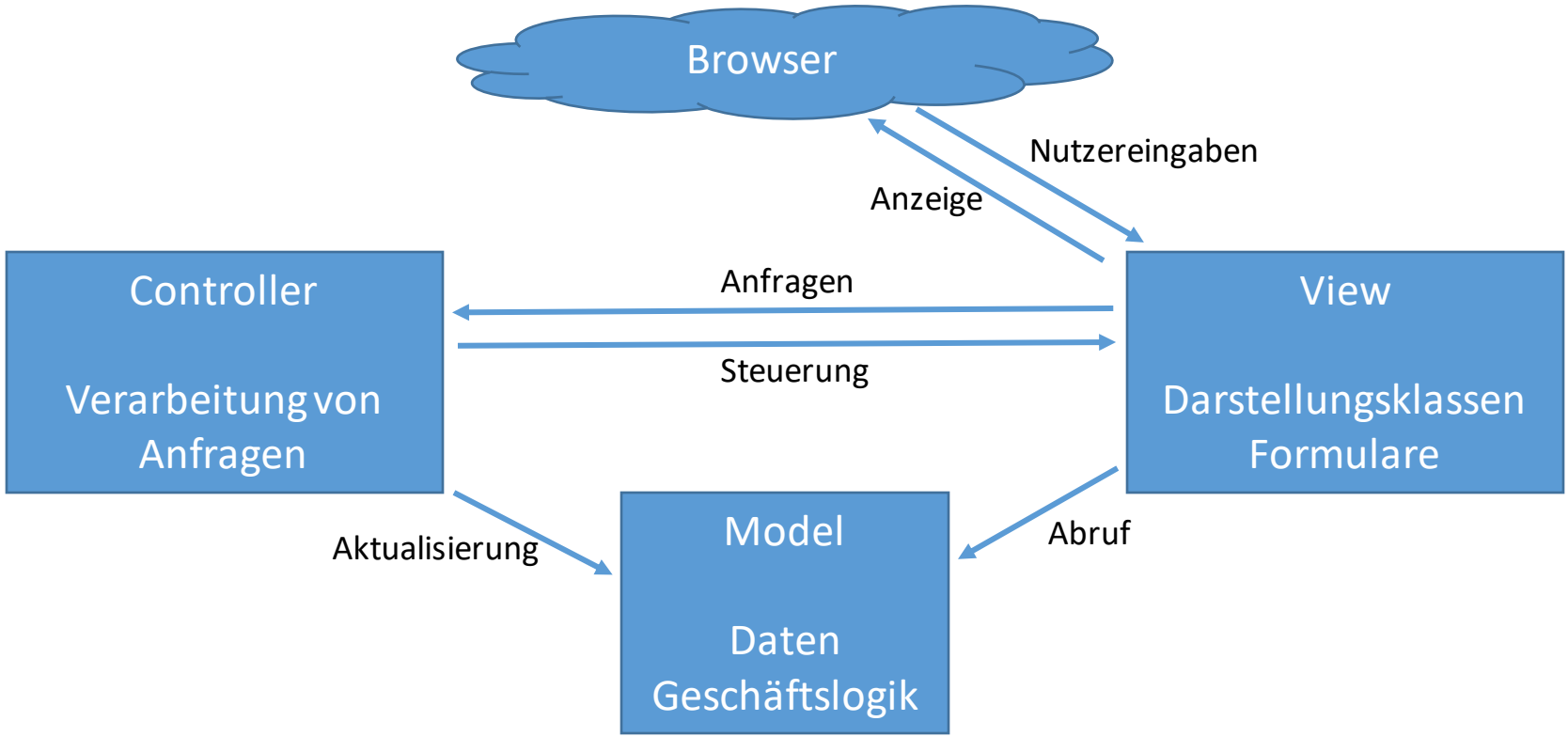
- **Darstellung** der Daten
- Entgegennahme von **Benutzerinteraktion**
- kennt Modell und Control
- **verschiedene Präsentationen** (Views) möglich

Controller (Steuerung)

- **verwaltet** die Präsentation
- **führt Benutzeranfragen aus** und gibt sie ggf. an das Modell weiter

Model View Controller

Web-Anwendung

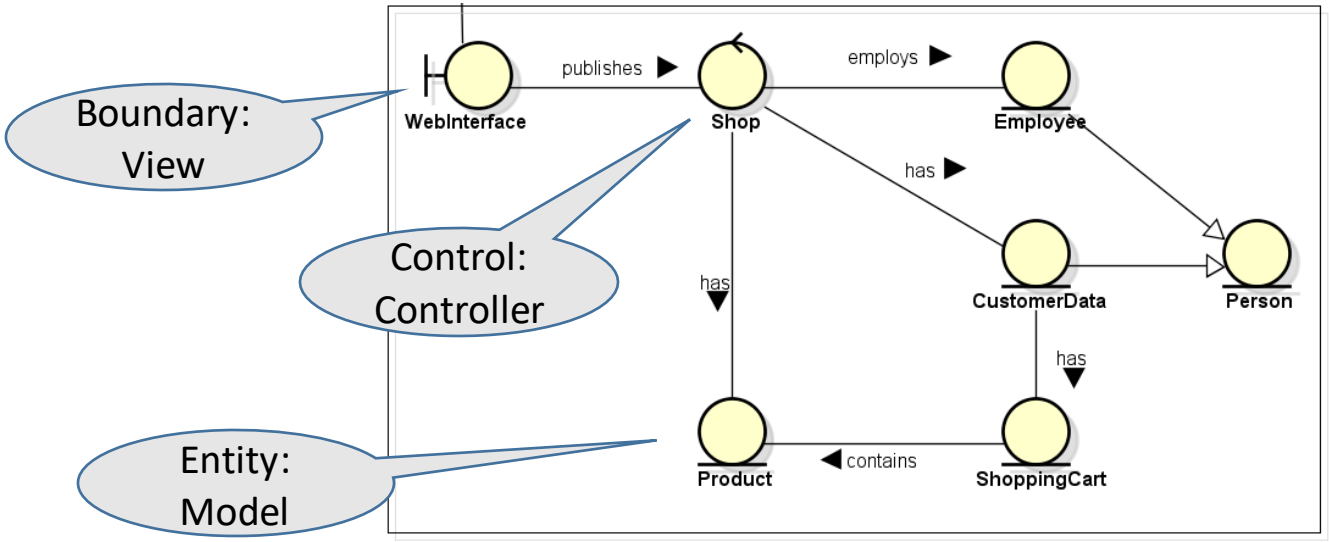


Model View Controller

Ermöglicht verschiedene **Interaktionen** mit dem System

- Typisch in Systemen mit Fokus auf Benutzerschnittstellen
- Erleichtert spätere Erweiterungen, z.B. neue Präsentationsarten (View)
- Ermöglicht Austausch getrennter Komponenten, z.B. der Datenbank

MVC ähnelt dem ECB-Pattern: Aufteilung in *boundary*, *entity* & *control*

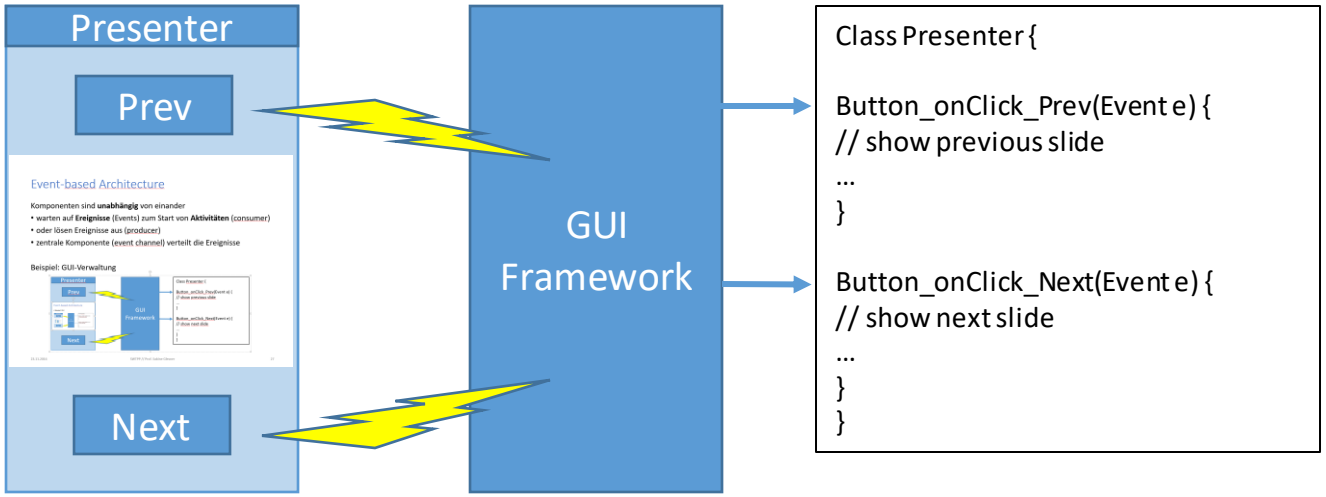


Event-based Architecture

Komponenten sind **unabhängig** von einander

- warten auf **Ereignisse** (Events) zum Start von **Aktivitäten** (consumer)
- oder lösen Ereignisse aus (producer)
- zentrale Komponente (**event channel**) verteilt die Ereignisse

GUI-Verwaltung



Event-based Architecture

Vorteile

- Reaktion auf Ereignisse kann **unmittelbar** erfolgen
- geeignet für **asynchrone/chaotische Umgebungen** (z.B. Benutzer-Interaktion)

Nachteile

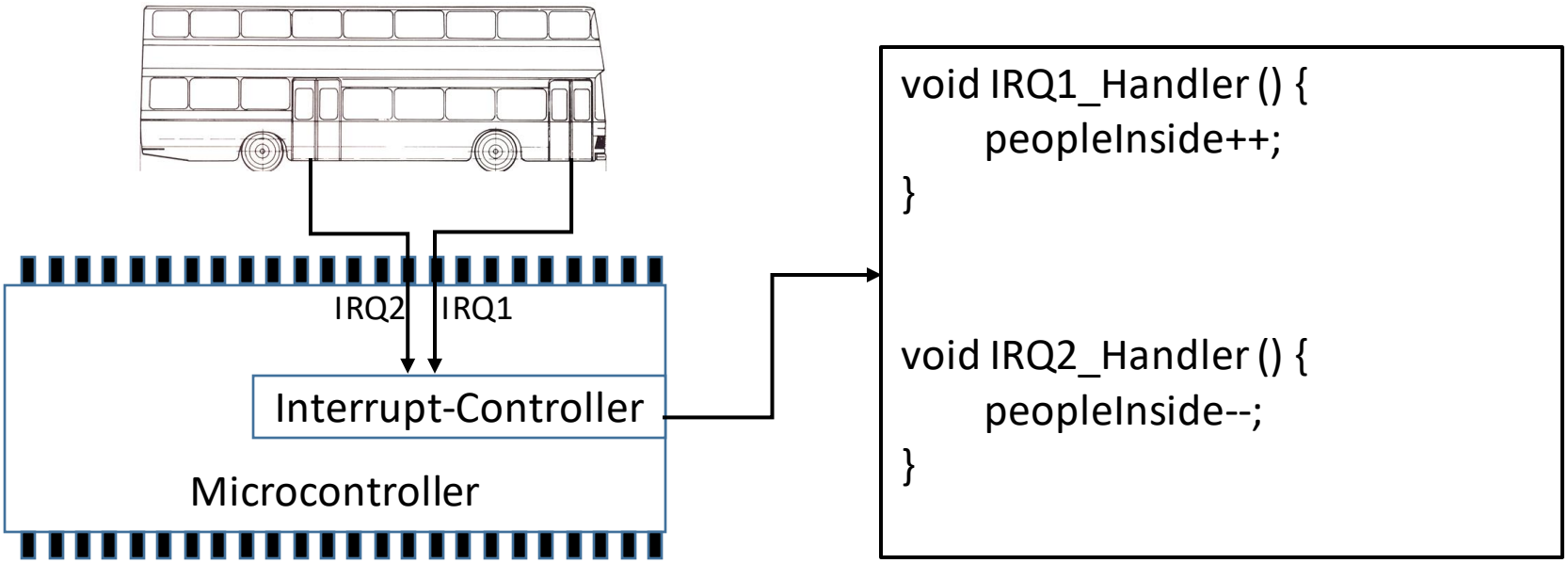
- Nicht behandelte Events müssen u.U. zwischengespeichert werden
- erhöhte Anforderungen an Synchronisation
- Verhalten **schwerer vorhersagbar**

Interrupt-based Architecture

Spezialfall von **event-based architectures**

- Verwendet Interrupts als Hardware(„low-level“)-Events
- Interrupts können maskiert (ignoriert) werden

Personenzähler im Bus

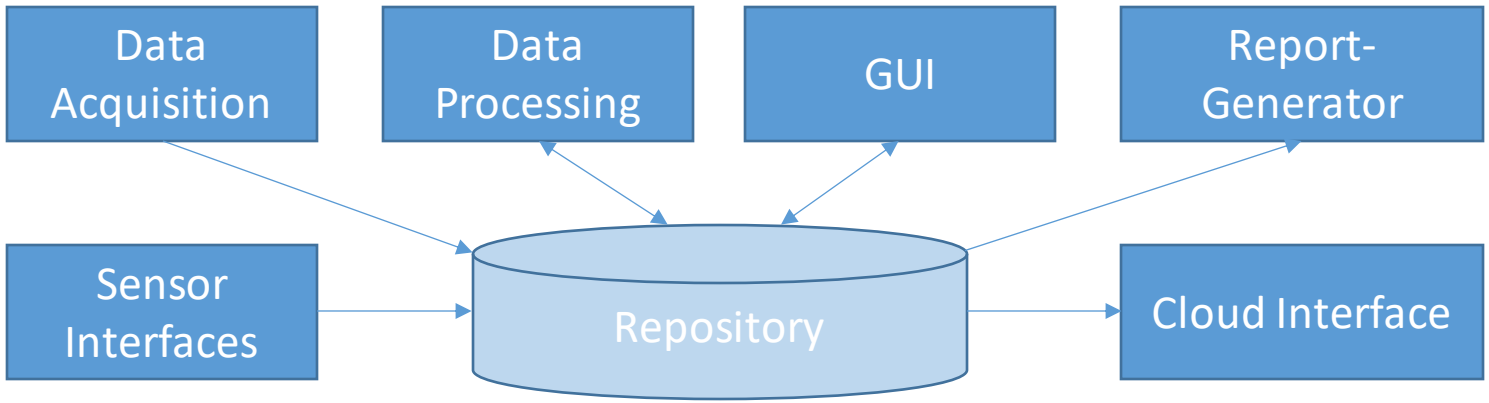


Repository-based Architecture

Organisation des Systems um **einen zentralen Datenspeicher**

- Komponenten sind über gemeinsame Daten (z.B. Datenbank) verbunden
- Koordination von Komponenten im Repository (z.B.: Trigger, Locks)

Beispiel: Zentrale Messdatenerfassung



Repository-based Architecture

Vorteile

- **wenig Schnittstellen**
- konsistente, zentrale Datenhaltung
- Einfache Erweiterbarkeit durch Hinzufügen neuer Komponenten

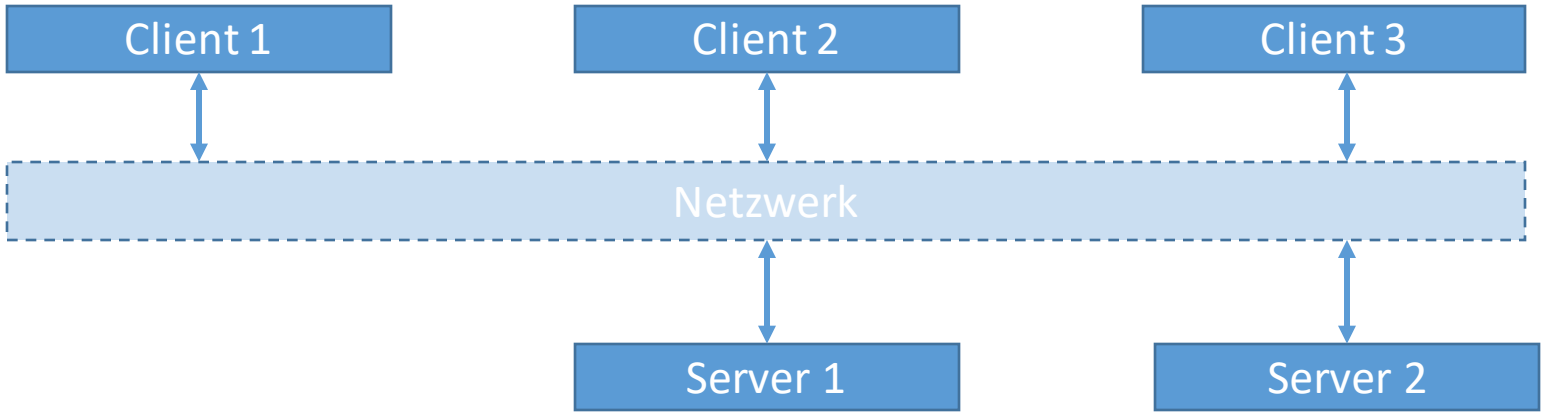
Nachteile

- Kommunikation zwischen Komponenten über die Daten teilweise ineffizient
  Die Anbindung ans Repository wird zum Bottleneck
- Repository ist „**single point of failure**“
  Ausfall der zentralen Komponente bedeutet Totalausfall

Client-Server Architecture

Architekturstil für **verteilte Systeme**

- jede Systemfunktion wird als **Dienst** auf einem zentralen **Server** angeboten
- **Clients** können diese Funktion in Anspruch nehmen

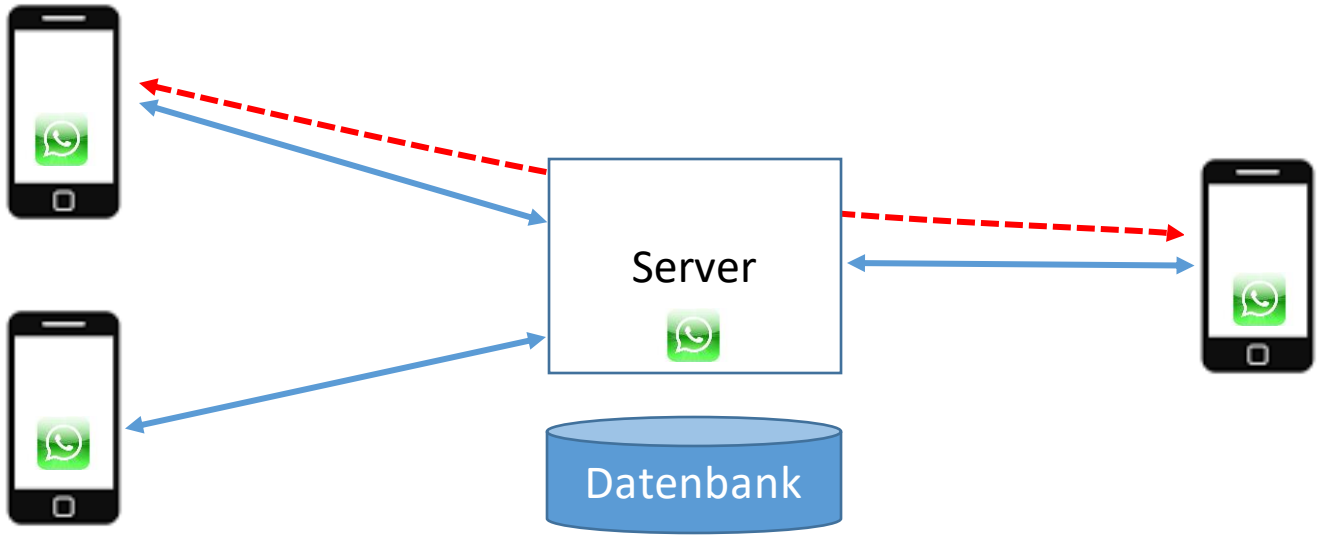


Client-Server Architecture

Keine direkte Kommunikation zwischen Clients

- Interaktion zwischen Clients nur über Server möglich

Instant Messaging



Client-Server Architecture

Vorteile

- Funktionen stehen **zentral zur Verfügung**, müssen nicht mehrfach implementiert werden
- einfache Verwaltung gemeinsam genutzter Ressourcen
- Leistungsschwache Clients können Arbeitslast auf den Sever auslagern

Nachteile

- **Single point of failure**
- zusätzliche Kommunikation
- ungleiche Lastverteilung / schlechte Ressourcennutzung
- **Zentrale Daten** (Privacy)

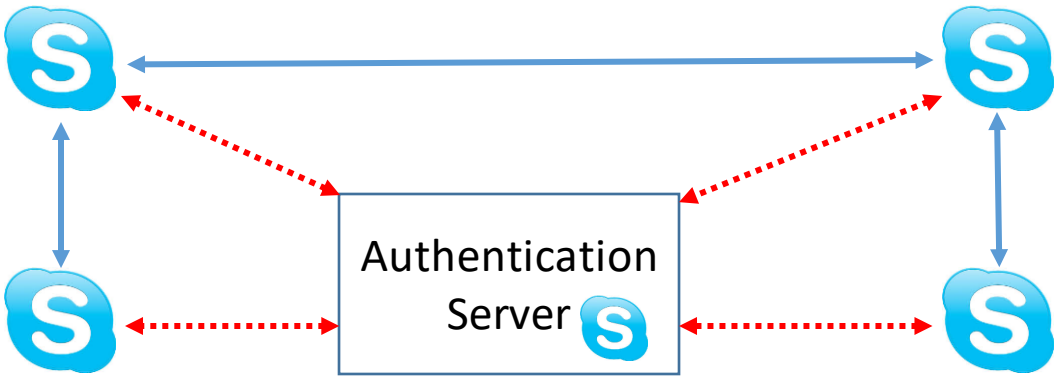
Peer-to-Peer Architektur

Komponenten in verteilten Systemen sind **gleichberechtigt**

- jede kann Funktionen bereitstellen und nutzen
- meist macht ein **zentraler Server** die Teilnehmer bekannt

Skype (in den Anfängen)

- Jeder Client meldet sich beim Server nur zur Authentifizierung und zum Abrufen des Status der Kontakte



Peer-to-Peer Architektur

Vorteile

- **effiziente** Kommunikation (keine Umwege)
- **gleichmäßige** Last/Ressourcenverteilung
- Ausfallsicherheit

Nachteile

- Gemeinsam genutzte Ressourcen **schwierig zu synchronisieren**
- Funktionen **mehrfach implementiert** (Komponenten komplexer)
- **Datenverkehr** nur zwischen einzelnen Teilnehmern

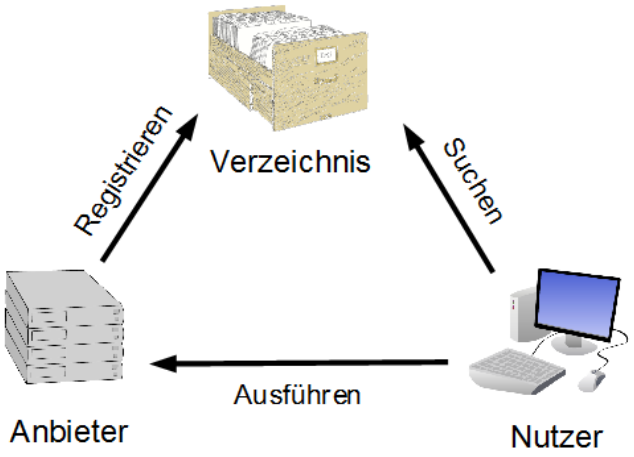
Verteilung kann auch **dezentral** geschehen (cf. Distributed Hash Tables)

Service Oriented Architecture

System besteht aus **verteilten, unabhängigen** Komponenten

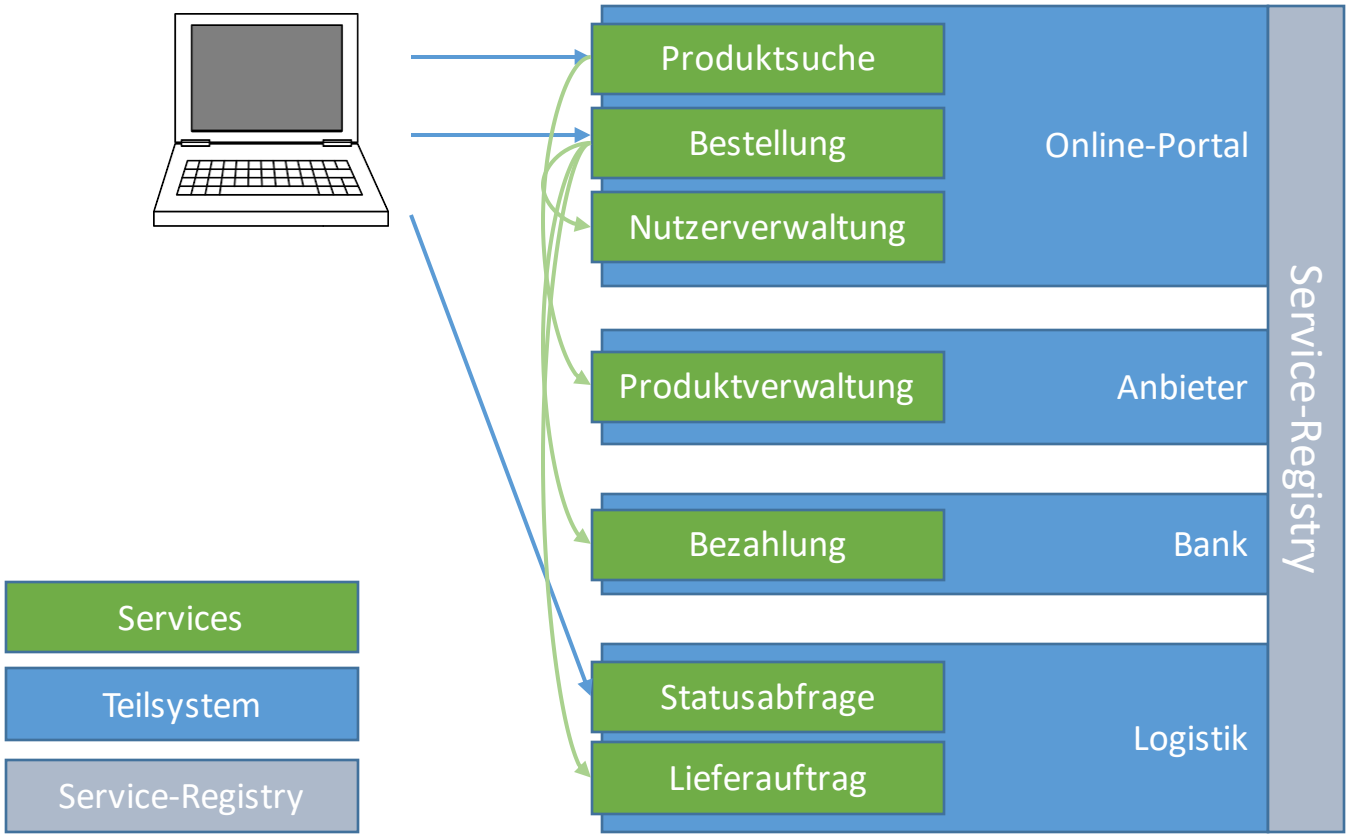
- Komponenten bieten **Funktionalitäten als Services** an
- **Globale Registry** für Service-Anbieter und Services
- Komplexe Komponenten können wieder andere Services verwenden
- **Standard-Protokoll** (z.B. SOAP, REST) für alle Services

Webservices



Service Oriented Architecture

Web-Shop



Service Oriented Architecture

Vorteile

- Services/Funktionalitäten **austauschbar**
- **Erweiterung** durch simple Registrierung neuer Services
- erlaubt **loose coupling** (dynamische Verbindung im Betrieb)
- **einheitliche/standardisierte** Protokolle für Schnittstellen

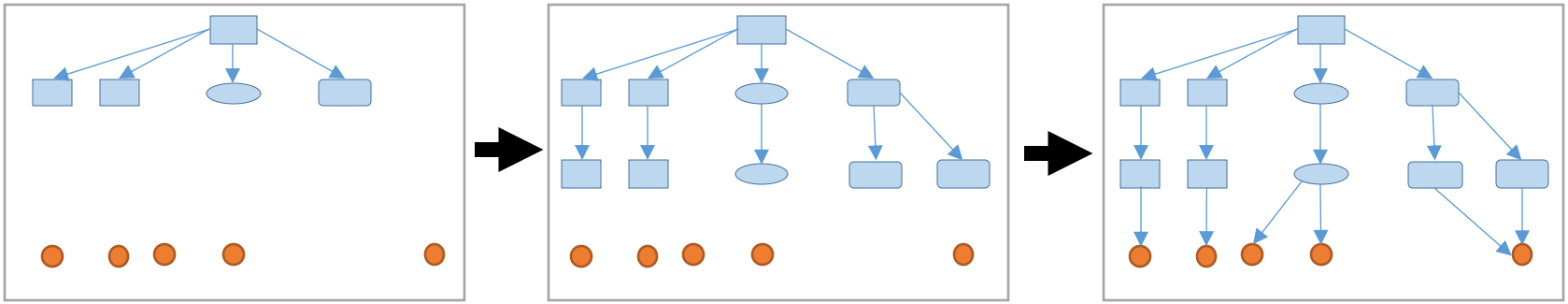
Nachteile

- komplexe technische Infrastruktur
- Registry ist „**single point of failure**“

Softwarearchitektur in der **plangesteuerten** Softwareentwicklung (1)

Vorgehensweise

- Anforderungen (●) stehen zu Beginn der Implementierungsphase fest
- Grundlegende Architektur (■ , ●) kann nach Anforderungsspezifikation entworfen werden
- Top-Down-Implementierung möglich: abstrakte Implementierung der Architektur (Grundgerüst), gefolgt von Details und Features



Softwarearchitektur in der **plangesteuerten** Softwareentwicklung (2)

Vorteile

- Berücksichtigung späterer Anforderungen, wodurch eine strukturierte Entwicklung der Architektur möglich wird
- Parallele Implementierung von Anforderungen innerhalb der Grundstruktur
- Identifikation von Fehlern in den Anforderungen

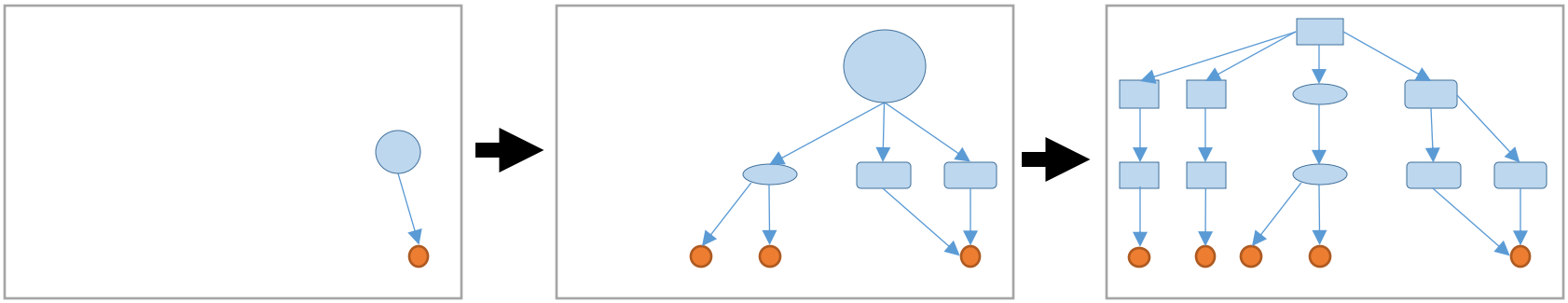
Nachteile

- Fehler in der Architektur aufgrund hoher Komplexität
- Änderungen der Architektur aufgrund von neuen Anforderungen ist teuer
- Geringe Akzeptanz der Architektur durch Entwickler
- Technische Probleme werden erst nach dem Architekturentwurf ersichtlich

Softwarearchitektur in der **agilen** Softwareentwicklung (1)

Vorgehensweise

- Stetige Änderungen der Anforderungen ( )
- Grundlegende Architektur (,) nicht planbar, sondern Ergebnis permanenter Veränderung
- Iterative Weiterentwicklung der Architektur anhand der nächsten Anforderungen
- Prinzip Einfachheit: Aktuelle Implementierung sollte simple sein und somit spätere Änderungen erlauben



Softwarearchitektur in der **agilen** Softwareentwicklung (2)

Vorteile

- Demokratischer Prozess: Architektur orientiert an technischer Umsetzung
- Schnelle, lauffähige Iterationen mit integrierter Implementierung
- Aktualität und zeitnahes Feedback zur Architektur

Probleme

- Mehraufwand durch häufiges Umbauen der Architektur bzw. vollständige Neuimplementierung notwendig
- Ständige Anpassungen bereits fertiger Implementierung (kann auch positiv sein – Refactoring)

***Achtung**: Die vorgestellten Architekturstile sind in der plangesteuerten und agilen Softwareentwicklung **identisch**!*

Zusammenfassung

Architektur spielt für die Implementierung eine **wesentliche Rolle**

- Eine klare Struktur sollte **frühzeitig** gewählt werden
- Verwendung **bewährter Architekturstile** empfohlen
- Funktionales Verhalten möglicherweise mit mehr Architekturstilen abbildbar als nicht-funktionale Anforderungen

Innerhalb von Systemen können **Architekturstile gemischt** auftreten

Einzelne Komponenten verteilter Systeme können wiederum jeweils eine **eigene Architektur** haben

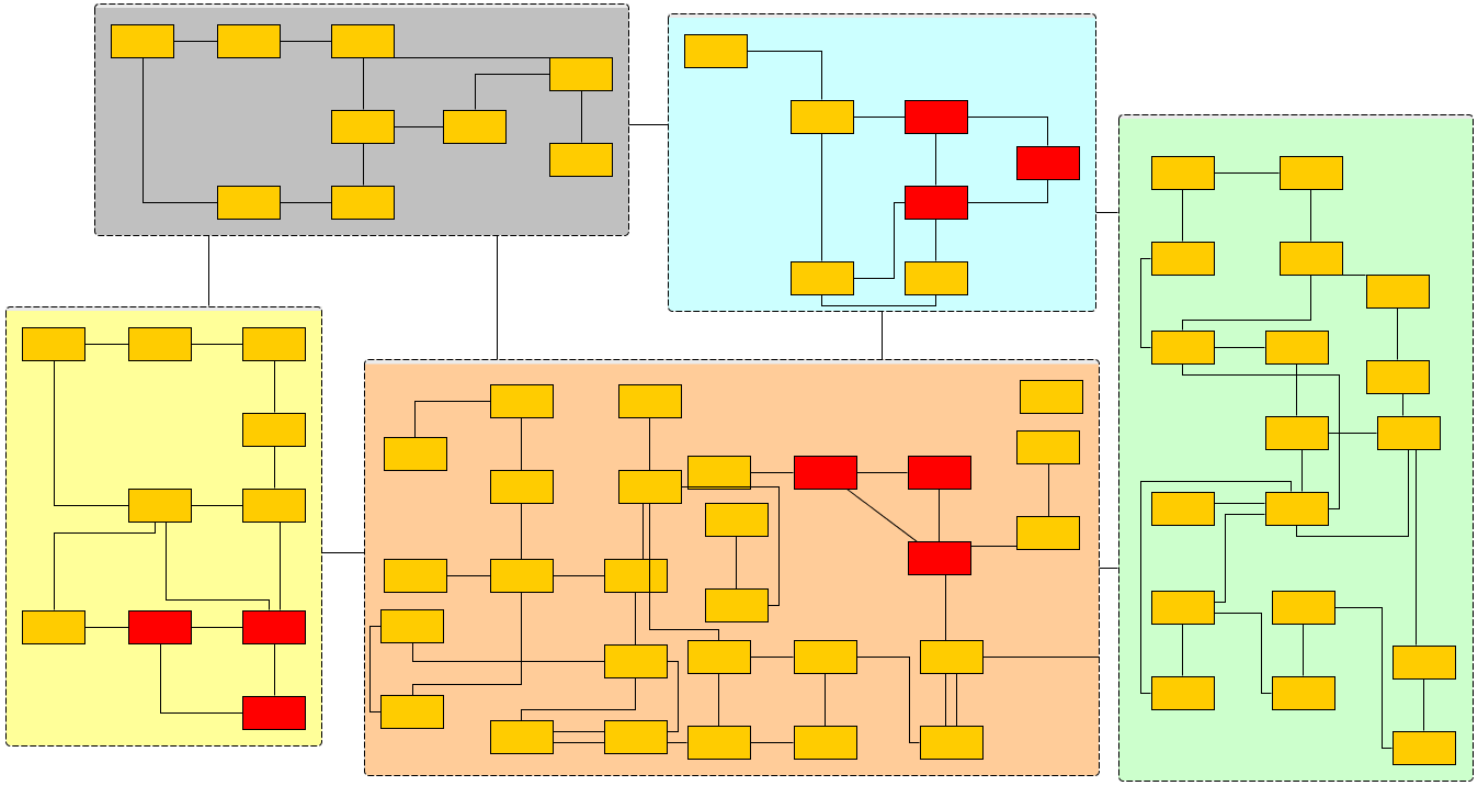
Inhalt

Implementierung

- Einführung
- Architekturstile
- Design Patterns

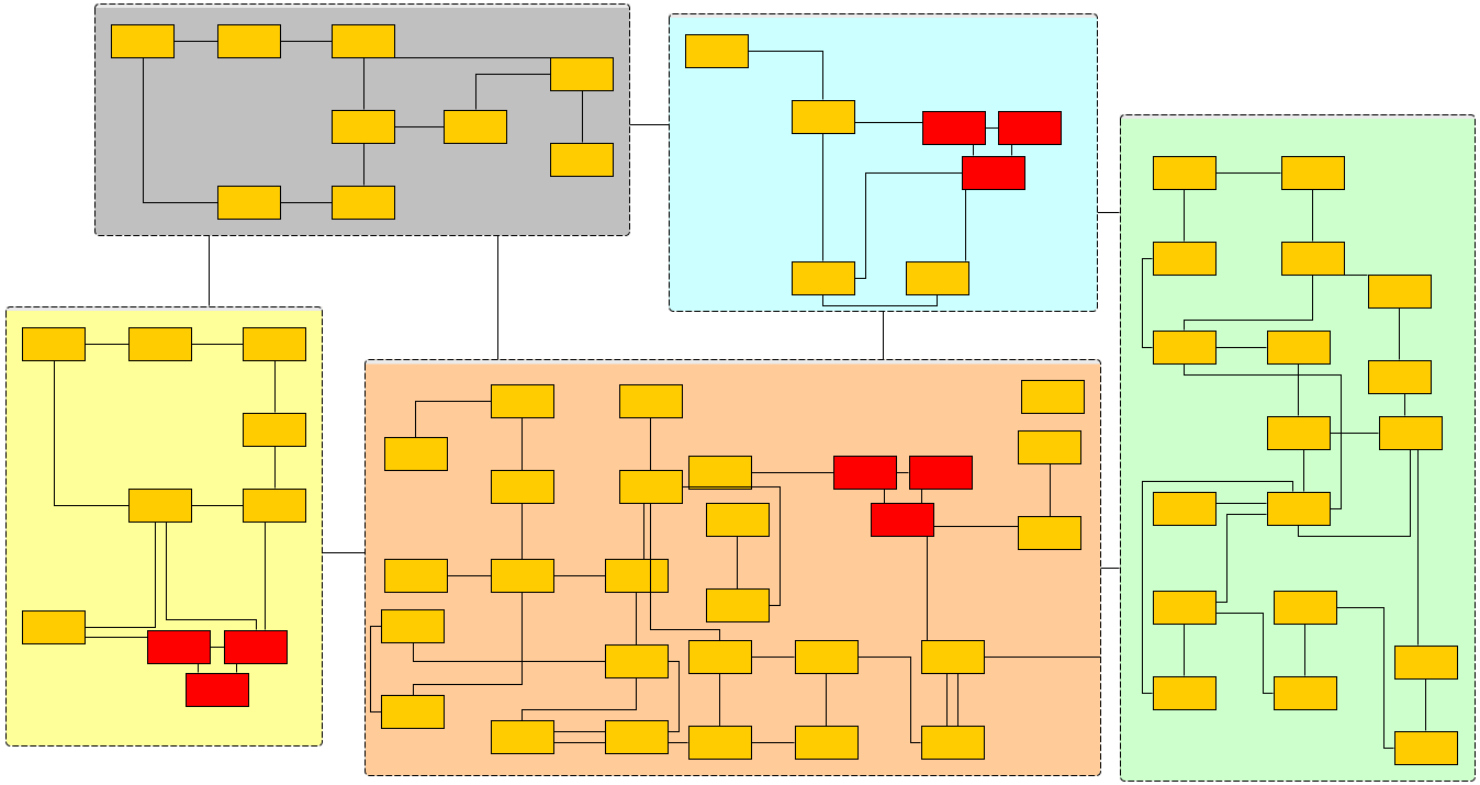
Design Patterns - Motivation

Auf manche Probleme trifft man immer wieder



Design Patterns - Motivation

Reicht es nicht jedes Problem einmal zu lösen?



Design Patterns

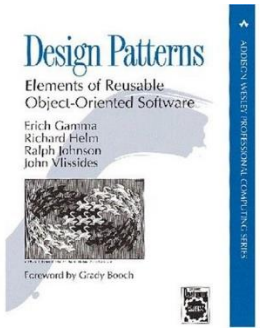
Generische Lösung für **wiederkehrendes Entwurfsproblem**

- Erfahrungen mit **erfolgreichen Lösungsansätzen** übertragbar machen
- Deutsch: Entwurfsmuster
- Überschneidungen mit Architekturstilen möglich

  Ist MVC Architekturstil oder Design Pattern?

1995 Populäre Veröffentlichung einer Sammlung der „Gang of Four“

- Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides. **Design Patterns: Elements of Reusable Object-Oriented Software.**
- Beinhaltet verschiedene Patterns zur Struktur,
- dem Verhalten und dem Erzeugen von Klassen und Objekten



Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software.

Erzeugungsmuster	Strukturmuster	Verhaltensmuster
factory method	bridge	template method
abstract factory	decorator	observer
singleton	façade	visitor
builder	flyweight	iterator
prototype	composite	command
	proxy	memento
		strategy
		mediator
		state
		chain of responsibility

Erzeugungsmuster – Singleton

Singleton

Problem

- für manche Klassen ist **nur eine Instanz** sinnvoll
- Beispiel: Dateisystem
- sicherstellen, dass wirklich nur eine Instanz erzeugt werden kann
- globale/statische Variable für die Instanz reicht nicht

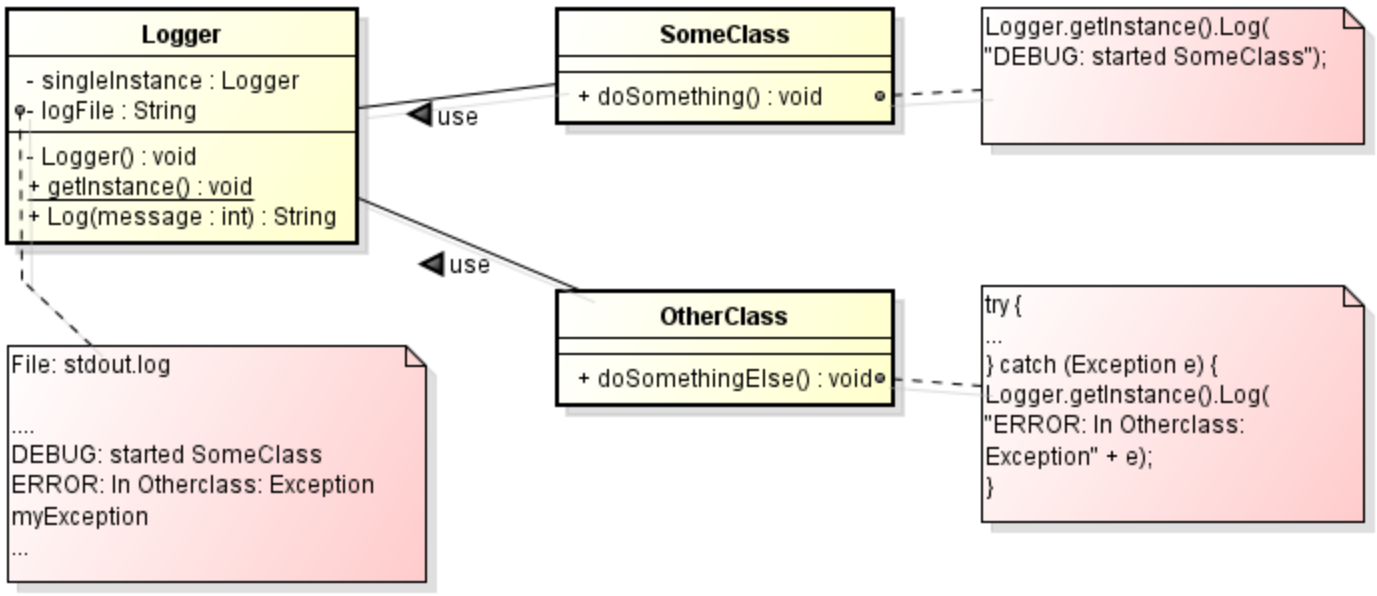
Lösung

- die Klasse **muss selbst dafür sorgen**, dass sie (oder Subklassen) **nur einmal instanziiert** wird

Erzeugungsmuster – Singleton

Logging

Logger schreibt applikationsweit in die gleiche Datei



Erzeugungsmuster – Singleton

Beispielhafte Implementierung in JAVA

- **Privater Konstruktor** nicht von außen erreichbar
- Instanziierung in einer **statischen Methode** versteckt
- erstellt die Instanz **beim ersten Aufruf** (lazy initialization)

```
public class MySingleton {
        // static singleton instance
        private static MySingleton instance = null;

        private MySingleton () { }  // private constructor

        public static synchronized MySingleton getInstance() {
                if (instance == null) {
                        instance = new MySingleton();
                }
                return instance;
        }
}
```

Erzeugungsmuster – Singleton

Alternativ können Singletons in JAVA über **Aufzählungstypen** gelöst werden

- Enums können auch Methoden und Attribute haben
- Werden bei der ersten Verwendung initialisiert

```
public enum Logger {
     INSTANCE;
     private String logFile;

     public void log (String message) {
          //....
     }
     //....
}
//....
Logger.INSTANCE.log("...");
```

Singletons wurden als Object in SCALA übernommen

Erzeugungsmuster – Builder

Builder

Problem

- Viele **optionale Parameter** im Konstruktor schlecht darstellbar
- Die Reihenfolge ist nicht offensichtlich und erschwer die Lesbarkeit
- Benötigt werden **benannte** optionale Parameter mit **Default-Werten**

Lösung

- Konstruktor wird von *Builder* aufgerufen
- *Builder* enthält Initialwerte und
- stellt Funktionen für optionale Parameterübergabe vor dem Aufrufen des Konstruktors bereit

Erzeugungsmuster – Telescoping

Teleskopkonstruktor

- Konstruktor für jede Kombination von Parametern

- Skaliert schlecht
- Schlecht erweiterbar
- Parameter hängt von Reihenfolge ab
- Konstruktoren können nicht verschieden benannt werden

```
public class Cake {
   private final int sugar; // required
   private final int flour; // required
   private final int butter; // optional
   private final int chocolate; // optional

   public Cake(int s, int f, int b, int c) {
      this.sugar = s; this.flour = f;
      this.butter = b; this.chocolate = c;
   }

   public Cake(int s, int f, int b) {
      this(s, f, b, 0);
   }

   public Cake(int sugar, int flour) {
      this(sugar, flour, 0);
   }
}
```

Erzeugungsmuster – Beans

Java Beans

- Objekt-Konstruktor und Setter-Methoden

- Erlaubt ungültige Zwischenzustände des Objekts
- Viel Schreibarbeit beim Erzeugen

```
public class Cake {
   private int sugar = -1; // required
   private int flour = -1; // required
   private int butter = 0; // optional
   private int chocolate = 0; // optional

   public Cake() {} // standard constructor

   public void setSugar(int s)
   {this.sugar = s;}
   public void setFlour(int f)
   {this.flour = f;}
   public void setButter(int b)
   {this.butter = b;}
   public void setChocolate(int c)
   {this.chocolate = c;}

   // ......
}
```

Erzeugungsmuster – Builder

Konstruktor ist Privat – Objekt Cake kann nur von Funktionen innerhalb der Klasse erstellt werden

Builder liefert alle Eingaben für die Attribute (auch optionale)

Builder ist lokale Klasse (inner class) – dort ist der private Konstruktor sichtbar

```
public class Cake {
   private final int sugar; // required
   private final int flour; // required
   private final int butter; // optional
   private final int chocolate; //optional

   private Cake(Builder b) { // private!
      // get parameters from builder
      this.sugar = b.sugar;
      this.flour = b.flour;
      this.butter = b.butter;
      this.chocolate = b.chocolate;
   }

   static class Builder {
      // let the builder do the job;
      // see next slide
   }
    // ......
}
```

Erzeugungsmuster – Builder

Optionale Parameter haben einen Default-Wert

Notwendige Parameter werden dem Builder-Konstruktor übergeben

Für die optionalen Parameter gibt es jeweils eine eigene „setter“-Funktion. `return this` ermöglicht, dass die Funktionen kompakter hintereinander aufgerufen werden können

Wenn alles fertig ist, erstellt der Builder das Objekt mit allen verfügbaren Eingaben

```
static class Builder {
   private final int sugar;
   private final int flour;
   private int butter = 0;
   private int chocolate = 0;

   public Builder (int s, int f)
   {
       this.sugar = s;
       this.flour = f;
   }

   public Builder butter(int val) {
      this.butter = val; return this; }
   public Builder chocolate(int val) {
      this.chocolate = val; return this; }

   // build all at once
   public Cake build()  {
      return new Cake(this);
   } // build
}
```

Builder

Erstellung von *Cake* komfortabel mit

```
Cake dry   = new Cake.Builder(500, 500).build();
Cake yummy = new Cake.Builder(500, 500).
             butter(250).chocolate(200).build();
```

- optionale Parameter in **benannten** Funktionen
- **Default-Werte** für nicht gesetzte Parameter
- Keine inkonsistenten Zwischenzustände
- Reihenfolge der optionalen Parameter irrelevant

PYTHON bietet diese Funktionalität standardmäßig an

Strukturmuster – Composite

Kompositum (composite)

Problem

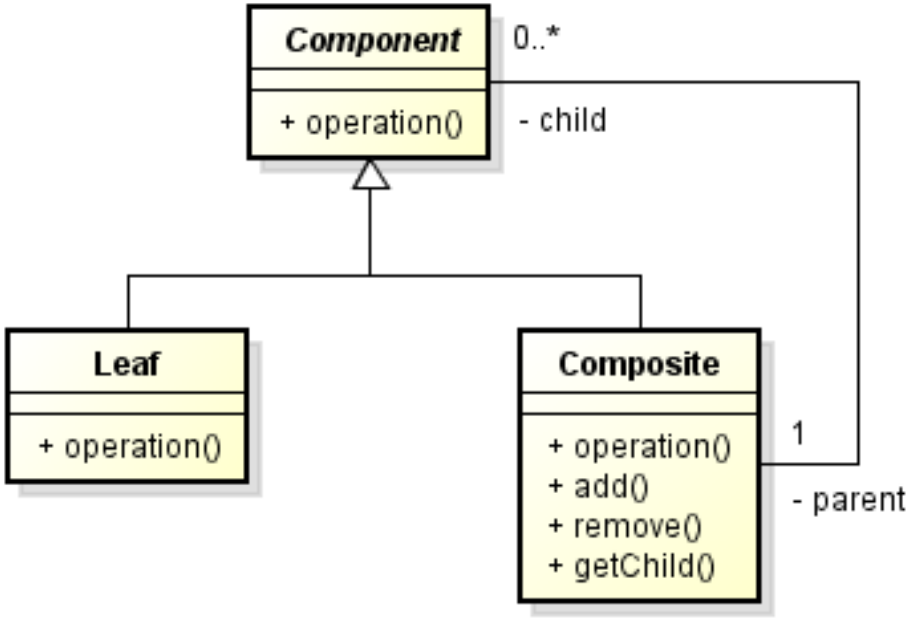
- Daten sind **hierarchisch** organisiert (baumförmig)
- Programm führt auf allen Knoten **gleichartige Operation** aus (Baum-Traversierung)

Lösung

- **Composite definiert Hierarchien**, die aus komplexen Objekten (composites) und einfachen Objekten bestehen
- für das Programm **transparent**, was für ein Objekt behandelt wird (gemeinsame abstrakte Operation für alle Knoten)

Strukturmuster – Composite

Struktur

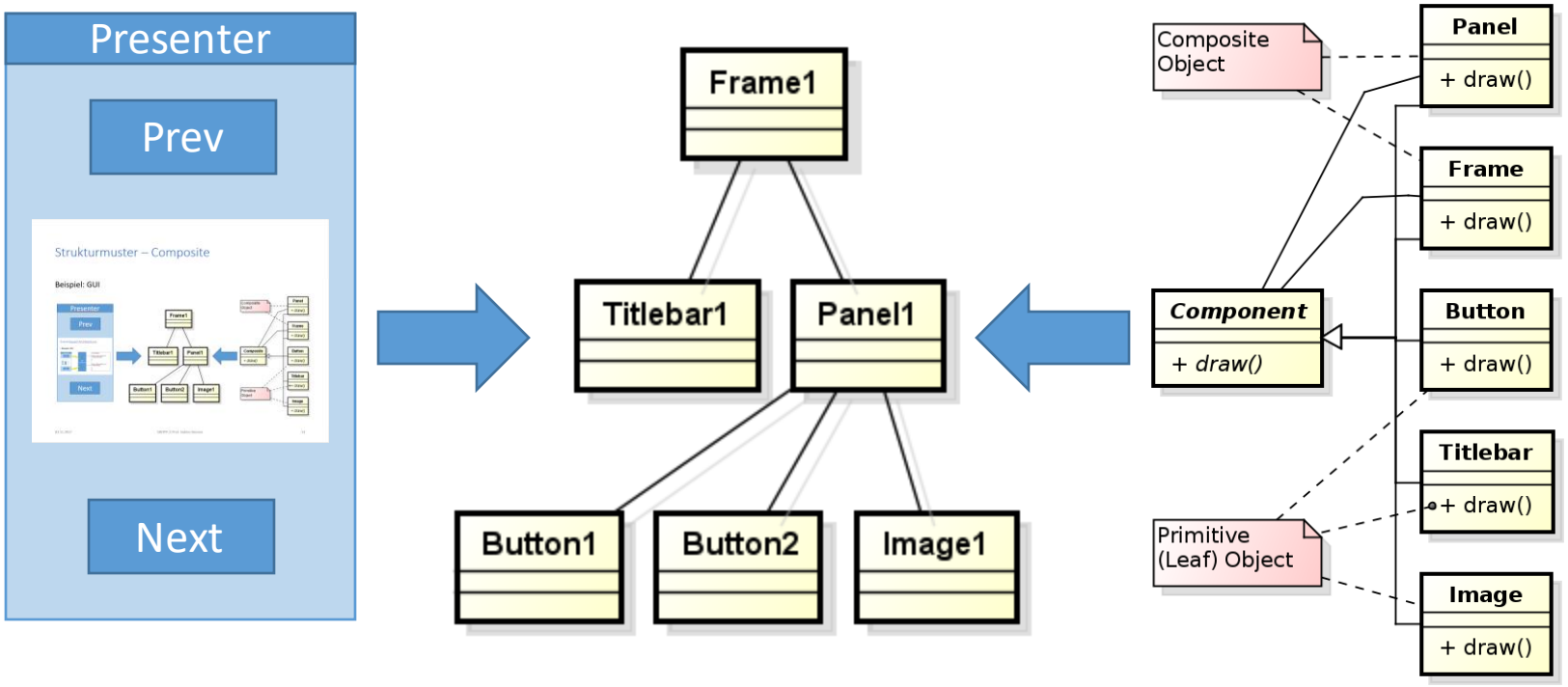


Vorteile

- vereinfacht den Client-Code
- neue Komponenten können leicht hinzugefügt werden

Strukturmuster – Composite

Beispiel: GUI



Strukturmuster – Proxy

Stellvertreter (proxy)

Problem

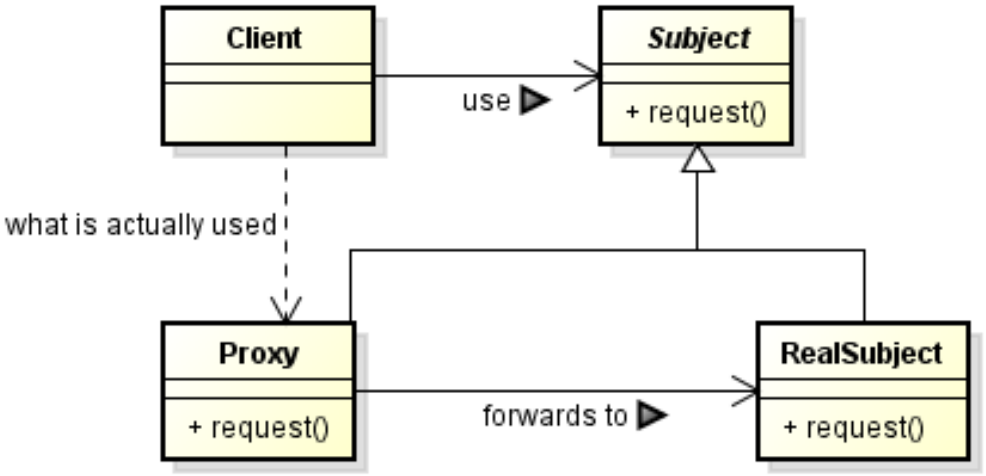
- Ein Zugriff/Verbindung zu einem Objekt kann durch einen Pointer nicht ausreichend dargestellt werden
- Zugriffsoperationen sind komplexer oder nur Teilmengen der Zugriffsmöglichkeiten sollen erlaubt sein

Lösung

- Proxy-Klasse ersetzt die tatsächliche Klasse an der Stelle der Verwendung
- Kapselt Zugriffe und implementiert zusätzliche Funktionalität

Strukturmuster – Proxy

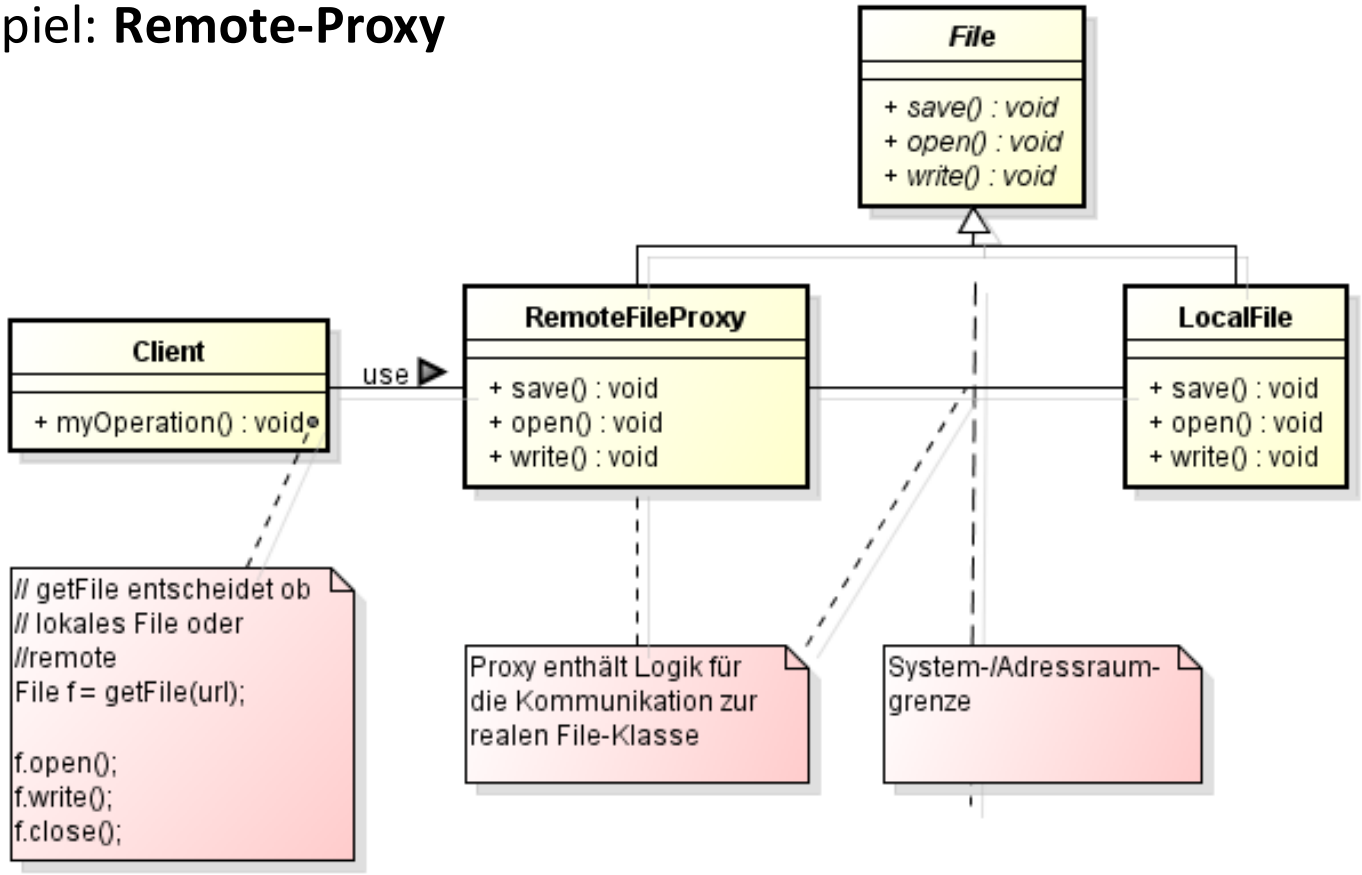
Struktur



- Proxy und „echte“ Klasse erben von abstrakten Typ
- Proxy reicht Abfragen weiter und fügt eigene Funktionalität hinzu

Strukturmuster – Proxy

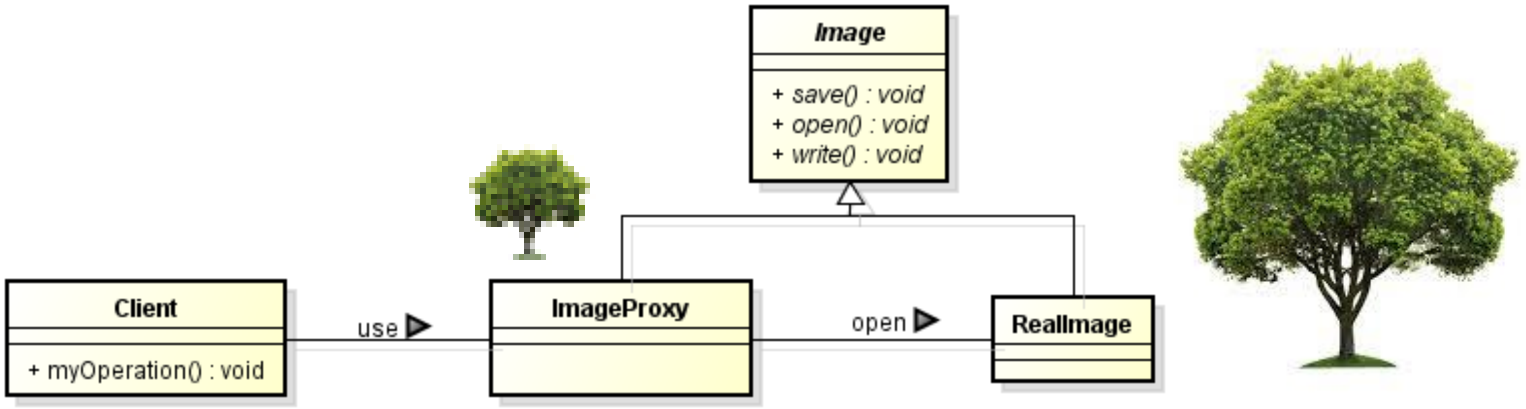
Beispiel: **Remote-Proxy**



Strukturmuster – Proxy

Beispiel: **Virtual Proxy**

- ganzes Bild laden ist aufwendig
- Proxy stellt Thumbnail zur Verfügung
- lädt echtes Bild nur, wenn nötig



Strukturmuster – Proxy

Anwendungsbeispiele

- **Remote Proxy**: Lokale Repräsentation eines Remote-Objekts (andere Bezeichnung: Botschafter)
- **Virtual Proxy**: Die Erzeugung des Objektes ist aufwendig, aber nicht immer notwendig. Proxy erstellt das Objekt erst bei Bedarf
- **Protection Proxy**: Bietet eingeschränkten Zugriff auf Objekte, die größeren Schutz benötigen
- **Smart Reference**: Führt zusätzliche Aktionen beim Zugriff aus (z.B. Zugriffszähler)

Kann in PYTHON auch für Properties eingesetzt werden

Verhaltensmuster – Observer

Beobachter (observer)

Problem:

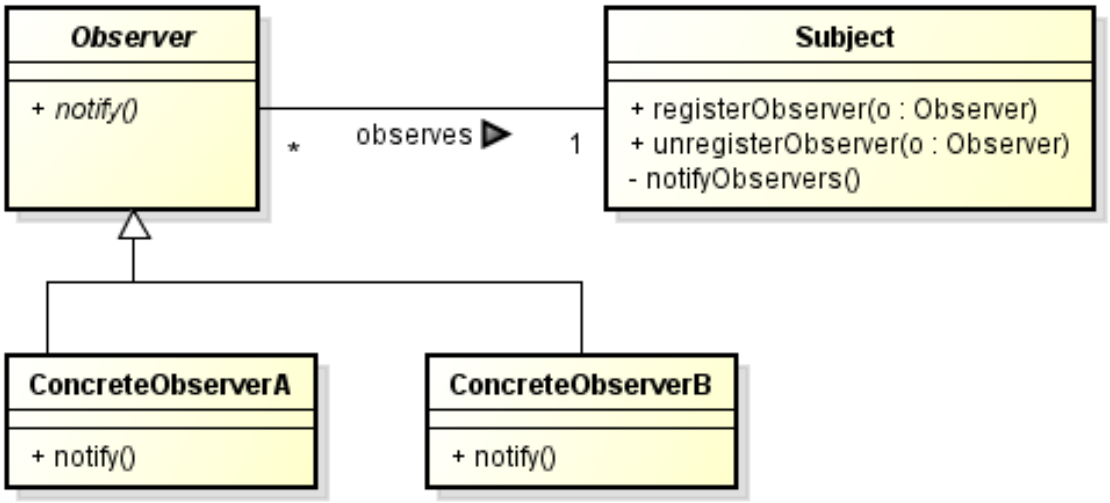
- Mehrere Objekte sollen **unmittelbar informiert** werden, wenn sich eines ändert (Beobachtung)
- Das beobachtete Objekt (Subjekt) kann nicht vorhersehen, welche Beobachter es gibt
- Beobachter können wechseln

Lösung:

- Subjekt stellt Möglichkeit bereit, sich **anzumelden** (publish)
- Beobachter **melden sich beim Subjekt an** (subscribe/register)
- Subjekt **aktualisiert** angemeldete Beobachter bei Änderung (notify/update)

Verhaltensmuster – Observer

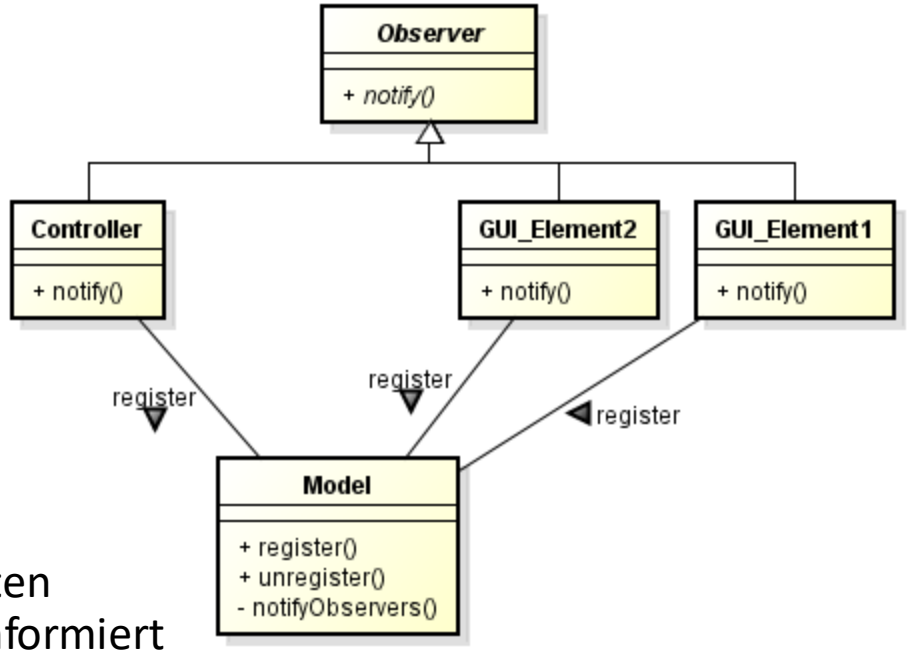
Struktur:



- Die Beobachter erweitern die abstrakte Klasse Observer und werden beim Subjekt registriert
- Das Subjekt führt die Aktualisierung in allen Beobachtern mit „notify“ durch

Verhaltensmuster – Observer

Beispiel: GUI mit MVC



- Model ist hier Subjekt
- Alle GUI-Elemente mit Inhalten aus dem Model sind Observer
- Auch der Controller kann Observer sein
- Grund: View und Controller sollten unmittelbar über Änderungen informiert werden
- Für Model nicht klar, welche GUI-Elemente es gibt

Verhaltensmuster – Command

Kommando (command)

Problem:

- Eine Anweisung soll nicht nur ausgeführt, sondern auch verwaltet werden
- Beispiele:
  - Verzögerung einer Ausführung durch Warteschlangen
  - Parametrierung von Objekten (Clients) mit Anforderungen
  - Aufzeichnung von Anforderungen

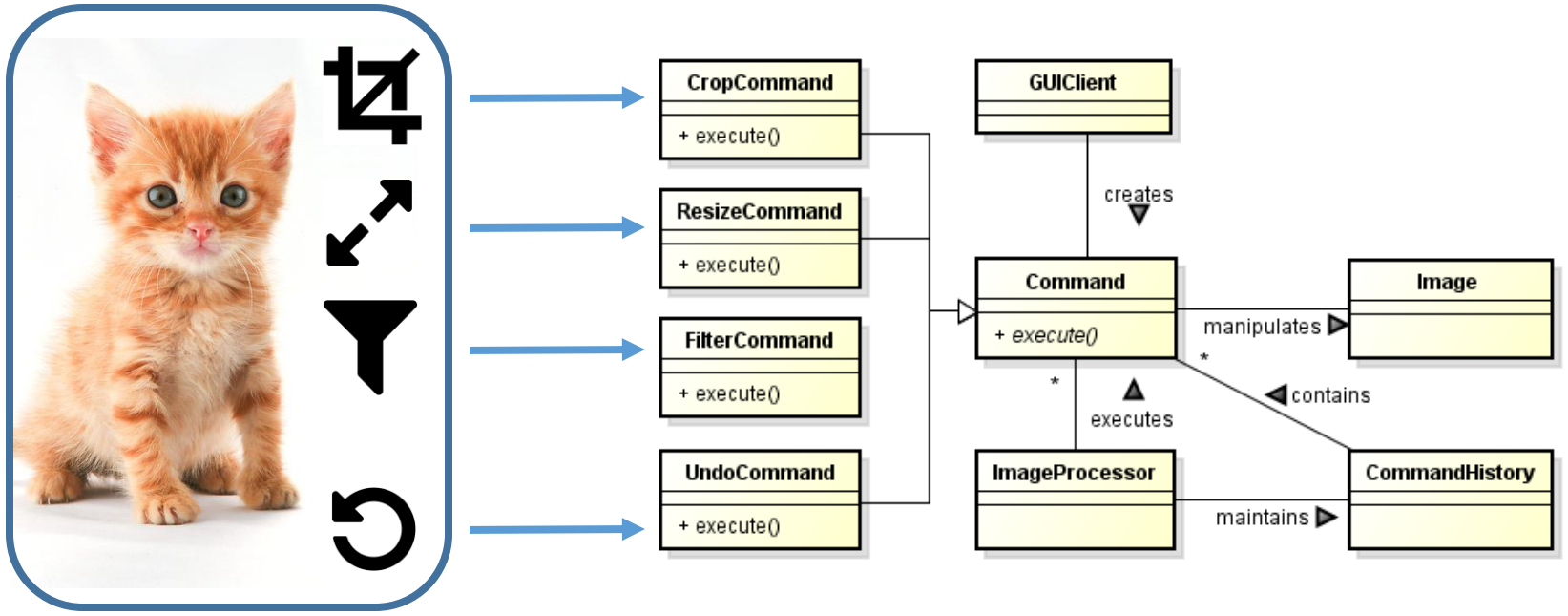
Lösung:

- Command: Interface für die Ausführung von Operationen
- Client erstellt Command statt eine Operation direkt zu starten
- Tatsächliche Ausführung der Funktionalität verzögert bzw. indirekt

Verhaltensmuster – Command

Beispiel: Bildbearbeitung

- Kommandos werden im GUI-Framework für Buttons konfiguriert
- Aufzeichnung aller Kommandos für Rückgängig-Funktion



Zusammenfassung

Implementierung stellt eigene Anforderungen an das Modell

- Durch Vielfalt der Probleme/Lösungsmöglichkeiten sind Vorgaben an Implementierung schwierig

„Gute Erfahrungen“ in Mustern beschrieben

- Muster für Gesamtstruktur: Architekturstile
- Muster für wiederkehrende Probleme: Design Patterns

Lernziele

- ❑ Wofür gibt es Architekturmuster?
- ❑ Wie unterscheiden sich die vorgestellten Architekturmuster?
- ❑ Welche Vor- und Nachteile haben sie jeweils?
- ❑ Für was für Systeme eignen sie sich?
- ❑ Was ist ein Design Pattern?
- ❑ Wofür wurden die vorgeschlagen?
- ❑ Wie kann man sicherstellen, dass es von einer Klasse nur eine Instanz gibt?
- ❑ Wie unterscheiden sich das Builder Pattern von den vorgestellten Alternativen?
- ❑ Wie funktioniert das Composite-Pattern?
- ❑ Für welche Zwecke kann man das Proxy-Pattern gebrauchen?
- ❑ Wie ist der Zusammenhang zwischen Observer-Pattern und MVC?
- ❑ Wie kann mit dem Command-Pattern der Aufruf von der Ausführung einer Funktion entkoppelt werden?

Quellen

- Ian Somerville: Software Engineering, 9. Auflage
- Heide Balzert: Lehrbuch der Objektmodellierung, 2. Auflage
- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns
- Joshua Bloch: Effective Java (2nd Edition)

Vorstellung des Fachgebiets **Software and Embedded Systems Engineering**

Technische Universität Berlin

SESE

Inhalt

- Aktuelle Forschung bei SESE
- Weitere Kurse bei SESE
- Aktuelle Stellenangebote

Inhalt

- Aktuelle Forschung bei SESE
- Weitere Kurse bei SESE
- Aktuelle Stellenangebote

Anforderungsspezifikation für RL

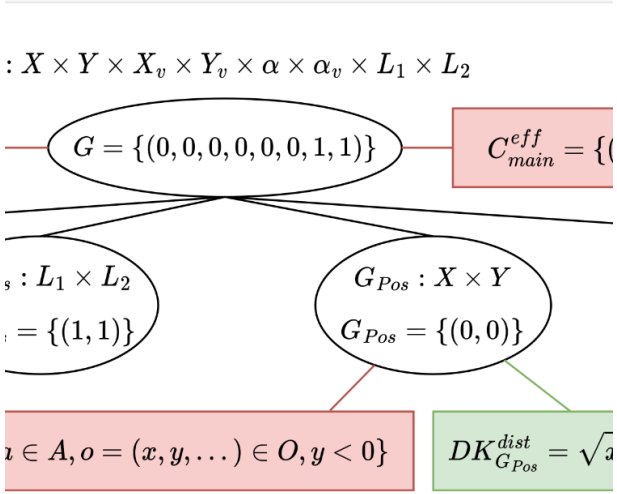
Ziel: Entwicklung einer Spezifikationssprache für Reinforcement Learning (RL) zur Konstruktion von RL-Agenten

Kontakt: Simon Schwan

Hohe technische Einstiegsbarrieren im Entwurf von RL-Agenten basierend auf modernen Algorithmen

- Komplexe Anforderungen in skalarer Belohnungsfunktion
- Viele Parameter / Entscheidungen im Entwurf

→ Garantien insbesondere im Kontext von CPS schwierig



Ansätze: Formalisierte Zielspezifikation nutzen, um von technischen Details zu abstrahieren.

Real-time Scheduling für CPS

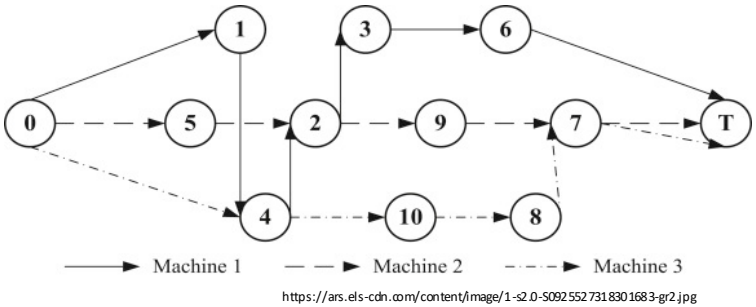
Ziel: Lösen von Scheduling Problemen mit diversen Nebenbedingungen in verteilten zeitkritischen Systemen, um Modularität in CPS zu ermöglichen…

Kontakt: Milko Monecke

Aktuelle Scheduling-Algorithmen sind häufig **Heuristiken ohne Garantien** oder generische **Suchverfahren** mit **hoher Zeitkomplexität**.

- Berücksichtigen spezielle Eigenschaften bestimmter Use Cases nicht

→ nicht optimale oder wenig spezialisierte Scheduling-Algorithmen!



https://ars.els-cdn.com/content/image/1-s2.0-S0925527318301683-gr2.jpg

Ansätze: genaue Klassifizierung von Scheduling-Problemen, Evaluation bestehender Algorithmen, Entwickeln spezialisierter Algorithmen (z.B. Nutzung von problemspezifischen Vorwissen in Suchprozessen, …)

Aktives Automatenlernen für CPS

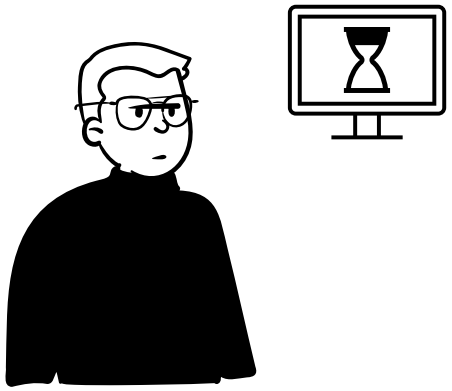
Ziel: Automatenmodell eines unbekannten Zielsystems erzeugen. Zur Verifikation, Verständnis, …

Kontakt: Paul Kogel, Wolffhardt Schwabe

Aktuelle Verfahren zum aktiven Automatenlernen nur **eingeschränkt** auf CPS anwendbar.

- Sehr große Eingabealphabete
- Zeitabhängiges Verhalten

→ Lernen dauert extrem lange!



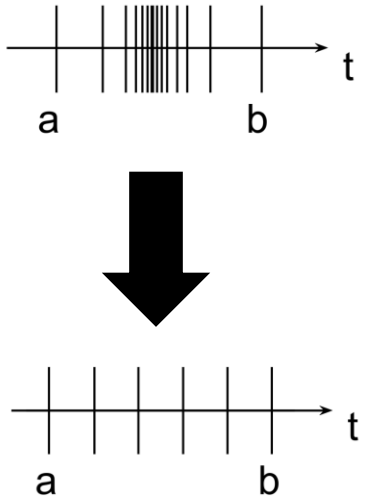
Ansätze: (ungenaues) Vorwissen nutzen, Filter + Abstraktion, spezialisierte Modelle

Opacity für Timed Automata

Ziel: Opacity verifizieren.
Das bedeutet: Systeme davon abhalten, geheime Informationen an die Außenwelt zu verraten.

Kontakt: Julian Klein

- Geheimnisse sind oft aus **Zeitverhalten** ableitbar (*Timed Opacity*)
- Die Verifikationen von Timed Opacity ist **nicht entscheidbar** (Beschränkung erforderlich)



Aktuelle Methoden für Verifikation **beschränken** daher das **Sicherheitslevel** oder die **Klasse** der betrachteten Systeme.

Ansätze:

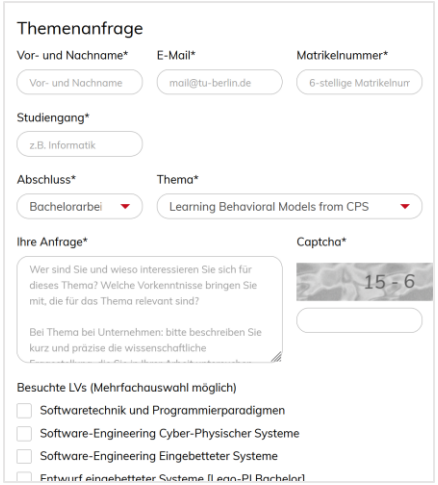
- Beschränkung des **Zeitmodells** statt Sicherheitslevel oder Systemklasse
- **Zeitabstraktion** für effizientere Berechnungen

Abschlussarbeiten

Begeistert von unseren Themen?

Dann bewerbt euch jetzt für eine Abschlussarbeit!

https://www.tu.berlin/sese/studium-lehre/abschlussarbeiten



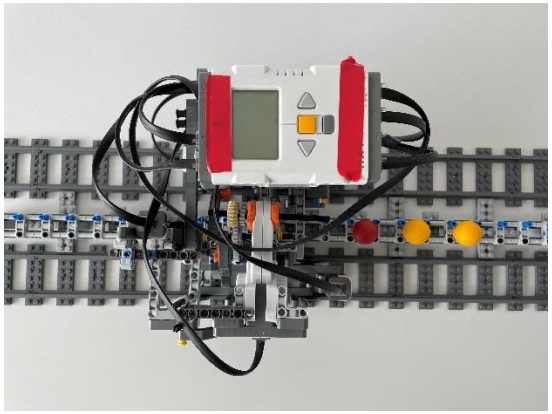
Inhalt

- Aktuelle Forschung bei SESE
- Weitere Kurse bei SESE
- Aktuelle Stellenangebote

Kurse im Sommersemester 2024

Lego-Projekt:

- Entwurf Eingebetteter Systeme (Bachelor)
- Master Project (+ Seminar) Software Engineering of Embedded Systems



Alle vorherigen Projekte
tu.berlin/sese/studium-lehre/studierendenprojeke

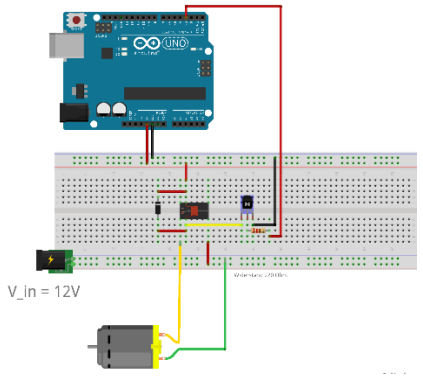
- Projekt mit 9 LP
- Arbeit in großen Teams
- Komplexe Aufgaben zu aktuellen Themen
- Sehr großer Andrang (Losverfahren)!

ISIS-Kurs mit mehr Infos zu Beginn des nächsten Semesters.

Kurse im Sommersemester 2024

Programmierpraktikum *Cyber-Physical Systems* (PCPS)

- Praktikum mit 6 LP
- Bevorzugt für Bachelor Informatik (Wahlpflicht)
- Nur für Personen, die noch kein Programmierpraktikum absolviert haben
- Zentrale Vergabe über Meta-Kurs auf ISIS

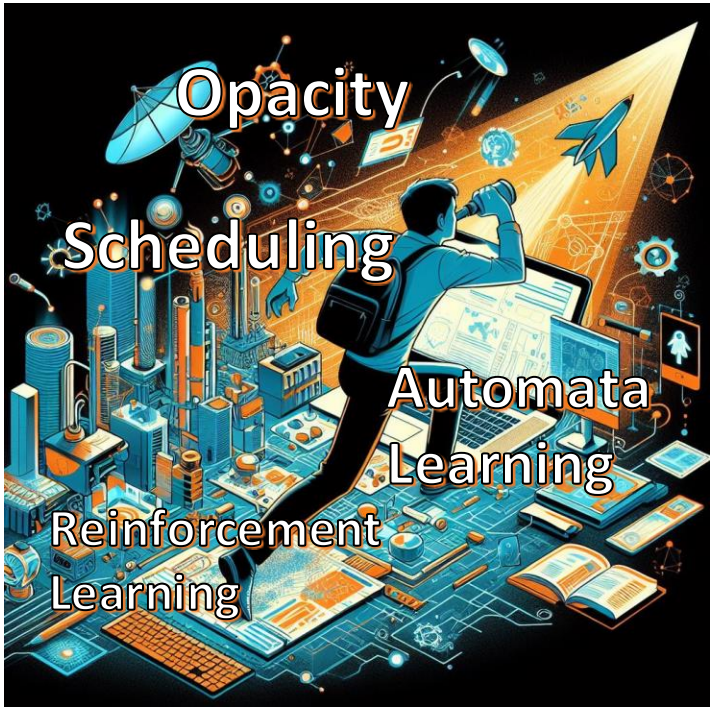


- Selbstorganisation und Arbeit in Teams
- Hardwarenahe Programmierung
- Spannende Use-Cases

Kurse im Sommersemester 2024

Aktuelle Themen zu *Software and Embedded Systems Engineering*

- Bachelor-Seminar mit 3 LP
- Auseinandersetzung mit aktuellen Arbeiten aus unseren Forschungsgebieten
- Schriftliche Ausarbeitung, Reviews + Präsentation



Inhalt

- Aktuelle Forschung bei SESE
- Weitere Kurse bei SESE
- Aktuelle Stellenangebote

Arbeit als Tutor*in

Wir (und die Fakultät) brauchen Tutor*innen
zur Unterstützung der Lehre

Du …

… **erklärst gerne** und willst mal üben das vor mehr Leuten zu tun?

… findest **unseren Stoff interessant** und verstehst ihn auch?

… magst **Geld**?

Dann bewirb dich bei der Fakultät IV und wähle SWTPP oder SEES/SECPS!

… für das WiSe 24/25 am besten während des Sommersemesters

Bei Fragen kannst du dich gerne an uns wenden!

SHK in der Forschung

Komplexitätstheorie und Formale Methoden mit praktischer Anwendung sind dein Ding?

Dann komme jetzt als **SHK** ins SESE Team und revolutioniere gemeinsam mit uns die Forschung in den Bereichen **Automata Learning** und **Real-time Scheduling**.
40h/Monat, flexible Arbeitszeiten

Fragen direkt an milko.monecke@tu-berlin.de

SHK im Forschungsprojekt ZoLA

Hast du Lust auf Robotik und Reinforcement Learning?

Dann komme jetzt als **SHK** ins Forschungsprojekt *Ziel-orientiertes Lernen von Agenten (ZoLA)*

und unterstütze uns bei Forschung, Implementierung und Experimenten.

40,60,80h/Monat, flexible Arbeitszeiten, Arbeit im Team, Projektbeginn März 2024

Fragen direkt an Simon unter s.schwan@tu-berlin.de