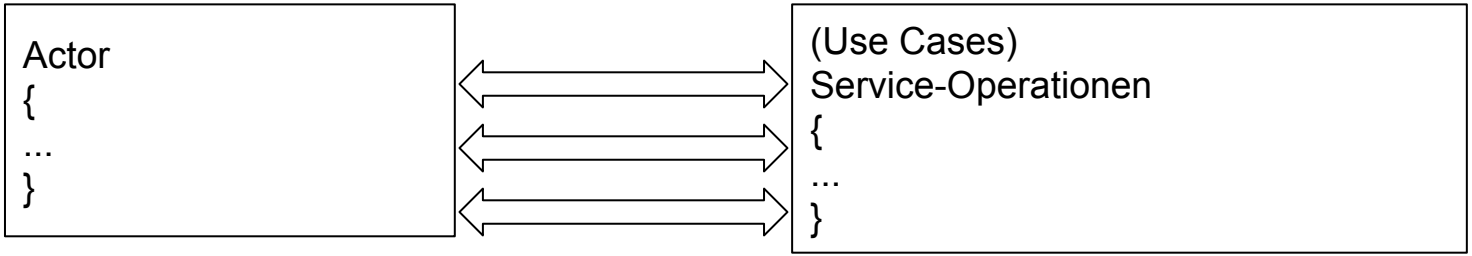
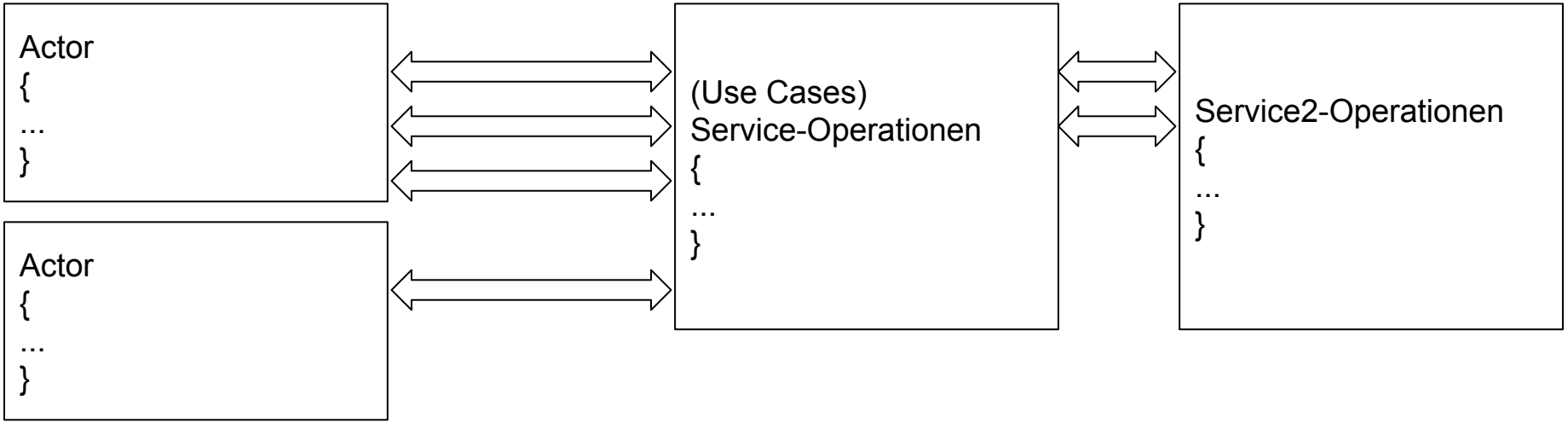
Clean Code

Ausgangssituation



Ausgangsssituation: Service Orientiert

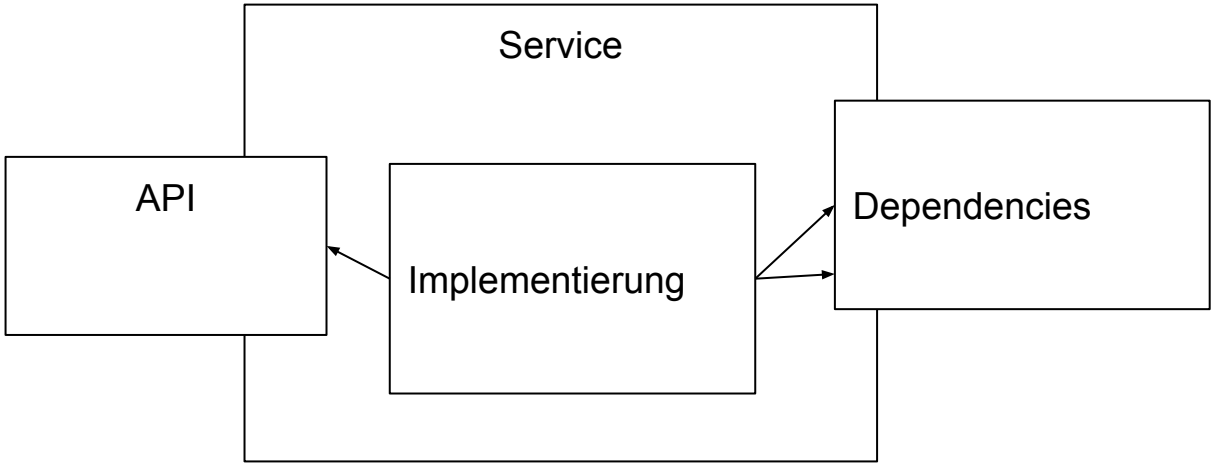


Clean Code

- Das Programm muss klar und vollständig dokumentiert sein
- Zusätzlicher Aufwand ist notwendig
  - Software-Entropie
- Aufwand muss kalkuliert und garantiert werden

Aufwände zur Erstellung der vollständigen Dokumentation

integrata cegos



Schätzung

integrata cegos

- Welche der Service-Dokumentationen verursacht den größten Aufwand?
- Dependencies sind eine Liste von bereitzustellenden anderen Services
  - Qualifier insbesondere die Versionsnummer
  - Formulierung erfolgt sinnvollerweise über eine Service-Koordinate
    - Name des Services
      - Hierarchische Namensstruktur, Einstiegspunkt der Domänen-Name
    - Version als fixe Nummer oder besser ein Version Range
      - 1.1 oder [1, 2)
- Implementierungsdokumentation ist extrem aufwändig
- API-Dokumentation ist ebenfalls sehr aufwändig, kann aber formal beschrieben werden

Service Repository

- Jeder Service im Unternehmen muss eindeutig im Service-Repository gefunden werden können
  - Identifier ist die Service-Koordinate
    - API-Beschreibung
    - Dependencies
    - Implementierungs-Dokumentation
- Optional: Such-Funktion über Tags
  - “Suche mir alle Services mit Bezug zu ‘Rechnungswesen’”

In der Praxis

integrata cegos

- API-Beschreibungen sind OpenAPI-Dokumente für RESTful WebServcies
- Dependencies können beispielsweise sein
  - Build-Dependencies
  - Container-Orchestrierung, docker-compose, Kubernetes pod
- Implementierungs-Dokumentation ist sehr heterogen
  - Self documented code
  - Test driven
  - Klassische Pflichtenhefte
  - …
- Service Repository ist beispielsweise WIKI/Confluence

Die Ebene der Service-Implementierung

Realisierung mit einer Objekt-orientierten Plattform

integrata cegos

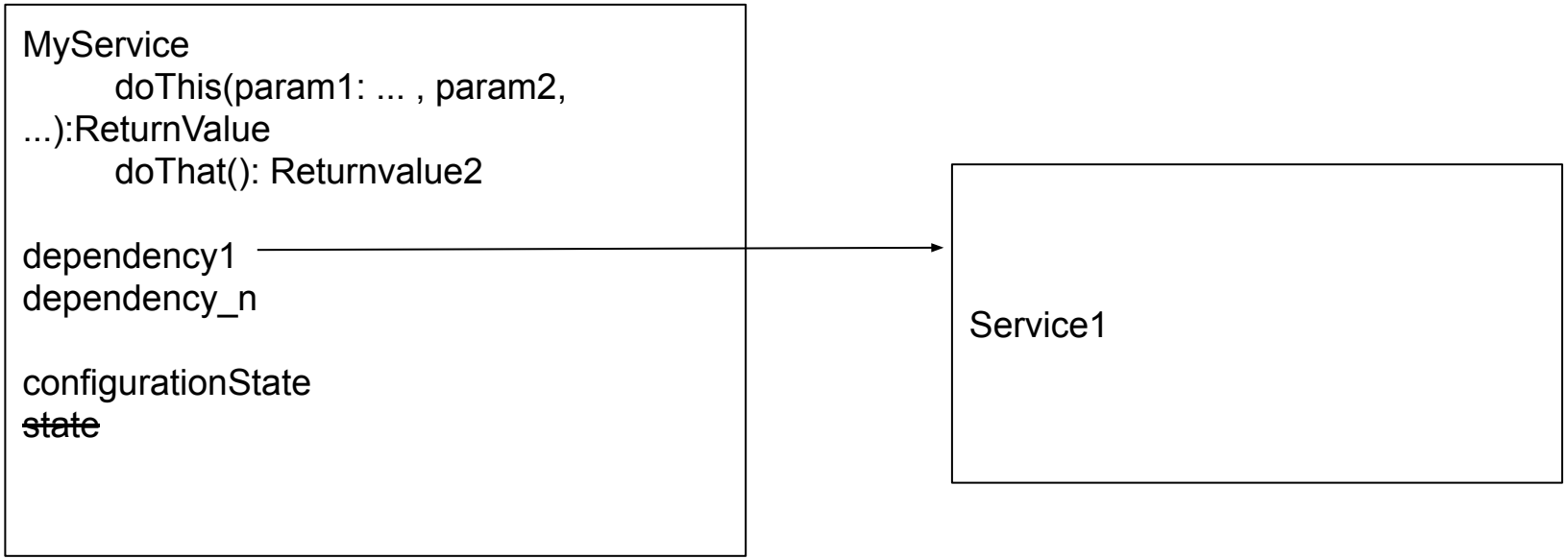
- Modellierung erfolgt jetzt z.B. mit den Klassendiagrammen aus UML
- Exkurs: Klassen und OOP
  - Weder C++ noch Java sind wirklich Objekt-orientierte Sprachen
  - Beide Sprachen sind Klassen-orientiert
  - Objekt-orientierte Sprachen fassen alle Objekte als potenziell individuell auf, Klassen-orientierte Sprache sprechen von “Instanzen einer Klasse”
  - Echte Objekt-orientierte Sprachen wären beispielsweise JavaScript, Scala, ...

Vorsicht: Klassen, Objekte und Services

integrata cegos

- Daten, die zwischen Services ausgetauscht werden sind “Ressourcen” und keine Objekte
  - Objekt hat Verhalten, ein Objekt “kann” etwas
  - Ressourcen sind blanke Daten, sind structs
- Diese Datenaustausch-Klassen sind “anemic”, enthalten keine Business Logik
  - Innerhalb einer Implementierung sind anemic Klassen ein Anti-Pattern
- Bereitgestellte Services sind eigentlich nur “Objekte”, die bereits fertig produziert/hergestellt sind
  - Aus Sicht des Actors ist der Service ein einziges Objekt

Ein Beispiel für eine Service-Klasse



Services und OOP

- Services sind prozedural
- Ein Service definiert Service-Operationen, die auf externen Daten-Strukturen operieren

Regeln für ein gutes OOP-Modell

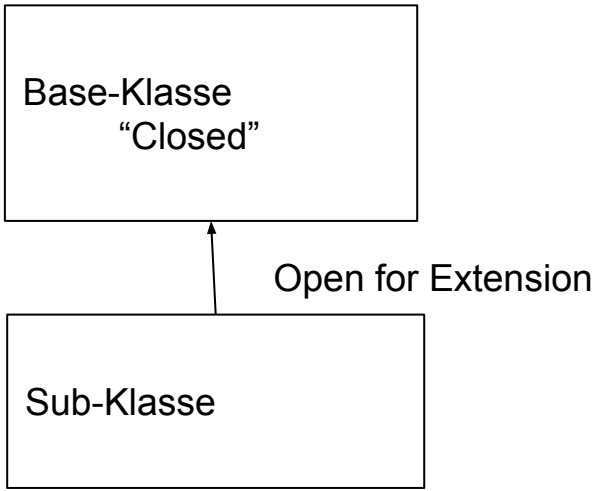
- Nicht-Einhaltung ist IMMER ein “Code Smell”
- Qualitäts-Metriken sind nicht immer fixen Regeln!
  - Große Klassen mit vielen Lines of Code sind qualitativ schlecht
  - Performance
  - Benutzer-Akzeptanz

“Goldene Regel 1” der OOP

- Single Responsibility Principle
  - Jede Klasse macht exakt eine Aufgabe, übernimmt im Rahmen der Anwendung eine eindeutige “Rolle”
- Eine Klasse ist so groß wie notwendig, aber niemals größer
  - “So klein wie möglich” ist unsinnig, Eine Methode pro Klasse???

“Goldene Regel 2” der OOP

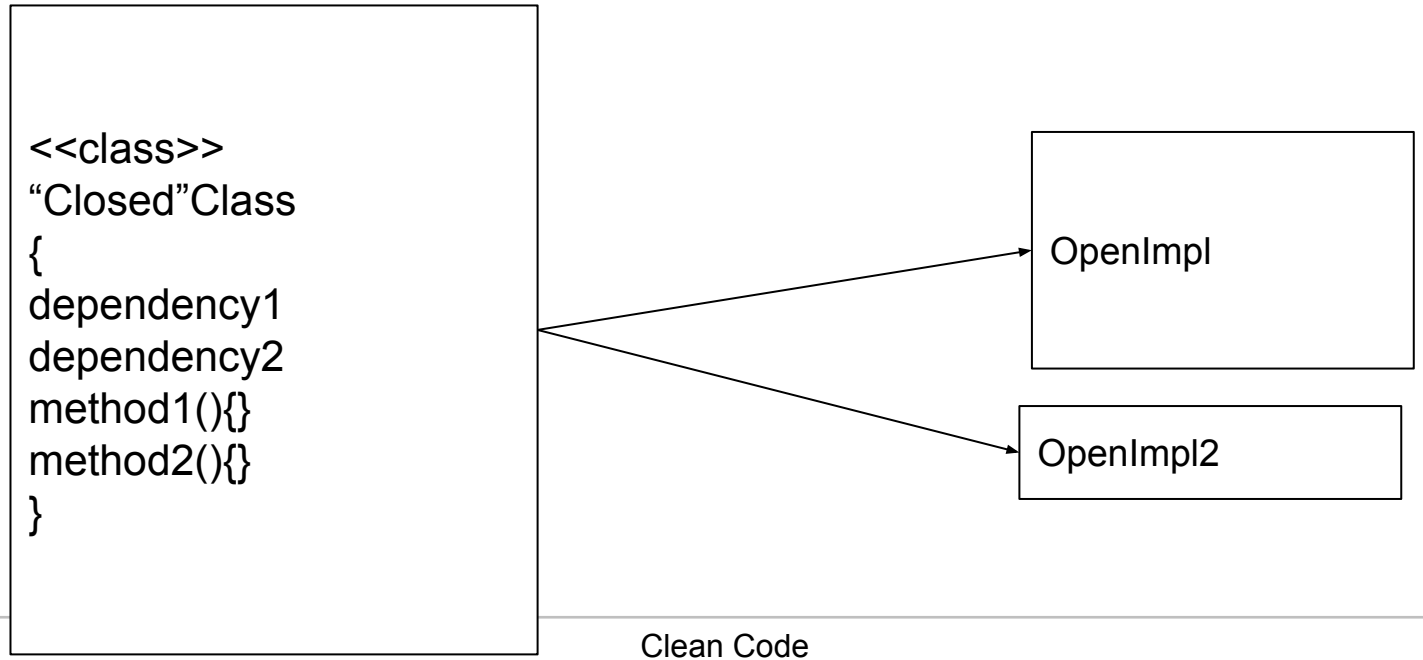
- Open-Closed Principle
  - Eine Implementierung soll ohne Änderung in der Lage sein, geänderte Vorgaben umzusetzen
- Klassische Umsetzung in der OOP-Welt ist “Vererbung”



Open-Closed-Prinzip für Klassen

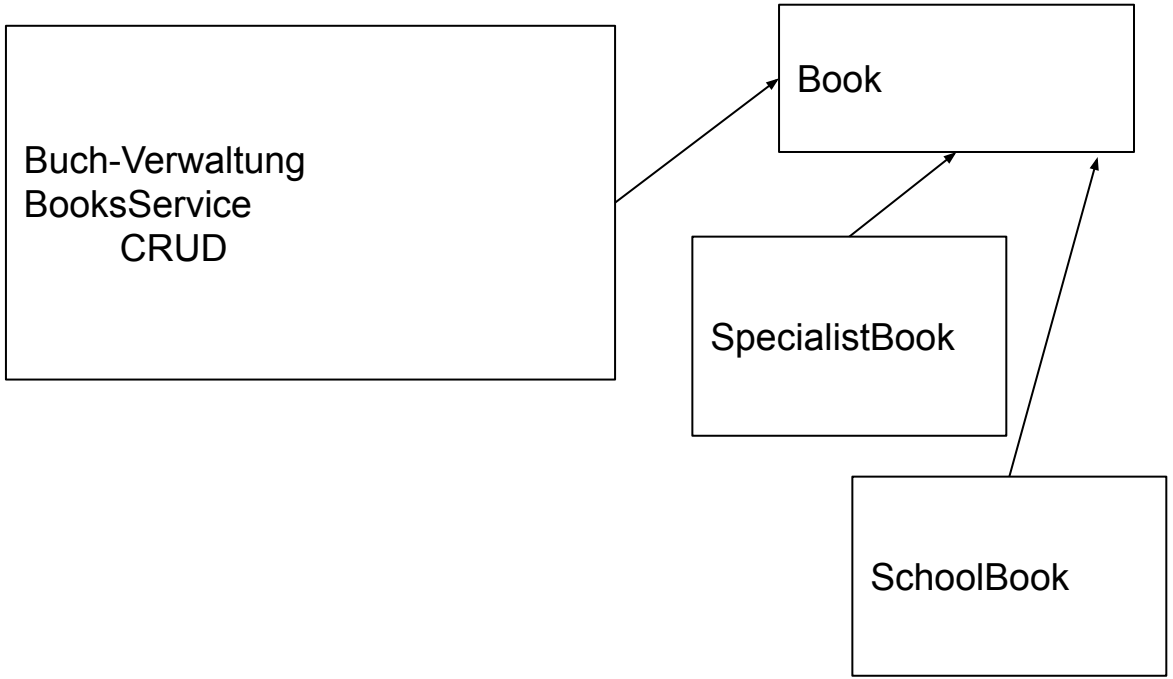
integrata cegos

- In Java und C++ nicht so einfach umzusetzen!
- Design Pattern “Strategy” ist hierfür notwendig



Ein kleines Beispielprojekt

Fachliche Seite der Anwendung



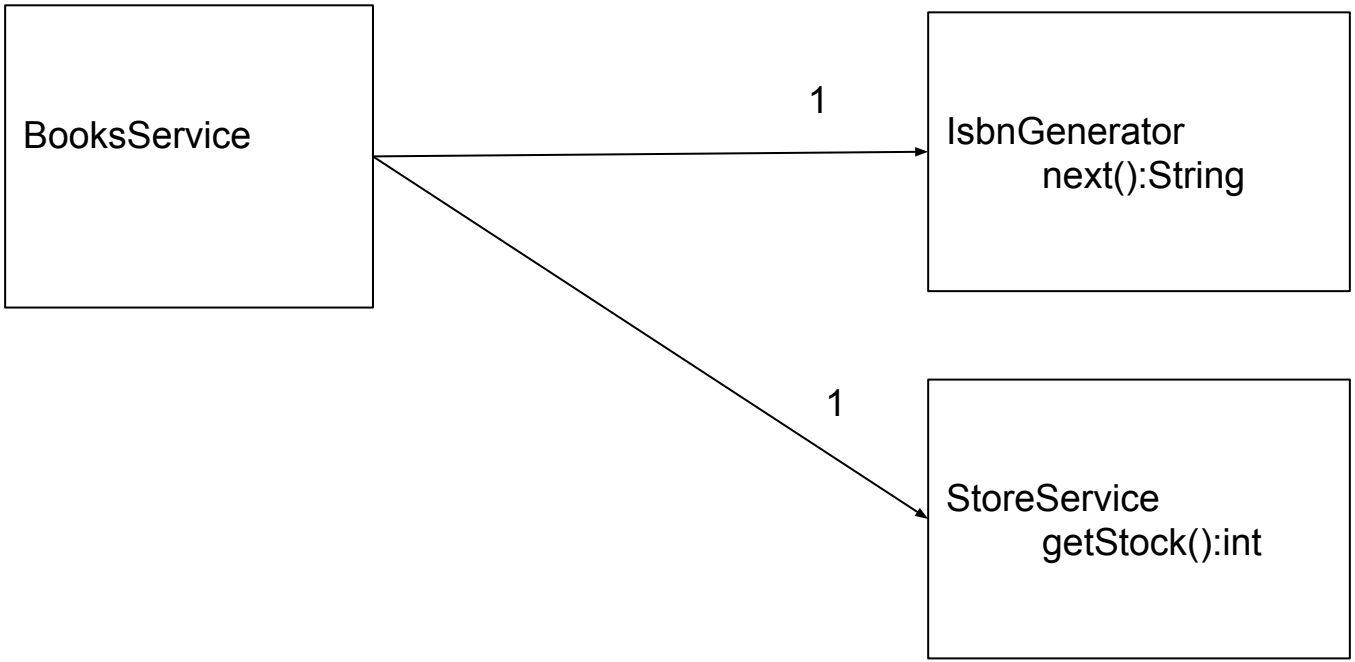
Erkannte Schwachstellen

- Single Responsibility
  - Rollen sind
    - Buchverwaltung
    - ISBN-Erzeugung
    - Bestandsabfrage
- Open-Closed
  - Keinerlei Möglichkeit, die Strategien zur Generierung und Bestandsabfrage zu ändern

ToDo

- Verändern Sie das Modell so, dass die beiden oben aufgeführten Prinzipien erfüllt sind

Muster: Technisches Modell

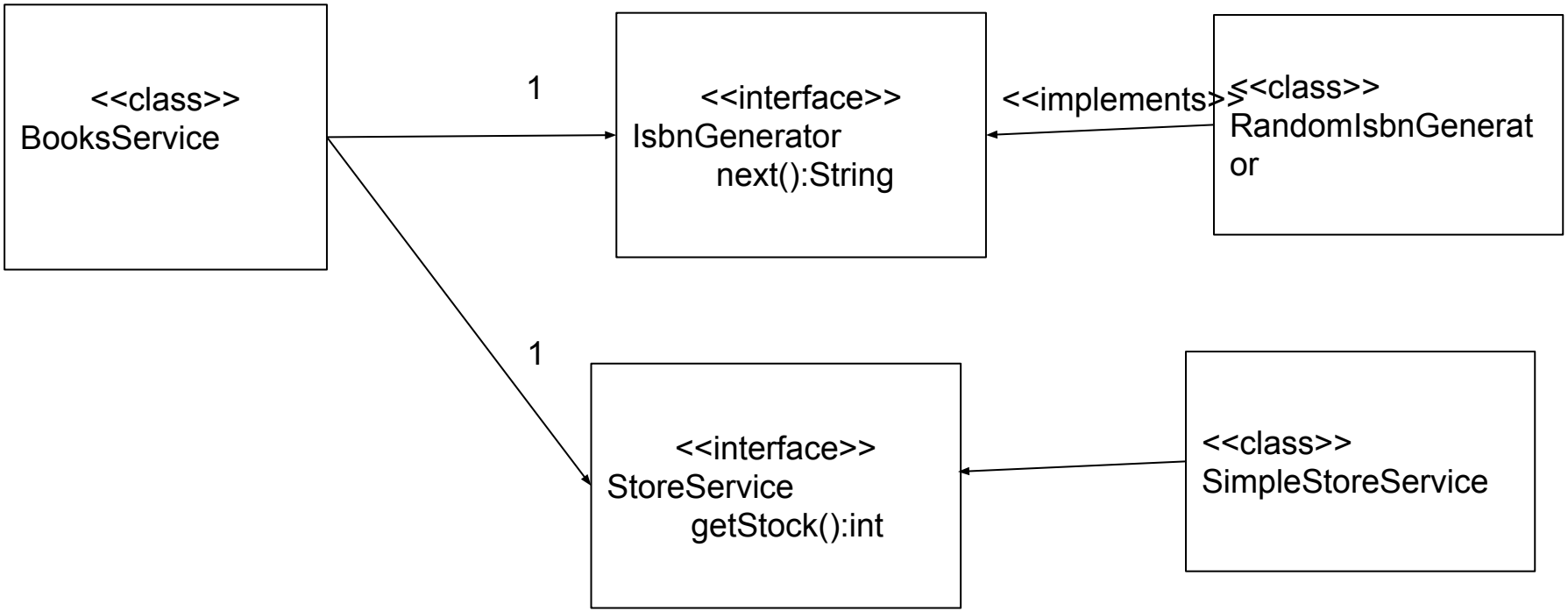


Umsetzung in einer statisch typisierten Sprache

integrata cegos

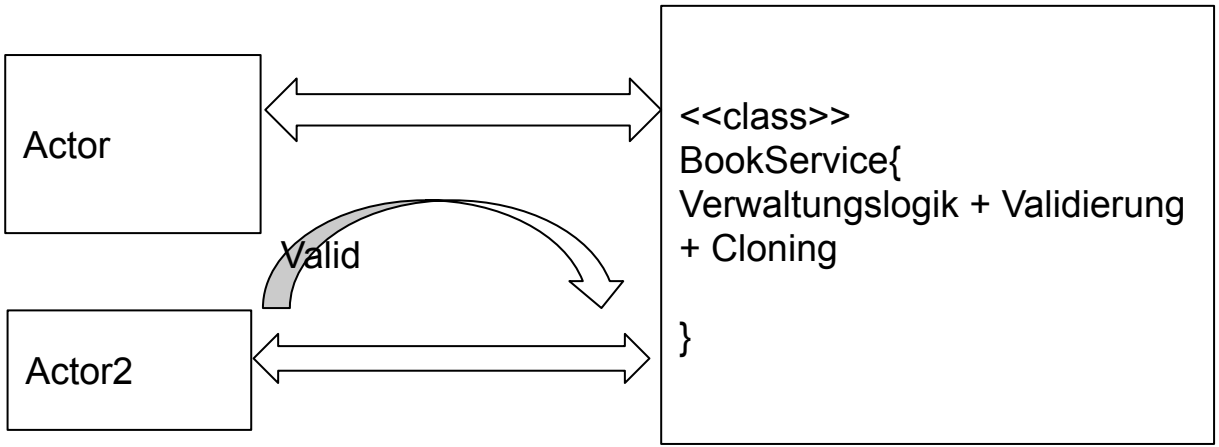
- Verlangt bei der Umsetzung zusätzlichen Aufwand
- Einführung eines Interfaces
- Regel: Abhängigkeiten, die aus dem Open Closed-Prinzip resultieren müssen über ein abstraktes Interface entkoppelt werden

Muster: Strategy-Design-Pattern



Schwachstellen, Teil 2

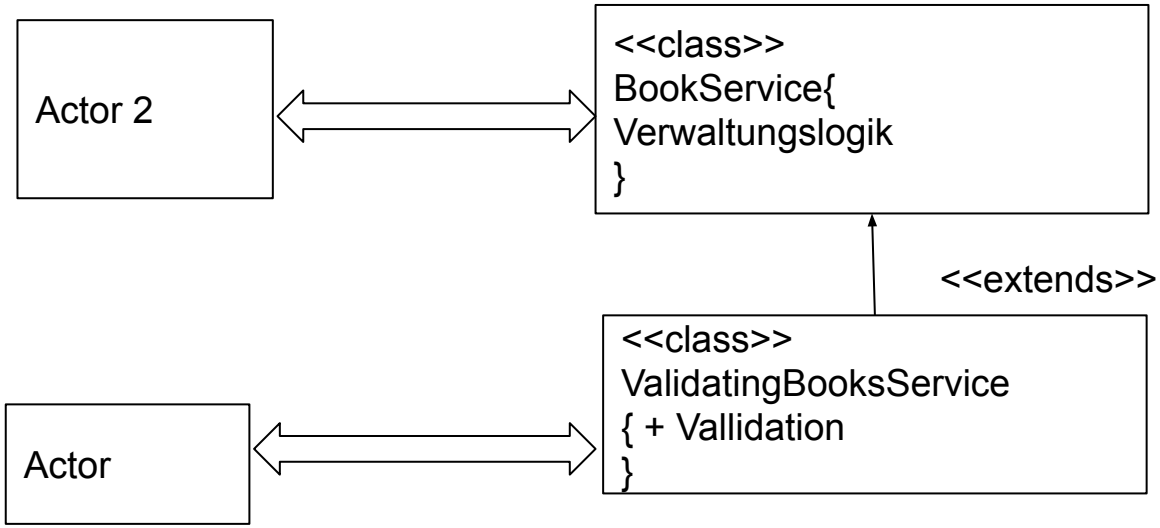
- Validierungslogik
- In Java-Code: SerializationUtil.clone



ToDo

- Wie schaut das Klasenmodell aus, das diese Verletzungen korrigiert?
-

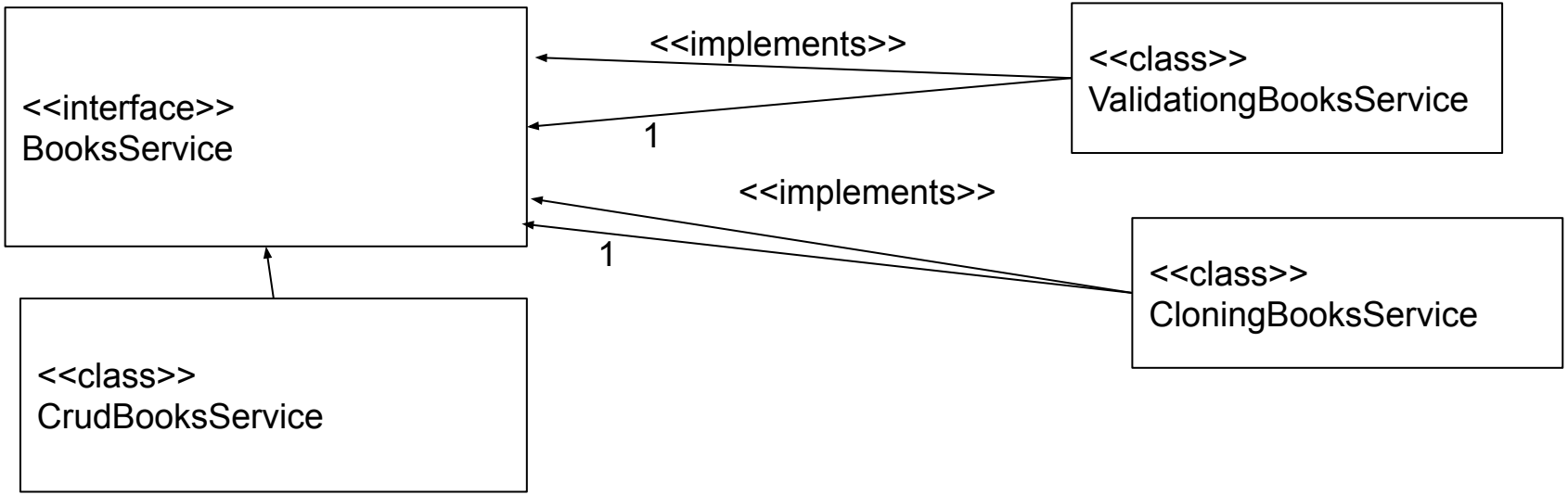
Muster: Ausgerichtet auf Vererbung



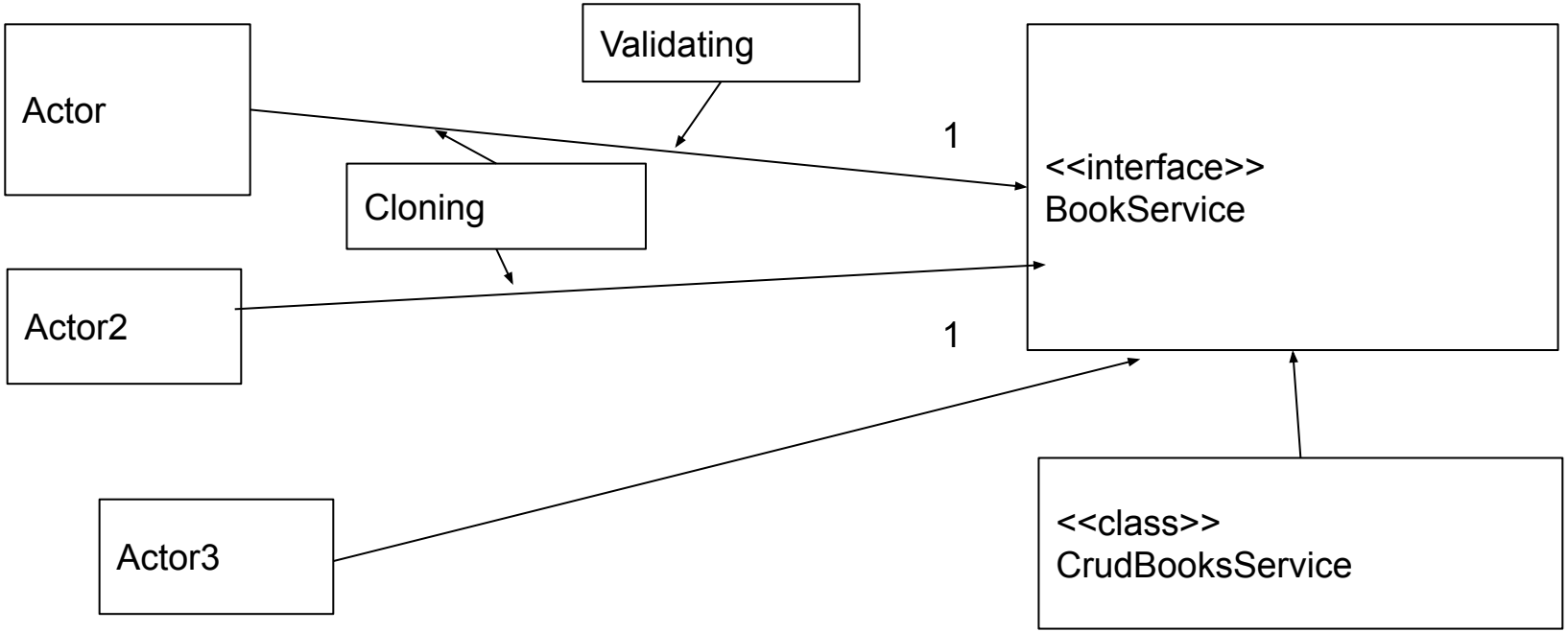
Eher so nicht zu realisieren

Starre Vererbungshierarchie ist meistens zu unflexibel

Muster: Decorator

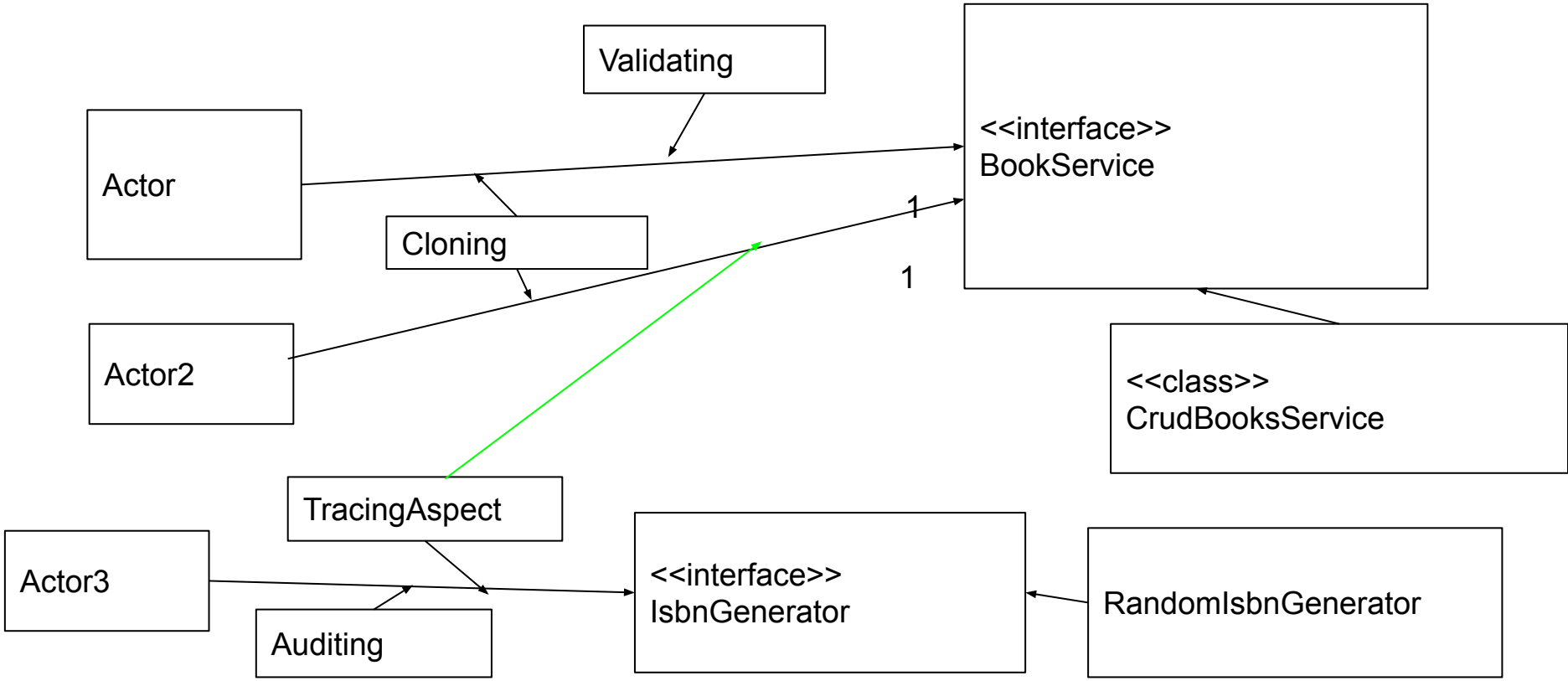


Klassendiagramm mit Actors und Decorators



Klassendiagramm mit Actors und Decorators und Aspect

integrata cegos



Was ist ein Aspekt?

integrata cegos

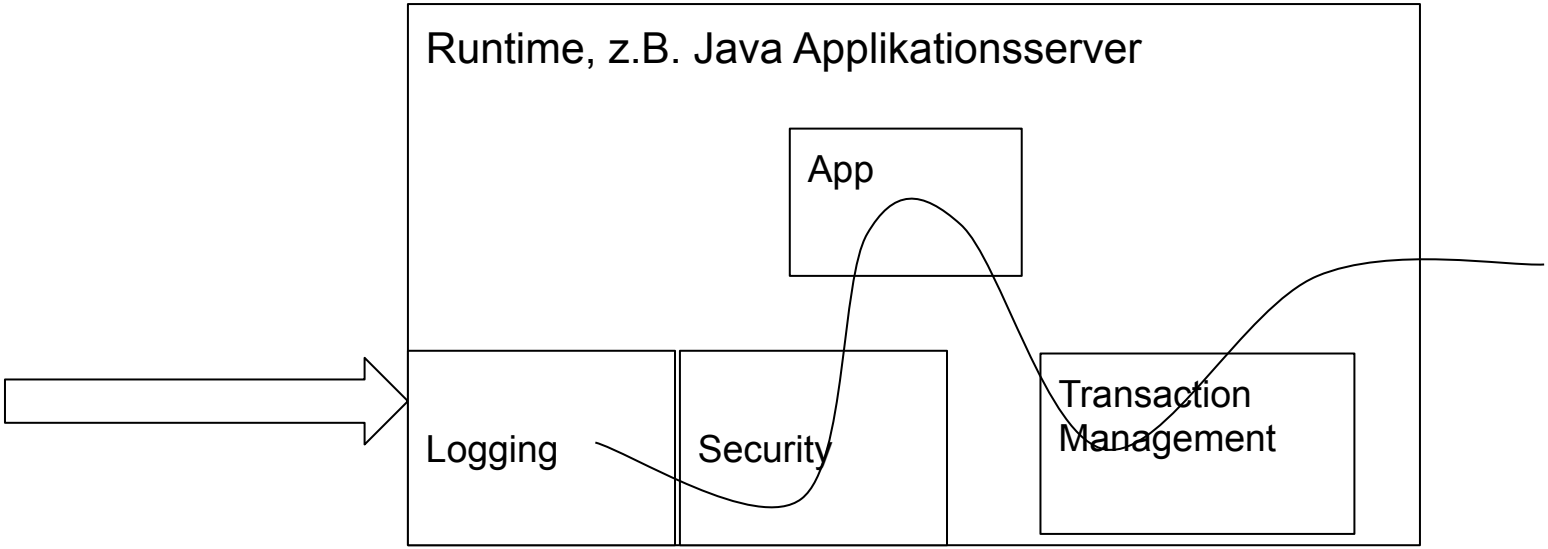
- Dekorationslogik
- Querschnittsfunktion über verschiedene Schnittstellen des Modells
  - Cross Cutting Concerns
- Beispiele
  - Tracing
  - Monitoring
  - Authentication and Authorization
  - Transaction Management

Aspekte und C++/Java

integrata cegos

- Statische Typisierung verhindert einfache Lösungen
  - Java:
    - AspectJ
    - Spring
    - Enterprise Edition
  - C++
    - ~~AspectC++~~
    - “Functional Decomposition”

Warum ist AOP nicht so direkt unterstützt?



Warum ist AOP nicht so direkt unterstützt, 2?

integrata cegos

- Aspekte können “unheimlich” sein, werden nicht unbedingt transparent eingeführt
- “Fachentwickler realisiert seine Fachklasse”
  - “Fehlerticket”: Du schreibst zu viel in die Konsole

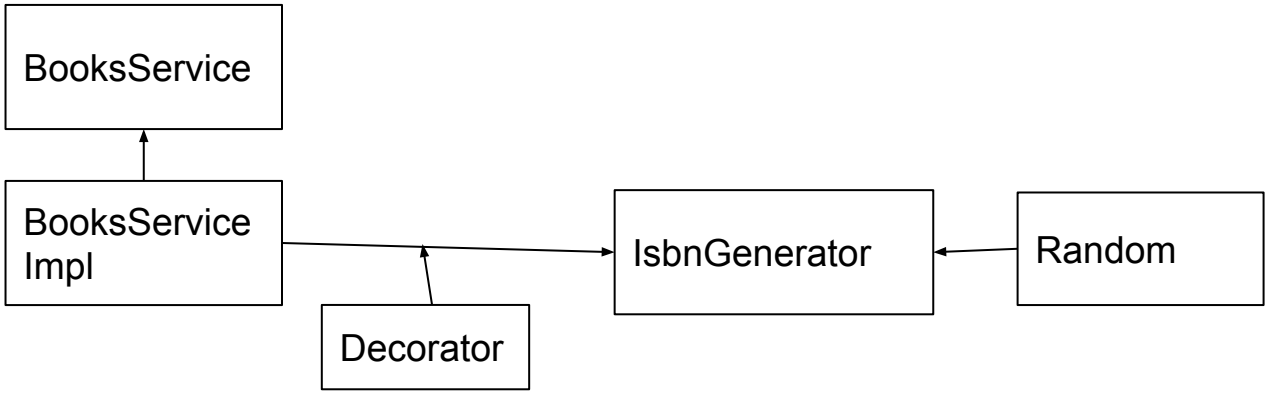
AOP ist gesetzt in...

integrata cegos

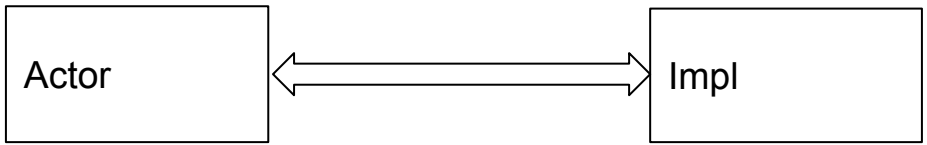
- Test und QS
- ~~if (MODE_DEBUG) … else if (MODE_TRACE)...~~

Aktueller Stand

- Vielzahl von implementierenden Klassen
  - Actors
  - Fachklassen
  - Decorators

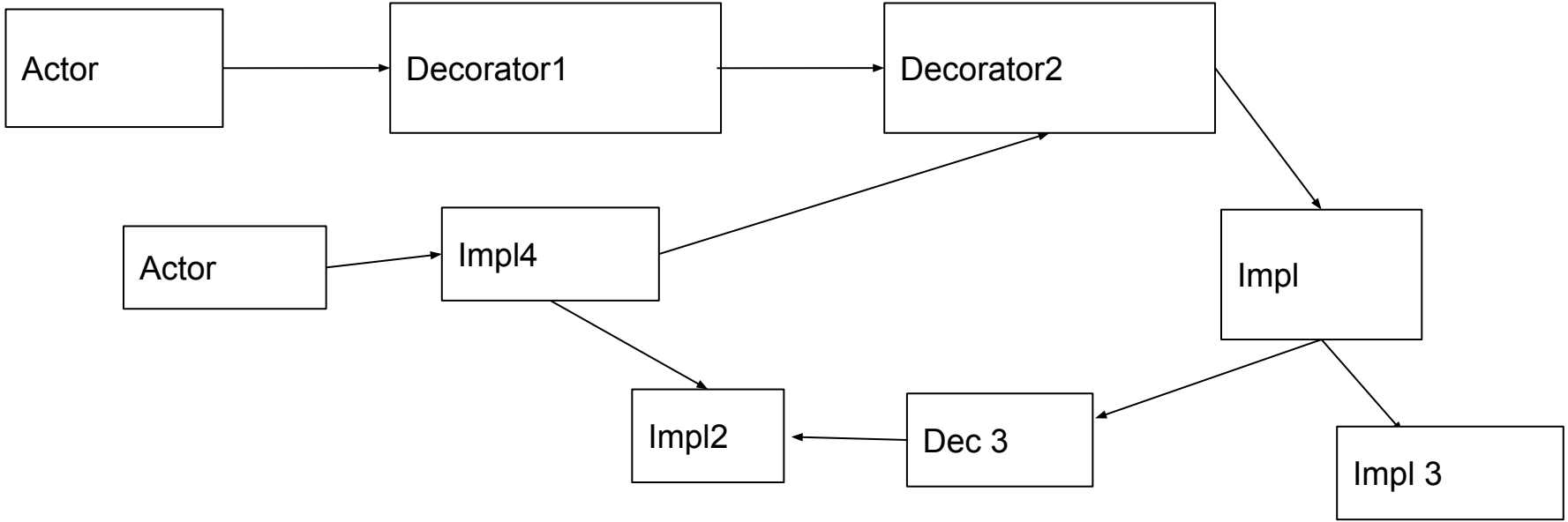


Vorher: Runtime



Aktueller Stand: Runtime

integrata cegos



Context & Dependency Injection

- Ursprünge von CDI entstammen den etablierten Pattern von Erich Gamma & Co
  - Strategy
  - Erzeugungsmuster
    - Factory
    - Singleton
- “Erfindung” war 2003 im Rahmen des Spring Frameworks in Java

Aufgaben des Context

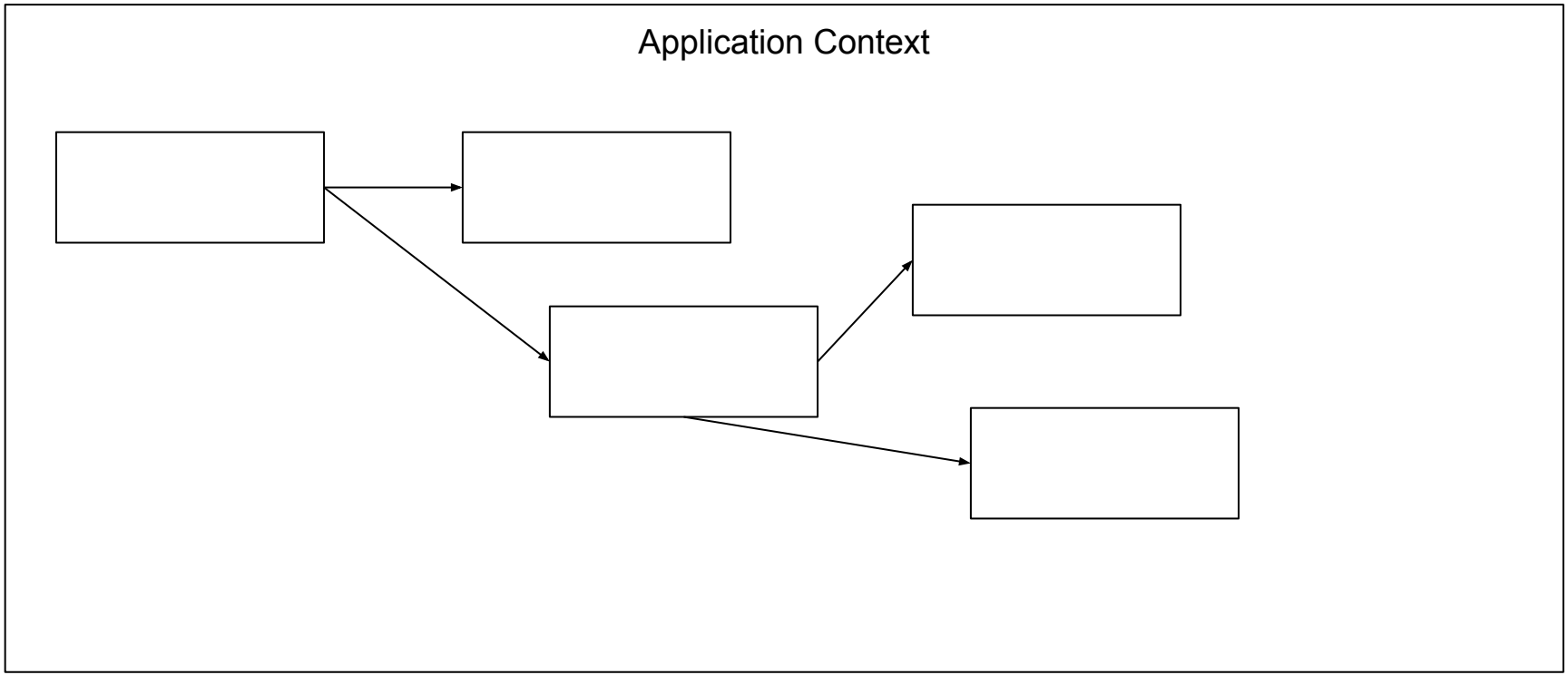
- Identifikation aller relevanter Fach-Klassen
- Erzeugung der zugehörigen Fach-Objekte unter Berücksichtigung eines “Scopes”
  - meistens der ApplicationScope
- Identifikation aller Dependencies eines Fach-Objekts
- Setzen, “Injecten” der Dependencies erfolgt zum Abschluss

Definition des Contexts

integrata cegos

- Nicht programmatisch, sondern deklarativ
- z.B.
  - Externe Konfigurationsdatei, z.B. XML
    - <object id=”impl1” class=”xyz” scope=“application”>
      - <property name=”myDependency” ref=”impl2” />
    - </object>
    - <object id=”impl2” class=”abc” scope=”application” />

Runtime mit Application Context



Refactoring

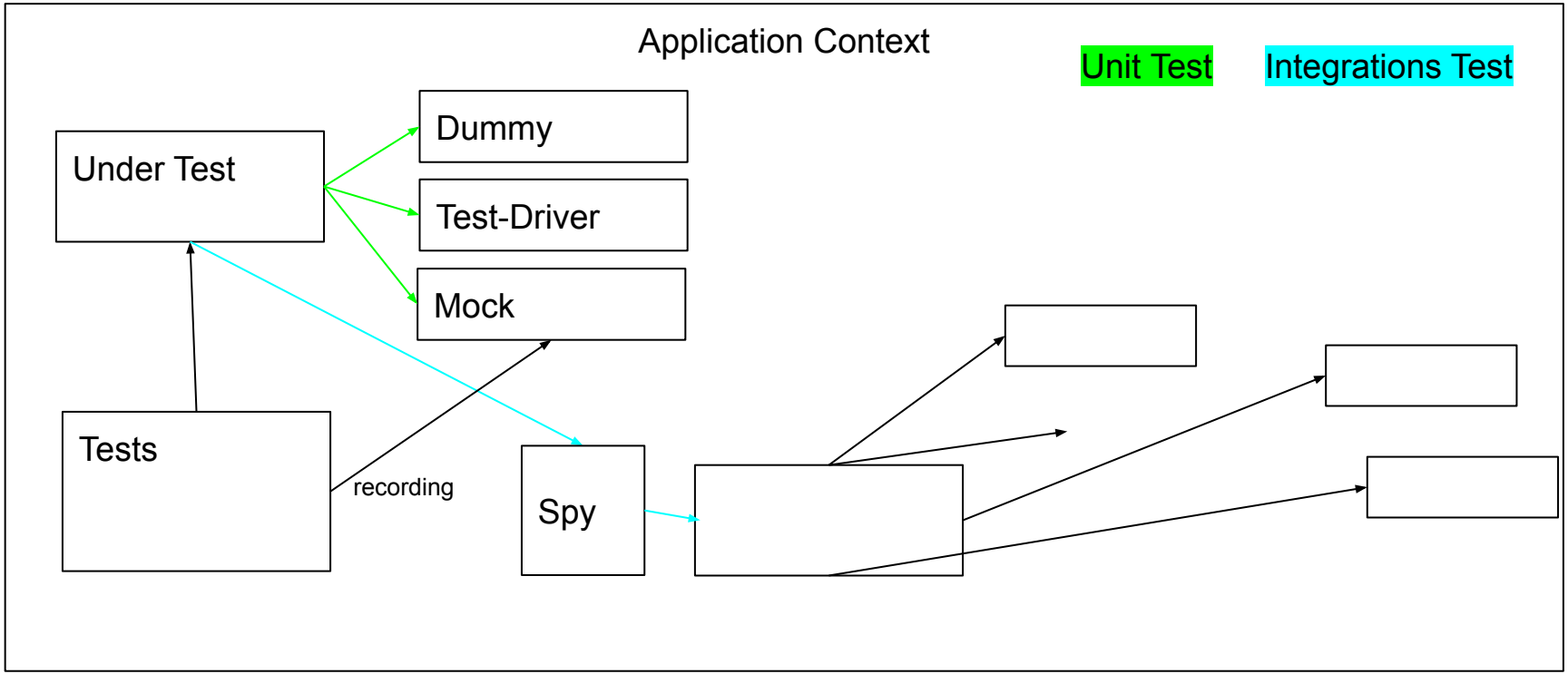
Grundvoraussetzung des Refactorings

- Funktionierende Codebasis
  - Verifizierung durch einen Satz erfolgreich bestätigter Tests

Grundziele des Refactorings

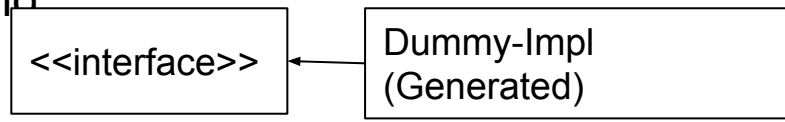
- Erzeugung qualitativ hochwertigen Codes ohne Verlust an Funktionalität
  - Automatisierbare Tests sollen dies verifizieren
    - Die Anwendung ist nach Context&Dependency Injection zu designen
    - class MyBusinessClass{
    - }
    - In einer anderen Anwendung
      - ~~new MyBusinessClass()~~

Eine testbare Applikation



Kategorisierung

- Dummy
  - Verlangt keinerlei eigene Implementierung
  - Liefert keine konsistenten Daten
- Test Driver
  - auch bekannt als Test Stubs
  - Bestimmen konsistente Daten
    - Damit faktisch eine Alternativ-Implementierung
  - Beträchtlicher Aufwand in Erstellung und Wartung
- Mock
  - Liefern konsistente Daten, die vom Unit-Test definiert werden
  - Phasen: Record versus Play
- Spy
  - Test-Decorator bzw. Test-Aspekt



Testing-Frameworks

integrata cegos

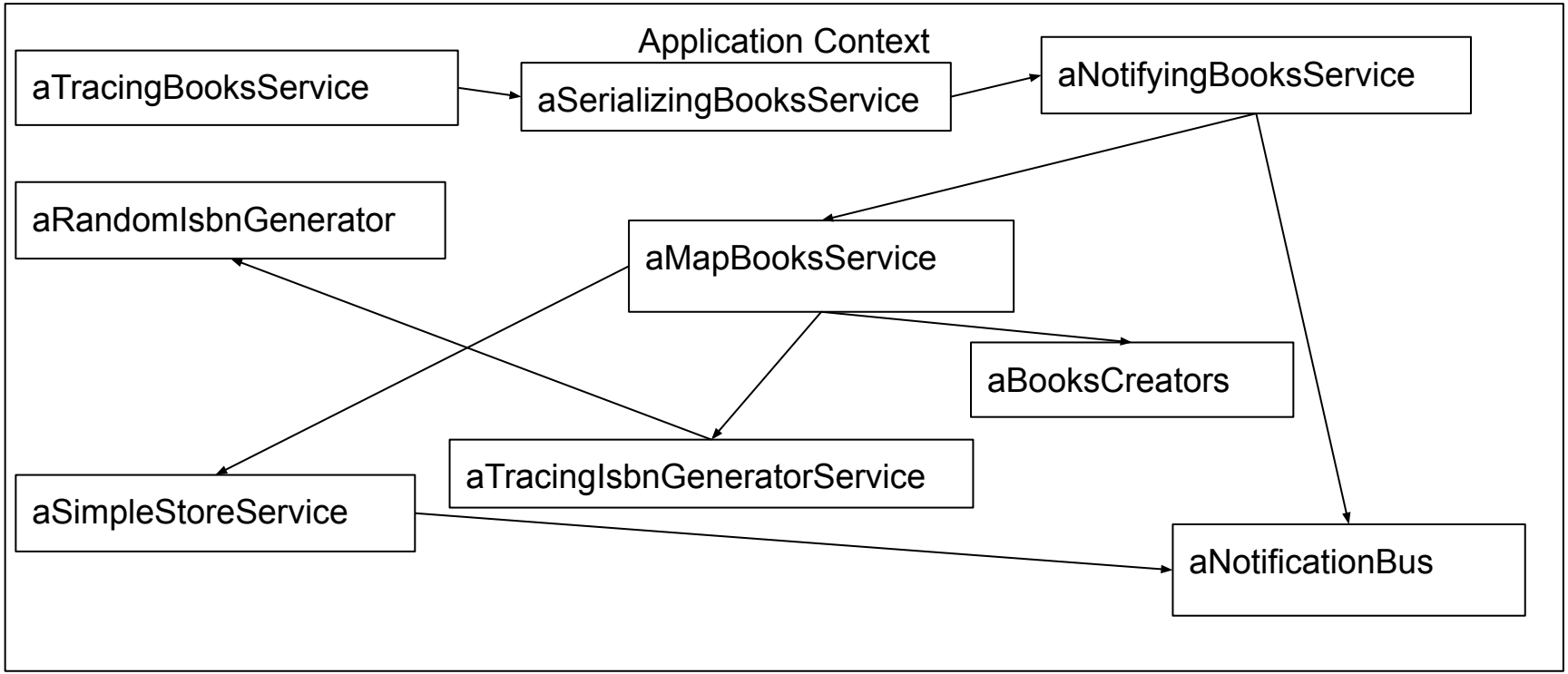
- Liefern
  - Dummies
  - Spies
  - Mocks

Grundvoraussetzung des Refactorings reviewed

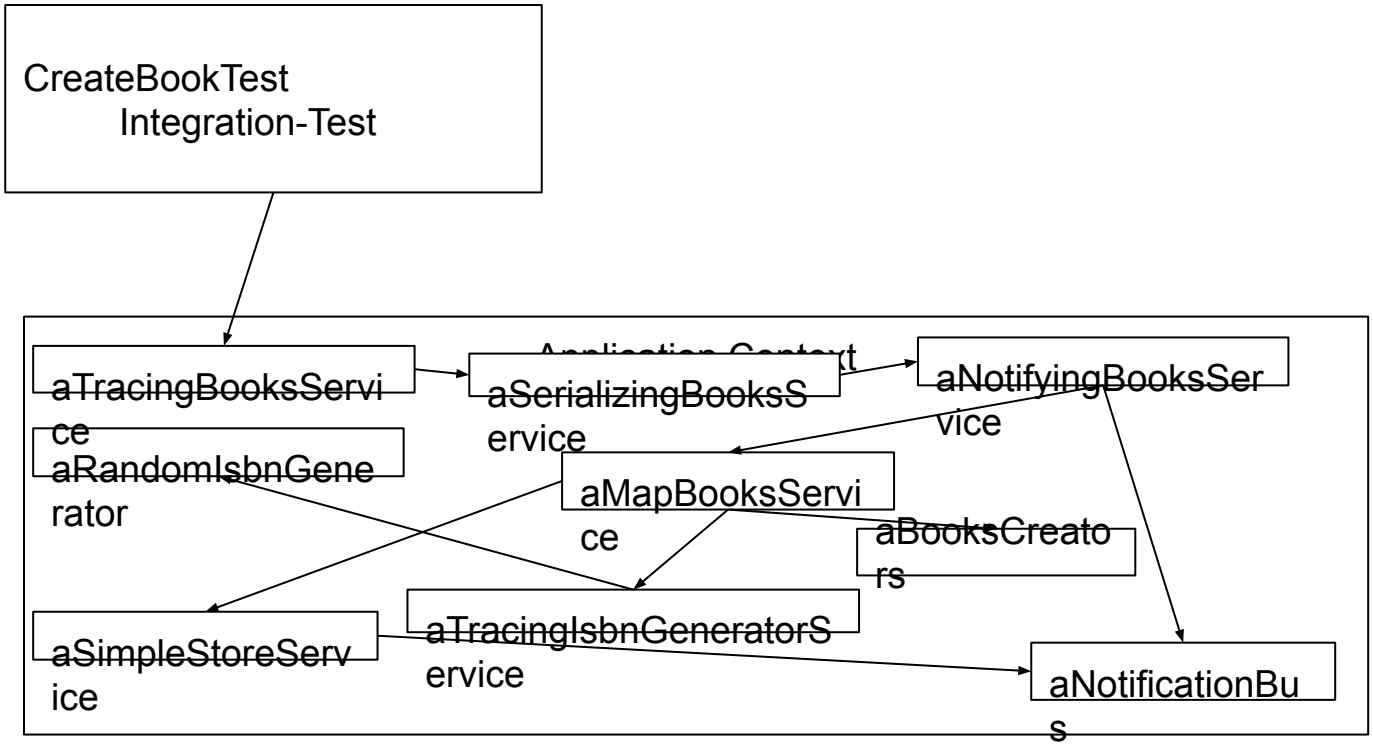
integrata cegos

- Funktionierende Codebasis designed nach CDI-Prinzipien
  - Verifizierung durch einen Satz automatisierter Unit-Tests + Automatisierte Integrationstest

ToDo: Application Context für die BooksApplication

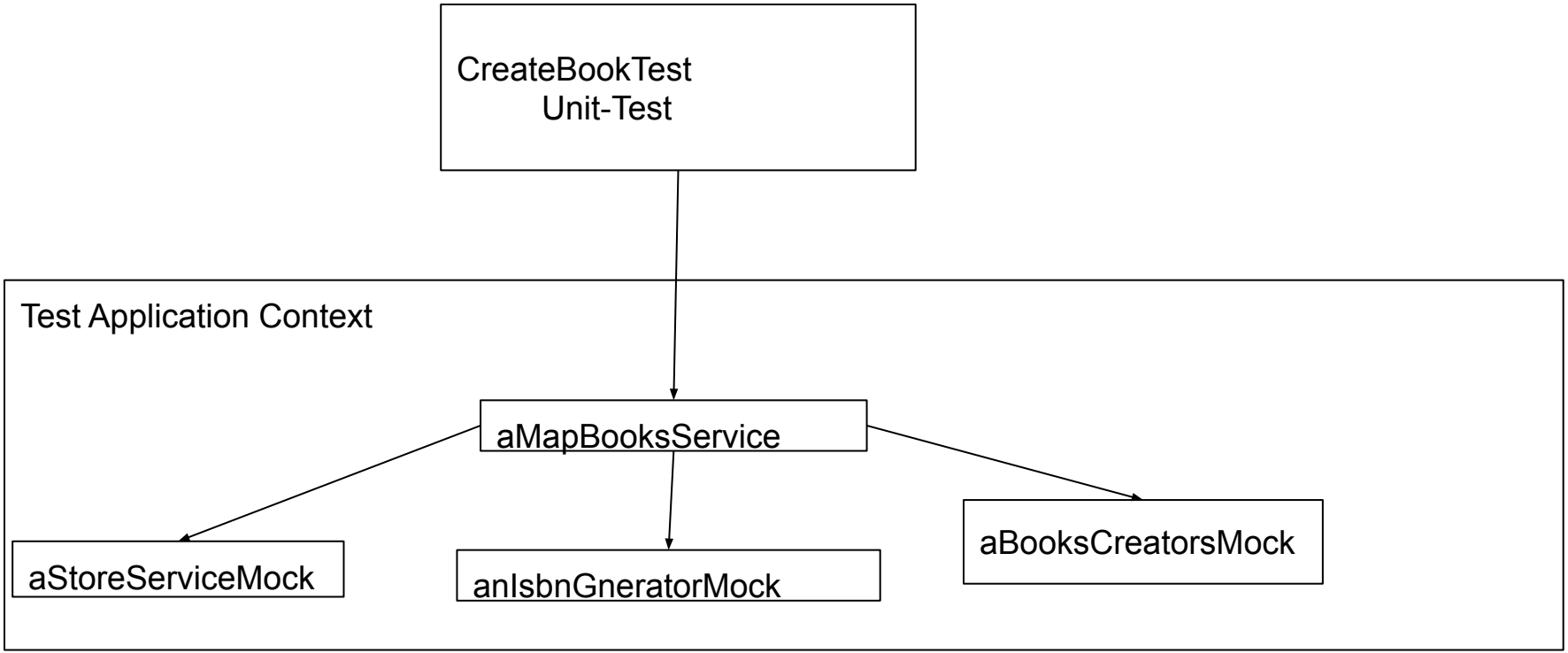


Test für den BooksService



51

Test für den BooksService



52

Clean Code

Einige Prinzipien

- SOLID
  - Single Responsibilty Principle
    - Konsequenz: Fein-granulare Konzeption unserer Klassen
  - Open-Closed-Principle
    - Entweder sind Klassen entweder final oder rein abstrakt (interfaces)
      - Relativ einfach automatisch zu bestimmen
        - z.B. Walker-Metrik
  - Liskov Substitution Principle
  - Interface Segregation Principle
  - Dependency Inversion Principle
    - CDI...

Liskov

- Kreis und Ellipse in einer Vererbungshierarchie
  - “Ein Kreis ist eine spezielle Ellipse”
  - Ellipse hat Methoden scaleX, scaleY
    - Ein Kreis erbt diese Methoden, die jetzt allerdings komplett sinnlos sind
    - ~~“Ein Kreis ist eine spezielle Ellipse”~~
    -

Interface Segregation Principle

integrata cegos

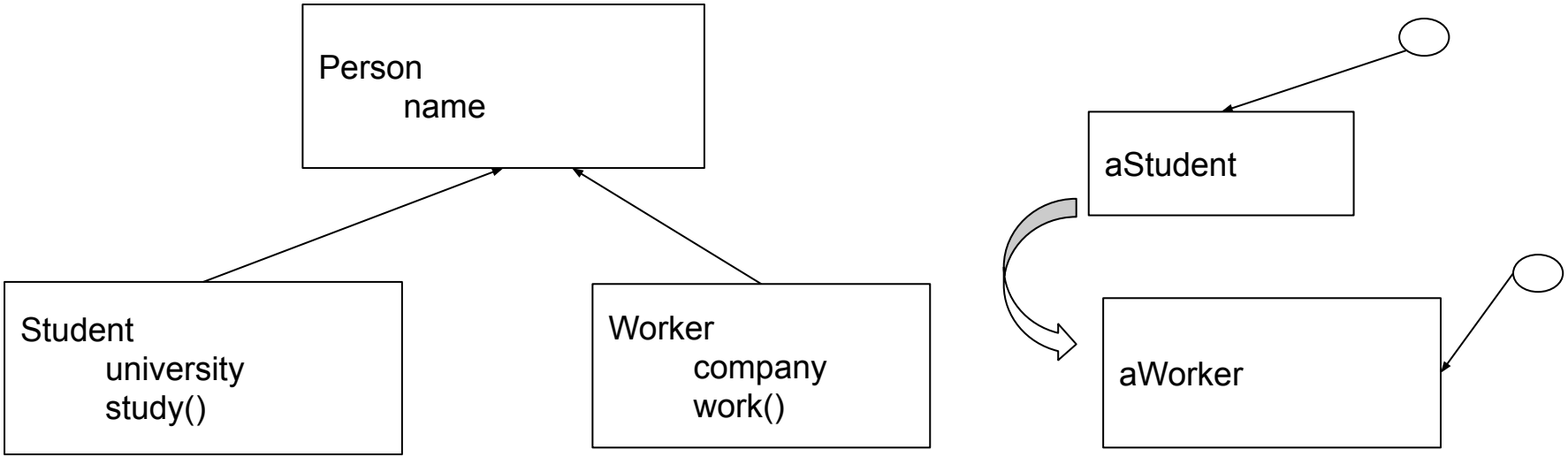
- CrudService
  - Create
  - Read
  - Update
  - Delete
- ReadService
  - Read
- MutableService
  - Create, Update
- AdminService
  - Delete

Einige Prinzipien, Teil 2

integrata cegos

- DRY
  - Don’t repeat yourself
- KISS
  - Keep it simple, stupid
    - Vorsicht: CDI IST SIMPLE!!!
  - Don’t optimize
- POLS
  - Principle of least surprise
    - Meine Formulierung: API-Dokumentation muss vollständig sein
- FCoI
  - Favour Composition over Inheritance
    - Ergänzung meinerseits: “bei statisch typisierten Sprachen”

FCoI

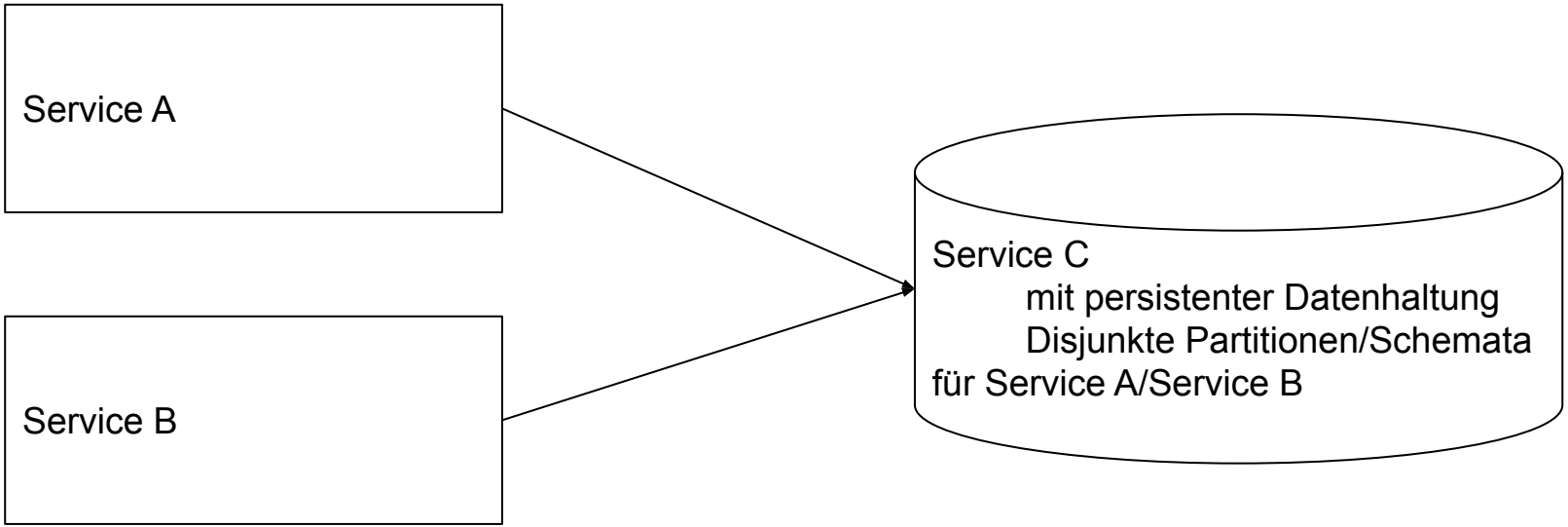

Person
name
Student
university
study()
Worker
company
work()
aStudent
aWorker

Exkurs: Microservices

Services

- Single Responsibility
  - “Jeder Service übernimmt eine wohl-definierte Aufgabe”
- “Context & Dependency Injection”
  - Service Oriented Architecture
  - Service Registry löst die Endpoints auf
-

Microservices



Weiter mit Clean Code

Kriterien für Clean Code

- Überlapp zu Code-Qualität
  - Schwer quantifizierbar/direkt messbar
  - Korrelation zu “Qualitäts-Metriken”
- Lesbarkeit/Verständlichkeit
  - Lines of Code
    - Hrair-Limit: 7
      - Ergebnis der Psychologie: “Das menschliche Gehirn ist durchschnittlich in der Lage, 7 Folgen/Zusammenhänge en Block zu erfassen”
  - Nur wenige if-else-Pfade
- Erweiterbarkeit
  - Robertson-Walker-Metrik
- Wiederverwendbarkeit
  - Eher auf Service-Ebene
- Testbarkeit
  - (Code Coverage)
  - Hot Spots, jeder erkannte Fehler muss einen Test bekommen

Refactoring Patterns

integrata cegos

- Katalog von “Best Practices”, die für verschiedene Situationen einen Vorschlag zur Verbesserung des Codes
- Nicht eindeutig…
  - fast jedes Pattern hat ein Reverses Pattern

Durchführung

- “Was wollen wir erreichen?”
  - Kompakt versus (potenziell) Wiederverwendbar
- Auswahl von Code-Metriken, die den Effekt des Refactorings belegen können
- Zum Schluss eine allgemeine, unabhängige Beurteilung der Ergebnisse

Typische Ergebnisse

- Identifikation weiterer Dependencies durch Wiederverwendbare Klassen/APIs
- Erhöhung der Anzahl von Unit-Tests
  - Kürzere Test-Ausführung
- Viel repräsentativere Tests durch spezifischere Assertions

Namens-Regeln

- Prefixes, “Ungarische Notation” etc. überflüssig
  - Statt Prefixes bitte sprechende Klassennamen und Namespaces/Packages benutzen
  - Duplikation vermeiden
    - class Person {String personName}
- Standard-Namen benutzen, Inkonsistenzen vermeiden
  - Bezug zur Aspektorientierten Programmierung
    - Joinpoints als “Wiedererkennbares Muster im Code”
    - Around(execution(* find*(..)), aspect=Cache)
    - Around(execution(* *Service.*(..)), aspect=Transactional)
- Trend zu Selbst-dokumentiertenden Code

Kommentare und Clean Code

integrata cegos

```
//In Arbeit, siehe Vorgabe unter...
//Version 1.1 erstellt am 10.11.2020 von Rainer Sawitzki
//Version 1.0 erstellt am 14.7.2015 von Egon Meier

class Person{
    // retrieves the name of person
    String getName() {return this.name;}
}
```

Namenskonventionen für Methoden

integrata cegos

- Sprechende Namen
  - ~~retrieveResult()~~
- calculatePrice(withTotalPrice:totalPrice, andDiscount:discount)
  - z.B. Apple Swift
- Spezialfall: Test-Methoden
  - statt testCreate
  - testCreatePersonWithLastnameSawitzkiCreatesPersonWithId42

Funktionale Programmierung

Funktionen sind Top-Level-Objekte

integrata cegos

- Eine Funktion ist nicht notwendigerweise als Bestandteil einer Klasse eine Methode
  - Kontext der Funktion ist eine Instanz einer Klasse
    - Kontext = this
- Impure Functions
  - Kontext der Funktion ist die “Umgebung” der Deklaration der Funktion
    - function outer(p){ var c = 42; function impureInner(){print(c)}}
- Pure Functions
  - Kennt “nur” die Aufruf-Parameter, es gibt keinen Zugriff auf den umhüllenden Kontext der Deklaration

Standardisierung

- Generator
  - Parameterlose Funktion mit Rückgabewert
- Unary Functions
  - Ein Parameter, ein Rückgabewert
  - Identisch zu einem Transformer
- Predicate
  - Eine spezielle unäre Funktion mit Rückgabetyp boolean
- Binary Function
  - Zwei Parameter, ein Rückgabewert

Datenverarbeitung mit “High Order Functions”

integrata cegos

- Nehmen andere Funktionen als Parameter entgegen
- Beispiel: Datencontainer, Collections
  - Array/List, Set, Map/Dictionary
  - Jede Collection kann
    - Selektion
      - filter (Predicate-Function)
      - sort (Unary Function mit Rückgabetyp int)
      - map (Binary Function)
        - entspricht einer Transformation
      - reduce(Binary Function, Parameter: Element der Collection, Aggregat: “Irgendwas”)
      - forEach/iterate
        - Unary Function, Parameter: Collection, kein Rückgabetyp

Clean Code und Functional

- Es gelten fast dieselben Prinzipien
  - Single Responsibility
  - Least surprise
-

Dies und Das

Clean Code und Design Patterns

- Anwendung, die nach gängigen Design Patterns konzipiert ist hat den Trend “Clean” zu sein

Literatur-Empfehlung

- Clean C++, Stephan Roth
  - https://www.springer.com/de/book/9781484227923