

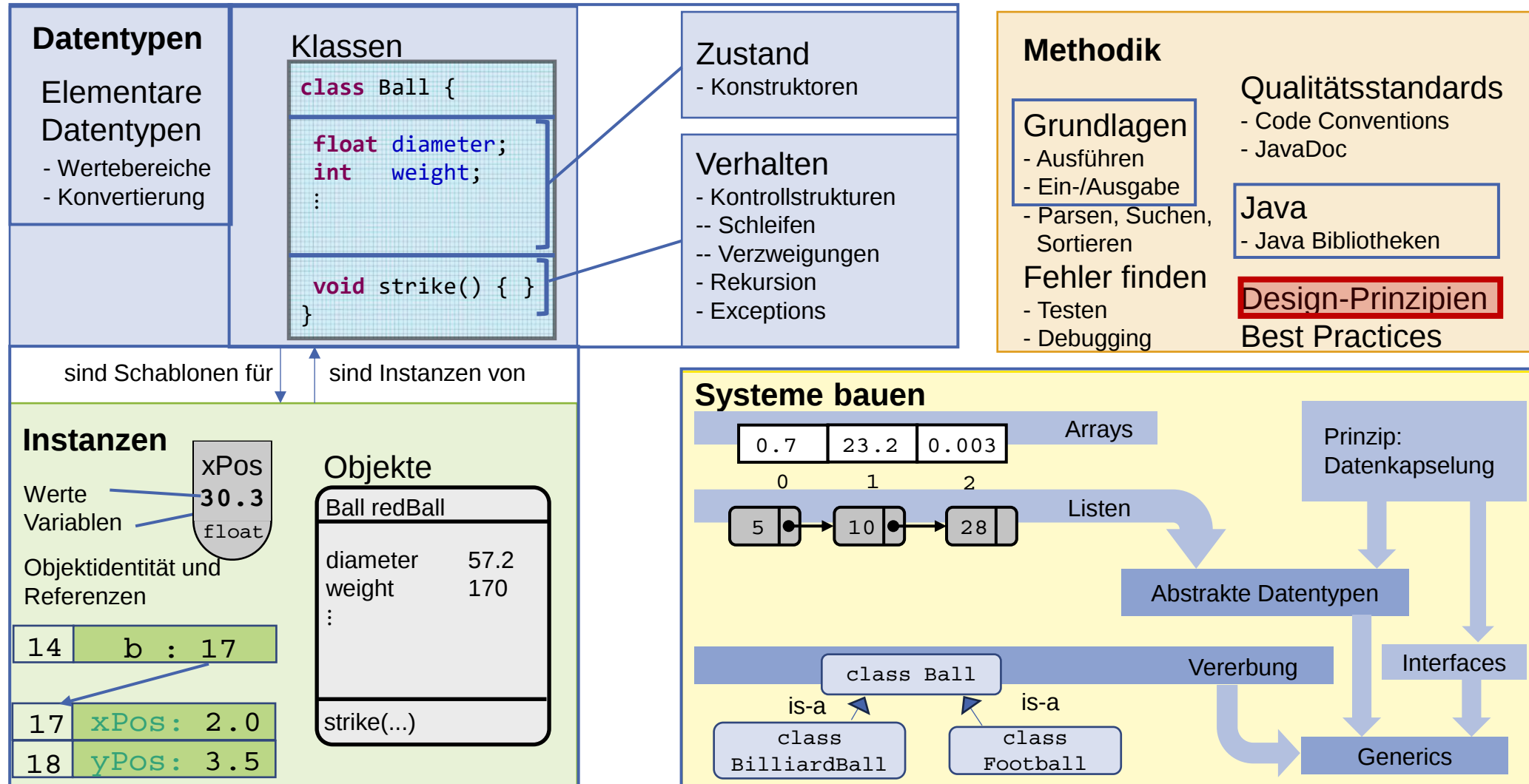
Vorlesung Programmieren

13. Objekt-orientierte Design-Prinzipien

Prof. Dr.-Ing. Anne Koziolk



Vorlesungsüberblick: Objekt-orientiertes Programmieren in Java



Vorlesungsüberblick: Vorläufiger Semesterplan

21.10.2024	Erstsemesterbegrüßung: Einführung
23.10.2024	Organisatorisches; Ein Einfaches Programm, Objekte und Klassen
30.10.2024	Typen und Variablen
06.11.2024	Kontrollstrukturen (+Scanner)
13.11.2024	Konstruktoren und Methoden
20.11.2024	Arrays; Konvertierung, Datenkapselung, Sichtbarkeit
27.11.2024	Listen und Abstrakte Datentypen
04.12.2024	Vererbung
11.12.2024	Exceptions; Interfaces
18.12.2024	Generics; Rekursion
08.01.2025	Java-API; Objektorientierte Design-Prinzipien
15.01.2025	Best Practices; Finden und Beheben von Fehlern
22.01.2025	Testen und Assertions
29.01.2025	JUnit; Parsen, Suchen, Sortieren
05.02.2025	Vom Programm zur Maschine; Ausblick auf zukünftige Lehrveranstaltungen
12.02.2025	Wrap-Up

Überblick

- Datenkapselung
- Komposition vor Vererbung
- Gegen Schnittstellen programmieren
- 5 „SOLID“-Prinzipien, insbesondere „Open-Closed-Principle“
- Schnittstellenentwurf, ausgewählte Prinzipien

Folien basieren auf Material von Bob Tarr (University of Maryland)

Lernziele

- Sie können Prinzipien des objektorientierten Entwurfs bei der Programmierung von Java anwenden.
- Sie können Programmcode anhand der Prinzipien bewerten und verbessern.
- Sie können gut wartbaren und gut lesbaren Programmcode erstellen.



Quelle: <http://phdcomics.com>

Prinzip 1: (*Datenkapselung*)

Minimiere die Zugriffsmöglichkeiten
auf Klassen und Attribute

Datenkapselung (*Information Hiding*)

- Verwendung von **private** mit zugehörigen setter- und getter-Methoden, wann immer möglich

- **Beispiel:**

Ersetze

```
public double speed;           // in km/h
```

durch

```
private double speed;         // in km/h

public double getSpeed() {
    return speed;
}
public void setSpeed(double newSpeed) {
    speed = newSpeed;
}
```

Vorteile Datenkapselung (I)

- Einschränkungen / Prüfbedingungen für Werte möglich

Beispiel

```
public void setSpeed(double newSpeed) {  
    if (newSpeed < 0.0) {  
        sendErrorMessage(...);  
        newSpeed = Math.abs(newSpeed);  
    }  
    speed = newSpeed;  
}
```

- Wenn Benutzer der Klasse direkten Zugriff auf die Attribute hätten, wäre es nicht möglich, eine Einschränkung (wie `speed >= 0.0`) sicherzustellen

Vorteile Datenkapselung (II)

- Die interne Repräsentation kann geändert werden, ohne die Schnittstelle zu verändern

Beispiel

```
private double speed; // jetzt in m/s anstatt km/h

public void setSpeed(double newSpeed) {
    speed = convertKMHToMS(newSpeed);
}

private double convertKMHToMS(double speed) {
    return speed * 1000 / 3600;
}
```

← Schnittstelle unverändert

Vorteile Datenkapselung (II)

- Es können beliebige Seiteneffekte ausgeführt werden

Beispiel

```
public void setSpeed(double newSpeed) {  
    speed = newSpeed;  
    notifyObservers();  
    logSpeedChange();  
}
```

- Wenn Benutzer der Klasse auf die Attribute direkt zugreifen würden, müsste jeder sich extra um die Seiteneffekte kümmern

Prinzip 2: Bevorzuge Komposition gegenüber Vererbung

Zum Weiterlesen:

- Gamma, E., Johnson, R., Helm, R., & Vlissides, J. (2011).

Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software. S. 26

- Bloch, Joshua. *Effective java*. Addison-Wesley Professional, 2017.

Komposition

- Komposition bezeichnet die Wiederverwendung durch Zusammensetzen bestehender Objekte zu einem neuen mit erweiterter Funktionalität

Beispiel mit Komposition

```
class Table {  
    private TableData data;  
    private GraphicalRepresentation grep;  
    ...  
}
```

- Neue Funktionalität wird (auch) durch Delegation an Komponenten erreicht
- Komposition wird manchmal auch Aggregation genannt

Vererbung

- Bei Vererbung wird Wiederverwendung dadurch erreicht, dass die Implementierung bestehender Objekte erweitert wird
- Oberklasse enthält gemeinsame Funktionalität (Methoden, Attribute)
- Abgeleitete Klasse erweitert die Basisfunktionalität durch zusätzliche Methoden und Attribute

Beispiel Komposition vs. Vererbung

- Wir wollen eine Variante von HashSet schreiben, die die Anzahl der eingefügten Elemente mitprotokolliert.
- Wir leiten dazu eine Klasse von HashSet ab:

```
public class InstrumentedHashSet extends HashSet {  
    // The number of attempted element insertions  
    private int addCount = 0;  
    public InstrumentedHashSet(Collection c) { super(c); }  
    public InstrumentedHashSet(int initCap, float loadFactor) {  
        super(initCap, loadFactor);  
    }  
    public boolean add(Object o) {  
        addCount++;  
        return super.add(o);  
    }  
    public boolean addAll(Collection c) {  
        addCount += c.size();  
        return super.addAll(c);  
    }  
    public int getAddCount() {  
        return addCount;  
    }  
}
```

Beispiel aus Bloch, Joshua. *Effective java*.
Addison-Wesley Professional, 2017.

Beispiel Komposition vs. Vererbung

- Wir wollen nun unsere neue Klasse InstrumentedHashSet testen:

```
public static void main(String[] args) {  
    InstrumentedHashSet s = new InstrumentedHashSet();  
    s.addAll(Arrays.asList(new String[] { "Hund", "Katze", "Maus" }));  
    System.out.println(s.getAddCount());  
}
```

- Ausgabe: 6 (nicht das erwartete 3). Warum?
- Die Methode addAll() von HashSet ruft intern die add()-Methode auf. Daher werden bei addAll() eingefügte Elemente doppelt gezählt!
- Allgemeines Problem: Vererbung erfordert Detailwissen über Oberklassen



Beispiel mit Komposition

```
public class InstrumentedSet implements Set {  
    private final Set s;  
    private int addCount = 0;  
  
    public InstrumentedSet(Set s) { this.s = s; }  
  
    public boolean add(Object o) {  
        addCount++;  
        return s.add(o);  
    }  
  
    public boolean addAll(Collection c) {  
        addCount += c.size();  
        return s.addAll(c);  
    }  
  
    public int getAddCount() {  
        return addCount;  
    }  
}
```

Beispiel mit Komposition

```
// Forwarding methods (rest of Set interface methods)
public void clear(){ s.clear(); }
public boolean contains(Object o){ return s.contains(o);}
public boolean isEmpty(){ return s.isEmpty(); }
public int size(){ return s.size(); }
public Iterator iterator(){ return s.iterator();}
public boolean remove(Object o){ return s.remove(o);}
public boolean containsAll(Collection c){ return s.containsAll(c);}
public boolean removeAll(Collection c){ return s.removeAll(c);}
public boolean retainAll(Collection c){ return s.retainAll(c);}
public Object[] toArray(){ return s.toArray();}
public Object[] toArray(Object[] a){ return s.toArray(a);}
public boolean equals(Object o){ return s.equals(o);}
public int hashCode(){ return s.hashCode();}
public String toString(){ return s.toString(); }
}
```

Hinweis: Das Beispiel aus „Bloch, Joshua. *Effective java*. Addison-Wesley Professional, 2017.“ geht noch einen Schritt weiter und teilt „Wrapper“ und „Forwarding Class“.

Beispiel mit Komposition

Zu beachten:

- Diese Klasse implementiert das Set-Interface
- Das enthaltene Set-Objekt kann ein beliebiges Objekt sein, welches das Set-Interface implementiert (nicht nur HashSet)
- Daher sehr flexibel

Beispiel

```
List list = new ArrayList();  
Set s1 = new InstrumentedSet(new TreeSet(list));  
  
int capacity = 5;  
float loadFactor = 0.7f;  
Set s2 = new InstrumentedSet(new HashSet(cap, loadFactor));
```

Vorteile / Nachteile Komposition

Vorteile:

- Zugriff auf Komponenten erfolgt ausschließlich über deren Interface
- „Black-Box“-Wiederverwendung, da interne Details der Komponenten nicht sichtbar sind
- Gute Datenkapselung, weniger Implementierungs-Abhängigkeiten
- Jede Klasse konzentriert sich auf genau eine Aufgabe
- Komposition kann dynamisch zur Laufzeit erfolgen durch Referenzen auf Komponenten

Nachteile:

- Komposition benötigt typischerweise mehr Objekte
- Interfaces müssen sauber definiert werden, damit Wiederverwendung in möglichst vielen Fällen ermöglicht wird

Vorteile / Nachteile Vererbung

Vorteile:

- Implementierung der abgeleiteten Klasse einfacher, da das meiste vererbt wird
- Basisklasse kann leicht modifiziert oder erweitert werden

Nachteile:

- Schlechte/keine Datenkapselung, da abgeleitete Klasse auf Implementierungsdetails der Oberklasse Bezug nehmen kann
- „White-Box“-Wiederverwendung, da Implementierungsdetails in abgeleiteten Klassen sichtbar sind
- Änderungen an abgeleiteten Klassen können erforderlich werden, wenn Oberklasse verändert wird
- Implementierungen, die von der Oberklasse geerbt werden, können zur Laufzeit nicht verändert werden

Prinzip 3:

Programmiere gegen Schnittstellen und nicht gegen eine Implementierung

Zum Weiterlesen:

Gamma, E., Johnson, R., Helm, R., & Vlissides, J. (2011).

Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software. S. 24

Vorteile/Nachteile Interfaces

Vorteile:

- Verwender des Interfaces (*Clients*) kennen die genaue Klasse des Objekts, das das Interface implementiert, nicht
- Objekte können leicht durch andere ausgetauscht werden
- Objektverbindungen müssen nicht „festverdrahtet“ programmiert werden
- Wiederverwendung wird verbessert

Nachteile:

- Etwas erhöhte Design-Komplexität

Beispiel Interface

Beispiel

```
/**
 * Interface Maneuverable provides the specification
 * for a maneuverable vehicle.
 */
public interface Maneuverable {
    public void left();
    public void right();
    public void forward();
    public void reverse();
    public void setSpeed(double speed);
    public double getSpeed();
}
```

Beispiel Interface

Beispiel

```
public class Car implements Maneuverable { ... }  
  
public class Boat implements Maneuverable { ... }  
  
public class Submarine implements Maneuverable { ... }
```

Beispiel Interface

- Eine Methode in einer anderen Klasse kann Verkehrsmittel steuern, ohne zu wissen, um welche es sich genau handelt (Auto, Boot, U-Boot, ...)

Beispiel

```
public void travel(Maneuverable vehicle) {  
    vehicle.setSpeed(35.0);  
    vehicle.forward();  
    vehicle.left();  
    vehicle.setSpeed(0.0);  
}
```

Prinzip 4: (*Open-Closed Principle*)

Software-Komponenten sollten offen für Erweiterung, aber geschlossen für Änderung sein

Zum Weiterlesen:
Martin, Robert C. Agile software development: principles, patterns, and practices.
Prentice Hall, 2002.

Open-Closed Principle

- Das Open-Closed Prinzip (OCP) besagt, dass man versuchen sollte, Programmmodule zu entwerfen, die nie geändert werden müssen
- Für Erweiterungen fügen wir neuen Code hinzu, wir ändern keinen alten Code

Zwei Kriterien:

- **Offen für Erweiterung:** Das Verhalten eines Moduls kann erweitert werden, um neue Funktionalität bereitzustellen
- **Geschlossen für Änderung:** Der Source-Code des Moduls darf sich nicht ändern

Beispiel OCP

Beispiel

```
public double totalPrice(Part[] parts) {  
    double total = 0.0;  
    for (int i = 0; i < parts.length; i++) {  
        total += parts[i].getPrice();  
    }  
}
```

- Falls es sich bei Part um ein Interface handelt, können verschiedene Teile ohne Änderung der Methode totalPrice() behandelt werden

Beispiel OCP

- Für bestimmte Teile soll ein besonderer Preis berechnet werden. Wie können wir das implementieren?

Beispiel

```
public double totalPrice(Part[] parts) {  
    double total = 0.0;  
    for (int i = 0; i < parts.length; i++) {  
        if (parts[i] instanceof Motherboard)  
            total += 1.20 * parts[i].getPrice();  
        if (parts[i] instanceof Memory)  
            total += 1.10 * parts[i].getPrice();  
        else  
            total += parts[i].getPrice();  
    }  
}
```

Erfüllt dieser Code das OCP?

- **Nein!**



Beispiel OCP

- **Besser:** Extra Klasse (PricePolicy) zur Berechnung spezieller Preise

Beispiel

```
public class Part {  
    private double price;  
    private PricePolicy pricePolicy;  
  
    public void setPricePolicy(PricePolicy pricePolicy) {  
        this.pricePolicy = pricePolicy;  
    }  
    public void setBasePrice(double price) { this.price = price; }  
    public double getPrice() { return pricePolicy.getPrice(price); }  
}  
  
public class PricePolicy {  
    private double factor;  
  
    public PricePolicy(double factor) { this.factor = factor; }  
    public double getPrice(double price) { return price * factor; }  
}
```

OO-Design-Prinzipien

SOLID:

■ Single Responsibility Principle

- Jede Klasse sollte nur eine Verantwortung („Grund zur Änderung“) haben.

■ Open/Closed Principle

- Klassen sollten Erweiterungen erlauben, ohne dabei ihr Verhalten zu ändern

■ Liskov Substitution Principle

- Eine Instanz einer abgeleiteten Klasse sollte sich so verhalten, dass jemand, der meint, ein Objekt der Basisklasse vor sich zu haben, nicht durch unerwartetes Verhalten überrascht wird.

■ Interface Segregation Principle

- Klassen sollten durch Interface nicht gezwungen werden, nicht notwendige Methoden zu implementieren; stattdessen Interface auftrennen

■ Dependency Inversion Principle

- Abstraktes soll nicht von Details abhängen

Zum Weiterlesen:
Martin, Robert C. Agile software development:
principles, patterns, and practices.
Prentice Hall, 2002.

Ausgewählte Prinzipien für den Schnittstellenentwurf

Warum Aufwand in den Entwurf guter Schnittstellen investieren?

“Good interfaces reduce costs”

[Gernot Starke, Stefan Tilkov (Eds.) “SOA-Expertenwissen”,
1st edition, dpunkt.verlag, Heidelberg, 2007, p. 357]

Gute Schnittstellen

- Gute Schnittstellen sind erster Schritt zu qualitativ hochwertigem Klassenentwurf
- Eigenschaften guter Schnittstellen
 - verbergen Implementierungsdetails („Information Hiding“)
 - besitzen konsistentes Abstraktionsniveau
 - besitzen angemessenes Abstraktionsniveau

Schnittstellenentwurf für Kapselung/Wiederverwendbarkeit

- **Entwurfsfrage:** Soll eine Schnittstelle die **interne Repräsentation** der späteren Implementierung widerspiegeln?
- Beispiel:

```
void addCustomers(List<Customer> customers)
```

im Vergleich zu:

```
void addCustomer(Customer c)
```

- **Nein**, das Exponieren der internen Repräsentation schränkt die Wiederverwendbarkeit ein
 - Die Möglichkeit zum Hinzufügen eines **Set** von **Customers** sollte ebenso gegeben sein, zusätzlich zu **List** von **Customers**
 - Benutzer (Aufrufer) einer Klasse sollte keine Annahmen über die interne Implementierung treffen müssen
- Deshalb:
 - Nutzen von generellen (möglichst abstrakten) Superklassen/Schnittstellen (z.B. `Collection`):

```
void addCustomers(Collection<Customer> customers)
```

Trennung von Befehl und Anfrage

- Eine Methode sollte entweder
 - den Objekt-Zustand zurückliefern ([Anfrage](#))
 - oder den Objektzustand ändern ([Befehl](#))
 - aber **nicht beides gleichzeitig**
- Grund: Mehrfache Hintereinanderausführung einer Methode sollte dasselbe Ergebnis liefern
- Methode wird dann als [idempotent](#) bezeichnet

Trennung von Befehl und Anfrage

- Nachteil: Mehr Methodenaufrufe
- Gegenbeispiel: Iterator-Muster in Java und C#

Java

```
while (i.hasNext()) {  
    Object ele = i.next();  
}
```

boolean hasNext(): Anfrage

Object next(): **Befehl und Anfrage**

C#

```
while (i.MoveNext()) {  
    Object ele = i.Current();  
}
```

Object Current(): Anfrage

boolean MoveNext(): **Befehl und Anfrage**

Außerdem zeigt Iterator (**IEnumerator**) nach dessen Instanziierung auf ein ungültiges Element.

Trennung von Befehl und Anfrage

■ Ein eleganter Iterator?

“New” Java:

```
while (i.isCurrentValid()) {  
    Object ele = i.current(); // no switch to next element  
    i.moveToNext();  
}
```

boolean isCurrentValid(): Anfrage

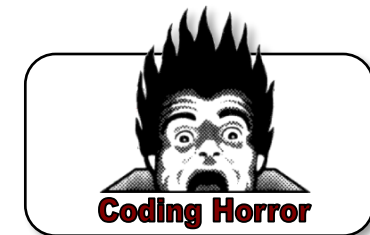
Object current(): Anfrage

void moveToNext(): Befehl (was aber geschieht beim letzten Element?)

Schlechte Abstraktion

```
public interface Program {  
    void initializeCommandStack();  
    void pushCommand(Command command);  
    Command popCommand();  
    void shutdownCommandStack();  
    void initializeReportFormatting();  
    void formatReport(Report report);  
    void printReport(Report report);  
    void initializeGlobalData();  
    void shutdownGlobalData();  
}
```

[Steve McConnell “Code Complete”, 2nd edition, Microsoft Press, 2004, S. 134]



Bessere Abstraktion

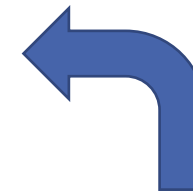
```
public interface Program {  
    void initializeUserInterface();  
    void shutdownUserInterface();  
    void initializeReports();  
    void shutdownReports();  
}
```

[Steve McConnell “Code Complete”, 2nd edition, Microsoft Press, 2004, S. 134]

Auftrennung von Klassen und Schnittstellen

- Aufteilen einer Klasse in zwei Klassen bzw. Schnittstellen, falls...
- einige Methoden arbeiten mit der einen Hälfte der Attribute, die restliche Methoden mit der anderen Hälfte

```
public interface IUserInterfaceCtrl {  
    void initializeUserInterface();  
    void shutdownUserInterface();  
}  
  
public interface IReportCtrl {  
    void initializeReports();  
    void shutdownReports();  
}
```



```
public interface Program {  
    void initializeUserInterface();  
    void shutdownUserInterface();  
    void initializeReports();  
    void shutdownReports();  
}
```



Gemischtes Abstraktionsniveau

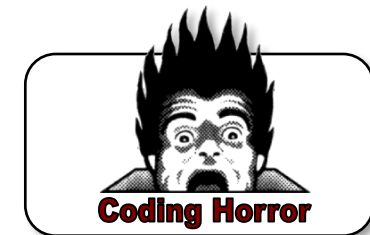
```
public interface IEmployeeCensus {  
    void addEmployee(Employee employee);  
    void removeEmployee(Employee employee);  
  
    Employee nextItemInList();  
    Employee firstItem();  
    Employee lastItem();  
}
```



Employee-Abstraktionsniveau

List-Abstraktionsniveau

[Steve McConnell “Code Complete”, 2nd edition, Microsoft Press, 2004, S. 135]



Konsistentes Abstraktionsniveau

```
public interface IEmployeeCensus {  
    void addEmployee(Employee employee);  
    void removeEmployee(Employee employee);  
  
    Employee nextEmployee();  
    Employee firstEmployee();  
    Employee lastEmployee();  
  
}
```

```
public class EmployeeCensus implements IEmployeeCensus {  
    ...  
  
    // von außen nicht sichtbare Verwendung von List  
    private List employeeList;  
  
    ...  
}
```

Employee-Abstraktionsniveau

[Steve McConnell “Code Complete”, 2nd edition, Microsoft Press, 2004, S. 135]

Wahl von Methodennamen

- Ausdrucksstarke Namen benutzen
- In der **Benutzerterminologie** beschreiben, was die Methode leistet
 - Nicht: **doComputations(...)**
 - Sondern: **calculateInvoiceSum(...)**
- Keine Methodennamen (allgemeiner: Bezeichner), die sich bloß durch eine Ziffer unterscheiden
 - `outputUser1`, `outputUser2`
 - Mangelnde Semantik der Ziffer
 - Gefahr von unentdeckt gebliebenen Bugs durch Tippfehler

[Steve McConnell “Code Complete”, 2nd edition, Microsoft Press, 2004]

Wahl von Methodennamen

- Konsistente Nutzung gegensätzlicher Begriffe
 - add/remove
 - create/destroy
 - first/last
 - get/put
 - get/set
 - increment/decrement
 - insert/delete
 - lock/unlock
 - min/max
 - next/prev(ious)
 - old/new
 - open/close
 - show/hide
 - source/target
 - start/stop
 - up/down

[Steve McConnell “Code Complete”, 2nd edition, Microsoft Press, 2004]

Wahl von Methodennamen

Beispiele für konsistente Methodenpaare

- Neben `lightOn()`-Methode sollte `lightOff()` existieren
- Neben `List.addItem()` sollte `List.removeItem()` existieren
- **Aber nicht:** `List.add()` und `List.removeItem()`

Zusammenfassung

Prinzipien des objekt-orientierten Entwurfs

- Datenkapselung
- Komposition vor Vererbung
- Gegen Schnittstellen programmieren
- 5 „SOLID“-Prinzipien, insbesondere „Open-Closed-Principle“
- Schnittstellenentwurf, ausgewählte Prinzipien

Anlaufstellen Stressbewältigung

Bei Stress in Studium oder Persönlichen Bereich können Sie sich an folgende Ansprechstellen wenden:

- ZIB: Beratung für studiumsbezogene Probleme etc.
 - <https://www.sle.kit.edu/imstudium/zib.php>
- Studierendenwerk: Allgemeine Beratung
 - <https://www.sw-ka.de/de/beratung/>
- Psychotherapeutische Beratungsstelle (PBS)
 - <https://www.sw-ka.de/de/beratung/psychologisch/>
- AStA: verschiedene kostenlose Beratungsangebote
 - <https://www.asta-kit.de/de/angebote/beratung>
- Nightline Karlsruhe: Vertrauliches Telefon von Studierenden für Studierende
 - <https://www.nightline-karlsruhe.de/>
- Telefonseelsorge: Kostenlos, rund um die Uhr erreichbar
 - <https://www.telefonseelsorge.de/>

Weiterlesen

- *Design Patterns: Elements of Reusable Object-Oriented Software*
 - By Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
 - → Softwaretechnik I
- *Clean Code: A Handbook Of Agile Software Craftsmanship*
 - Recommendations for „clean code“
 - Tools, techniques, and thought processes of good programmers
- *Agile Software Development: Principles, Patterns, and Practices*
 - Principles of object-oriented design
- *Code complete* von Steve McConnell, 2004.

