Software Engineering II

10 Architekturmuster

SS 2020

Prof. Dr. Dirk Müller

Übersicht

- Einführung
- Begriff
- Arten
- Schichten, *Pipe-and-Filter*, Datenhaltung, Client-Server
- *Model-View-Controller*
  - *Web*-Architektur
  - wichtige angewandte Entwurfsmuster
  - Anwendung in *Android*
- *Dependency Injection*
  - Ziele und Einordnung
  - Beispiel in *Java*
- Zusammenfassung

Begriff

- „Architekturmuster [..] beschreiben Systemstrukturen, die die Gesamtarchitektur eines Systems festlegen. Sie spezifizieren, wie Subsysteme zusammenarbeiten.“ [2], S. 37
  - beschreiben grundlegende Organisation und Interaktion zwischen den Komponenten einer Anwendung
- feinere Granularität: „Entwurfsmuster geben bewährte generische Lösungen für häufig wiederkehrende Entwurfsprobleme an, die in bestimmten Situationen auftreten. Sie legen die Struktur von Subsystemen fest.“ [2], S. 37
- abstrakte Beschreibung eines bewährten Ansatzes
  - in verschiedenen Umgebungen angewendet und getestet
  - Empfehlungen, wann es angemessen ist, das Muster zu verwenden und wann nicht
  - Stärken und Schwächen

Arten von Architekturmustern

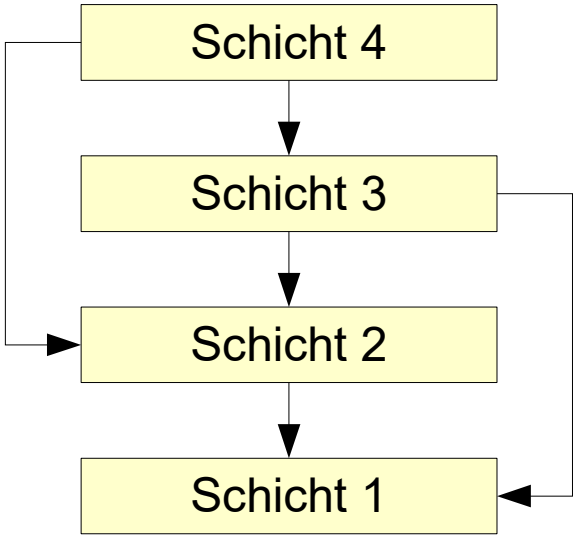
- allgemeine (*Mud-to-Structure*)
  - Schichtenarchitektur
  - *Pipe-and-Filter*-Architektur
  - Datenhaltung (*Repository*)
- für verteilte Systeme
  - Client-Server-Architektur
- für interaktive Systeme
  - *Model-View-Controller*-Muster (zugleich Entwurfsmuster [3])
- für Echtzeitsysteme
  - Monitorstruktur
- für adaptive Systeme
  - Mikrokernel
  - *Dependency Injection*

Schichtenarchitektur

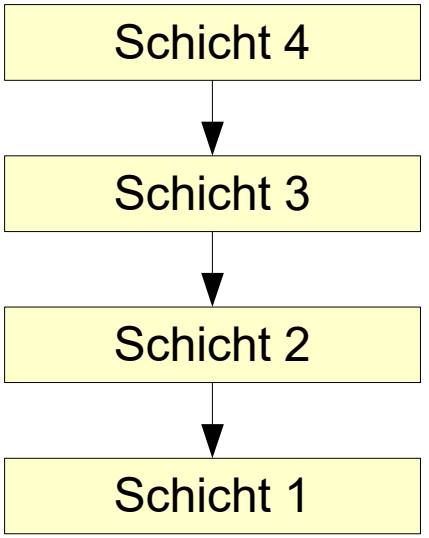
- Subsysteme werden horizontalen Schichten zugeordnet
  - Jede Schicht stellt nur den höheren Schichten Dienste bereit.
  - innerhalb einer Schicht beliebige Zugriffe (keine Regeln)
  - innere Struktur einer Schicht nach außen unsichtbar
- Ziele
  - engere Bindung und losere Kopplung, stabile Schnittstellen => bessere Änderbarkeit und physikalische Verteilbarkeit
  - Portabilität verbessert (meist nur unterste Schicht nötig)
  - Sicherheit wird gut unterstützt (Überwachungsschichten)
- Granularität: richtiges Maß
  - zu wenige: Ziele nicht erreicht
  - zu viele: zu viel Overhead
- Fehlerbehandlung
  - selbst behandeln oder an höhere Schicht durchreichen

Quelle: [2], S. 46 ff.

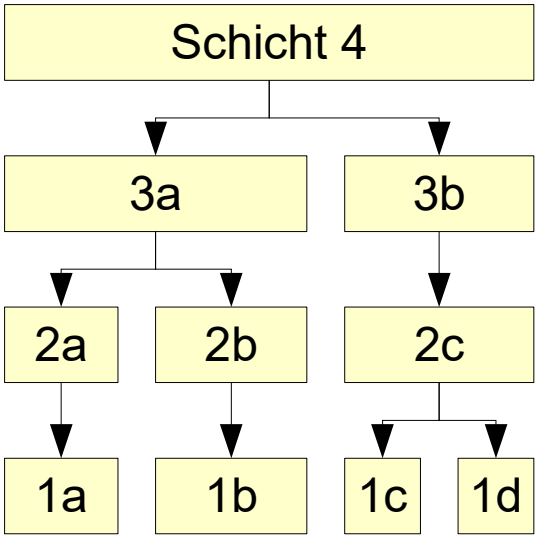
Schichtenarchitektur: Strukturvarianten



allgemein, nicht strikt



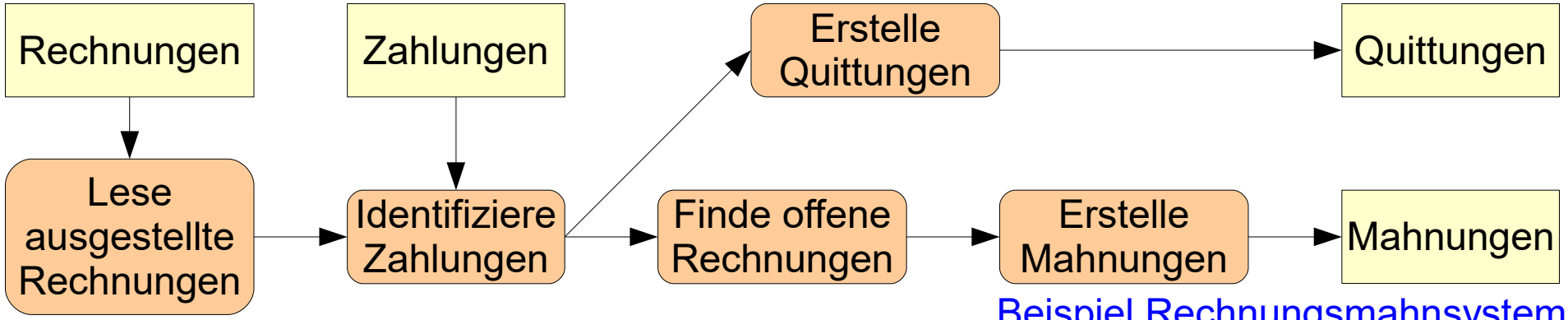
strikt



baumartig

- strikt: Schichten dürfen nicht übersprungen werden
  - nur n-1 Schnittstellen, beste Änderbarkeit
- nicht strikt: Schichten werden übersprungen
  - extra Schnittstellen, schlechtere Änderbarkeit
  - flexibler, effizienter
- baumartig:
  - kein Austausch zwischen Schichten gleicher Knotenebene

Pipe-and-Filter-Architektur



Beispiel Rechnungsmahnsystem

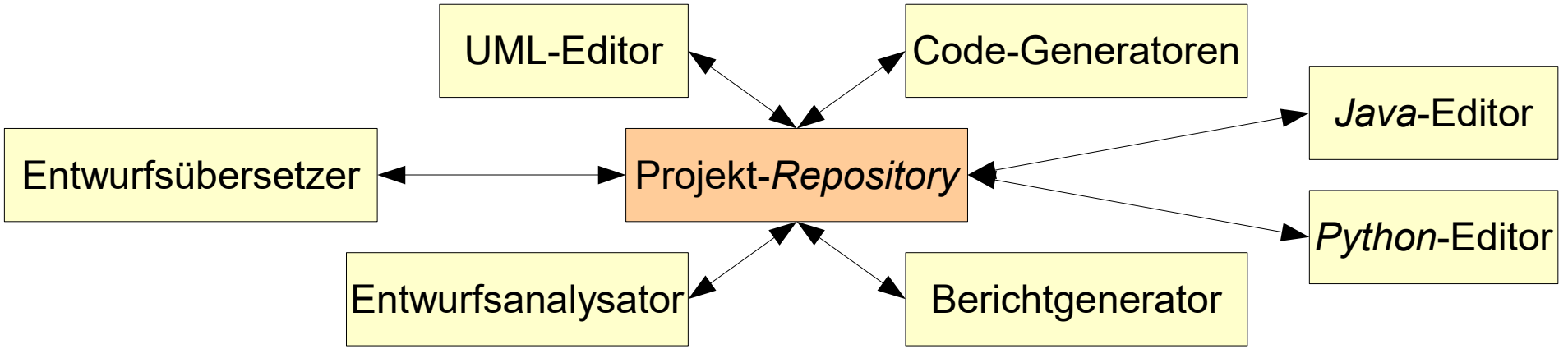
- Funktionsblöcke transformieren (sequenziell oder parallel) Datenstrom schrittweise von Datenquellen zu Datensenken
  - *Pipes* synchronisieren Filter
- *UNIX*/*Linux*-Kommandozeile: z. B. Rechtschreibprüfung

```
tr 'A-Z' 'a-z' <myText.txt | tr -cs 'a-z' '\n' | sort | uniq | comm -23 - /usr/share/dict/words
```

- Änderbarkeit gut unterstützt
- schwierig umzusetzen für interaktive Systeme mit GUIs
  - ereignisgesteuert, komplexe I/O-Formate

Quelle: [Som10], S. 163 f.

Datenhaltung

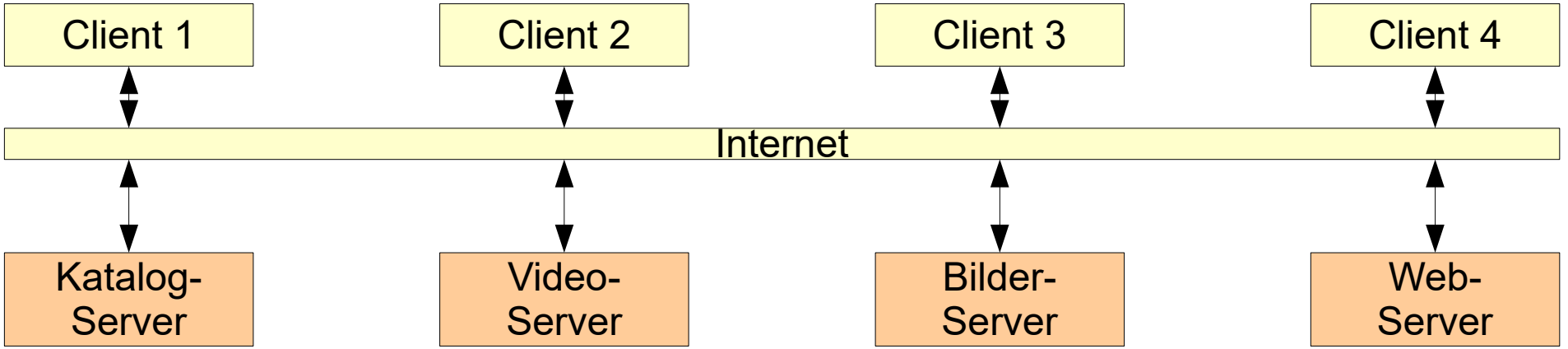


Beispiel Integrierte Entwicklungsumgebung (IDE)

- zentrale Datenhaltung in einem *Repository*
- keine explizite Sender-Empfänger-Übertragung nötig
  - Parallelverarbeitung, effizient
- Datenmodell des *Repository* ist zwangsläufig Kompromiss aus individuellen Anforderungen
- Änderbarkeit (neue Komponenten müssen kompatibel zum Datenmodell sein) und Verteilbarkeit (*Repository* muss auf einem Rechner bleiben) schwierig
- aktives *Repository*: *Blackboard* (schwarzes Brett), z. B. in KI

Quelle: [Som10], S. 159 f.

Client-Server-Architektur

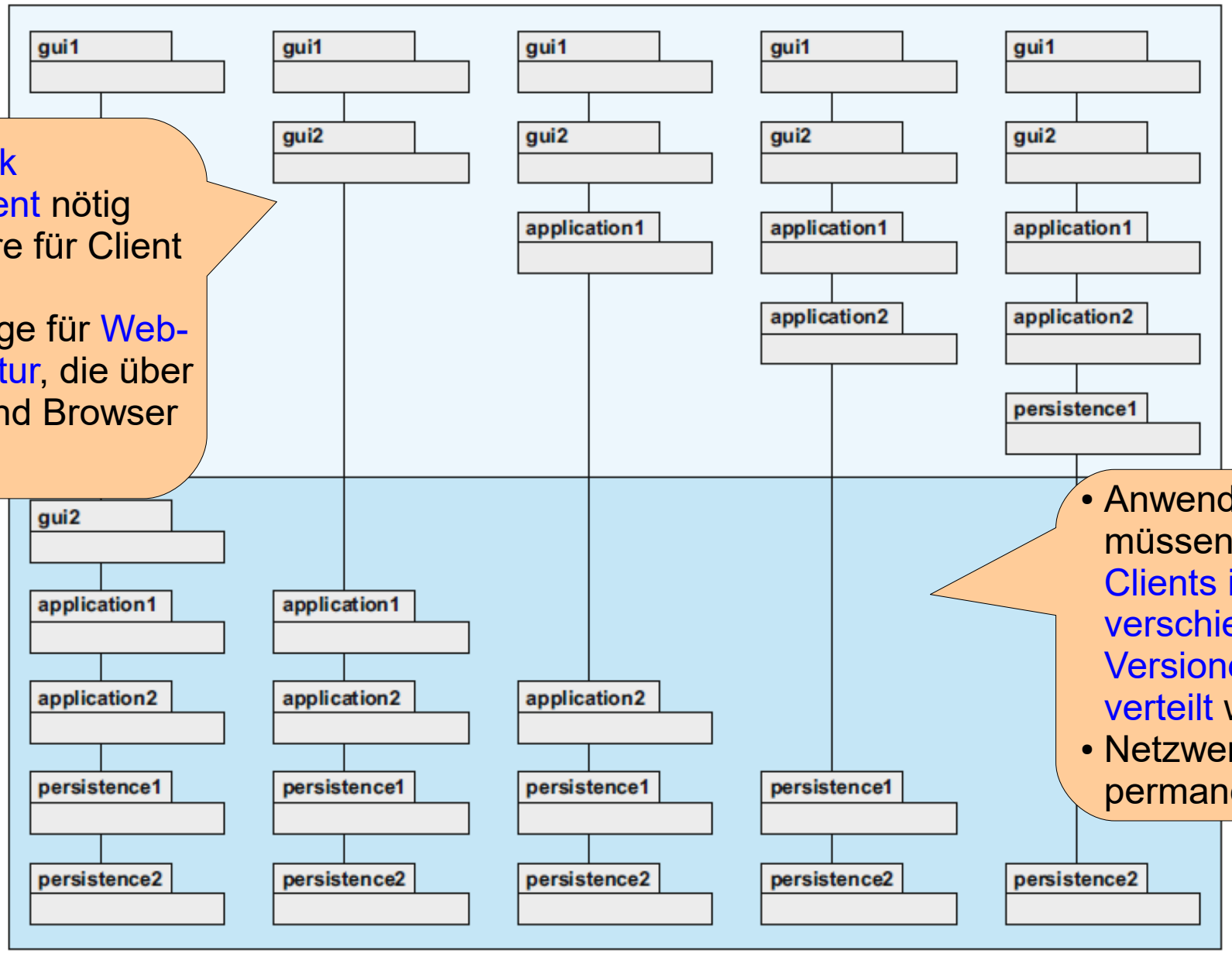


Beispiel Online-Videothek

- Clients und Server über Netzwerk (meist Internet) verbunden
- Software muss auf Clients und Servern verteilt werden
- Clients stellen Anfragen an Server, die sie beantworten
- engere Bindung, losere Kopplung
- Skalierbarkeit schlecht, aber: Replikation von Servern mgl.
  - Effizienz von Netz und System abhängig
- Server als *Single Point of Failure* => *DoS*-Attacken möglich

Quelle: [Som10], S. 160 f.

Client-Server-Architektur: Verteilungsarten

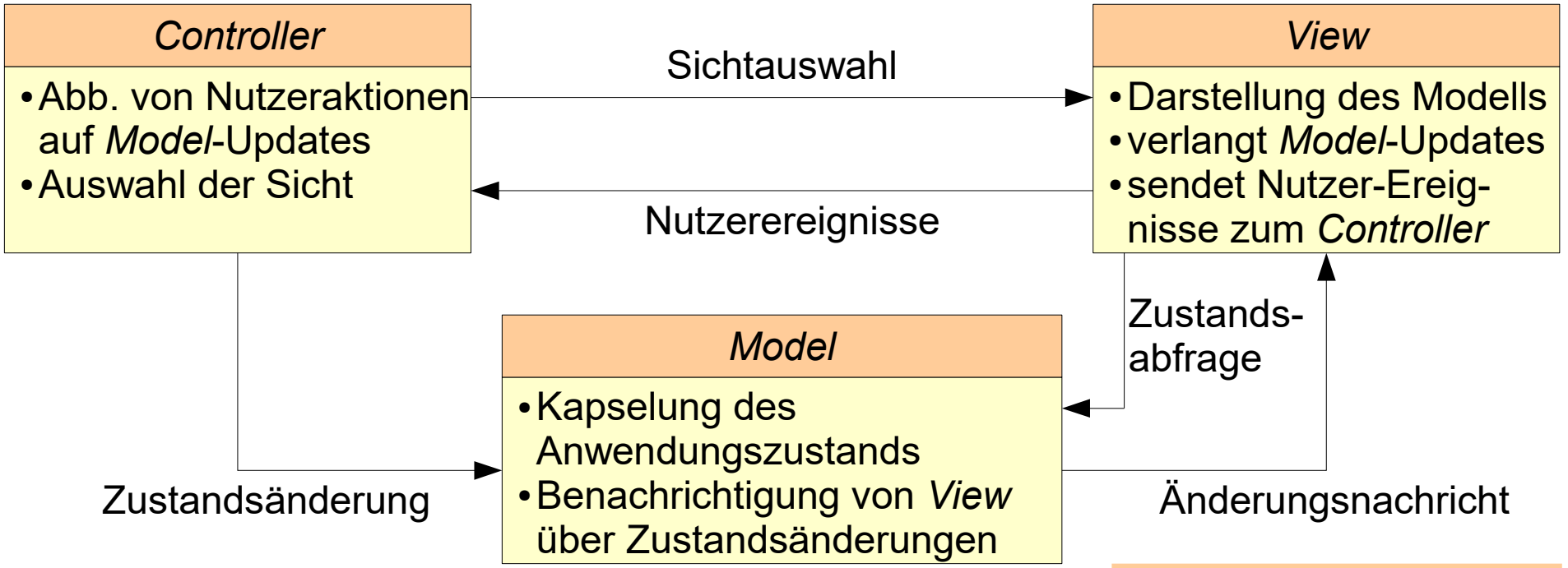


- Netzwerk permanent nötig
- Hardware für Client billiger
- Grundlage für Web-Architektur, die über HTTP und Browser läuft

- Anwendungen müssen an alle Clients in verschiedenen Versionen (OS) verteilt werden
- Netzwerk nicht permanent nötig

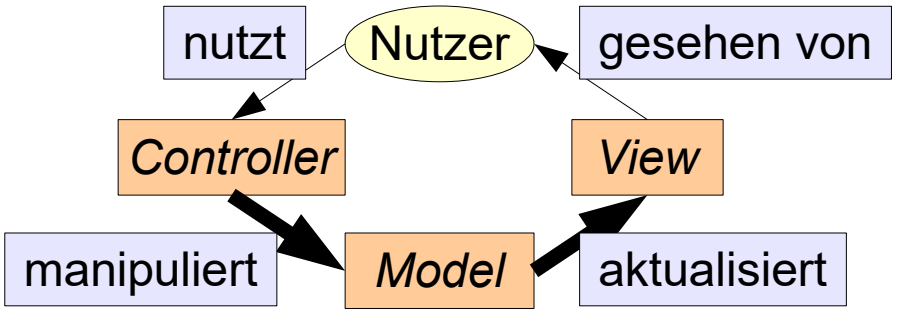
Quelle: [2], S. 194 ff.

Model-View-Controller-Muster

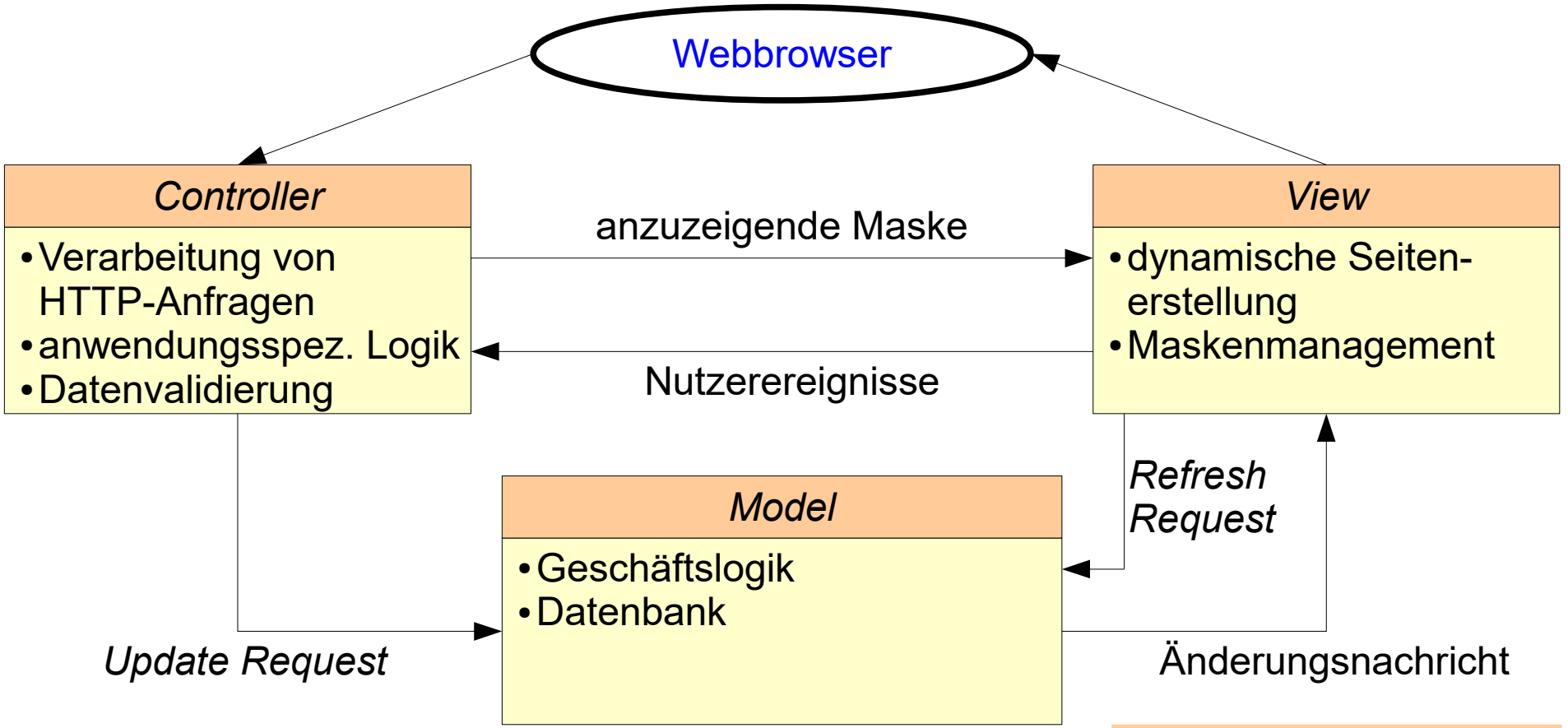


- *Model*: fachliche/Anwendungs-/ Geschäftslogik
- *View*: Benutzeroberfläche, häufig GUI
- *Controller*: Steuerung der Anw., Abb. Ereignisse->Funktionen

Quelle: [Som10], S. 156



Web-Architektur mit MVC-Muster

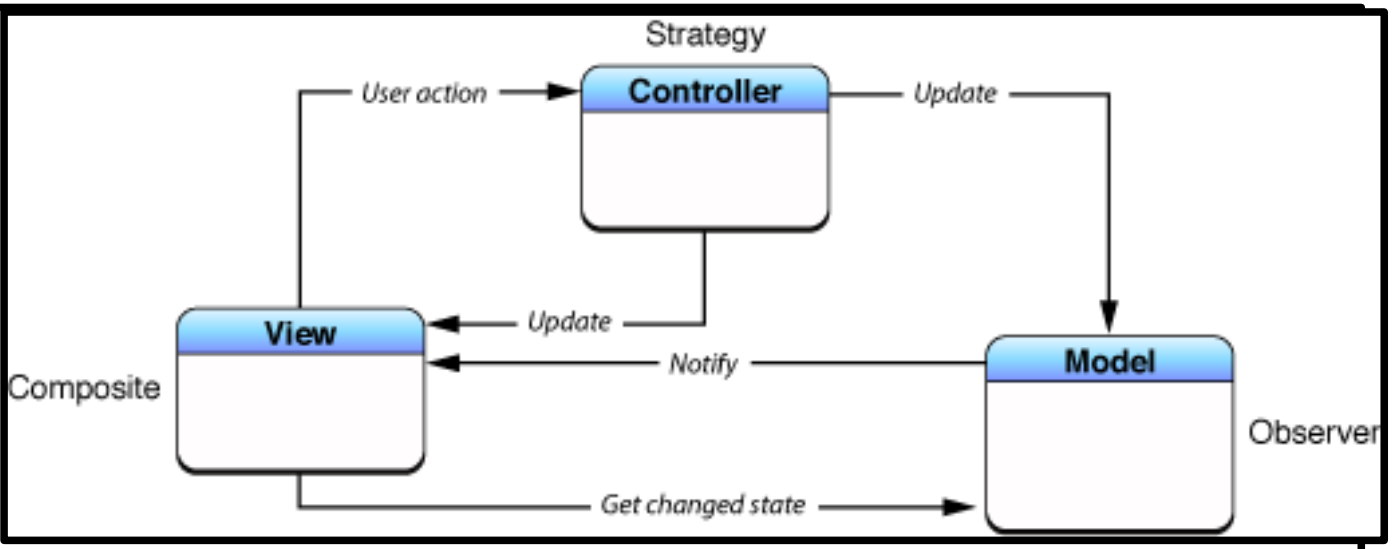


Quelle: [Som10], S. 157

- keine permanente Verbindung, bei Bedarf TCP-Verbindung
- Trend vom *Thin Client* zum *Rich Client*, z. B. *JavaScriptMVC*

Entwurfsmuster im MVC-Architekturmuster

- Beobachter
  - *Model* nutzt das Muster, um *View* + *Controller* entsprechend der letzten Zustands-änderungen zu aktualisieren



- Strategie (*View* und *Controller*)
  - *Controller* als Verhalten von *View*, leicht austauschbar
- Kompositum (*View*)
  - intern genutzt, um Hierarchie von GUI-Objekten zu verwalten
  - Aktualisierung der Wurzel eines Unterbaums stößt Aktualisierung desselben an

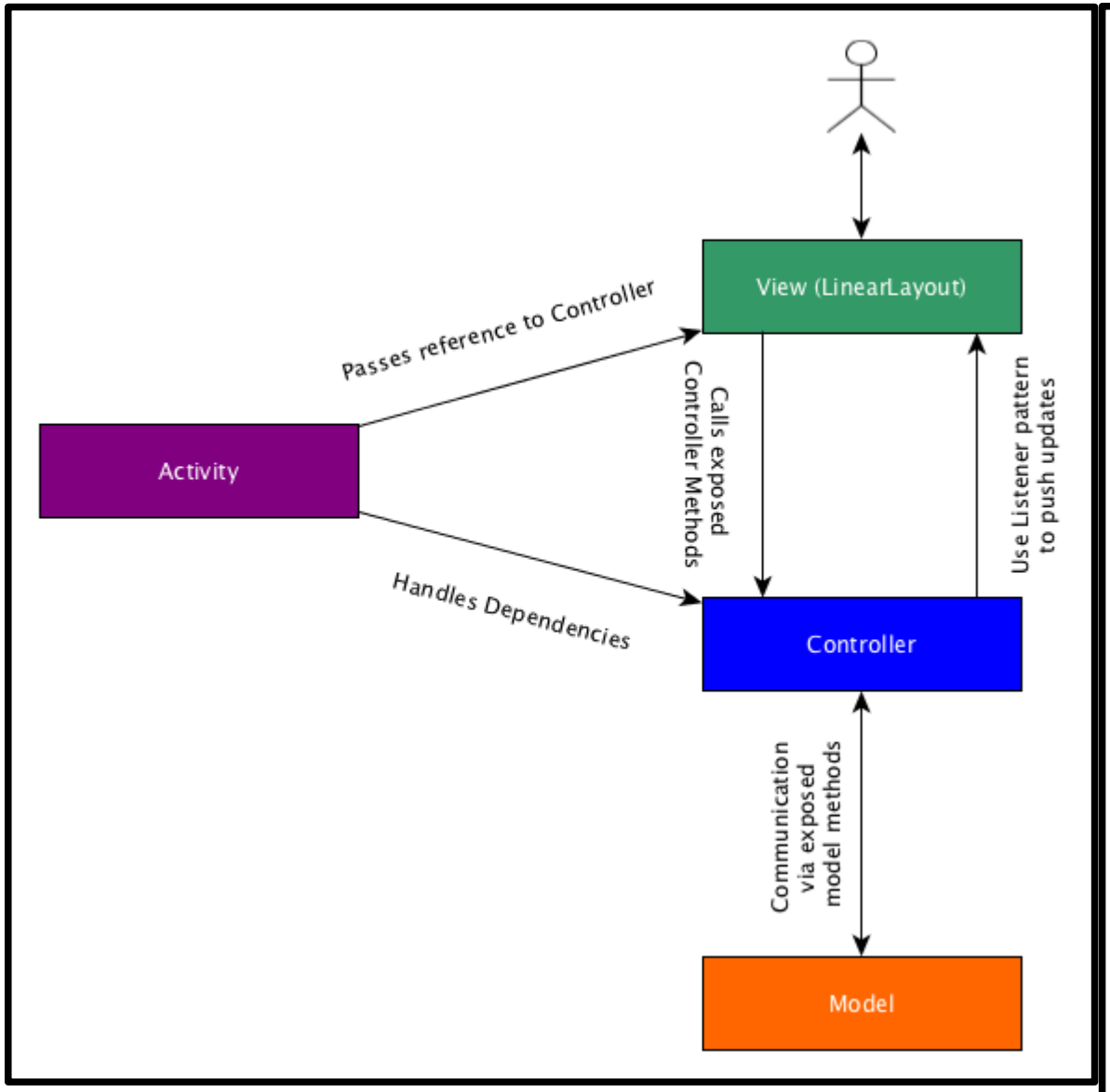
Quelle: [5]

MVC-Muster: Beurteilung

- Änderbarkeit und Portabilität durch enge Bindung und lose Kopplung gut unterstützt
- gute Skalierbarkeit bei Einsatz für *Web*-Architektur
- klare Trennung der Zuständigkeiten (*Separation of Concerns*) für Darstellung, Interaktion und Datenhaltung
- Komponenten können getrennt entwickelt werden
- *View* + *Controller* können zur Laufzeit gewechselt werden
- erhöhter Entwurfs- und Implementierungsaufwand
- Nachrichtenfluss von und zum *Model* muss in verteilten Systemen gut organisiert werden (Abfrage auf einen Schlag), um relativen Overhead zu beschränken

Quelle: [2], S. 68, S. 196 ff.

MVC-Muster in *Android*



- *View*-Schicht sollte von UI-Klassen, z. B. `LinearLayout` oder `ViewGroup` erben
  - nur vom *Controller* abhängig
- schlanke *Controller*-Klasse, die nicht erbt
  - Verbindungscode zwischen *Model* und *View*
  - wiederverwendbar
- *Activity* kann zu *Dependency Injection* entwickelt werden

Quelle: [4]

MVC-Codebeispiel für *Android* (1/2)

```
MyController controller;
protected void onCreate(){
    MyModel model = new MyModelUserDB(this);
    AnalyticsFacade analytics = MyApp.getAnalyticsFacade(this);
    UserWebService userWebService = new UserWebServiceImp();
    MyView view = findViewById("R.id.myview");
    controller = new MyController(model, userWebService, analytics)
    view.registerController(controller);
}
protected void onDestroy(){
    view.unRegisterController();
}
```

Quelle: [4]

```
protected User getUser(){
    return model.getUser();
}
protected void onClickBuy(){
    analytics.customEvent(new Event(Analytics.SCREEN_NAME, Analytics.ACCOUNT_EVENT,
                                    Analytics.NEW_SUBCRIPTION, user.getUUID));
    userWebService.subscribe(type, vat);
}
protected interface Listener {
    void onLogIn(User user);
    analytics.onUserLogin(user);
}
```

MVC-Codebeispiel für *Android* (2/2)

```
@override
public void registerController(MyController
controller) {
    this.controller = controller;
    controller.setListener(this);
    user = controller.getuser;
    userNameLabel.setText(user.getName);
}

@Override
public void unRegisterController(){
    controller.setListener(null);
    this.controller = null;
}
@Override
protected void onLogIn(User user){
    if (!user.isCustomer) {
        displayPlzBuyDialog();
    }
}
```

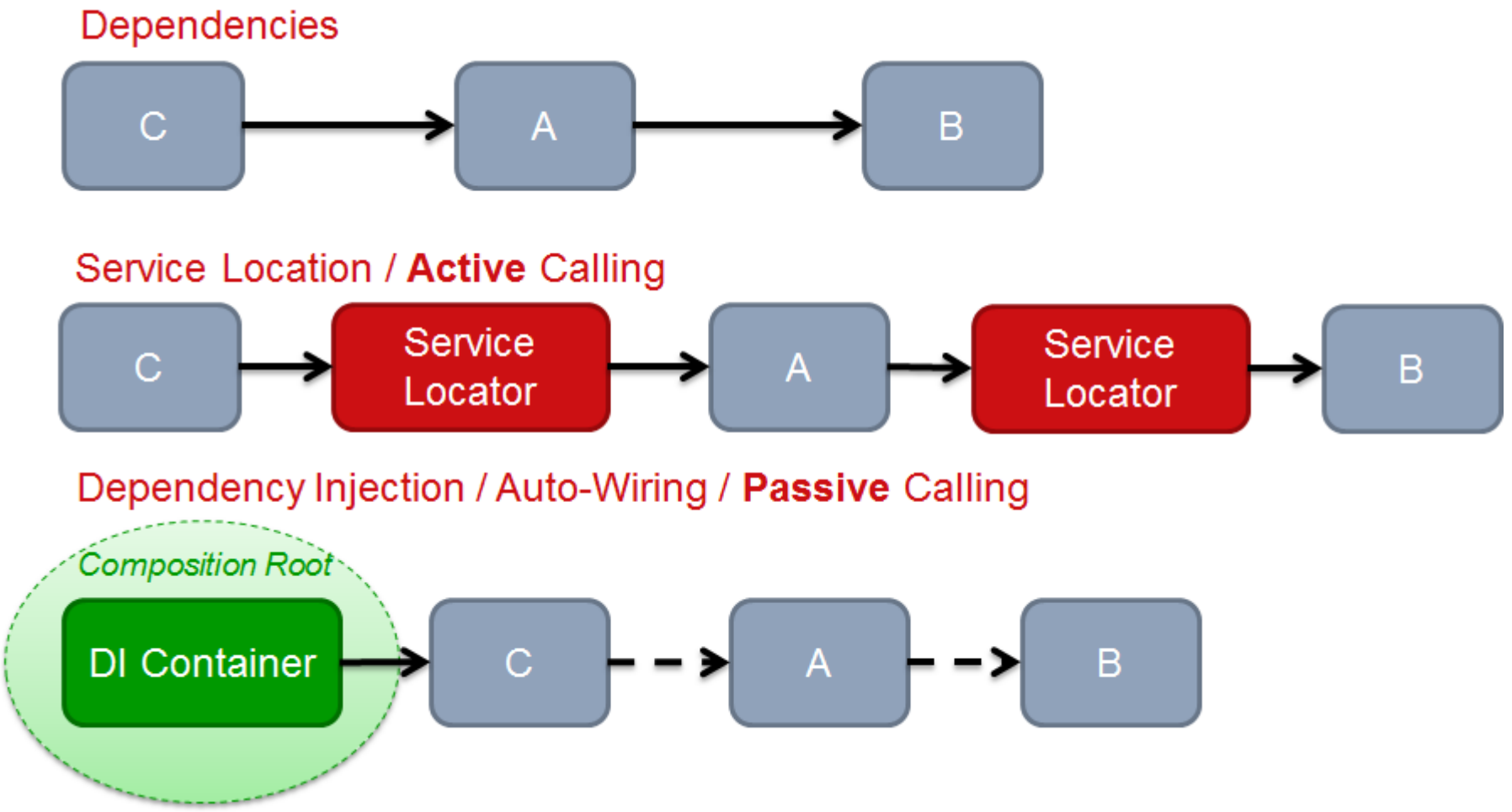
- hier benutzte Entwurfsmuster:
  - Beobachter (hier „*Listener pattern*“ genannt)
  - Strategie
  - Fassade
- Vorstufe für das Architekturmuster *Dependency Injection*

Quelle: [4]

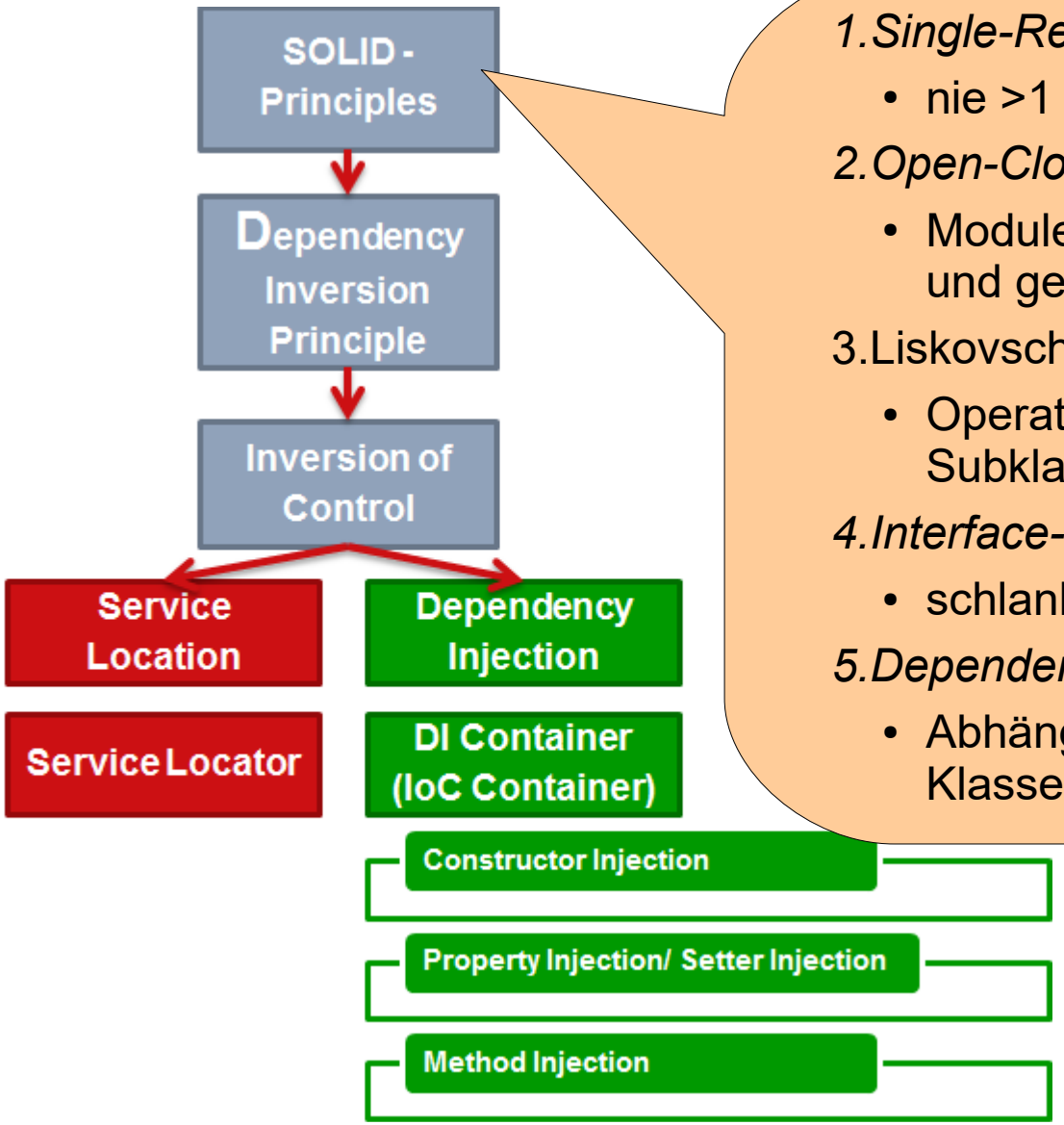
Dependency Injection (DI)

- Ziel: losere Kopplung zwischen Klassen
  - besser zu testen
  - einfacher austauschbar und damit wiederverwendbar
  - Setzen der Abhängigkeiten nicht mehr zur Übersetzungszeit nötig
- Grundlage: Umkehrung der Kontrolle (*Inversion of Control*)
- 2004 durch *Martin Fowler* systematisch beschrieben [6]
- noch flexibler als die Alternative *Service Locator*
- Abhängigkeiten in Programmcode oder in Konfigurationsdateien (z. B. in XML) ausgelagert
- implementiert in *Frameworks* wie z. B. *Spring* und *Google Guice*
  - auch völlig selbst erstellt möglich [7]

Dependency Injection



Dependency Injection



1.Single-Responsibility-Prinzip
- nie >1 Grund, Klasse zu ändern

2.Open-Closed-Prinzip
- Module offen für Erweiterungen und geschlossen für Modifikationen

3.Liskovsches Substitutionsprinzip
- Operationen auf Objekte der Subklassen korrekt ausgeführt

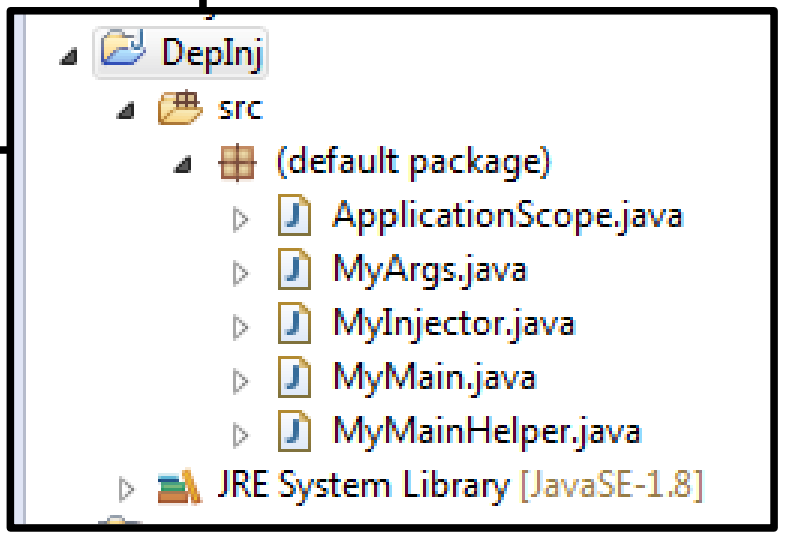
4.Interface-Segregation-Prinzip
- schlanke Schnittstellen

5.Dependency-Inversion-Prinzip
- Abhängigkeit nur von abstrakteren Klassen

Dependency Injection: Anwendungskontext

```java
public class ApplicationScope {

    private final String[] args;

    public ApplicationScope(String[] myArgs) {
        this.args = myArgs;
    }

    public String[] getArgs() {
        return this.args;
    }
}
```

Kontext durch Argumente bestimmt



Quelle: [7]

Dependency Injection: Parsen der Argumente

```
public class MyArgs {
    private String filename;
    private int timeout;

    public MyArgs(String[] args) {
        this.filename = args[0];
        this.timeout = (int)Integer.parseInt(args[1]);
    }
    public String getFileName() {
        return this.filename;
    }

    public int getTimeOut() {
        return this.timeout;
    }
}
```

Parsen der ersten beiden Argumente ohne Ausnahmebehandlung

Quelle: [7]

Dependency Injection: Hilfsklasse, mit `run()`

```
public class MyMainHelper {
    String myFileName;
    int myTimeout;

    public MyMainHelper(String myFileName,
                        int myTimeout) {
        this.myFileName = myFileName;
        this.myTimeout = myTimeout;
    }

    public void run() {
        System.out.println(myFileName);
    }
}
```

ist so für Modultests geeignet

hier nur Kontrollausgabe des Dateinamen-Kontexts

Quelle: [7]

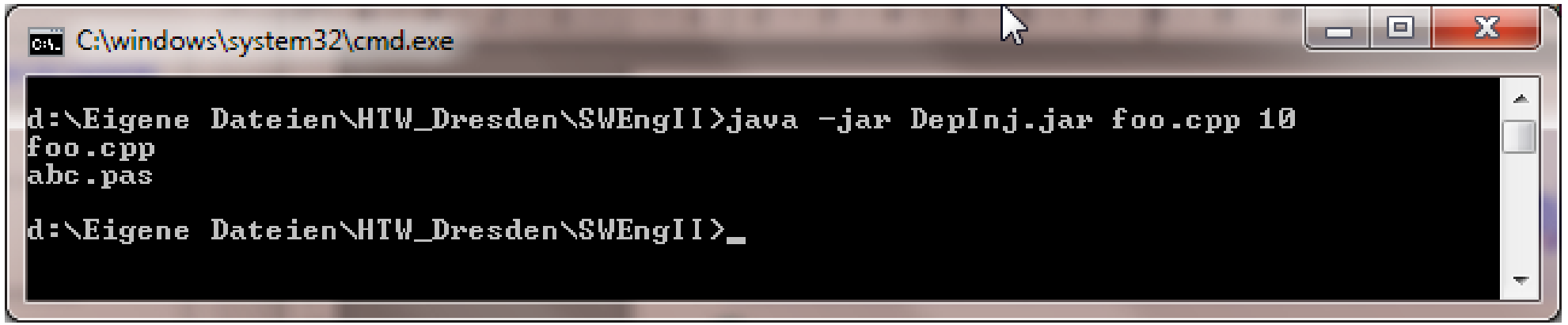
Dependency Injection: Injizierer

```
public class MyInjector {
    public static MyMainHelper injectMyMainHelper(ApplicationScope applicationScope) {
        return new MyMainHelper(injectMyFileName(applicationScope),
                                injectMyTimeout(applicationScope));
    }

    public static String injectMyFileName(ApplicationScope applicationScope) {
        return injectMyArgs(applicationScope).getFileName();
    }

    public static MyArgs injectMyArgs(ApplicationScope applicationScope) {
        return new MyArgs(applicationScope.getArgs());
    }

    public static int injectMyTimeout(ApplicationScope applicationScope) {
        return injectMyArgs(applicationScope).getTimeOut();
    }
}
```

Quelle: [7]

Dependency Injection: `main()`

```
public class MyMain {
    public static void main(String[] args) {
        ApplicationScope scope = new ApplicationScope(args);
        MyMainHelper helper = MyInjector.injectMyMainHelper(scope);
        helper.run();
        args[0]="abc.pas";
        ApplicationScope scope2 = new ApplicationScope(args);
        helper = MyInjector.injectMyMainHelper(scope2);
        helper.run();
    }
}
```



Quelle: [7]

Zusammenfassung

- Architekturmuster wie Entwurfsmuster unabhängig von Sprache, Wiederverwendung der Idee, aber grobgranular
  - grundlegende Organisation und Interaktion zwischen den Komponenten einer Anwendung beschrieben
  - bringen Struktur in Systeme, gut für Änderbarkeit
- MVC-Muster
  - dient sowohl als Architektur- als auch als Entwurfsmuster
  - Basis für alle modernen Nutzerschnittstellen
  - implementiert als Architekturmuster drei Entwurfsmuster: Beobachter, Strategie und Kompositum
- *Dependency Injection*
  - modernes Prinzip zur Erreichung einer loseren Kopplung zwischen Klassen
  - gut für Wiederverwendung und Testen

Literatur

[1] Ian Sommerville: „Software Engineering 10“, Addison-Wesley, 2016

[2] Helmut Balzert: „Lehrbuch der Softwaretechnik. Entwurf, Implementierung, Installation und Betrieb“, 3. Auflage, Springer-Verlag, 2011

[3] Kamal Wickramanayake: „Is MVC a design pattern or an architectural pattern?“, 2010, Download am 6.6.2016, http://www.swview.org/blog/mvc-design-pattern-or-architectural-pattern

[4] Thanos Karpouzis: „Android Architecture - A simple guide for MVC, MVP and MVVM on Android projects“, 24.8.2015, Download am 6.6.2016, https://medium.com/android-news/android-architecture-2f12e1c7d4db#.ayvtkwqci

[5] Toni Sellarès: „The Model View Controller: a Composed Pattern.“, 2006, Download am 6.6.2016, http://ima.udg.edu/~sellares/EINF-ES1/MVC-Toni.pdf

[6] Martin Fowler: “Inversion of Control Containers and the Dependency Injection pattern“, 23.01.2004, Download am 6.6.2016, http://martinfowler.com/articles/injection.html

[7] Chad Parry: “DIY-DI”, 9. März 2010. Download am 6.6.2016, http://blacksheep.parry.org/wp-content/uploads/2010/03/DIY-DI.pdf