DHBW
Duale Hochschule
Baden-Württemberg
Karlsruhe

PROGRAMMIER PRINZIPIEN

Lars Briem

(briem.lars@googlemail.com)

Duale Hochschule Baden Württemberg - Standort Karlsruhe

Prinzipien in der Softwareentwicklung

- Prinzipien sind eine Art Leitfaden
  - Programmierprinzipien sind ein Leitfaden für die Entwicklung
- Entstehen aus jahrelanger Erfahrung und Diskussion
  - Werden von vielen Menschen mitgestaltet

Prinzipien in der Softwareentwicklung

- Verantwortung aufteilen
  - Festlegen, wer für was verantwortlich ist
- Prinzipien sind allgemeiner als Muster
- SOLID, GRASP, SLAP, DRY, ...

SOLID

- **S**ingle Responsibility Principle
- **O**pen Closed Principle
- **L**iskov Substitution Principle
- **I**nterface Segregation Principle
- **D**ependency Inversion Principle

Single Responsibility Principle

- Eine Klasse sollte nur eine Ursache oder einen Grund haben sich zu ändern
- Niedrige Komplexität und Kopplung
- *Was macht die Klasse?*
  - Konjunktionen in Antwort deuten auf mehrere Zuständigkeiten hin

Single Responsibility Principle

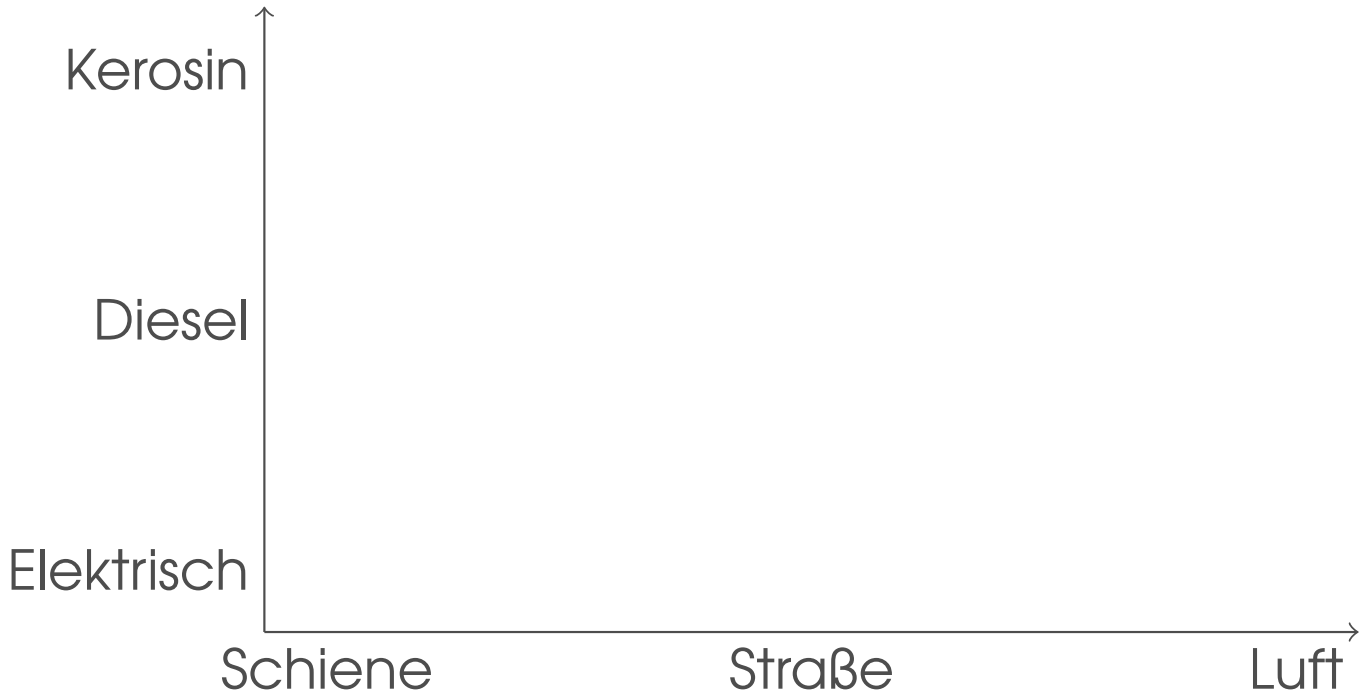
- Jede Klasse sollte nur eine Zuständigkeit haben
  - Eine Klasse erhält eine klar definierte Aufgabe
  - Komplexeres Verhalten entsteht durch Kombination mehrerer Objekte
- Eine Klasse enthält *Achsen* auf der sich Anforderungen ändern können
  - Jede Zuständigkeit fügt eine weitere Achse hinzu
  - Jede Klasse sollte nur eine Achse haben

Single Responsibility Principle

Änderungsdimensionen am Beispiel eines Fahrzeugs in der Simulation

Single Responsibility Principle

Änderungsdimensionen am Beispiel eines Fahrzeugs in der Simulation

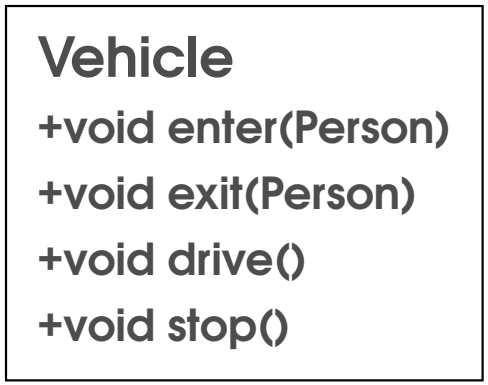


Single Responsibility Principle

Änderungsdimensionen am Beispiel eines Fahrzeugs in der Simulation

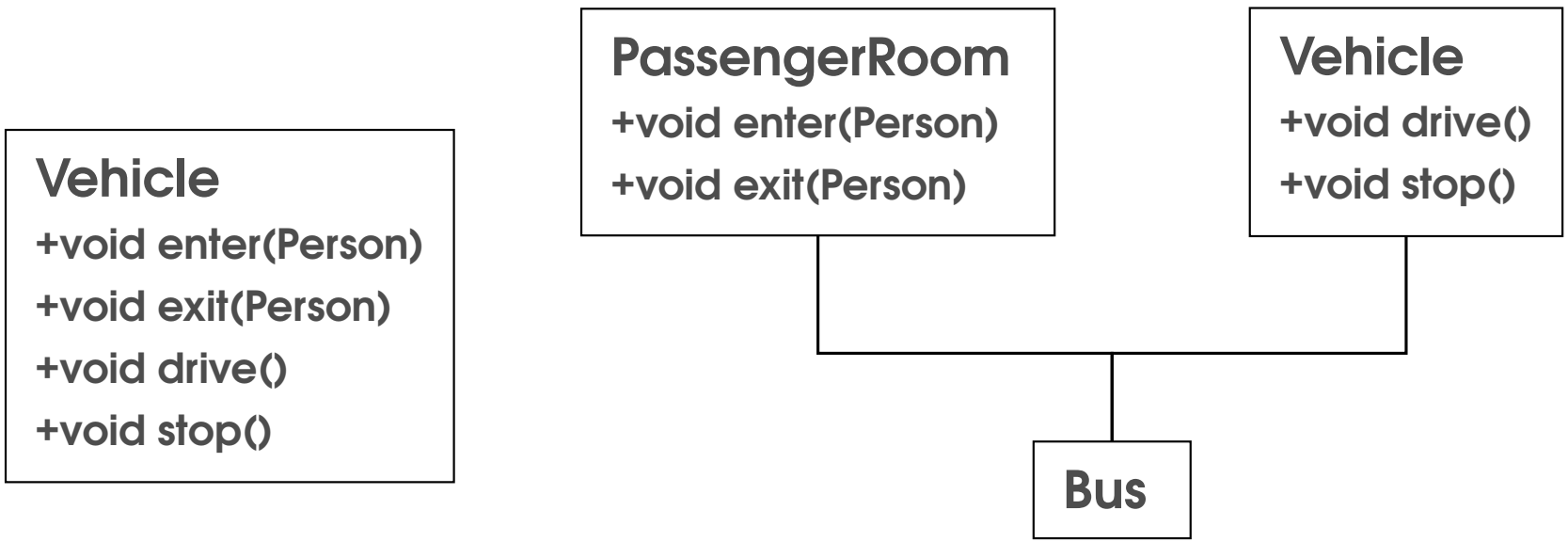
Single Responsibility Principle

Beispiel für ein Fahrzeug



Single Responsibility Principle

Beispiel für ein Fahrzeug



Open Closed Principle

- Elemente der Software wie Klassen, Module, Funktionen sollten . . .
  - offen für Erweiterungen und
  - geschlossen für Änderungen sein
- Erweiterungen durch Vererbung bzw. Implementierung von Interfaces
  - Neue Unterklasse mit angepasstem Verhalten ergänzen
- Bestehender Code wird nicht geändert

Open Closed Principle

Beispiel mit Switch/If-Else Kaskade

```
public class CommentChecker
  boolean isValid(Comment comment) {
    if(comment.isEmpty()) {
      return false;
    }
    if(!comment.isUnique()) {
      return false;
    }



    return true;
  }
}
```

Open Closed Principle

Beispiel mit Switch/If-Else Kaskade

```
public class CommentChecker
  boolean isValid(Comment comment) {
    if(comment.isEmpty()) {
      return false;
    }
    if(!comment.isUnique()) {
      return false;
    }
    if(comment.isSpam()) {
      return false;
    }
    return true;
  }
}
```

Open Closed Principle

Beispiel mit Switch/If-Else Kaskade

```
public class CommentChecker
  boolean isValid(Comment comment) {
    if(comment.isEmpty()) {
      return false;
    }
    if(!comment.isUnique()) {
      return false;
    }
    if(comment.isSpam()) {
      return false;
    }
    return true;
  }
}
```

Überprüfung auf Spam ist eine Änderung

Open Closed Principle

Beispiel mit Switch/If-Else Kaskade

```
public class CommentChecker
  private List<CommentRule> rules;

  boolean isValid(Comment comment) {
    for(CommentRule rule : rules) {
      if(rule.isObservedBy(comment)) {
        return false;
      }
    }
    return true;
  }
  ...
}
```

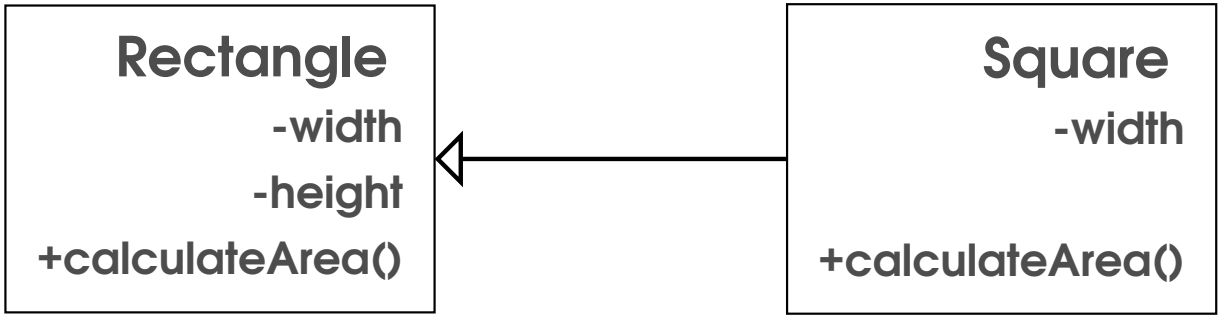
Open Closed Principle

- Abstraktionen fördern die Erweiterbarkeit
  - Zu viele Abstraktionen sind ebenfalls schlecht
  - Erfahrung in der Domäne und der Umsetzung sind vorteilhaft
- Software ist nie immun gegen Änderungen
- Der Entwickler entscheidet . . .
  - welche Erweiterungen möglich sind
  - was durch Änderungen ergänzt werden soll
- Stabilität einer Klasse ist ausschlaggebend

Liskov Substitution Principle

- Objekte eines abgeleiteten Typs müssen als Ersatz für Instanzen ihres Basistyps funktionieren ohne die Korrektheit des Programms zu ändern
- Starke Einschränkung der Ableitungsregeln
- Führt zur Einhaltung von Invarianzen
- Anschauliches Beispiel bei der Flächenberechnung
  - Quadrat erbt von Rechteck

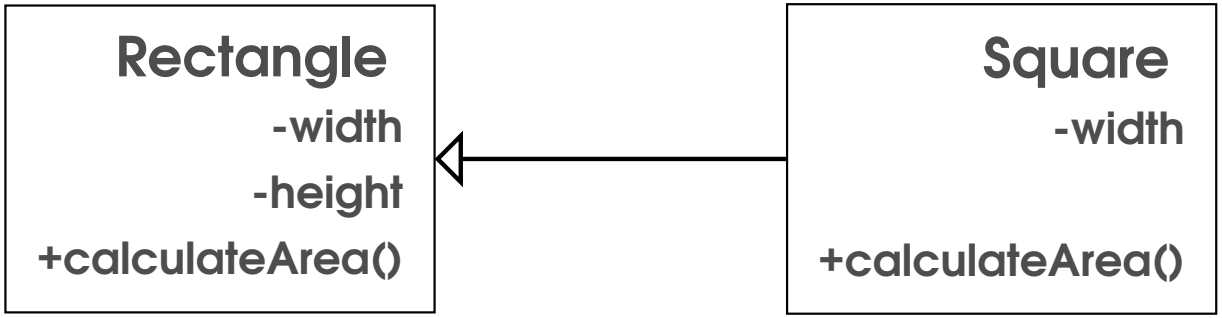
Liskov Substitution Principle




```
class AreaCalculator {
  void calculate() {
    int width = 2;
    int height = 3;
    Rectangle rectangle = new Rectangle(width, height);
    Area area = rectangle.calculateArea();

    assertThat(area, is(equalTo(new Area(6))));
  }
}
```

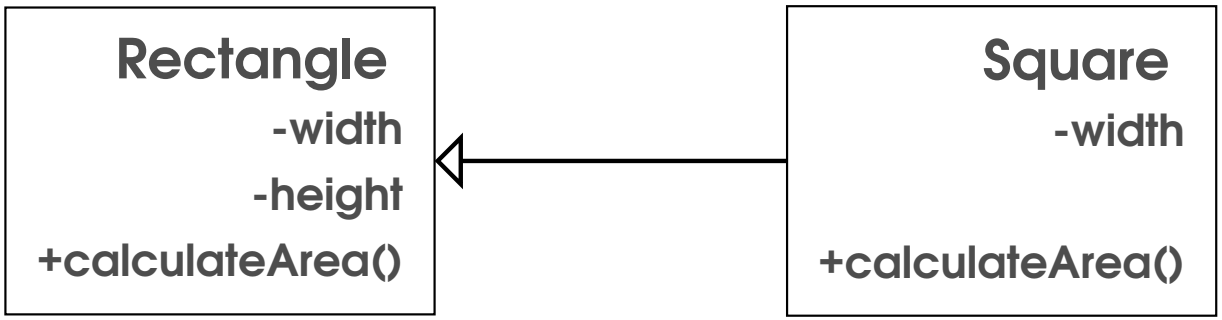
Liskov Substitution Principle




```
class AreaCalculator {
  void calculate() {
    int width = 2;
    int height = 3;
    Rectangle rectangle = new Square(width);
    Area area = rectangle.calculateArea();

    assertThat(area, is(equalTo(new Area(6))));
  }
}
```

Liskov Substitution Principle




```
class AreaCalculator {
  void calculate() {
    int width = 2;
    int height = 3;
    Rectangle rectangle = new Square(width);
    Area area = rectangle.calculateArea();

    assertThat(area, is(equalTo(new Area(6))));
  }
}
```

Fläche stimmt nicht

Liskov Substitution Principle

- Invarianzen von Klassen berücksichtigen
  - Abgeleitete Typen müssen schwächere Vorbedingungen haben
  - Abgeleitete Typen müssen stärkerer Nachbedingungen haben
    - `assert (width == newWidth) && (height == old.width)`
- *Design by Contract* kann helfen Verstöße zu finden
- Ableitung in OOP ist mehr eine "verhält sich wie" Beziehung anstatt einer "ist ein" Beziehung

Interface Segregation Principle

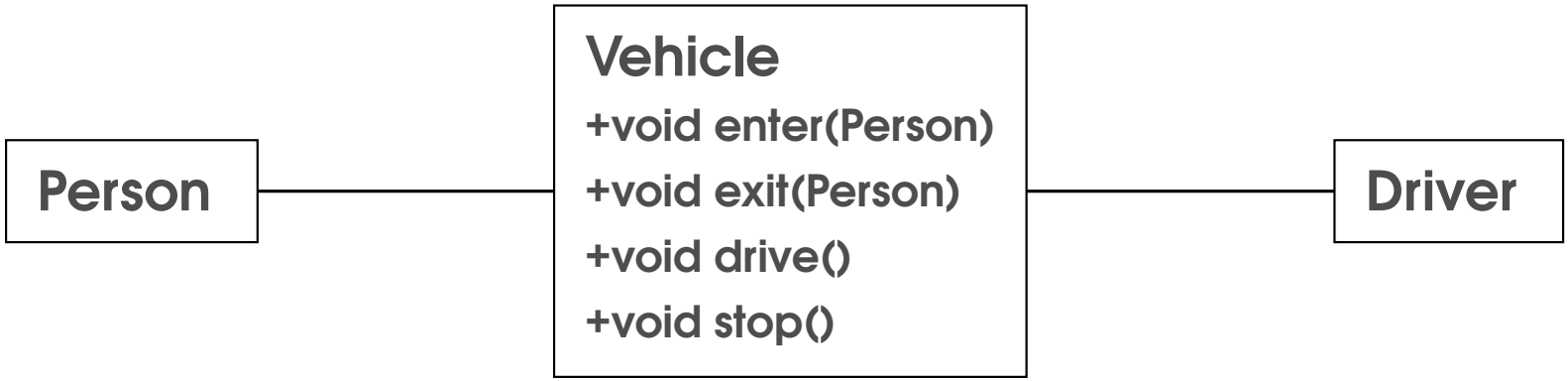
- Anwender sollten nicht von Funktionen abhängig sein, die sie nicht nutzen
- *Schwere* (fat) Interfaces und Klassen bündeln viel Funktionalität
  - Ein Anwender einer Methode eines Interfaces ist automatisch abhängig von Änderungen an anderen Methoden des Interfaces
  - Ein Anwender hat Zugriff auf Methoden, die nicht für ihn bestimmt sind
- Interfaces passend zu den Anwendern gestalten

Interface Segregation Principle

- Führt dazu, dass Typen meist mehrere Interfaces implementieren
  - In Java z.B. Clonable, Serializable, Comparable
- Ein Typ bedient dadurch mehrere Anwender
- Schwere Klassen können nach wie vor bestehen, aber Anwender ist nur von leichten Interfaces abhängig

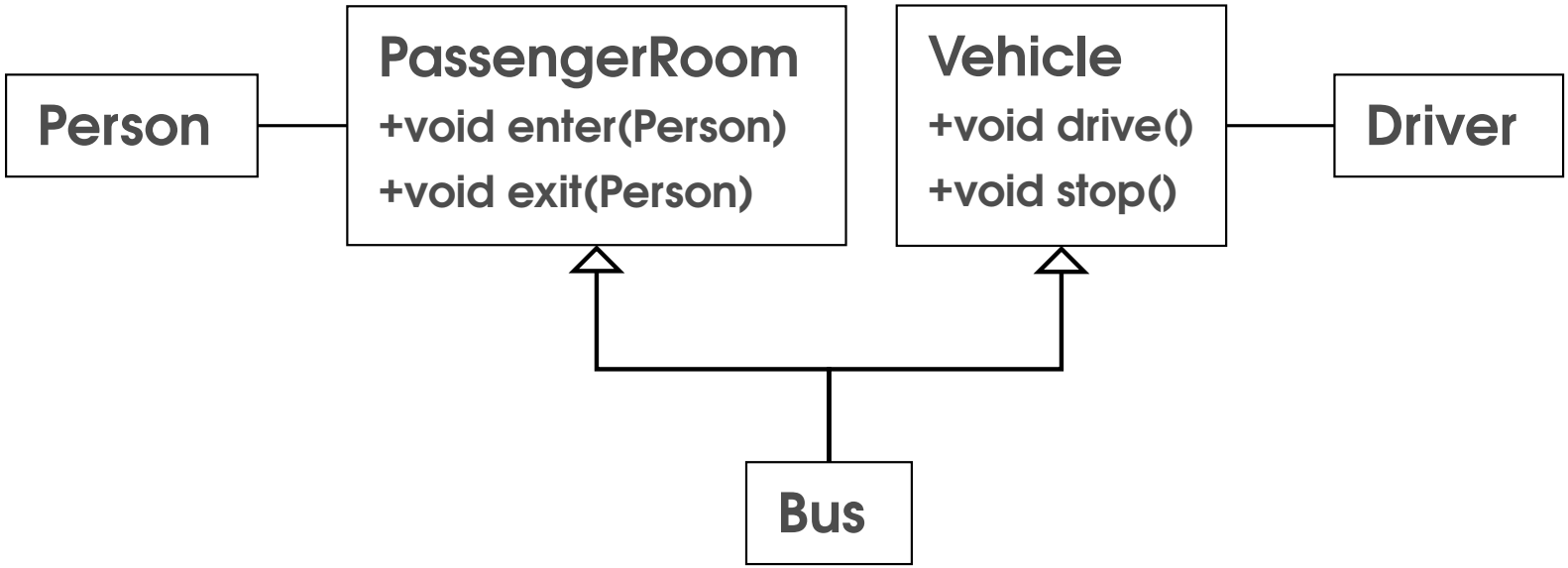
Interface Segregation Principle

Beispiel für ein Fahrzeug



Interface Segregation Principle

Beispiel für ein Fahrzeug



Dependency Inversion Principle

- Klassischerweise sind High-Level Module von Low-Level Modulen abhängig
  - Änderung in einer Low-Level Implementierung führt zu Änderung in High-Level Modul
  - Änderung in High-Level Modul führt eventuell zu Änderung in anderen Low-Level Modulen

$\Rightarrow$ Umkehrung (Inversion) der Abhängigkeit

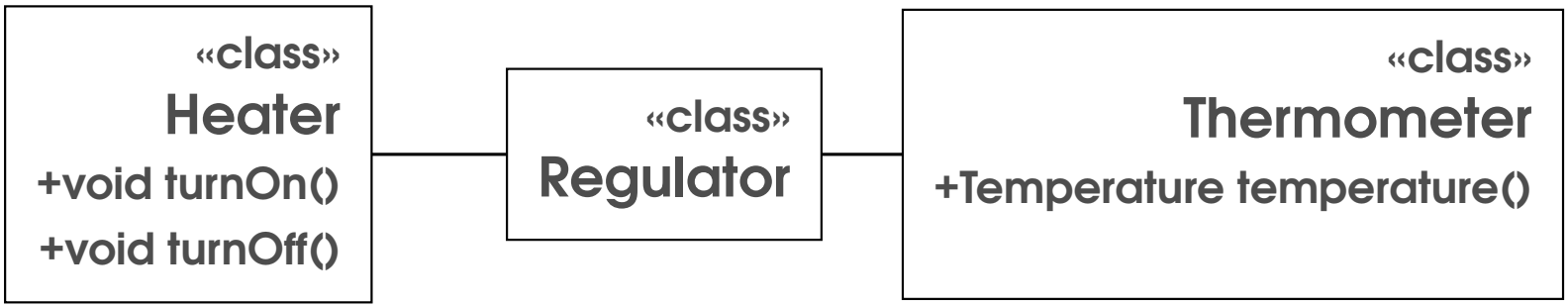
Dependency Inversion Principle

- High-Level Module sollten nicht von Low-Level Modulen abhängig sein. Beide sollten von Abstraktionen abhängen.
- Abstraktionen sollten nicht von Details abhängig sein. Details sollten von Abstraktionen abhängen.

Dependency Inversion Principle

- Regeln werden durch High-Level Module vorgegeben
- Low-Level Module sind Implementierungen der Regeln
- High-Level Module können wiederverwendet werden
  - High-Level Module bilden ein Framework

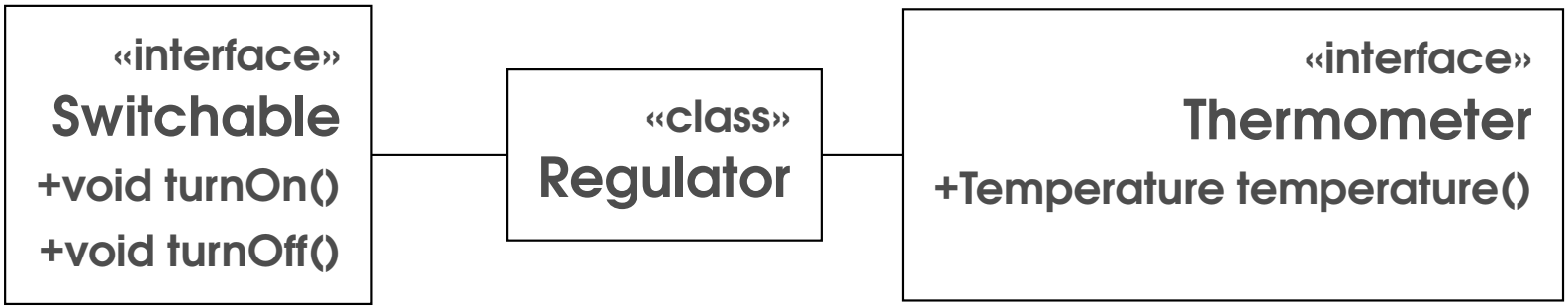
Dependency Inversion Principle




```
class Regulator {
  Heater heater;
  Thermometer current;
  ...

  void regulate(Temperature min, Temperature max) {
    if (current.temperature().isLowerThan(min)) {
      heater.turnOn();
    }
    else if (current.temperature().isHigherThan(max)) {
      heater.turnOff();
    }
  }
}
```

Dependency Inversion Principle

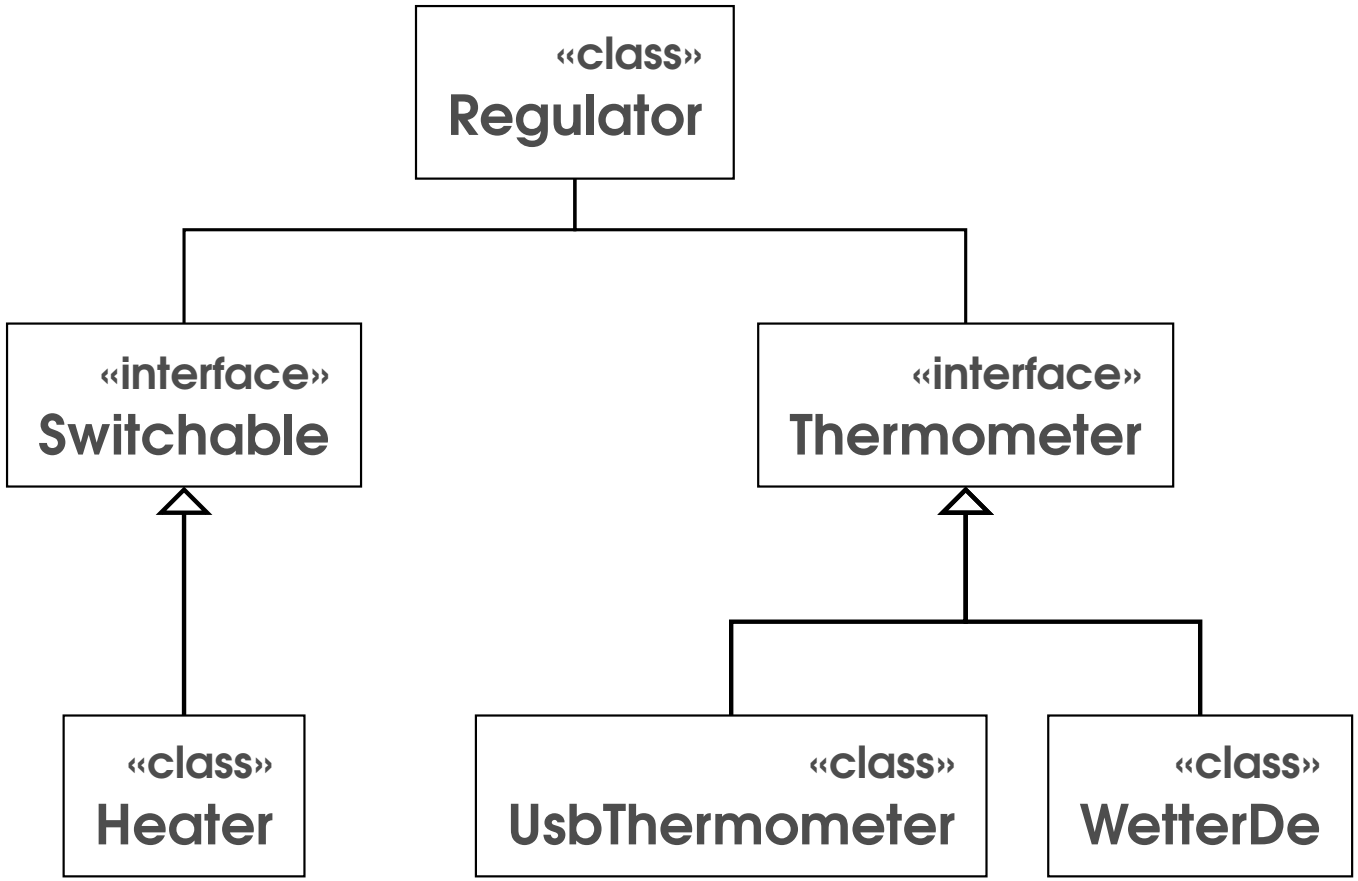



```
class Regulator {
  Switchable device;
  Thermometer current;
  ...

  void regulate(Temperature min, Temperature max) {
    if (current.temperature().isLowerThan(min)) {
      device.turnOn();
    }
    else if (current.temperature().isHigherThan(max)) {
      device.turnOff();
    }
  }
}
```

Dependency Inversion Principle

UML Klassen Diagramm



Dependency Inversion Principle

- Immer nur von Abstraktionen abhängig sein bedeutet . . .
  - Variablen oder Member sollten eine abstrakte Klasse oder ein Interface als Typ haben
  - Klassen sollten nur abstrakte Klassen oder Interfaces ableiten bzw. implementieren
  - Nur abstrakte Methoden implementieren
    - Methoden eines Interfaces sind abstrakt
- Beim initialen Aufbau der Anwendung werden Instanzen konkreter Klassen erzeugt
  - Bei Dependency Injection Frameworks passiert dies zum Teil in Konfigurationsdateien

Tell, don't ask

- Procedural Code gets information then makes decisions. Object oriented code tells objects to do things.
- Prozeduraler Code holt sich Informationen und trifft dann Entscheidungen. Objektorientierter Code sagt Objekten, dass sie etwas ausführen sollen (gibt Anweisungen).

Tell, don't ask

- Prozedurale Vorgehensweise
  - Status eines Elements abfragen
  - Entscheidung treffen
  - Element etwas ausführen lassen

$\Rightarrow$ Prozedurale Vorgehensweise führt zu zentralisierter Businesslogik

Tell, don't ask

- Objektorientierte Vorgehensweise
  - Element etwas ausführen lassen
  - Objekte sind Experten bezüglich ihrer internen Informationen
  - Objekt hat alle Informationen um eine Entscheidung selbst zu treffen

$\Rightarrow$ Objektorientierte Vorgehensweise führt zu verteilter Businesslogik

Tell, don’t ask

- Prozeduraler Code koppelt sich stark an andere Elemente
- Kopplung ist auch in objektorientiertem Code vorhanden, aber innerhalb des Objekts gekapselt (besser)
- Kommandos an Objekte stellen ist besser als Abfragen an Objekte richten
- Führt zu Command Query Separation
  - Abfragen sind seiteneffektfrei
  - Kommandos führen sauber definierte Aktionen aus

KISS

- Keep it simple, stupid
- Herkunft in der U.S. Navy 1960
  - Einfache Systeme arbeiten am besten
  - Ursprünglich von Kelly Johnson von Lockhead Skunk Works
  - Ein Flugzeug muss mit wenigen Werkzeugen von einem durchschnittlichen Mechaniker im Feld unter Kampfbedingungen reparierbar sein

KISS

- Vergleichbar mit Occam's Razor bei der Suche nach Hypothesen
  - Bevorzuge eine Hypothese oder Theorie, die möglichst wenige Annahmen oder Unbekannte besitzt
- Variationen
  - Keep it Simple, Silly
  - Kepp it short and simple
  - Keep it simple and straightforward
  - Keep it small and simple

KISS

- Komplexität erhöht die Wahrscheinlichkeit einen Fehler zu machen
  - Je größer die Anspannung, desto mehr steigt die Wahrscheinlichkeit den Fehler tatsächlich zu machen
- Komplexität unter allen Umständen vermeiden

KISS

- Entwickler tendieren zu komplexer Software
  - *Einfach kann jeder* ist ein weit verbreiteter Irrglaube
  - *Geniale* Software muss kompliziert sein ebenso
- Komplexe Algorithmen sind schwer zu verstehen
  - Fehler können nur schwer gefunden werden
- Wirklich geniale Algorithmen sind einfach
  - Routensuche mit Dijkstra
  - Sortierung mit Mergesort (Teile und Herrsche Ansatz)

SLAP

- Single Level of Abstraction Principle
- Prinzip des einfachen Abstraktionsniveaus
- Code innerhalb einer Methode ist auf einem Abstraktionsniveau
  - Keine Vermischung von Arbeit und Delegation (Technische Vermischung)
  - Keine Vermischung aus DB und Businesslogik (Vermischung der Belange/Semantik)

SLAP

- Bei Einhaltung entstehen Composed Methods
  - Zusammengesetzte Methoden
  - Einstiegsmethode in eine Klasse delegiert an private Hilfsmethoden
  - Einstiegsmethode liest sich wie ein Inhaltsverzeichnis
  - Hilfsmethoden übernehmen die Arbeit
  - Auch Hilfsmethoden können an andere Objekte delegieren

SLAP

- Fördert die Wiederverwendbarkeit
  - Hilfsmethoden können als Templatemethoden genutzt werden
- Bei Missachtung des Prinzips finden Sprünge in den Abstraktionsebenen einer Methode statt
- Sprünge zwischen Abstraktionsebenen sind tendentiell schwer zu verstehen
  - Es entsteht kognitiver Overhead
  - Sprünge sind schwerer lesbar

GRASP

- General Responsibility Assignment Software Patterns
- Basis Prinzipien auf denen Entwurfsmuster aufbauen
- Ziel ist die *Low Representational Gap (LRG)* möglichst klein zu halten
  - Die Lücke zwischen gedachten Domänenmodell und Softwareimplementierung (Designmodell) sollte klein sein

GRASP

- Zuweisung von Verantwortlichkeiten bzw. Zuständigkeiten
- Zuständigkeiten haben 2 Typen
  - Ausführend bedeutet . . .
    - Objekte erstellen
    - Objekte kontrollieren
    - Aktionen ausführen
  - Wissen über . . .
    - gekapselte Daten
    - Beziehungen zu zugehörigen Objekten
    - ableitbare bzw. berechenbare Informationen

GRASP

- Low Coupling
- High Cohesion
- Information Expert
- Creator
- Indirection
- Polymorphism
- Controller
- Pure Fabrication
- Protected Variations

Low Coupling

- Lose bzw. geringe Kopplung
- Kopplung bzw. Coupling beschreibt die Beziehungen zwischen Objekten
- Kopplung ist ein Maß für die Abhängigkeit zwischen Objekten

Low Coupling

- Positive Effekte durch geringe Kopplung
  - \+ Geringere Abhängigkeit zu Änderungen in anderen Teilen
  - \+ Einfacher testbar
  - \+ Verständlicher, da weniger Kontext notwendig ist
  - \+ Einfacher wiederverwendbar

Low Coupling

- Formen der Kopplung im Code z.B. in Java
  - X implementiert Interface Y
  - X ist abgeleitet von Klasse Y (auch indirekt)
  - X hat ein Attribut vom Typ Y
  - X hat eine Methode mit Referenz zu Klasse Y
    - Parameter, lokale Variable oder Rückgabewert
  - X verwendet eine statische Methode von Klasse Y
  - X verwendet eine polymorphe Methode von Klasse oder Interface Y

$\Rightarrow$ Komponenten werden austauschbar, wenn die Kopplung lose ist

Low Coupling - Weitere Formen

- Kopplung an konkrete oder abstrakte Datentypen
  - Klassen und Interfaces
- Kopplung verschiedener Threads
  - Gemeinsame Sperren bzw. Locks
- Kopplung durch Resourcen
  - Gemeinsame Dateien, Speicher, CPU

$\Rightarrow$ Kopplung zu stabilen Komponenten weniger problematisch

High Cohesion

- Hohe bzw. starke Kohäsion
- Kohäsion ist ein Maß für den Zusammenhalt einer Klasse
  - Beschreibt die semantische Nähe der Elemente einer Klasse
- Hohe Kohäsion und Lose Kopplung als Fundament für idealen Code

\+ Einfacheres und verständlicheres Design

\+ Komponenten werden wiederverwendbarer

High Cohesion - Beispiel

Simulation einer Person

```
public class Agent {
  private final Date dateOfBirth;
  private final Sex sex;
  private final Job job;
  private Point2D location;

  public void moveTo(Point2D next) {
    this.location = next;
  }

  public Destination selectDestination(Choices destinations) {
    return destinations.selectBasedOn(dateOfBirth, sex, job);
  }
}
```

High Cohesion - Beispiel

Simulation einer Person

```
public class Agent {
  private final Date dateOfBirth;
  private final Sex sex;
  private final Job job;
  private Point2D location;

  public void moveTo(Point2D next) {
    this.location = next;
  }

  public Destination selectDestination(Choices destinations) {
    return destinations.selectBasedOn(dateOfBirth, sex, job);
  }
}
```

High Cohesion - Beispiel

```
public class Agent {
  private final Sociodemography sociodemography;
  private Location location;

  ...
}

public class Sociodemography {
  private final Date dateOfBirth;
  private final Sex sex;
  private final Job job;

  ...
}

public class Location {
  private final Point2D location;

  ...
}
```

High Cohesion

- Semantische Nähe der Attribute und Methoden bestimmen
  - Semantik nur schwer automatisiert testbar
  - Menschliche Einschätzung notwendig
- Automatisch bestimmte technische Metriken
  - Anzahl Attribute und Methoden einer Klasse
  - Häufigkeit der Verwendung der Attribute in allen Methoden
  - Nicht immer treffend

Information Expert

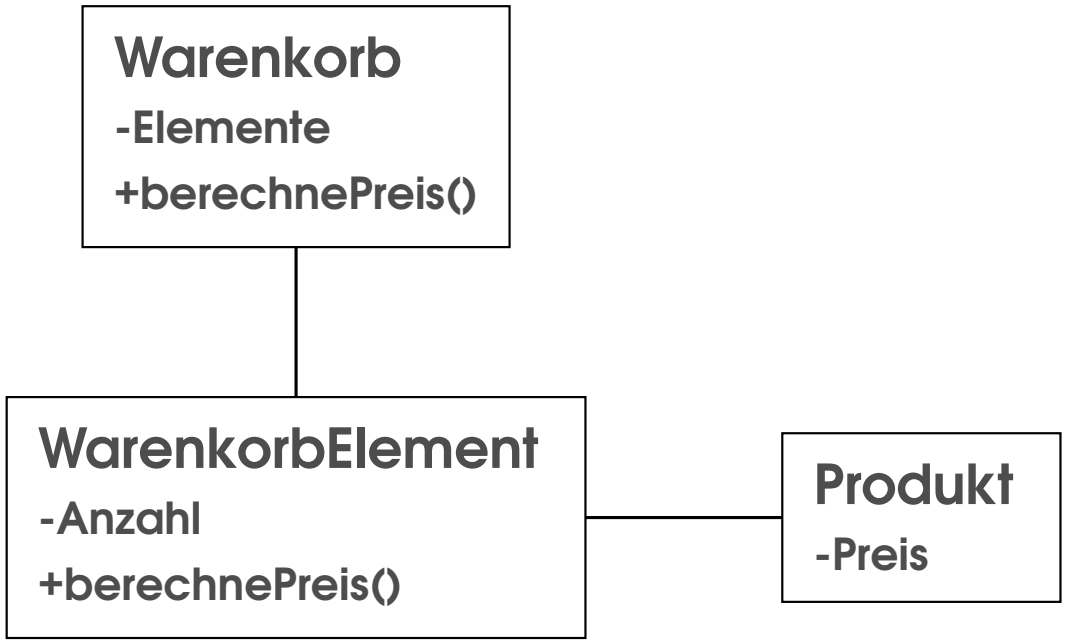
- Allgemeine Zuweisung einer Zuständigkeit zu einem Objekt
- Einfachste Möglichkeit
  - Das Objekt, das die Informationen besitzt, erhält die Verantwortung dafür
- Befragung von Domänen- und Designmodell
  - Wenn im Designmodell eine passende Klasse existiert wird diese verwendet
  - Ansonsten wird im Domänenmodell eine passende Repräsentation gesucht und dafür eine Klasse im Designmodell erstellt

Information Expert

- Objekte sind zuständig für Aufgaben über die sie Informationen besitzen
  - Informationen können auch auf Teilexperten verteilt sein
  - Experte sammelt Informationen von Teilexperten um Aufgabe zu erledigen

Information Expert - Beispiel

Berechnung des Preises bestehend aus Teilpreisen



Information Expert

+ Kapselung von Informationen
+ Leichtere Klassen, da Businesslogik zu den Daten verteilt wird

– Kann zu Problemen mit anderen Prinzipien führen
  - Separation of Concerns kann eine Lösung sein

Creator

- Das Erzeuger-Prinzip legt fest, wer für die Erzeugung von Objekten zuständig ist
- Ein Objekt der Klasse B ist zuständig für die Erzeugung von Objekten der Klasse A, wenn
  - B eine Aggregation von A ist
  - B enthält Objekte von A
  - B erfasst Objekte von A
  - B nutzt Objekte von A mit starker Kopplung
  - B hat sämtliche Informationen zur Initialisierung von A
    - B ist Experte zur Erstellung von A

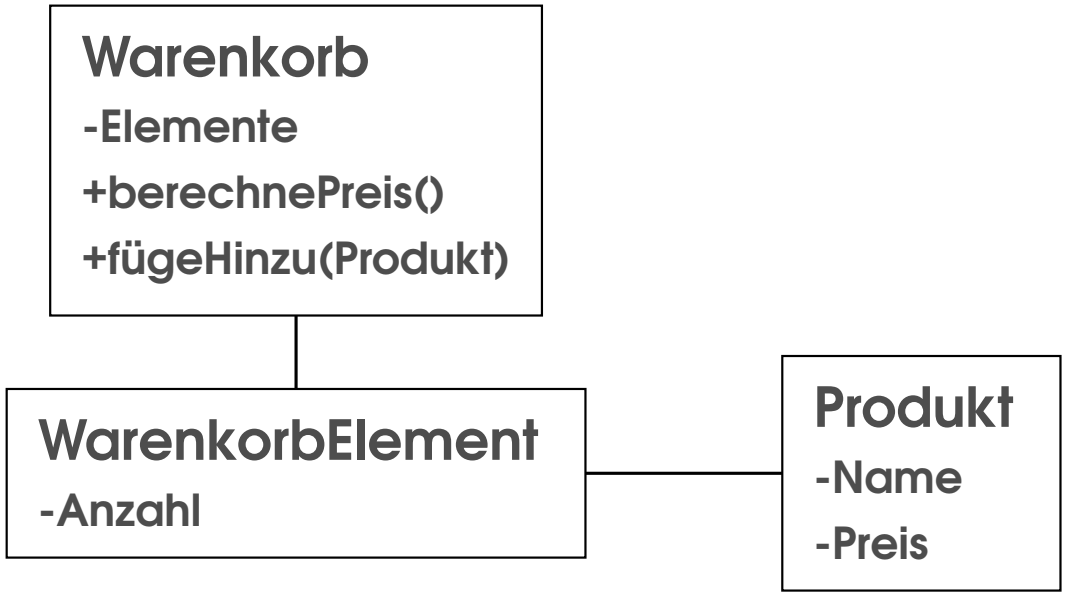
Creator

- Allgemein gehalten kommt ein Objekt als Creator eines anderen in Frage, wenn es zu jedem erstellten Objekt eine Beziehung hat
  - Eine Komposition in UML deutet auf einen Creator hin

+ Ein geeigneter Creator verringert die Kopplung von Komponenten

Creator - Beispiel

Einträge in einem Warenkorb anlegen



$\Rightarrow$ Warenkorb hat alle Informationen (Produkt) um WarenkorbElemente zu erstellen

Indirection

- Indirektion bzw. Delegation
- Kann Systeme oder Teile von Systemen voneinander entkoppeln
- Indirektion bietet mehr Freiheitsgrade als Vererbung bzw. Polymorphismus
  - Benötigt aber auch mehr Aufwand bzw. Code

Indirection - Beispiel

```
class MyStack<T> {
  List<T> elements;

  public MyStack() {
    elements = new ArrayList<>();
  }
  public void push (T element) {
    elements.add(0, element);
  }
  public T pop() {
    return elements.remove(0);
  }
  public boolean isEmpty() {
    return elements.isEmpty();
  }
  public int size() {
    return elements.size();
  }
}
```

- Schnittstelle ist auf den Anwendungszweck angepasst
- Mehr Flexibilität
- Komposition verschiedener Objekte erzielt das gewünschte Ergebnis

Polymorphism

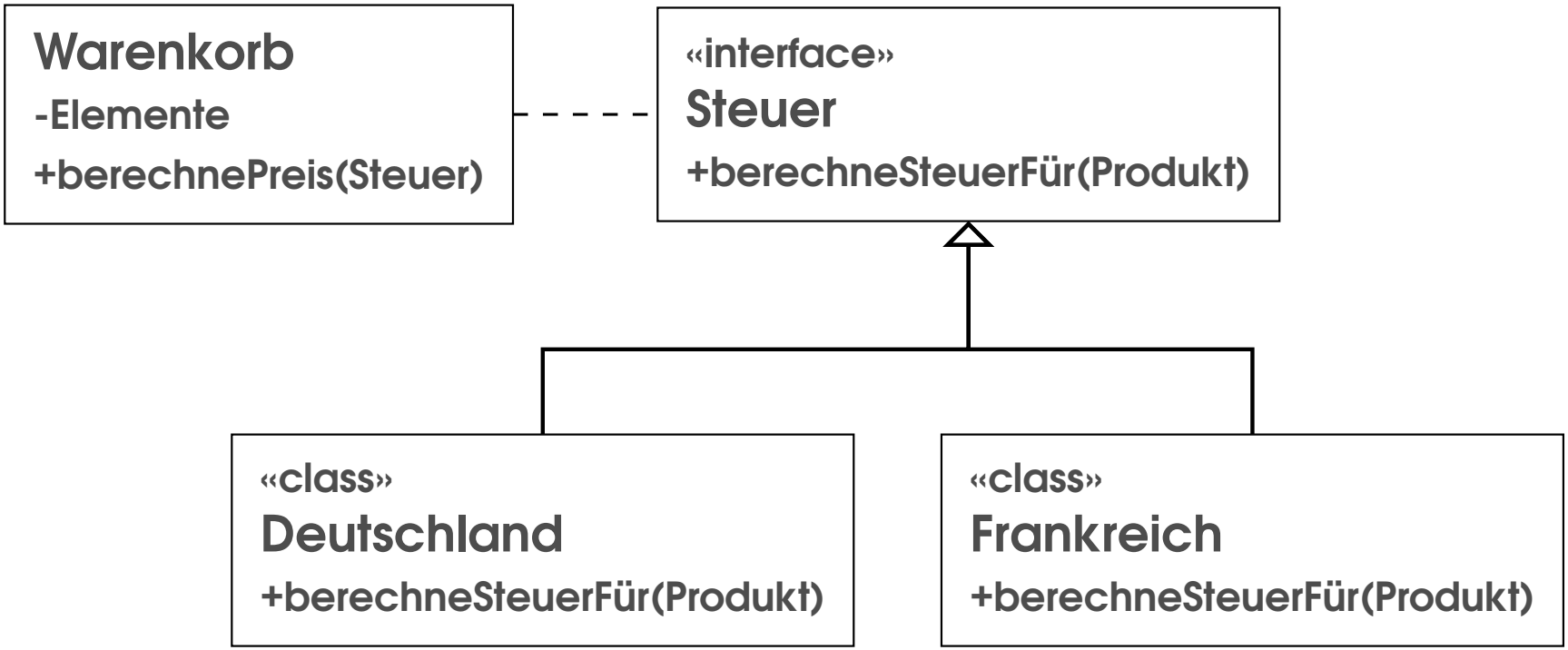
- Polymorphismus
- Behandlung von Alternativen abhängig von einem konkreten Typ
- Grundlegendes OO Prinzip zum Umgang mit Variation
  - Methoden erhalten je nach Typ eine andere Implementierung
- Vermeidung von Fallunterscheidungen
  - Kein If-Else bzw. Switch
  - Konditionalstruktur wird im Typsystem codiert

Polymorphism

- Abstrakte Klasse oder Interface als Basistyp
  - Interfaces binden den Anwender nicht an eine Klassenhierarchie
- Führt zur Verwendung des Entwurfsmusters *Strategie*
- Polymorphe Methodenaufrufe werden erst zur Laufzeit gebunden

Polymorphism - Beispiel

Berechnunge der korrekten Steuer von Produkten



Polymorphism

+ Einfacher erweiterbar
+ Bestehende Implementierung muss nicht verändert werden
+ Extrahierung von Frameworks wird vereinfacht

Controller

- Verarbeitung von einkommenden Benutzereingaben
- Koordination zwischen Benutzeroberfläche und Businesslogik
  - Einziger Ansprechpartner der Benutzeroberfläche
- Hauptsächlich Delegation zu anderen Objekten
  - Controller enthält keine Businesslogik
- Zustand der Anwendung kann in Controller gehalten werden
  - Aktion deaktivieren, während eine andere läuft

Arten eines Controllers

- System Controller
  - 1 Controller für alle Aktionen
  - Nur bei kleinen Anwendungen praktikabel
- Use Case Controller
  - 1 Controller pro Use Case
  - Viele kleine Controller

Pure Fabrication

- Reine bzw. völlige Erfindung
- Reine *Verhaltens-* oder *Arbeits-*Klasse
  - Klasse besitzt keinen Bezug zur Problemdomäne
- Trennung zwischen Technologie und Problemdomäne
  - Kapselung von Algorithmen

Pure Fabrication - Beispiel

Laden einer Bevölkerung aus der Datenbank

```
public void loadPopulation(Connection connection) {
  try {
    Statement statement = connection.createStatement();
    ResultSet results = statement.executeQuery(population());
    while (results.next()) {
      String name = results.getString("Name");
      String location = results.getString("Location");
      Person person = new Person(name, location);
      simulate(person);
    }
  } catch (SQLException exception) {
    process(exception)
  }
}
```

Pure Fabrication - Beispiel

Laden einer Bevölkerung aus der Datenbank

```
public void loadPopulation(Connection connection) {
  try {
    Statement statement = connection.createStatement();
    ResultSet results = statement.executeQuery(population());
    while (results.next()) {
      String name = results.getString("Name");
      String location = results.getString("Location");
      Person person = new Person(name, location);
      simulate(person);
    }
  } catch (SQLException exception) {
    process(exception)
  }
}
```

Pure Fabrication - Beispiel

```
public void loadPopulation(Connection connection) {
  try {
    Statement statement = connection.createStatement();
    ResultSet results = statement.executeQuery(population());
    while (results.next()) {
      handleElementOf(results)
    }
  } catch (SQLException exception) {
    process(exception)
  }
}

private void handleElementOf(ResultSet results) {
  String name = results.getString("Name");
  String location = results.getString("Location");
  Person person = new Person(name, location);
  simulate(person);
}
```

Pure Fabrication

+ Einfach wiederverwendbar auch außerhalb der Domäne
+ Begünstigt *high cohesion* durch Kapselung spezieller Funktionalität

$\Rightarrow$ Sollte möglichst wenig vorkommen

Protected Variations

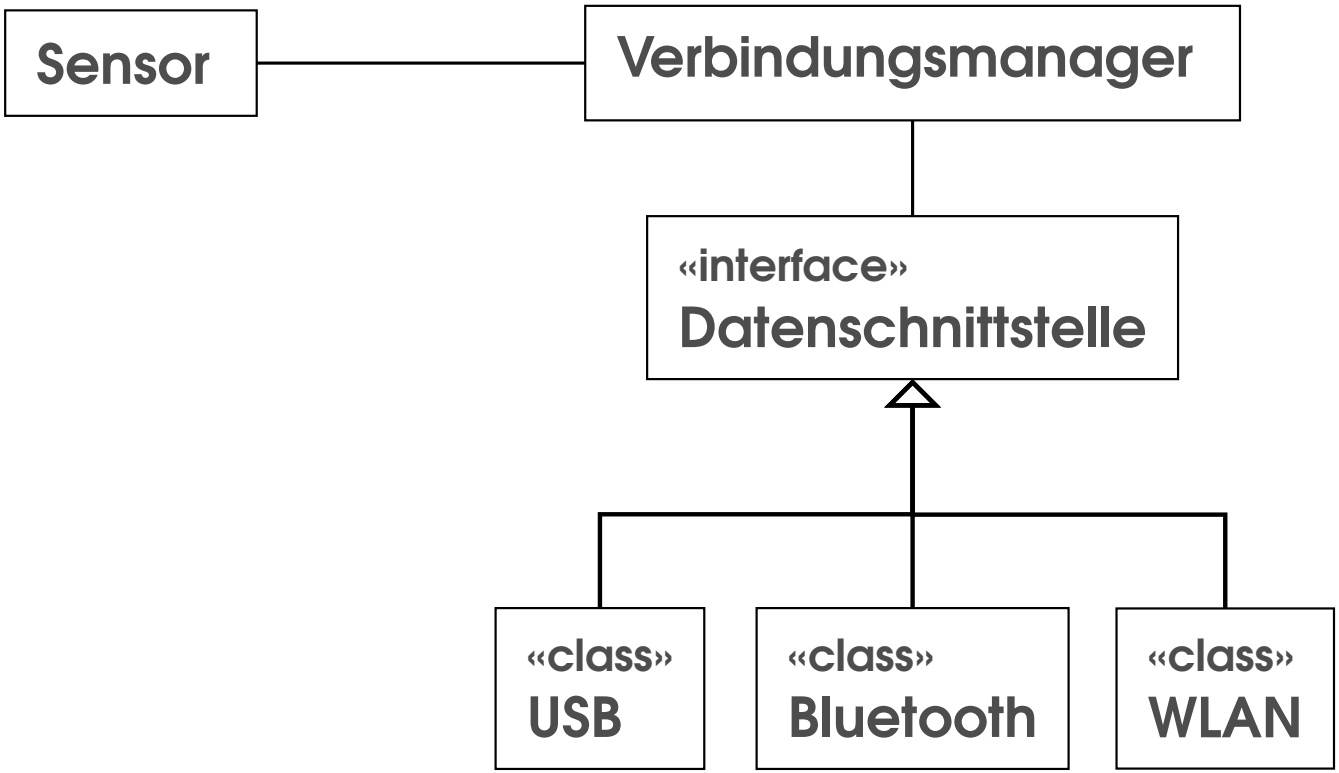
- Sicherung vor Variation
- Kapselung verschiedener Implementierungen hinter einer einheitlichen Schnittstelle (API)
  - Ursprünglich bekannt als Information Hiding
- Der Einfluss von Variabilität einzelner Komponenten soll nicht das Gesamtsystem betreffen
- Polymorphie und Delegation sind gute Schutzmöglichkeiten
  - Wechsel der Implementierung ist nicht relevant für das Gesamtsystem

Protected Variations

- Stylesheets im Webumfeld
  - Schützt vor konkretem Aussehen
- Spezifikation von Schnittstellen
  - Schützt vor Implementierungsdetails
- Betriebssysteme und Virtuelle Maschinen
  - Schützen vor konkreter Hardware
- Begrenzt auch SQL
  - Schützt vor konkreter Datenbank

Protected Variations - Beispiel

Verbindungsmanagement einer Konfigurationssoftware für Sensoren



DRY

- Don’t Repeat Yourself!
  - Wiederhole dich nicht!
- Anwendbar auf alles mögliche
  - Datenbankschemata
  - Testpläne
  - Buildsystem
  - Dokumentation

DRY

- *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system*
  - Es darf nur eine Quelle der Wahrheit geben
  - Alle anderen Quellen werden davon abgeleitet
- Vergleichbar zu den Normalformen bei RDBMS
- Mechanische Duplikation ist erlaubt
  - Header (*.h) Dateien und Code (*.c) Dateien in C
  - Header dient als Originalquelle
  - Compiler überprüft die Duplikation der Signatur

DRY

- Auswirkungen der Modifikation eines Teils haben eine definierte Reichweite
  - Keine unbeteiligten Teile sind betroffen
  - Alle relevanten Teile ändern sich automatisch
- Singleton ist keine Umsetzung des DRY Prinzips
  - Die Anzahl eines automatisch erzeugten Objekts ist irrelevant

DRY - Arten der Duplikation

- Imposed Duplication
  - Auferlegte Duplikation
  - Entwickler glaubt die Duplikation ist unumgänglich
- Inadvertent Duplication
  - Versehentliche Duplikation
  - Entwickler bemerkt die Duplikation nicht
- Impatient Duplication
  - Ungeduldige Duplikation
  - Entwickler ist zu faul die Duplikation zu beseitigen

DRY vs. WET

- WET ist das Gegenteil von DRY
- Steht für
  - Write everything twice
  - We enjoy typing
  - Waste everyones time

$\Rightarrow$ Kein ernsthaftes Programmierprinzip

YAGNI

- You ain't gonna need it
- Du wirst es nicht brauchen
- Unnötige Feature bzw. Funktionen erhöhen die Komplexität
  - Spekulatives Programmieren erhöht die Komplexitätssteuer
  - Komplexitätssteuer so niedrig wie möglich halten
  - Verständlichkeit sinkt bei zu viel Komplexität

YAGNI

- Jede vorhandene Funktionalität bindet Ressourcen
  - Unnötige Funktionalität bindet unnötig Ressourcen
  - Für die Entwicklung gewollter Funktionalität bleiben weniger Ressourcen
- Eigene Ideen sind nur schwer objektiv zu betrachten
  - Pair Programming liefert eine objektivere Meinung

YAGNI

- Spekulatives Programmieren führt zu Frameworks, die keiner benutzen will
  - Oftmals auch Elfenbeinturm-Projekte genannt
- Frameworks sind sinnvoll, wenn sie aus einer existierenden Anwendung entstehen
  - Bei ähnlicher Anwendung wichtige Komponenten aus einer bestehenden Anwendung extrahieren
  - Frameworks erhöhen die Wiederverwendbarkeit
  - Frameworks sind kein Allheilmittel

YAGNI

- Featuritis (auch schleichend) ist schlecht für Software
  - Marketing versteht (idealerweise) was das Beste für den Kunden ist
  - Marketing hat keine Ahnung von Komplexität in der Software
- Kommunikation zwischen Entwicklung und Marketing bzw. Kunde wichtig
  - Siehe die Vasa in Stockholm
- Fähigkeit zu Software hinzufügen, nicht Komplexität

  $\Rightarrow$ Weniger ist mehr

Conway's Law

- Any organisation that designs a system will inevitably produce a design whose structure is a copy of the organization's communication structure
- Kommunikationsstruktur findet sich in Code bzw. Architektur wieder
- Kommunikationsschnittstellen entsprechen Modulschnittstellen im Code

$\Rightarrow$ Kommunikation ist wichtig

Conway's Law

- Passen die Kommunikationsschnittstellen nicht zum Produkt, führt das zu Problemen
  - Siehe Databinding im JDK
- Beispiel: Konzernwebseiten spiegeln oft die Organisationsstruktur des Konzerns wieder
  - Besser wäre es die Webseite nach den Bedürfnissen des Kunden auszurichten
  - Noch besser die Organisation nach den Bedürfnissen des Kunden ausrichten

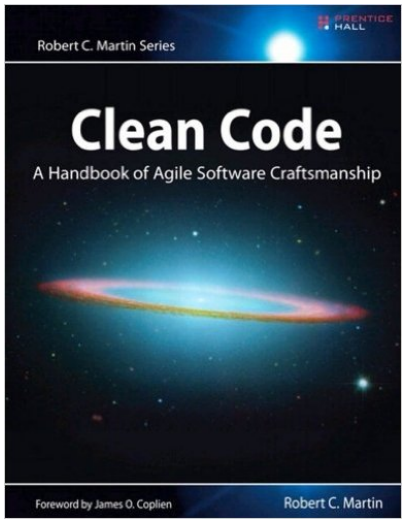
Conway's Law

- Bei der Neuausrichtung eines Produkts bzw. Software muss die Kommunikationsstruktur angepasst werden
- Projekte scheitern durch mangelhafte Kommunikation, nicht durch Menschen oder Technologien
- Kommunikationsstrukturen können nur gefördert werden, nicht geplant
  - …und schon gar nicht erzwungen

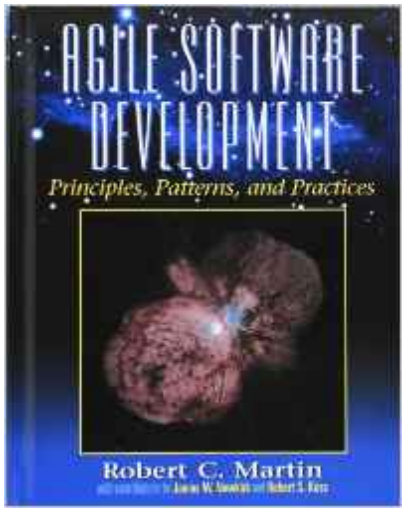
Prinzipien . . .

- weisen die Richtung bei der Entwicklung
- existieren in unterschiedlichen Abstraktionen
- müssen zusammen betrachtet werden
- müssen nicht erzwungen werden

Literatur

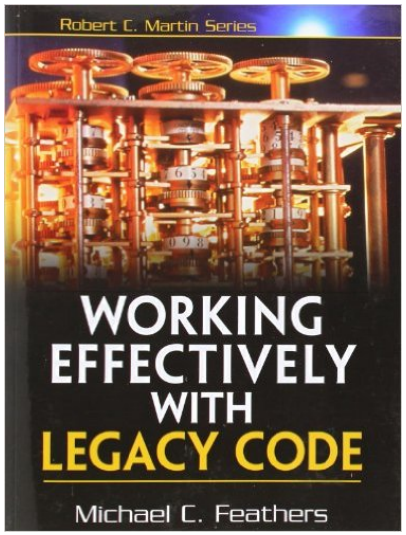


- Clean Code
  - Robert C. Martin
  - Pearson Education
  - ISBN: 978-0132350884

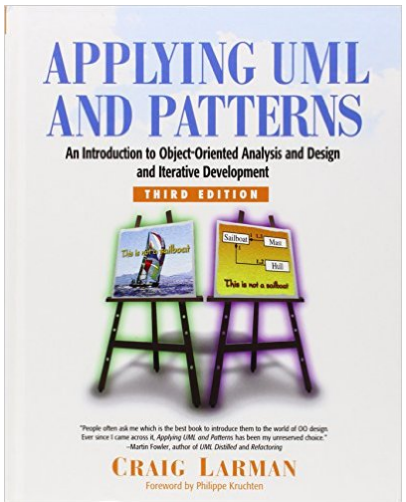


- Agile Software Development
  - Robert C. Martin
  - Pearson Education
  - ISBN: 978-0135974445

Literatur

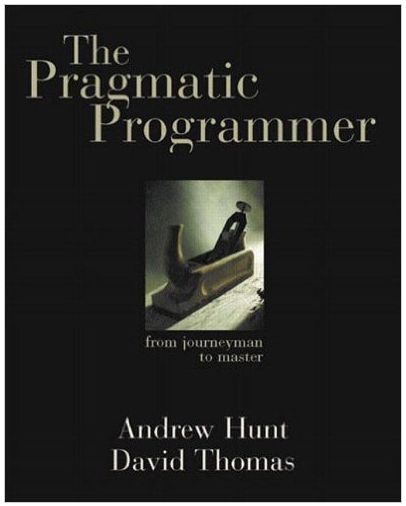


- Working Effectively with Legacy Code
  - Michael C. Feathers
  - Pearson Education
  - ISBN: 978-0131177055

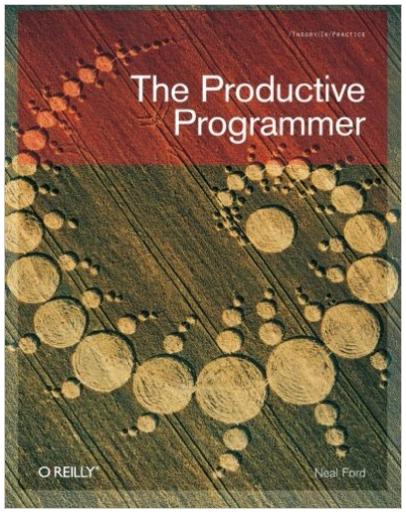


- Applying UML and Patterns
  - Craig Larman
  - Prentice Hall
  - ISBN: 978-0131489066

Literatur



- The Pragmatic Programmer
  - Andrew Hunt und David Thomas
  - Addison Wesley
  - ISBN: 978-0201616224



- The Productive Programmer
  - Neal Ford
  - O'Reilly
  - ISBN: 978-0596519780