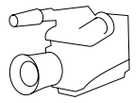
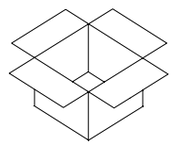
22.10.18

Aktuelle Hinweise

Entwicklungsprojekt interaktiver Systeme
Wintersemester 18/19

Technology
Arts Sciences
TH Köln



Viel drin
Medieninformatik an der Fachhochschule Köln

Bezug zu eigenen Arbeiten aus anderen Modulen

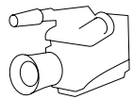
- Es muss aus den Dokumenten und Artefakten klar erkennbar sein, welche Ergebnisse von Anderen oder aus eigenen Projekten übernommen wurden, die verwendeten Ergebnisse sind also zu **zitieren**. Dies trifft u.a. auf Ideen, Modelle, Code und Texte zu.
- Die übernommen Teile werden **nicht als Teil der EIS Leistung** bewertet. Sofern Implementierungen übernommen werden - etwa aus WBA2 - so sollten diese als **externer Dienst** in die EIS Anwendung eingebunden werden.

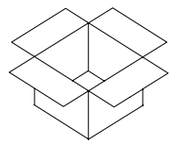
22.10.18

Implementierung

EIS Workshop 3

Technology
Arts Sciences
TH Köln



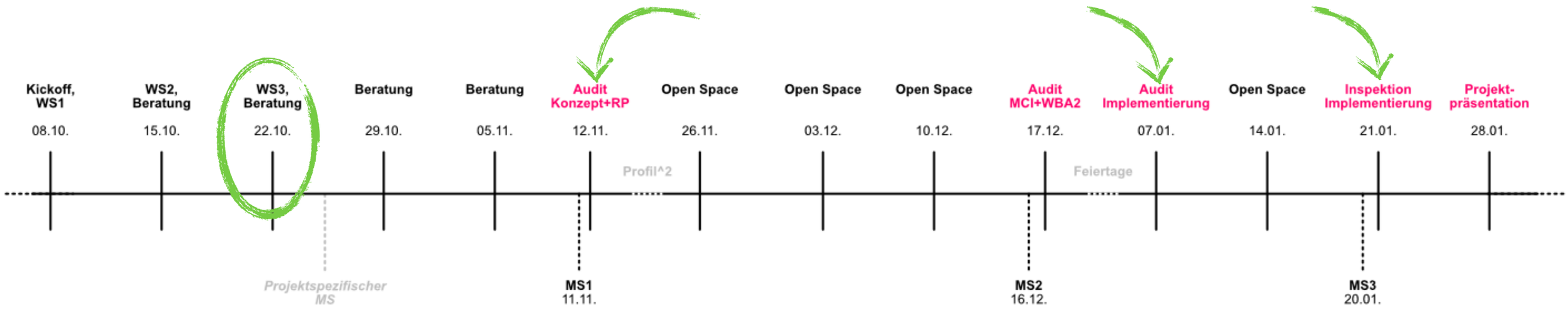
Viel drin
Medieninformatik an der Fachhochschule Köln

Implementation

Was **Vertikaler Prototyp**, relevante Anwendungsfälle:
Alleinstellungsmerkmale,
verteilte Anwendungslogik, PoCs, …

Warum Präsentationsmedium, Machbarkeit, Qualitätssicherung, …

Für wen Potentielle Geldgeber, Domänenexperten, Fachpublikum, …

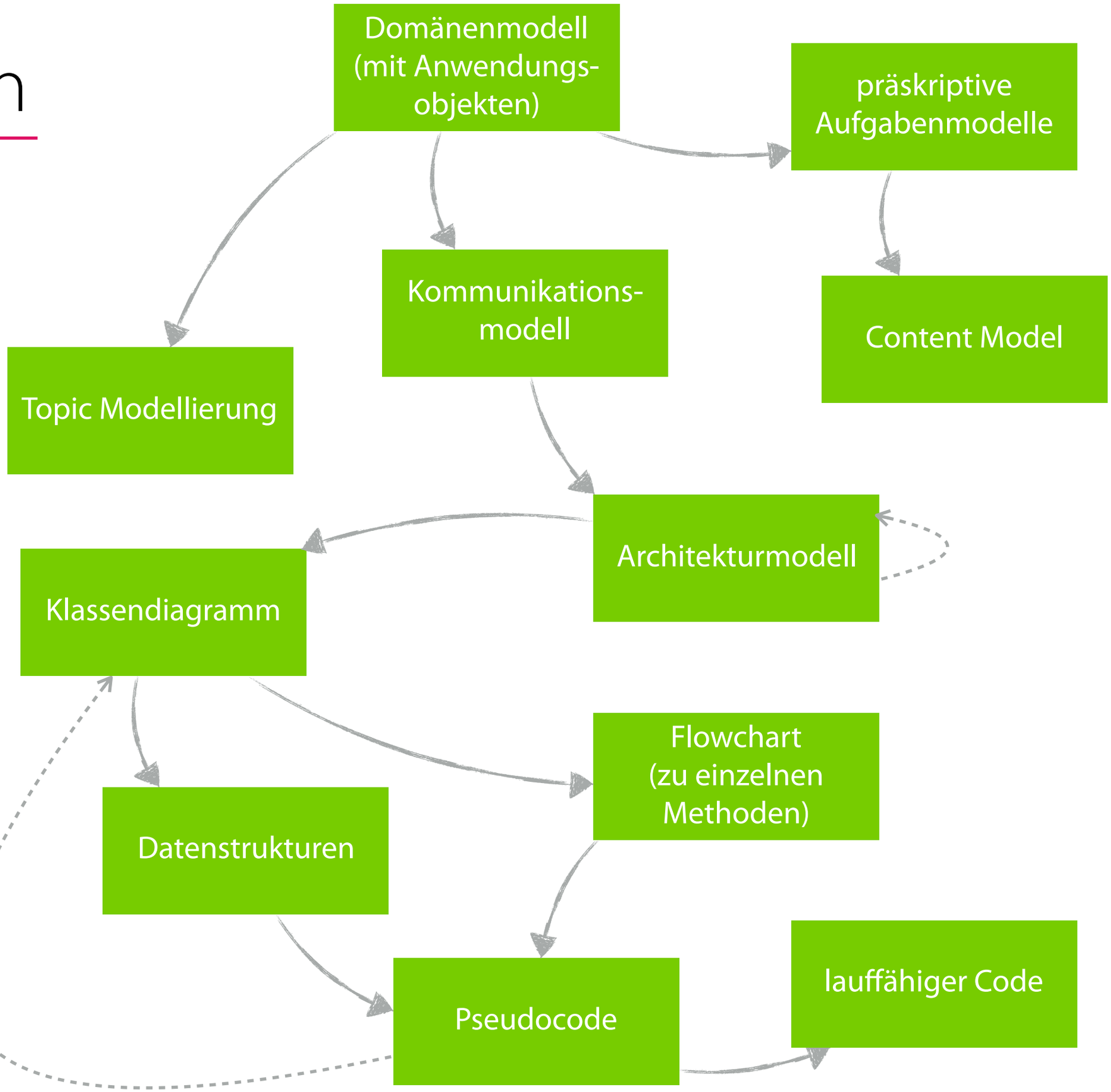


Agenda

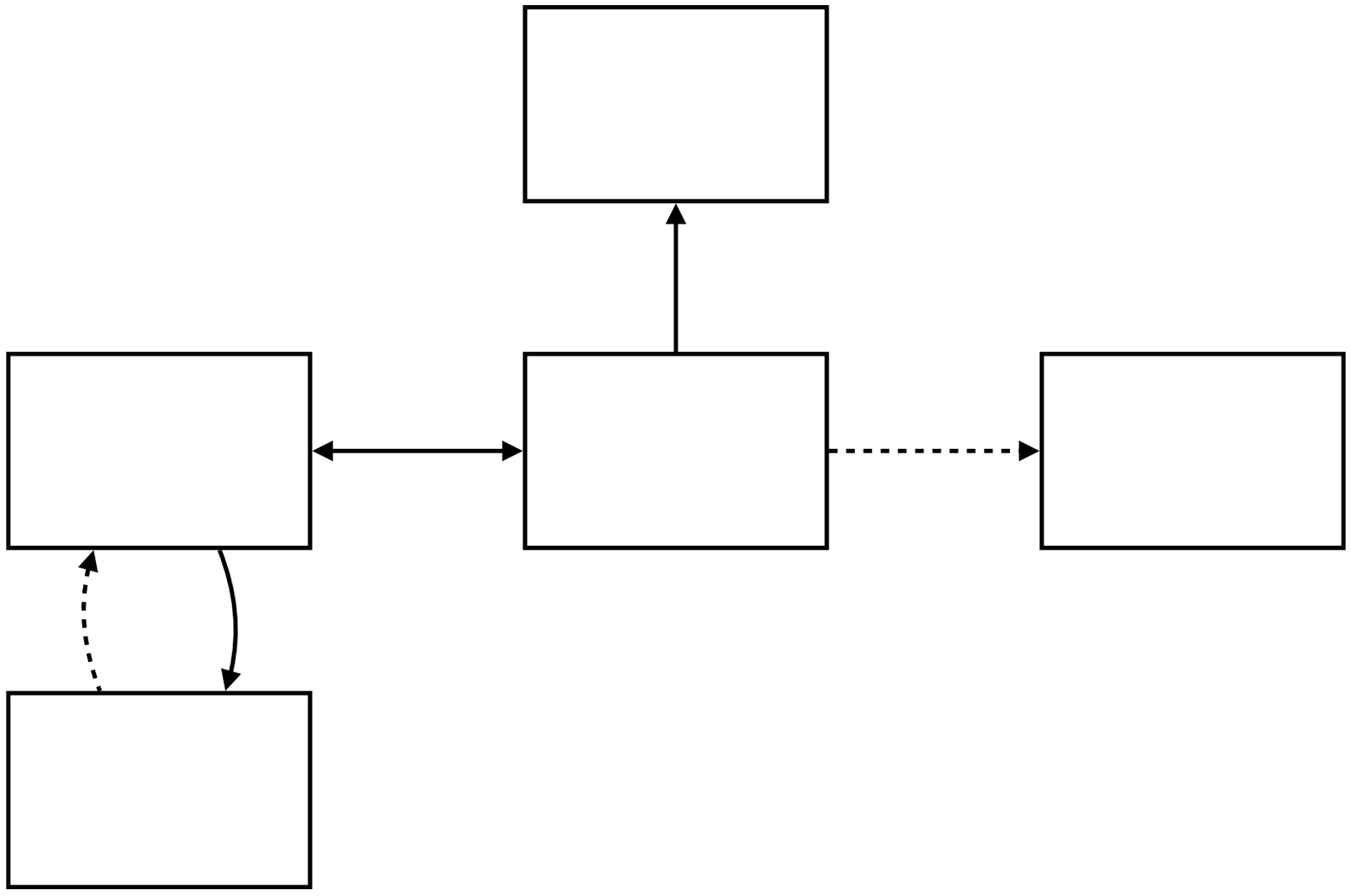
- Vorgehen
- Clean Code
- Versionskontrolle - Git
- Testen
- Formelles zur Abgabe
- Audit & Inspektion Implementierung

Vorgehen

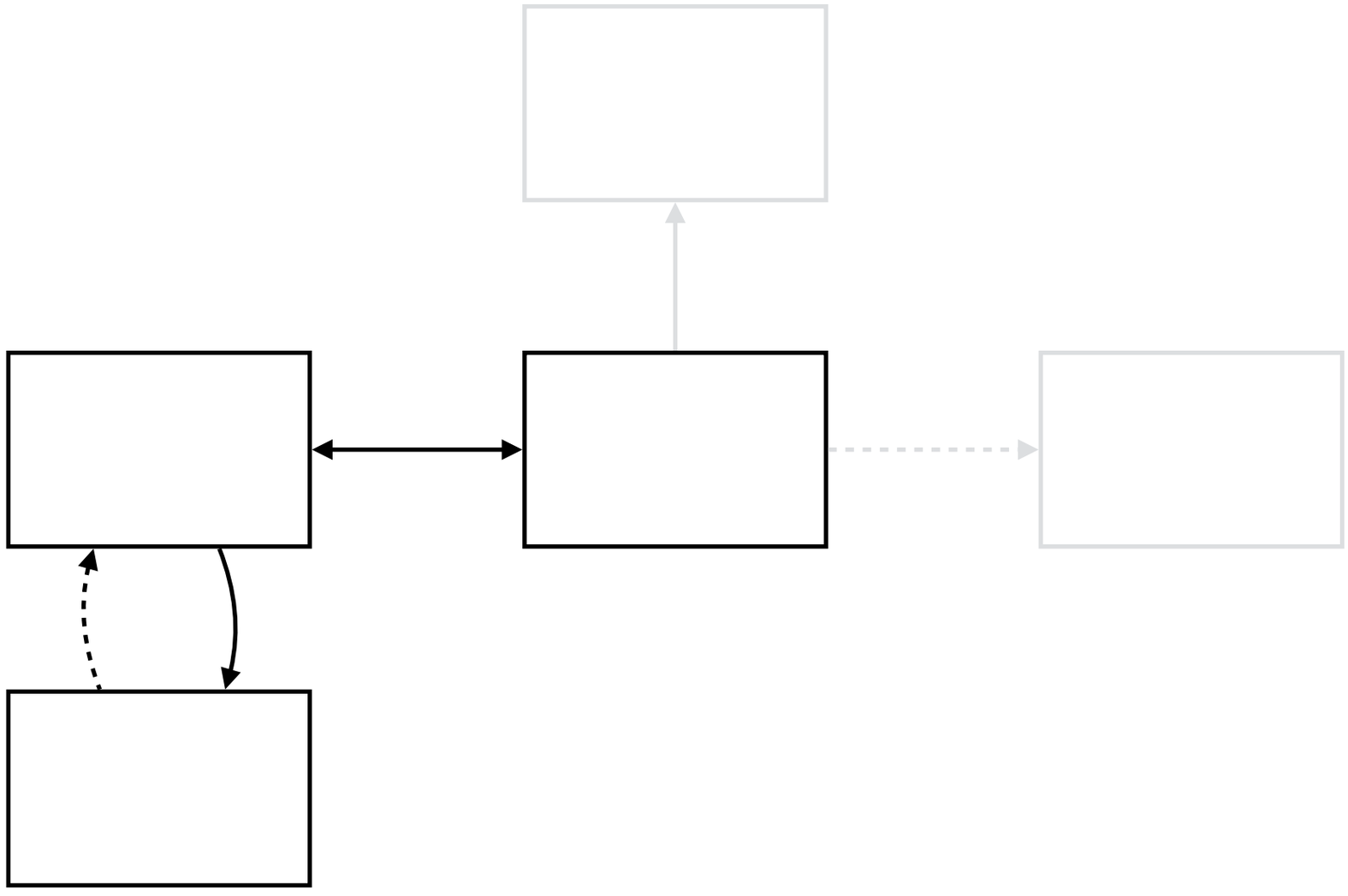
Das ist nur ein Beispiel!



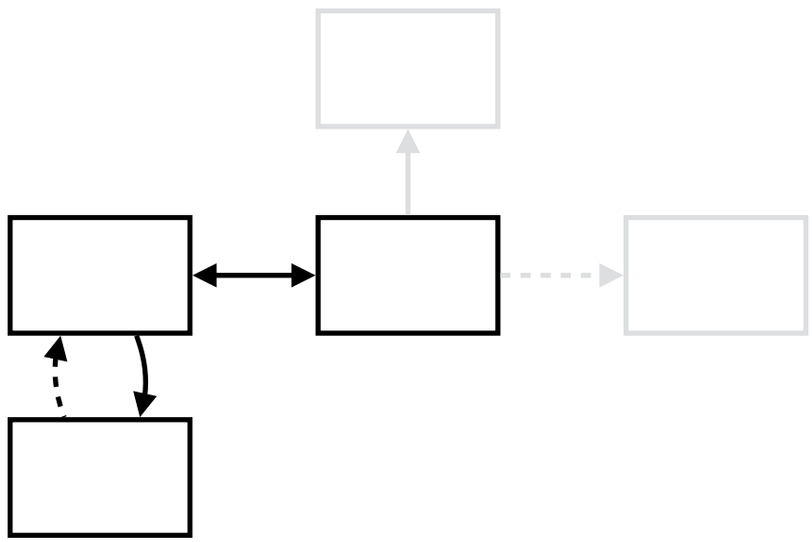
Vertikaler Prototyp



Vertikaler Prototyp



Vertikaler Prototyp



- mind. ein Alleinstellungsmerkmal
- und Verteiltheit der Anwendungslogik sind umgesetzt
- → mind. 1 Use Cases demonstrierbar

Clean Code

- Lesbarkeit durch Namensgebung & Codekonventionen
- DRY = Don‘t repeat yourself → modularisieren

sinnvolle Klassenhierarchie

Module

- Zu lange und unübersichtliche Quellcode-Dateien
  → Indikator dafür, dass modularisiert werden sollte
- Sinnvolle Ordnerstruktur nutzen

Namensgebung

„There are only two hard things in Computer Science: cache invalidation and naming things.“

— *Phil Karlton*

Namensgebung

- Sprechend und konsistent
- Nomen für Klassen, Variablen
- Verben für Methoden
- Positive Namensgebung
  - **`isValid`** & **`!isValid`** anstatt **`isInvalid`**
- Alle Magic Numbers benennen
  - **`months.length`** ist besser als **`12`**

Codekonventionen

- z.B.
  - http://www.torsten-horn.de/techdocs/java-codingconventions.htm
  - https://github.com/felixge/node-style-guide
- Nützlich für
  - Lesbarkeit
  - Übersicht
  - Klarheit
  - Refactoring

```
public double division(double divident, double divisor) {
   return divident/divisor;
}
```

```
User
   .findOne({ name: 'foo' })
   .populate('bar')
   .exec(function(err, user) {
     return true;
   });
```

Kommentieren

- Knapp und gehaltvoll
- Javadoc und Co.
  - Javadoc allein genügt nicht als Codedokumentation
- Nicht das ***Was*** sondern das ***Wieso*** dokumentieren
- Übernommenen Code kennzeichnen (!)

Fehlerbehandlung

- Fehler sollten nicht ignoriert werden
- Gute Fehlerbehandlung kann sich auch positiv auf MCI-Aspekte auswirken
- Exceptions/Errors nutzen (auch in JavaScript!)
  - https://docs.oracle.com/javase/tutorial/essential/exceptions/
  - https://www.joyent.com/developers/node/design/errors
- Nicht für den normalen Kontrollfluss verwenden

```
int sumArray(int[] array) {
   int sum = 0;
   try {
      for (int i = 0;; i++) // BAD: no loop test
         sum += array[i];
   } catch (IndexOutOfBoundsException e) {
      return sum;
   }
}
```

So nicht!

Versionskontrolle

- **Git verwenden** - und zwar sinnvoll!
  - feature branches für einzelne Funktionen oder Komponenten
  - Issues, um aktuelle Aufgaben zu tracken
  - Commit messages für lesbaren Arbeitsverlauf
  - .gitignore zum Ausklammern von Abhängigkeiten (KEIN Repository sollte „node_modules“-Ordner enthalten!)

Testen

- Test Driven Development (TDD) ist Standard
- Testen hilft um:
  - Fehler zu vermeiden und zu finden
  - Die Auswirkungen von Änderungen unmittelbar abzuschätzen
  - Die Zusammenarbeit zu vereinfachen
  - Die Qualität zu gewährleisten
- Tests können vor der Impl. definiert werden
- Tests sind selbst Code
- Verschiedene Frameworks: z.B. JUnit, Mocha

Abgabe

- Gesamter Quellcode (ohne nachladbare Abhängigkeiten)
  - Code-Leichen und auskommentierten Code entfernen
- Zusätzlich: Ausführbare Form
  - **JAR oder APK-Archive; in JS vollständige Package.json für „npm start“**
  - Nicht nur das Projektverzeichnis eurer IDE (!)
  - Für Node.js: vollständige **package.json**
  - **Keine unnötigen Abhängigkeiten mitliefern (.gitignore!)**
- Installationsdokumentation + Systemanforderungen
- Testdaten falls nötig mitliefern (z.B. SQL-Dumps, Testbenutzerdaten)
- Konfigurationsdateien für Parameter, die angepasst werden müssen
- Abgabe selbst in einer „frischen“ Umgebung installieren (z.B. leeres System)

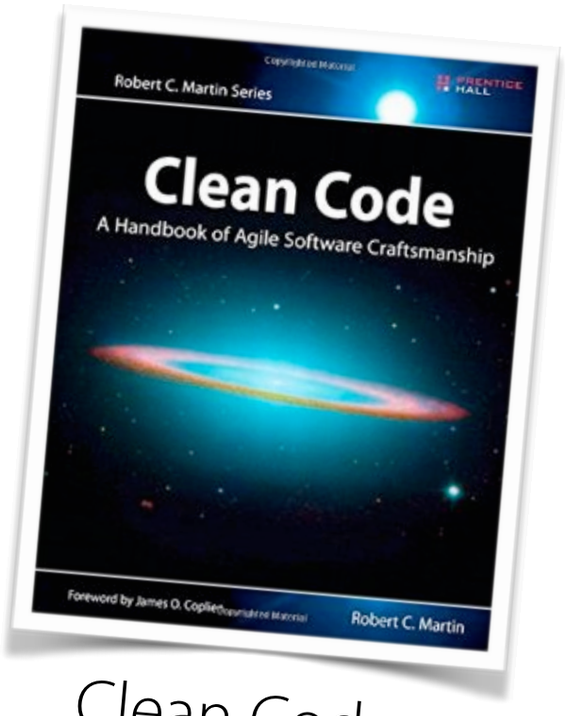
Audit & Inspektion
Implementierung

- relevante Stellen im Code sollen erläutert werden

  (→ Demonstration in Abschlusspräsentation!)

  - Anwendungslogik auf verschiedenen Komponenten
  - wichtige architekturelle Merkmale
- Diskurs zu Code-Designentscheidungen

Fragen?



Clean Code
Robert C. Martin