University of Basel

Software Engineering

Marcel Lüthi, Universität Basel

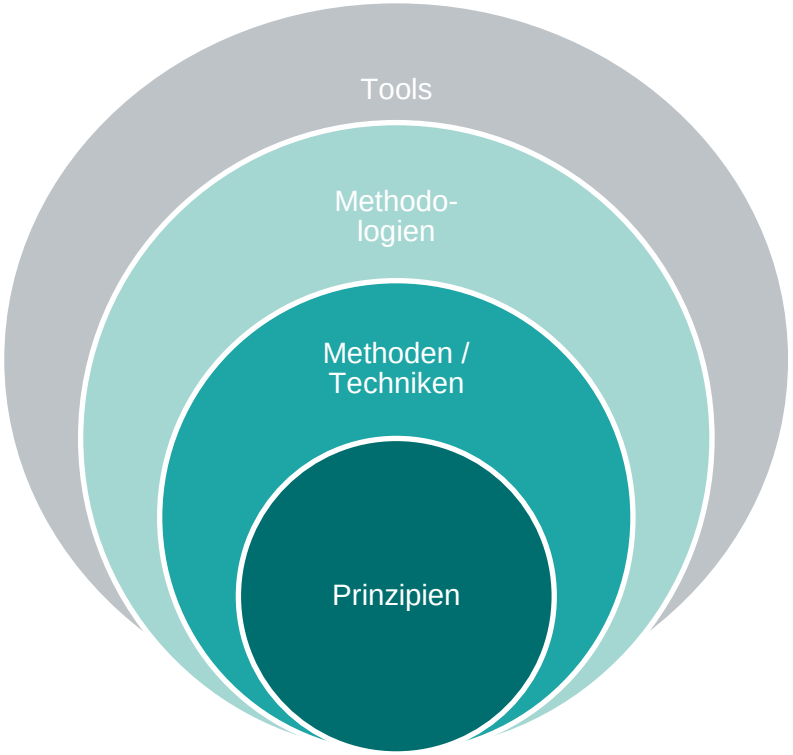
SOLID-Prinzipien

SOLID Prinzipien

- Single Responsibility Prinzip
- Open closed Prinzip
- Liskovsches Substitutionsprinzip
- Interface Segregation
- Dependency Inversion

Prinzipien zielen auf einfache Änderbarkeit durch Entkopplung der Module ab.

Standard Software-Engineering Prinzipien, angewendet auf OO



Single responsibility Prinzip

Es sollte nie mehr als einen Grund geben, eine Klasse zu ändern.

- Objekte sollten hohe Kohäsion aufweisen
- Jede Klasse sollte nur für eine Sache verantwortlich sein

Verwandtes Prinzip
- Separations of Concern

Beispiel: Single-responsibililty Prinzip

```
class BankAccount {

  private double balance;
  private Format format;

  void deposit(double amount) {
      //implementation
  }

  void formatAccountStatement() {
      // implementation
  }
}
```

Verantwortlichkeiten:
Kontoführung und Berichterstellung.

```
class BankAccount {

  private double balance;

  void deposit(double amount) {
       //implementation
  }
  void withdraw(double amount) {
       // implementation
  }
  boolean checkHasMoney() {
      // implementation
  }
}
```

Verantwortlichkeit:
Kontoführung

Open-closed Prinzip

Module sollten sowohl offen (für Erweiterungen), als auch geschlossen (für Modifikationen) sein.

- Ermöglicht System um neue Features zu erweitern, ohne ursprünglichen Code zu ändern.
  - Minimiert Risiko, dass existierende Funktionalität wegen Änderung nicht mehr funktioniert.

Verwandtes Prinzip
- Design for Change

Beispiel: Open-closed Prinzip

```
class Printer1 {
  void print(Document d);
}

class WordProcesser {
 ...

  void printDoc(Printer1 p) {
   p.print(document);
  }
}
```

```
interface Printer {
    void print(Document d);
}
class Printer1 implements Printer {
    void print(Document d) {}
}
class Printer2 implements Printer {
    void print((Document d) {}
}

class WordProcesser {
     void printDoc(Printer p) {
      p.print(document);
     }
   }
```

Änderung an Klasse Wordprocessor nötig 

Ursprünglicher Code bleibt unverändert 

Liskovsches Substitutionsprinzip

Sei $\phi(x)$ eine beweisbare Eigenschaft von Objekt x von Typ T. Dann soll $\phi(y)$ für Objekte y des Typs S wahr sein, wobei S ein Untertyp von T ist.

- Jedes Objekt kann durch ein Objekt der Subklasse ersetzt werden, ohne dass sich das Verhalten ändert.
- Kein überraschendes Verhalten → Einfacherer Code

- Problematisch zu prüfen bei nicht gut definierten Schnittstellen und Klassen
  - Braucht Spezifikation

(Negativ-)Beispiel: LSP

```
class Stack {
    private List data = new LinkedList();
    void push(Object o) {data.add(o);}
    int size() { return data.size(); }
 }
 class BoundedStack extends Stack {
     private Object[] data = new Array[10];
     void push(Object o) {
         if (data.size() < 10) {  data.add(o) }
     }
 }
 void testSize(Stack s) {
     int stackSizeBeforePush = s.size();
     s.push(o);
     assert(stackSizeBeforePush + 1 == s.size())
 }
```

`testSize(boundedStack)` schlägt fehl

Interface Segregation

Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden!

- Kleine Interfaces mit wohldefinierter Funktionalität sind zu bevorzugen
- Klassen werden entkoppelt

Verwandtes Prinzip

- Trennung der Verantwortlichkeiten

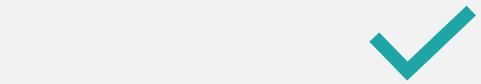
Beispiel: Interface Segregation

Negativbeispiel: `Printer` muss immer die `scan` und `print` Funktion anbieten

```
public interface DeviceOps {
   void print();
   void scan();
}

class Printer implements DeviceOps {
    …
}
```

Verbesserung: Separates Interface für pro Funktionalität

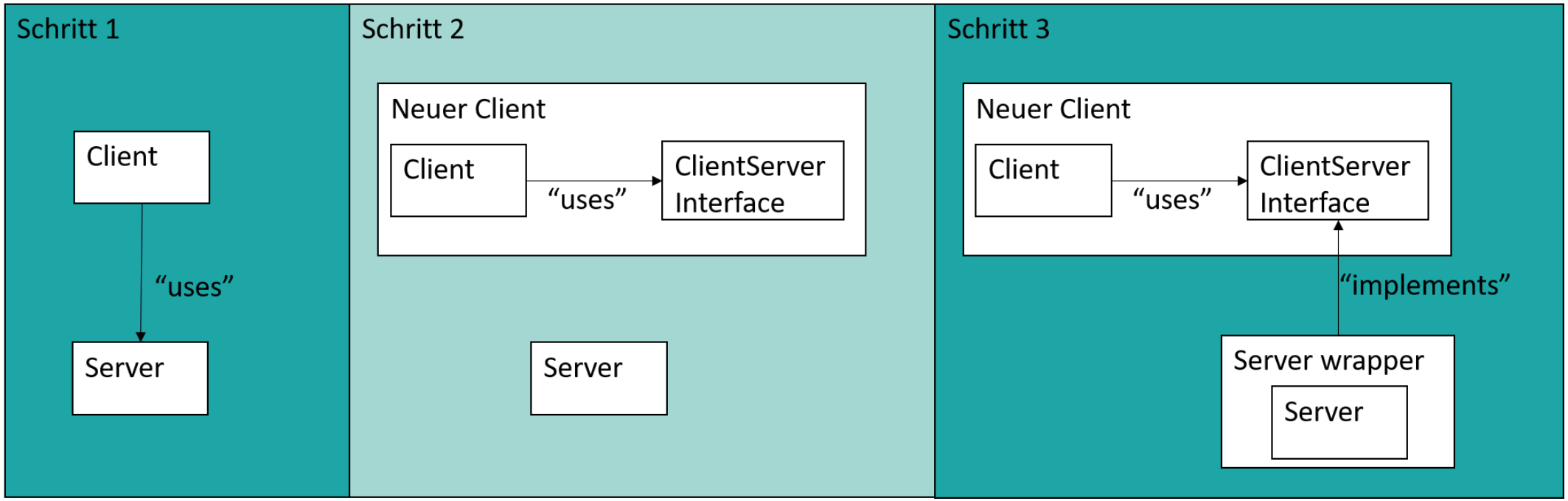
```
public interface PrinterOps {
   void print();
}

public interface ScannerOps {
   void scan();
}
```



Dependency inversion

- *Module hoher Ebenen sollten nicht von Modulen niedriger Ebenen abhängen.*
- *Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.*

Verwandte Prinzipien:
- Design for Change
- Modularität



Beispiel: Dependency inversion (DI)

```
class Logger {
   void log(String s) {…}
}

public class Account {
   private Logger logger = new Logger();

   void doTransaction(double amount) {
         logger.log("transferring" + amount);
         …
   }
}
```

Ohne Dependency inversion

```
public interface LoggingService {
    void log(String s);
}

public class Logger implements LoggingService
{
    @override void log(String s) {…}
}

public class Account {
   private LoggingService logger;

   Account(LoggingService logger) {
         this.logger = logger;
   }

   void doTransaction(double amount) {
         logger.log("transferring" + amount);
         …
   }
}
```

Mit Dependency inversion

Gesetz von Demeter

Eine Methode *m* einer Klasse *K* soll ausschließlich auf folgende Programm-Elemente zugreifen:

- Methoden von *K* selbst
- Methoden von Objekten, die als Argumente an *m übergeben werden*
- Methoden von Objekten, die in Instanzvariablen von *K* abgelegt sind
- Methoden von Objekten, die *m* erzeugt

Merksatz: Only talk to your immediate friends!

Beispiel: Gesetz von Demeter

```
class Engine {
    public void start() {// starts engine}
}

class Car {
    private Engine engine;
    public Car() { engine = new Engine(); }
    public getEngine() { return engine; }
}

class Driver {
    public void drive() {
        Car car= new Car();
        car.getEngine().start();
    }
}
```

Gesetz von Demeter wird verletzt

```
class Engine {
    public void start() {// starts engine}
}
class Car {
    private Engine engine;
    public getEngine() { return engine; }
    public Car() { engine = new Engine(); }
    public turnOn() { engine.start(); }

}
class Driver {
    public void drive() {
        Car car= new Car();
        car.turnOn();
    }
}
```

Gesetz von Demeter wird eingehalten