Durch private Hilfsmethode behebbare Codewiederholung - Negativbeispiel

```
public class Heightmap {
    private double [][] mapData;

    public Heightmap(int rows, int   columns) {
        mapData = new double[rows][columns];
    }

    public boolean setHeightAt(int x, int y, double value) {
        if(x >= 0 && y >= 0 && x > mapData.length && y > mapData[0].length) {
            mapData[x][y] = value;
            return false;
        } else {
             return false;
        }
     }

     public double getHeightAt(int x, int y) {
         if(x >= 0 && y >= 0 && x > mapData.length
        && y > mapData[0].length) {
             return mapData[x][y];
         } else {
             throw new IllegalArgumentException();
         }
     }
}
```

Durch private Hilfsmethode behebbare Codewiederholung - Positivbeispiel

```java
public class Heightmap {
    private double [][] mapData;

    public Heightmap(int rows, int columns) {
        mapData = new double[rows][columns];
    }
    public boolean setHeightAt(int x, int y, double value) {
        if(inBounds(x,y)) {
            mapData[x][y] = value;
            return false;
        } else {
            return false;
        }
    }
    public double getHeightAt(int x, int y) {
        if(inBounds(x,y)) {
            return mapData[x][y];
        } else {
           throw new IllegalArgumentException();
        }
     }
     private boolean inBounds(int x, int y) {
         return x >= 0 && y >= 0 && x < mapData.length && y < mapData[0].length;
     }
}
```

Durch private Hilfsmethode behebbare Codewiederholung

- Codewiederholung senkt die Wartbarkeit von Methoden
  - Änderungen müssen an allen kopierten Stellen mehrfach gemacht werden
  - fehlende Übersichtlichkeit & Fehleranfälligkeit
- bei Wiederholtem Code innerhalb einer Klasse
  - lagere diesen in private Hilfsmethode aus
  - kann auch Code mit nur einer Zeile betreffen
- Zum Beispiel:
  - in Heightmap muss mehrfach überprüft werden, ob angegebene Koordinaten innerhalb der definierten Karte sind
  - die Methode inBounds ist leichter lesbar als die Verkettung aus Bedingungen und kann in weiteren Methoden der Heightmap wiederverwendet werden

Durch Vererbung behebbare Codewiederholung - Negativbeispiel

```
public class DigitalClock {
    private int time;
    private final static int S_IN_DAY = 86400;
    private final static int S_IN_HOUR = 3600;
    private final static int SS_IN_MINUTE = 60;
    public DigitalClock(int hours, int minutes,
    int seconds) {
        time = S_IN_HOUR * hours + S_IN_MINUTE
    * minutes +  seconds;
    }
    public void tick() {
        time = (time + 1) % S_IN_DAY;
    }
    public int getHours() {
        return (time / S_IN_HOUR);
    }
    public int getMinutes() {
        return (time % S_IN_HOUR) / S_IN_MINUTE;
    }
    public int getSeconds() {
        return (time % S_IN_HOUR) % S_IN_MINUTE;
    }
}
```

```
public class AnalogClock {
    public AnalogClock(int hours,
    int minutes, int seconds) {
        time = S_IN_HOUR * hours +
     S_IN_MINUTE * minutes + seconds;
    }
    //Methoden tick(), getMinutes(),
getSeconds() wie in DigitalClock
    public int getHours() {
        return (time / S_IN_HOUR) %
12;
    }
}
```

Durch Vererbung behebbare Codewiederholung - Positivbeispiel

```java
public abstract class Clock {
    protected int time;
    private final static int SECONDS_IN_DAY = 86400;
    protected final static int SECONDS_IN_HOUR = 3600;
    private final static int SECONDS_IN_MINUTE = 60;
    public Clock(int hours, int minutes, int seconds) {
        time = SECONDS_IN_HOUR * hours +
    SECONDS_IN_MINUTE * minutes + seconds;
    }
    public void tick() {
        time = (time + 1) % SECONDS_IN_DAY;
    }
    public abstract int getHours();
    public int getMinutes() {
        return (time % SECONDS_IN_HOUR) / SECONDS_IN_MINUTE;
    }
    public int getSeconds() {
        return (time % SECONDS_IN_HOUR) % SECONDS_IN_MINUTE;
    }
}
```

```java
public class AnalogClock extends Clock{
    public AnalogClock(int hours, int minutes,
    int seconds) {
        super(hours,minutes,seconds);
    }
    public int getHours() {
        return (time / SECONDS_IN_HOUR) % 12;
    }
}
```

```java
public class DigitalClock extends Clock{
    public DigitalClock(int hours, int minutes,
    int seconds) {
        super(hours,minutes,seconds);
    }
    public int getHours() {
        return (time / SECONDS_IN_HOUR);
    }
}
```

Durch Vererbung behebbare Codewiederholung

- Codewiederholung senkt die Wartbarkeit von Methoden
  - Änderungen müssen an allen kopierten Stellen mehrfach gemacht werden
  - fehlende Übersichtlichkeit & Fehleranfälligkeit
- bei gemeinsamen Code über mehrere Klassen
  - falls gemeinsame Abstraktion vorhanden ist ␣ lagere in abstrakte Oberklasse aus
- Zum Beispiel:
  - analoge und digitale Uhr unterscheiden sich nur in der Ausgabe der Stunden
  - bei Hinzufügen neuer Methode getSecondsAfterMidnight() müsste man beide Klassen ändern
  - daher: gemeinsames Verhalten in abstrakte Klasse Clock auslagern

Karlsruher Institut für Technologie

Software Design and Quality

Java Datenstrukturen

Java Datenstrukturen

- Wiederverwendung ist eines der Grundprinzipien des Programmierens
- Wenn Datenstrukturen bereits implementiert sind verwenden wir diese
- Vorteile
  - Alle erwarteten Funktionen sind performant implementiert
  - Intensiv getestete und millionenfach verwendete Implementierung

Java Datenstrukturen

- Wir geben einige Beispiele für benötigte Datenstrukturen
  - Arrays,
  - Listen (java.util.List<T>) mit Implementierungen
    - LinkedList, oder
    - ArrayList
  - Stacks (java.util.Stack<T>)
  - Queues (java.util.Queue<T>)
  - Mengen (java.util.Set<T> mit Implementierung HashSet oder TreeSet)

Einheitliche Sprache

- Kommentare und Ausgaben sollen in einheitlicher Sprache verfasst sein.
- Entweder deutsch oder englisch
- Nicht beides gemischt in einem Programm

- Warum?
  - Stört den Lesefluss
  - Kann zu Missverständnissen führen
  - Handelt es sich bei zwei verschiedenen Wörtern um eine Übersetzung oder um verschiedene Konzepte

Einheitliche Sprache: Negativbeispiel

```
/**
* Legt ein Gegenstand in das Lager.
*
* @param Der einzulagernde Gegenstand.
*/
void storeItem(Item item);

/**
* Checks if an item is in stock.
*
* @param item The item to search for.
* @return True if item is in stock, false otherwise.
*/
boolean checkStock(Item item);
```

Einheitliche Sprache

```
/**
* Stores an item. After this the item we be in stock.
*
* @param item The item to store
*/
void storeItem(Item item);

/**
* Checks if an item is in stock. This will be the case if and only if the item
* was previously stored with {@link #storeItem(Item)}.
*
* @param item The item to search for.
* @return True if item is in stock, false otherwise.
*/
boolean checkStock(Item item);
```

Fallunterscheidung mit enums statt dynamischer Typbindung

- gibt es für ein Objekt eine vordefinierte Anzahl von Typen, ist die Verwendung von enums von naheliegend
- Problem: Hinzufügen von neuen Typen macht Änderung des Codes an vielen Stellen nötig
- Lösung: Vererbung statt enums
  - Umsetzung von **switch-less-Programming**
  - Klassen sollten geschlossen für Änderung des Codes aber offen für Erweiterung in Form von Vererbung sein
  - nutze dynamische Typbindung durch überschriebene Methode in Unterklasse statt Fallunterscheidung
- Zum Beispiel
  - in payEntryFee muss im Negativbeispiel der Typ des Kunden überprüft werden
  - im Positivbeispiel „automatisch“ durch dynamische Typbindung der richtige Fall ausgeführt

Negativbeispiel

```
public class Customer {

    private double balance;
    private CustomerType type;

    public Customer(double balance,
     CustomerType type) {
        this.balance = balance;
        this.type = type;
    }

    double payEntryFee() {
        double fee;
        switch(type) {
            case CHILD : fee = 5.00;
break;
            case ADULT : fee = 10; break;
            case SENIOR : fee = 8; break;
        }
        balance =- fee;
        return fee;
    }
}
```

Positivbeispiel

```
public abstract class Customer {

    private double balance;

    public Customer(double balance) {
        this.balance = balance;
    }

    double payEntryFee() {
        double fee = getEntryFee();
        balance =- fee;
        return fee;
    }

    abstract double getEntryFee();
}


public class Child extends Customer
{
    public Child(double balance) {
        super(balance);
    }
    double getEntryFee() {
        return 7.5;
    }
}
```

Karlsruher Institut für Technologie

Software Design and Quality

Enums

Enums

- Verwendung: Modelliere feste, bekannte Wertemenge mit klar identifizierbaren Inhalten:
- Negativbeispiel

```
public class Weekday {
    int day;
}
```

- Nachteil: Verwendet muss Implementierungsdetails kennen
  - Entspricht 0 Montag, oder ist 1 Montag, oder doch 42?
  - Wie wird mit den ungültigen $2^{32} – 5$ Werten umgegangen?

Enums

- Ausdrucksstarke Modellierung mit Enums

```
public enum Weekday {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY
}
```

- Vorteile:
  - Werte sind durch ausdrucksstarke Bezeichner angegeben
  - Gültigkeit kann bereits durch den Compiler überprüft werden

Unspezifische Fehlermeldungen

- Fehlermedlungen müssen aussagekräftig sein.
- Inhalt einer guten Fehlermeldung
  - Grund für den Fehler
  - Wo in der Eingabe ist der Fehler
  - Hinweis auf eine richtige Eingabe

Negativbeispiel:
$ move red e
Error!
$ move red 4
Error!

Besser:
$ move red e
Error in second argument! An integer in the range between 1 and 6 is expected.
$ move red 3
Error, field 3 is not free! Type "print" to view the game board.

Fehlende Trennung von Programmlogik und UI - Negativbeispiel

Karlsruher Institut für Technologie

```java
public class UI {

    public static void main (String [] args) {
        Account a = new Account(100);
        while(true) {
            System.out.println("Input an
Integer");
            String input = "";
            //get user input
            int amount = 0;
            try {
                amount =
Integer.parseInt(input);
            } catch (NumberFormatException e) {
                continue;
            }
            a.transfer(amount);
        }
    }
}
```

```java
public class Account {

    private int balance;

    public Account(int startingBalance) {
        balance = startingBalance;
    }

    public void transfer(int amount) {
        if(balance + amount > 0) {
            balance += amount;
            System.out.println("balance:" + balance);
        } else {
            System.out.println("Unable to make
transfer. Your balance is " + balance);
        }
    }
}
```

Fehlende Trennung von Programmlogik und UI - Positivbeispiel

```
public class UI {
    public static void main (String [] args) {
        Account a = new Account(100);
        while(true) {
            System.out.println("Input an Integer");
            String input = "";
            //get user input
            int amount = 0;
            try {
                amount = Integer.parseInt(input);
            } catch (NumberFormatException e) {
                continue;
            }
            if(a.transfer(amount)) {
                System.out.println("balance: " +
a.getBalance());
            } else {
                System.out.println("Unable to make transfer.
Your          balance is " + a.getBalance());
            }
        }
    }
}
```

```
public class Account {

    private int balance;

    public Account(int startingBalance)
{
        balance = startingBalance;
    }

    public boolean transfer(int amount)
{
        if(balance + amount > 0) {
            balance += amount;
            return true;
        } else {
            return false;
        }
     }

     public int getBalance() {
         return balance;
     }
}
```

Fehlende Trennung von Programmlogik und UI

- es sollte eine klare Trennung zwischen User Interaktion und der eigentlichen Programmlogik geben
  - UI sollte von Programmlogik abhängig sein aber nicht umgekehrt
  - die Art der UI sollte ohne Änderung der Klassen der Programmlogik austauschbar sein
- Symptome von fehlender Trennung:
  - Ausgabe über das Terminal in Klassen der Programmlogik
  - Methoden in Logikklassen geben statt „rohen“ Datentypen die Stringrepräsentation dieser zurück
  - Methoden in Logikklassen erhalten die Eingabe in Stringform und parsen diese
- Zum Beispiel:
  - im Negativbeispiel gibt Account das Resultat des Transfers nur über das Terminal aus
  - im Positivbeispiel gibt Account den Erfolg des Transfers zurück und hat Methode, um den Kontostand auszugeben ␣ kann auch im Kontext einer GUI verwendet werden

Karlsruher Institut für Technologie

Software Design and Quality

Implementieren Gegen Interface

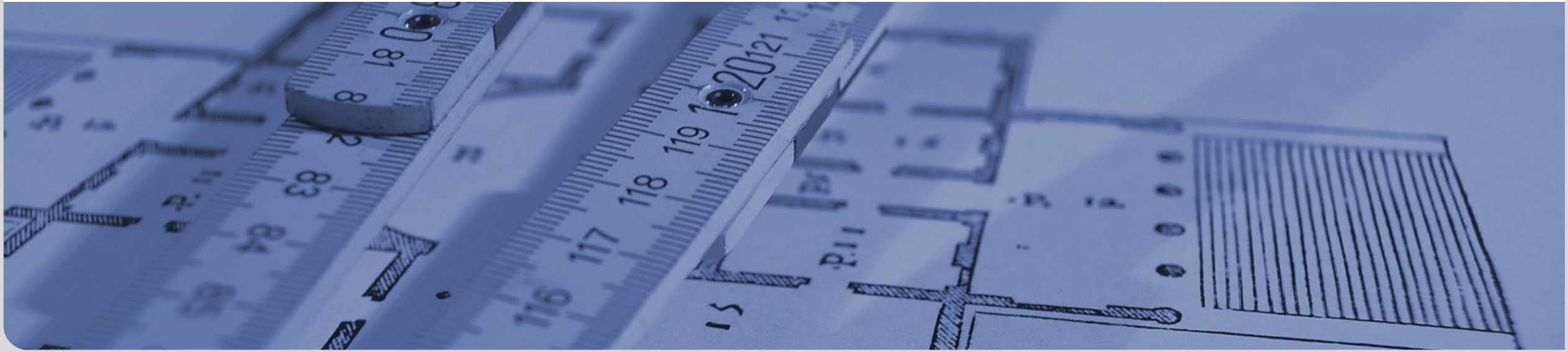
Implementieren Gegen Interface

- Es sollte grundsätzlich das Interface im Gegensatz zur konkreten Klasse als konkreter Typ verwendet werden.
- Gängige Frage: Welche Implementierung ist zu bevorzugen?
  - (a) LinkedList list = new LinkedList()
  - (b) List list = new LinkedList()
- Zwei Gründe für (b) (und gegen (a))
  - Entkopplung der Verwendung (als Liste) von der Implementierung (LinkedList).
  - Einfachere Wartbarkeit: Wenn wir die Datenstruktur ändern möchten (bspw. zu ArrayList) müssen alle Vorkommen bei Variante (a) angepasst werden. Bei (b) genau dieses eine Vorkommen

Karlsruher Institut für Technologie

Bewertungsrichtlinie: JavaDoc Leer

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Richtlinie

- Die Bewertungsrichtlinie "JavaDoc Leer" besagt,
  - dass es für nicht-private, also
    - öffentliche (public)
    - geschützte (protected)
    - paket-private (package-private, no modifier)
    - Klassen
    - Enums
    - Interfaces
    - Methoden
    - Konstanten
    - Attribute
  - stets einen aussagekräftigen JavaDoc Kommentar geben sollte.

Negativbeispiel

```
public String getName() {
    return this.name;
}
```

Verbessertes Beispiel

```
/**
 * Gets the full name of the person, i.e. the first name(s) separated by
 * whitespaces, followed by a whitespace and the last name.
 * This method returns {@code null} if no name was set.
 *
 * @return the full name of the person or {@code null} if no name was set
 */
public String getName() {
    return this.name;
}
```

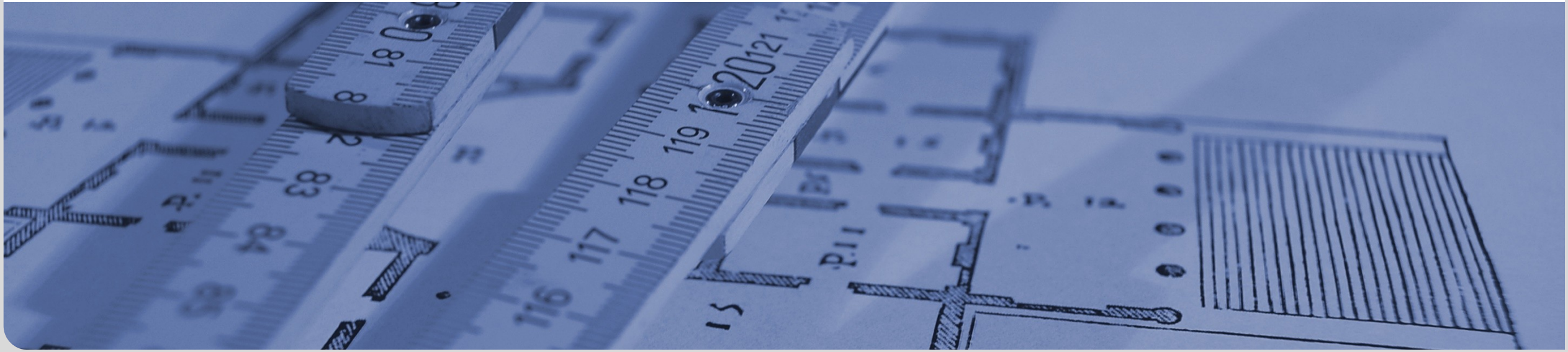
Siehe auch

- Bewertungsrichtlinie: JavaDoc Trivial
- Programmieren Wiki: Methodik / Dokumentation durch Kommentare
  - https://ilias.studium.kit.edu/goto.php?target=wiki_wpage_31795_1275181&client_id=produktiv

Karlsruher Institut für Technologie

Bewertungsrichtlinie: JavaDoc Trivial

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Richtlinie

- Die Bewertungsrichtlinie "JavaDoc Trivial" besagt,
  - dass es für nicht-private, also
    - öffentliche (public)
    - geschützte (protected)
    - paket-private (package-private, no modifier)
    - Klassen
    - Enums
    - Interfaces
    - Methoden
    - Konstanten
    - Attribute
  - stets einen **aussagekräftigen** JavaDoc Kommentar geben sollte.

Negativbeispiel

```
/**
 *
 * @return name
 */
public String getName() {
    return this.name;
}
```

Verbessertes Beispiel

```
/**
 * Gets the full name of the person, i.e. the first name(s) separated by
 * whitespaces, followed by a whitespace and the last name.
 * This method returns {@code null} if no name was set.
 *
 * @return the full name of the person or {@code null} if no name was set
 */
public String getName() {
    return this.name;
}
```

Siehe auch

- Bewertungsrichtlinie: JavaDoc Leer
- Programmieren Wiki: Methodik / Dokumentation durch Kommentare
  - https://ilias.studium.kit.edu/goto.php?target=wiki_wpage_31795_1275181&client_id=produktiv

Karlsruher Institut für Technologie

Software Design and Quality

Java Sortierung

Java Sortierung

- Wiederverwendung ist eines der Grundprinzipien des Programmierens
- Wenn Funktionalität bereits implementiert ist verwenden wir diese
- Java stellt Klassen mit Methoden zum Sortieren bereit
- Für Listen verwenden wir
- Collections.sort(List<T> list)
  - Hiermit können alle Listen sortiert werden, deren Typ T das Interface Comparable<T> implementiert

Java Sortierung

- Wiederverwendung ist eines der Grundprinzipien des Programmierens
- Wenn Funktionalität bereits implementiert ist verwenden wir diese
- Java stellt Klassen mit Methoden zum Sortieren bereit
- Für Arrays verwenden wir
- Arrays.sort(T[] array)
  - Hiermit können alle Arrays sortiert werden, deren Typ das Interace Comparable<T> implementiert
- Wir geben zwei Beispiele

Java Sortierung

```java
import java.util.Collections;
import java.util.List;

public class HelloWorld {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("ist");
        list.add("cool");
        list.add("programmieren");

        Collections.sort(list);

        System.out.println(list);
    }
}
```

Java Sortierung

```java
import java.util.Arrays;

public class HelloWorld {
    public static void main(String[] args) {
        String[] list = {"ist", "cool", "programmieren"};
        Arrays.sort(list);
        System.out.println(Arrays.toString(list));
    }
}
```

Unnötige Komplexität

- Es soll ein möglichst verständliches Programm geschrieben werden
- Minimale Komplexität, um Funktionalität zu implementieren
- Kein Wettbewerb wer kürzestes Programm schreibt (Codecolf)

Negativbeispiel:

```
boolean checkBuy() {
    if (checkBalance() && !checkShoppingCartEmpty()) {
        return true;
    }
    else {
        return false;
    }
}
```

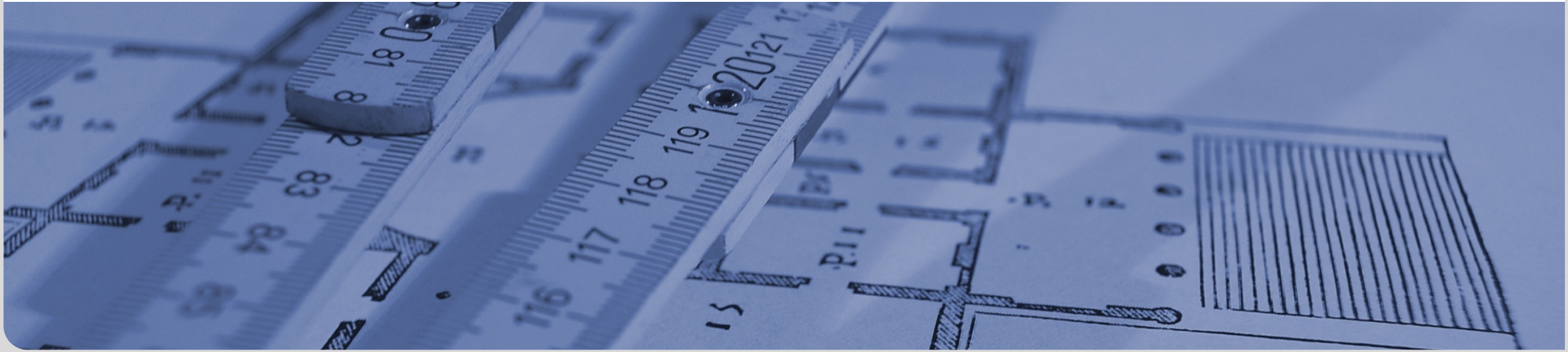
Besser:

```
boolean checkBuy() {
    return checkBalance() && !checkShoppingCartEmpty();
}
```

Karlsruher Institut für Technologie

Bewertungsrichtlinie: Magic Number

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Richtlinie

- Die Bewertungsrichtlinie "Magic Number / Strings" besagt,
  - dass
    - konstante Zahlen
    - Zeichenketten (Strings)
  - als Konstanten modelliert werden sollten, um den Code
    - lesbarer
    - verständlicher
    - besser wartbar
  - zu machen.

Negativbeispiel

```
final String password = "Test";
if (password.length() < 8) {
    System.err.println("Password too short!");
}
```

Verbessertes Beispiel

```
final String password = "Test";
final int minPasswordLength = 8;
if (password.length() < minPasswordLength) {
    System.err.println("Password too short!");
}
```

- Anmerkungen:
  - Wenn die Konstante in mehreren Methoden der selben Klasse verwendet werden soll, dann sollte die Konstante entsprechend als eine `private static final` Klassen-Konstante modelliert werden.
  - Wenn die Konstante auch in anderen Klassen benötigt wird, sollte die Konstante als eine `public static final` Klassen-Konstante modelliert werden.

Siehe auch

- https://en.wikipedia.org/wiki/Magic_number_(programming)
- Programmieren Wiki: Methodik / Magic Numbers
  - https://ilias.studium.kit.edu/goto.php?target=wiki_wpage_31815_1275181&client_id=produktiv

Verwendung von Object als Datentyp

- Datentypen sollten immer so spezifisch wie nötig aber so allgemein wie möglich gewählt werden
- Object ist als Datentyp eigentlich immer zu allgemein
  - man benötigt fast immer eine Methode einer spezifischeren Klasse
- wenn Klasse mit mehreren verschiedenen Datentypen funktionieren soll, sind Generics geeigneter
- Zum Beispiel:
  - im Negativbeispiel wird für den Containerdatentyp Pair Object als Typ der Elemente verwendet ⇨ mit getFirst() zurückgegebene Elemente müssen explizit auf das benötigte Objekt gecasted werden

Negativbeispiel

```
public class Pair {
    private Object first;
    private Object second;
    public Pair(Object first,
     Object second) {
         this.first = first;
         this.second = second;
    }
    public Object getFirst() {
        return first;
    }
    public Object getSecond() {
        return second;
    }
}
public static void main(String [] args)
{
    Date startTime = /* get start time*/
    Date endTime = /* get end time*/
    Pair timeSpan = new Pair(startTime,
     endTime);
    //some code ...
    if(((Date) timeSpan.getFirst())
     .before(now)
     && ((Date) timeSpan.getSecond())
     .before(now)) {
         System.out.println("Current
time
     within timespan");
     }
}
```

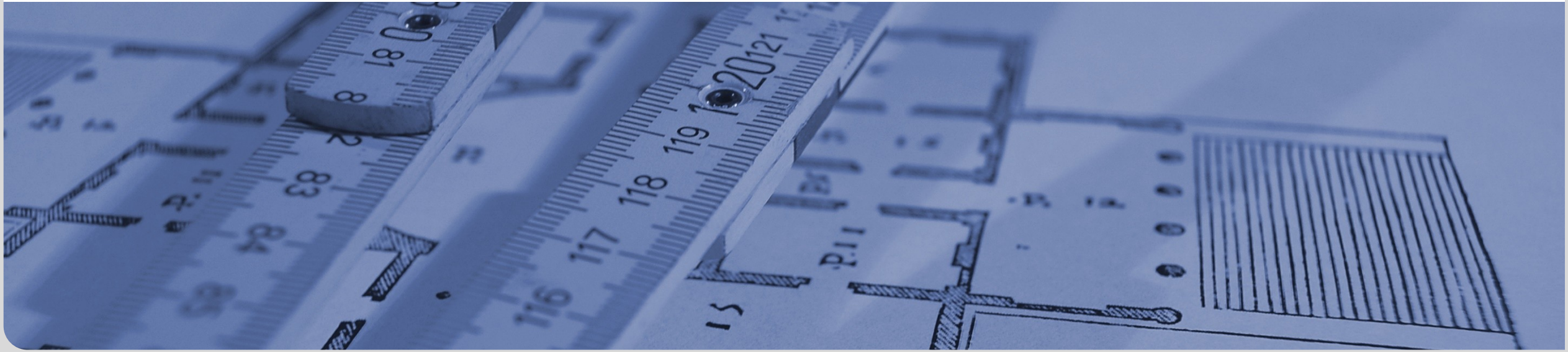
Positivbeispiel

```
public class Pair<T> {
    private T first;
    private T second;
    public Pair(T first, T second)
{
        this.first = first;
        this.second = second;
    }
    public T getFirst() {
        return first;
    }
    public T getSecond() {
        return second;
    }
}
public static void main(String [] args)
{
    Date startTime = /* get start time*/
    Date endTime = /* get end time*/
    Pair timeSpan =
       new Pair<Date>(startTime,
endTime);
    //some code ...
    if(timeSpan.getFirst().before(now)
     && timeSpan.getSecond()
     .before(now) {
         System.out.println("Current
time
     within timespan");
     }
}
```

Karlsruher Institut für Technologie

Bewertungsrichtlinie: Pakete

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Richtlinie

- Die Bewertungsrichtlinie "Pakete" besagt,
  - dass der Code sinnvoll in Pakete unterteilt werden sollte, um
    - Namenskonflikte zu verhindern und
    - ähnliche Typen zu gruppieren.
- Pakete sind außerdem gut geeignet um
  - (große) Projekte sinnvoll zu strukturieren.

Negativbeispiel

```
package edu.kit.informatik;
```

- für alle Klassen

Verbessertes Beispiel

`package edu.kit.informatik.ui;`

- für die User-Interface Klassen

`package edu.kit.informatik.data;`

- für die Data Klassen

- Anmerkung: Am besten noch genauere Aufteilung falls nötig und möglich

Siehe auch

- https://en.wikipedia.org/wiki/Java_package
- https://docs.oracle.com/javase/tutorial/java/package/packages.html

Ungeschickte Lösung (Quatsch ¯_(ツ)_/¯)

- Was von einer Abgabe **nicht** erwartet wird:
  - Effizitester Algorithmus
  - Lehrbuchmodellierung
- Aber sinnvolle Lösung innerhalb des bisherigen Stoff der Vorlesung

Negativbeispiel (siehe de.wikipedia.org/wiki/Bogosort):

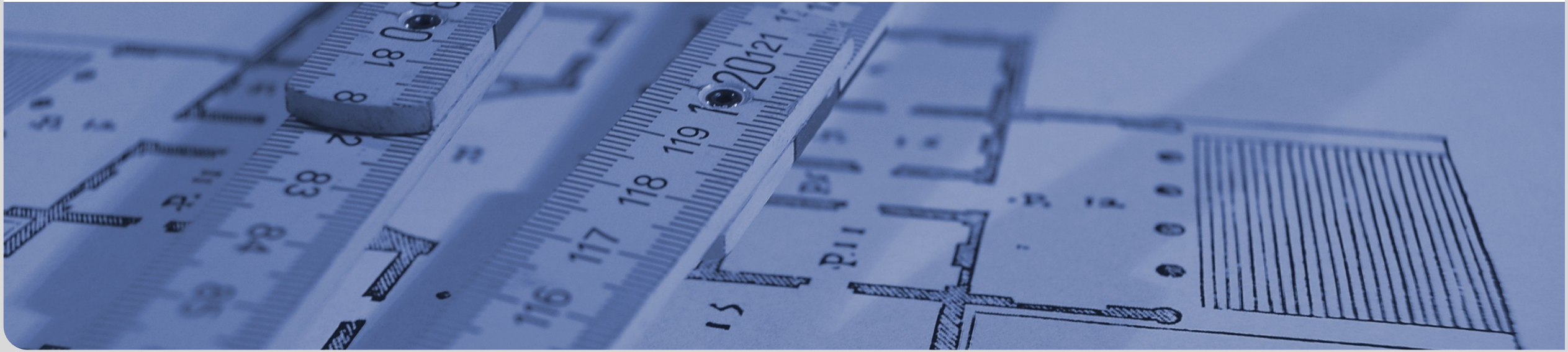
```
Random r = new Random();
while (!isSorted(toSort)) { // Prüfen, ob sortiert
        int a = r.nextInt(toSort.length);
        int b = r.nextInt(toSort.length);
        int temp = toSort[a];
        toSort[a] = toSort[b];
        toSort[b] = temp;
}
return toSort;
```

Besser: Mergesort, Bubblesort oder, falls erlaubt, Arrays.sort(int[])

Karlsruher Institut für Technologie

Bewertungsrichtlinie: Schlechte Bezeichner

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Richtlinie

- Die Bewertungsrichtlinie "Schlechte Bezeichner" besagt,
  - dass Bezeichner immer sprechend sein sollten, d.h. es sollten
    - keine unnötigen Abkürzungen oder
    - nichtssagende Bezeichner
  - gewählt werden.
  - Außerdem sollten sich Bezeichner immer an die Java Namenskonventionen halten.

Negativbeispiel

```
public static int s(final int a, final int b) {
    return a - b;
}
```

Verbessertes Beispiel

```
public static int subtract(final int a, final int b) {
    return a - b;
}
```

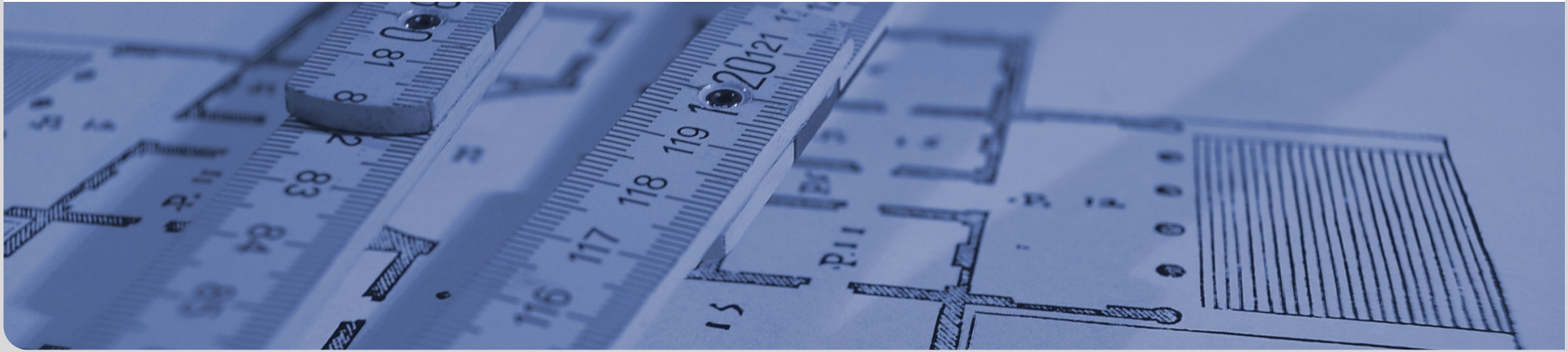
Siehe auch

- https://en.wikipedia.org/wiki/Naming_convention_(programming)
- https://www.oracle.com/java/technologies/javase/codeconventions-namingconventions.html
- Programmieren Wiki: Methodik / Programmierstil
  - https://ilias.studium.kit.edu/goto.php?target=wiki_wpage_31819_1275181&client_id=produktiv

Karlsruher Institut für Technologie

Bewertungsrichtlinie: Schwieriger Code

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Richtlinie

- Die Bewertungsrichtlinie "Schwieriger Code" besagt,
  - dass schwierige Codestellen
    - mit einem oder mehreren **Entwicklerkommentar(en)** kommentiert werden sollten und
    - - falls möglich - der schwierige Code in **geeignete Hilfsmethoden** strukturiert werden sollte.

Negativbeispiel

```
/**
 * Creates a new LudoBoard with standard start board state.
 */
public LudoBoard() {
    this.normalFields = new ArrayList<Field>();
    this.startFieldMap = new HashMap<PlayerColor, Field>();
    this.finalFieldsMap = new HashMap<<PlayerColor, List<Field>>();
    this.playerToFieldsMap =
            new HashMap<PlayerColor, SortedSet<Field>>();
    for (int i = 0; i < NUMBER_OF_NORMAL_FIELDS; i++) {
        final Field field = new Field(i);
        this.normalFields.add(field);
    }
```

Negativbeispiel

```
final PlayerColor[] orderedColors = PlayerColor.valuesOrdered();
for (final PlayerColor color : orderedColors) {
    final Field startField = new Field("" + PREFIX_NAME_START_FIELD + color.getPattern());
    this.startFieldMap.put(color, startField);
    final List<Field> finalFields = new ArrayList<>();
    for (int i = 0; i < NUMBER_OF_MEEPLES_PER_PLAYER; i++) {
        final Meeple meeple = new Meeple(color, i);
        startField.pushMeepleOnThisField(meeple);

        final char prefix = (char) (PREFIX_NAME_FINAL_FIELD + i);
        final Field finalField = new Field("" + prefix + color.getPattern());
        finalFields.add(finalField);
    }
    this.finalFieldsMap.put(color, finalFields);
}
```

Negativbeispiel

```
        for (final PlayerColor color : orderedColors) {
            final SortedSet<Field> set = new TreeSet<>();
            final Field startField = this.startFieldMap.get(color);
            if (!startField.isEmpty()) {
                set.add(startField);
            }
            for (final Field field : getNormalFields()) {
                if (!field.isEmpty() && field.getFirstMeepleOnThisField().getColor() == color) {
                    set.add(field);
                }
            }
            for (final Field field : getFinalFields(color)) {
                if (!field.isEmpty()) {
                    set.add(field);
                }
            }
            this.playerToFieldsMap.put(color, set);
        }
    }
```

Verbessertes Beispiel

```
/**
 * Creates a new LudoBoard with standard start board state.
 */
public LudoBoard() {
    this.normalFields = new ArrayList<Field>();
    this.startFieldMap = new HashMap<PlayerColor, Field>();
    this.finalFieldsMap = new HashMap<<PlayerColor, List<Field>>();
    this.playerToFieldsMap = new HashMap<PlayerColor, SortedSet<Field>>();
    initialize();
    initializePlayerToFieldsMap();
}
```

Verbessertes Beispiel

```
private void initialize() {
    // creating normal fields
    for (int i = 0; i < NUMBER_OF_NORMAL_FIELDS; i++) {
        final Field field = new Field(i);
        this.normalFields.add(field);
    }
    final PlayerColor[] orderedColors = PlayerColor.valuesOrdered();
    for (final PlayerColor color : orderedColors) {
        // creating start field
        final Field startField = new Field("" + PREFIX_NAME_START_FIELD + color.getPattern());
        this.startFieldMap.put(color, startField);
        final List<Field> finalFields = new ArrayList<>();
        // filling start field and creating final fields
        for (int i = 0; i < NUMBER_OF_MEEPLES_PER_PLAYER; i++) {
            final Meeple meeple = new Meeple(color, i);
            startField.pushMeepleOnThisField(meeple);
            final char prefix = (char) (PREFIX_NAME_FINAL_FIELD + i);
            final Field finalField = new Field("" + prefix + color.getPattern());
            finalFields.add(finalField);
        }
        this.finalFieldsMap.put(color, finalFields);
    }
}
```

Verbessertes Beispiel

```
/*
 * Initializes the player to fields map, i.e. for each player p: puts all the fields on which a meeple of p stands as p's field set.
 */
private void initializePlayerToFieldsMap() {
    final PlayerColor[] orderedColors = PlayerColor.valuesOrdered();
    for (final PlayerColor color : orderedColors) {
        final SortedSet<Field> set = new TreeSet<>();
        final Field startField = this.startFieldMap.get(color);
        if (!startField.isEmpty()) {
            set.add(startField);
        }
        for (final Field field : getNormalFields()) {
            if (!field.isEmpty() && field.getFirstMeepleOnThisField().getColor() == color) {
                set.add(field);
            }
        }
        for (final Field field : getFinalFields(color)) {
            if (!field.isEmpty()) {
                set.add(field);
            }
        }
        this.playerToFieldsMap.put(color, set);
    }
}
```

Siehe auch

- https://en.wikipedia.org/wiki/Comment_(computer_programming)
- https://java.soeinding.de/content.php/kommentare

Seiteneffekte

- Seiteneffekt: Alles, was eine Methode abseits der Berechnung des Rückgabetyps tut.
  - Z. B. Ausgaben, setzen von Attributen, …
- Seiteneffekte müssen explizit dokumentiert sein
- Guter Stil: Methoden mit Seiteneffekt haben Rückgabetyp void (Command-query separation)

Seiteneffekt Negativbeispiel

```
double computeMedian(int[] data) {
    Arrays.sort(data);
    if (data.length % 2 == 0) {
        return ((double) data[data.length / 2] + (double) data[data.length / 2 - 1]) / 2;
    }
    return data[data.length / 2];
}
```

Seiteneffekte

- Alternative 1: Seiteneffekt entfernen

```
double computeMedian(int[] data) {
    int[] copy = new int[data.length];
    // Efficient Java-API method to copy arrays
    System.arraycopy(data, 0, copy, 0, data.length);
    Arrays.sort(copy);
    if (copy.length % 2 == 0) {
        return ((double) copy[copy.length / 2] + (double) copy[copy.length / 2 - 1]) / 2;
    }
    return copy[copy.length / 2];
}
```

Seiteneffekte

- Alternative 2: Seiteneffekt dokumentieren

```
/**
* Sort and compute the median of an given array.
*
* @param data The data to calculate the median of. The passed in array-Object will be sorted.
* @return the median of data.
*/
double sortAndcomputeMedian(int[] data) {
    Arrays.sort(data);
    if (data.length % 2 == 0) {
        return ((double) data[data.length / 2] + (double) data[data.length / 2 - 1]) / 2;
    }
    return data[data.length / 2];
}
```

Karlsruher Institut für Technologie

Software Design and Quality

Ungeeigneter Schleifentyp

Ungeeigneter Schleifentyp

- Java stellt verschiedene Schleifentypen bereit
  - for-Schleifen zum Iterieren von indizierten Datenstrukturen
  - for-each Schleifen zum Iterieren von Datensturkturen ohne Indizierung
  - while-Schleifen zum Iterieren abhängig von einer Bedingung
  - do-while-Schleifen zum Iterieren abhängig von einer Bedingung

Ungeeigneter Schleifentyp

- Java stellt verschiedene Schleifentypen bereit
- Grundsätzlich ist es möglich jede Iteration mit jedem Schleifentyp zu realisieren
- Jedoch eignen sich bestimmte Schleifentypen für bestimmte Zwecke besser als andere
- Wir geben ein Beispiel

Ungeeigneter Schleifentyp

- Wir geben zwei Implementierungen um die Elemente eines Arrays auszugeben

```
String[] a = new String[100];
// fülle Array mit 100 Elementen
```

```
int i = 0;
while (i < 100) {
    System.out.println(a[i]);
    i++;
}
```

```
for (int i = 0; i < 100; i++) {
    System.out.println(a[i]);
}
```

Karlsruher Institut für Technologie

SDQ Software Design and Quality

Wildcard Import

Wildcard Import

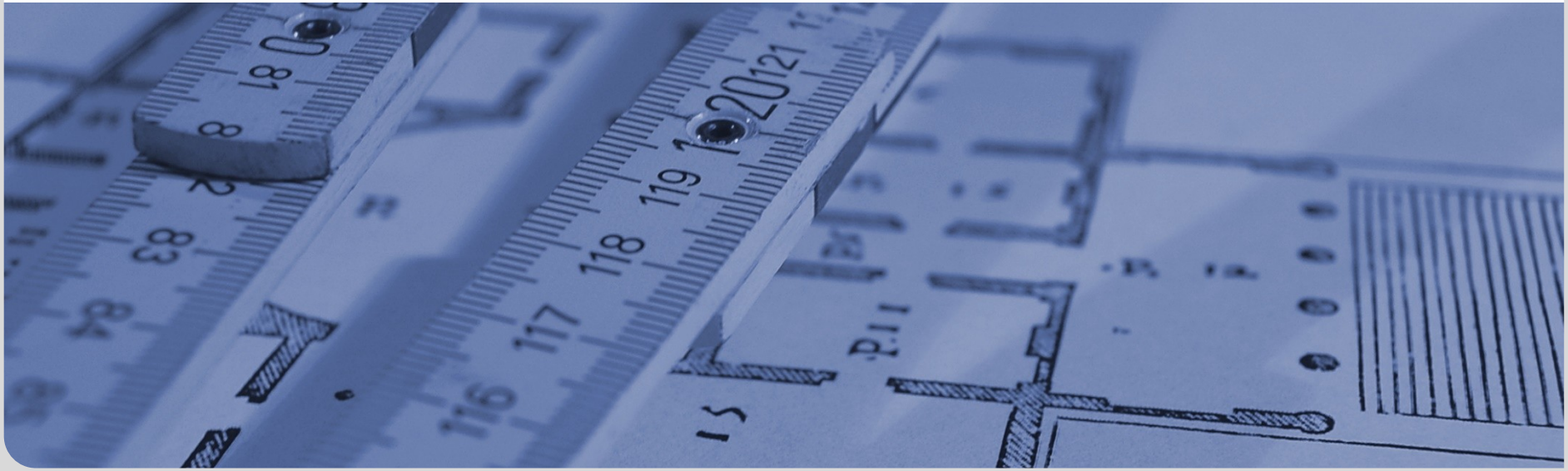
- Wie: import java.util.*
- Was: Importiert alle Klassen des packages
- Problem: Überfüllung des Namensraums
- Einfaches Beispiel
  - Sowohl java.sql.* als auch java.util.* importieren eine Klasse Names Date
  - Initialisierung Date date nicht mehr eindeutig
    - Ein speziellerer Bezeichner wäre nötig
- Lösung: Importiere nur java.sql.Date (und die gewünschten in util)

Karlsruher Institut für Technologie

Bewertungsrichtlinien

Getter & Setter für Listen

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Getter & Setter für Listen

- Nie komplexe veränderliche Objekte, Arrays oder Collections (z.B. ArrayList, HashMap) in direkter Form setzen oder ausgeben
- Mögliche Folgen: Geheimnisverrat & Manipulation
- Beispiel:

```
// Innerhalb unserer Klasse
private int[] array;

public void setSomeArray(int[] array) {
    this.array = array;
}

// Und an einer anderen Stelle
int[] somewhere = {0, 42, 1337, 4711};
setSomeArray(somewhere); // array = {0, 42, 1337, 4711}
somewhere[2] = 666; // array = {0, 666, 1337, 4711}
```

- Erstes Ziel: Schnittstellen ohne Arrays und Listen gestalten, durch Methoden für einzelne Elemente anhand der Eigenschaften, statt der Gesamtmenge
- Alternativ *System.arraycopy(…)*, *clone()* oder nicht veränderliche Objekte verwenden

Verwendung von instanceof - Negativbeispiel

```
public class Bibliography {
    private LinkedList<Literature> literatureList;
    public Bibliography() {
        literatureList = new LinkedList<Literature>();
     }
     public void addSource(Literature source) {
        literatureList.add(source);
     }
     public String printBibliography() {
         String output = "";
         for(Literature l : literatureList) {
            output += l.print();
            if((l instanceof Book)) {
                output += ", pages " + ((Book) l).getStart()+ " - " +
((Book)l).getEnd();
            } else if (l instanceof Paper) {
                output +=  ", " + ((Paper) l).getJournal();
            } else if (l instanceof Website) {
                output += " accessed at " + ((Website)
l).getAccessDate().toString();
             }
            output+= "\n";
         }
         return output;
     }
}
```

```
public class Literature {
    private final String author;
    private final String title;
    public Literature(String author, String title) {
        this.author = author;
        this.title = title;
    }
    public String getAuthor() {
        return author;
    }
    public String getTitle() {
        return title;
     }
    public String print() {
         return author + ": \"" + title + "\"";
    }
}
```

```
public class Paper extends Literature{
    private String journal;
    public Paper(String author, String title, String journal)
 {
        super(author, title);
        this.journal = journal;
    }
    public String getJournal() {
        return journal;
    }
}
```

```
public class Website extends Literature {
    private Date accessDate;
    public Website(String author,
       String title, Date access)  {
        super(author, title);
        accessDate = access;
    }
    public Date getAccessDate() {
        return accessDate;
    }
}
```

```
public class Book extends Literature{
    private int pageStart;
    private int pageEnd;
    public Book(String author, String
title, int pageStart, int pageEnd) {
        super(author, title);
        this.pageStart = pageStart;
        this.pageEnd = pageEnd;
    }
    public int getStart() {
        return pageStart;
    }
    public int getEnd() {
        return pageEnd;
    }
}
```

Verwendung von instanceof - Positivbeispiel

```
public class Bibliography {
    private LinkedList<Literature> literatureList;
    public Bibliography() {
        literatureList = new LinkedList<Literature>();
    }
    public void addSource(Literature source) {
        literatureList.add(source);
    }
    public String printBibliography() {
        String output = "";
        for(Literature l : literatureList) {
            output += l.print() + "\n";
        }
        return output;
    }
}
```

```
public class Literature {
    private final String author;
    private final String title;
        public Literature(String author, String title) {
            this.author = author;
            this.title = title;
        }
     public String print() {
         return author + ": \"" + title + "\"";
     }
}
```

```
public class Paper extends Literature{
    private String journal;

    public Paper(String author, String title,
    String journal) {
        super(author, title);
        this.journal = journal;
    }
    @Override
    public String print() {
       return super.print() + ", " + journal;
    }
}
```

```
public class Book extends Literature{
    private int pageStart;
    private int pageEnd;
    public Book(String author, String title,
     int pageStart, int pageEnd) {
        super(author, title);
        this.pageStart = pageStart;
        this.pageEnd = pageEnd;
    }
    @Override
    public String print() {
        return super.print() + ", pages „ at "
+ pageStart + " - " + pageEnd;
    }
}
```

```
public class Website extends Literature{
    private Date accessDate;
    public Website(String author, String
title,    Date access) {
        super(author, title);
        accessDate = access;
    }
    @Override
    public String print() {
        return super.print() + " accessed

    + accessDate.toString();
    }
}
```

Verwendung von instanceof

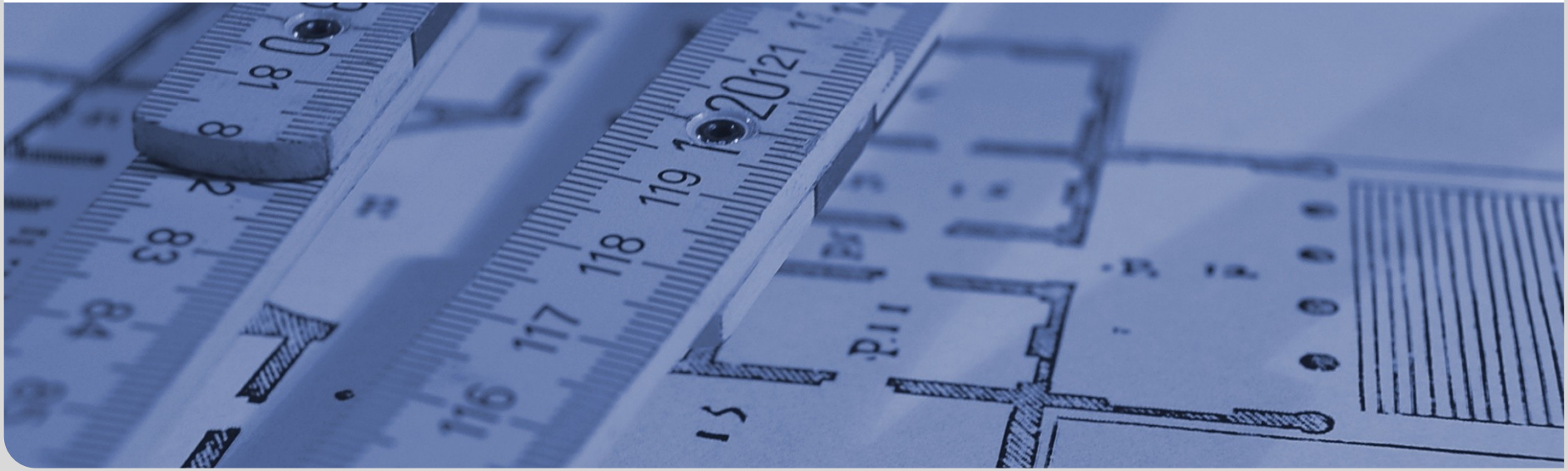
- instanceof darf nur in Ausnahmefällen verwendet werden
  - z.B. in equals()-Methode
- instanceof verhindert dynamische Typbindung, da explizit auf bestimmte Unterklasse geprüft werden muss
  - kein switch-less-Programming
- Verwendung von instanceof ist oft Symptom schlechter Abstraktion
- Zum Beispiel:
  - im Negativbeispiel muss instanceof verwendet werden, da keine sinnvolle Methode zur Stringrepräsentation der Literatur vorhanden ist
  - im Positivbeispiel wird durch dynamische Typbindung immer die print-Methode der korrekten Unterklasse aufgerufen

Karlsruher Institut für Technologie

Bewertungsrichtlinien

Keine Objektorientierte Modellierung

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Keine Objektorientierte Modellierung

- Objektorientierte Modellierung ist ein Grundkonzept der Vorlesung
- Indizien für schlechten Aufbau:
  - Übergroße Klassen (Gottklassen)
  - Nur Transferobjekte (Datenpakete durch das Gesamtprogramm schicken)
  - Viele statische Methoden in Hilfsklassen (*Utility classes*)
- Beispiel: Eine Klasse *Flugzeug*, die die gesamte Ein- und Ausgabe, Verarbeitung und Speicherung einer etwaigen Simulation übernimmt, statt Aufteilen in *Simulation*, *Kommando-System*, *Flug*, *Datum*, *Flugzeug*, usw.

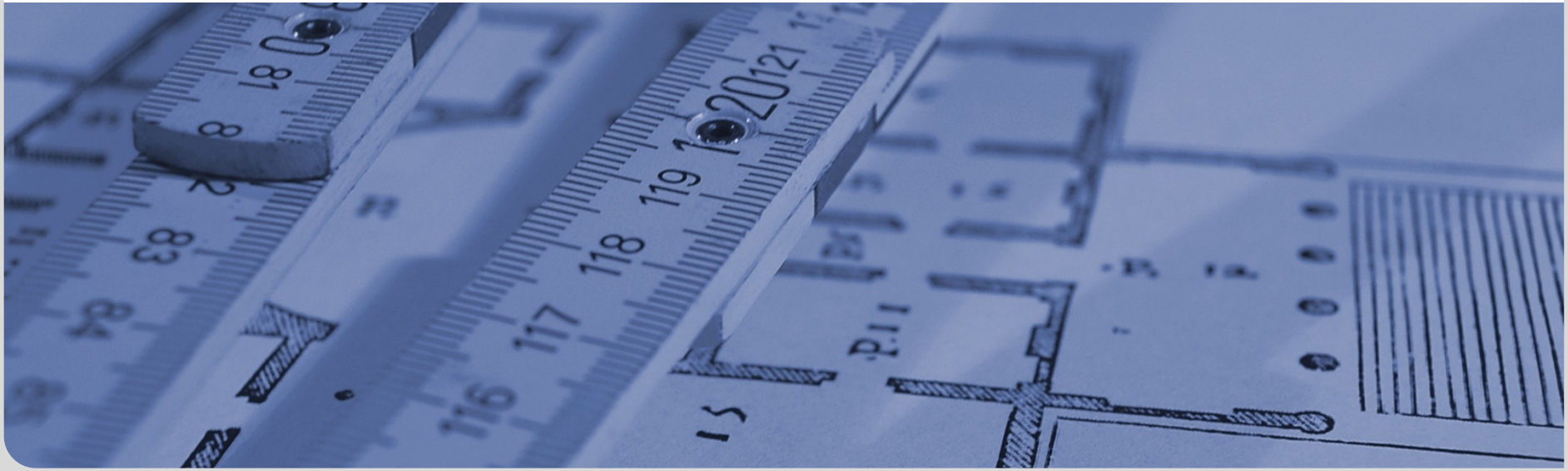
Karlsruher Institut für Technologie

Software Design and Quality

Bewertungsrichtlinien

Kommentare

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Kommentare

- Entfernen von TODOS, auskommentierten Methoden und unnötigen Aussagen
- Stören den Lesefluss & die Übersichtlichkeit
- Versionskontrolle, bspw. Git, anstelle von Auskommentieren verwenden
- Beispiel (alle Kommentare sind zu entfernen):

```
// TODO Testfälle erstellen
// TODO auto-generated method
public int multiply(int first, int second) {
    // Checkstyle lässt mich nicht Multiply schreiben
    // FIXME Variablennamen überdenken
    // Mein Tutor ist doof
    // Multipliziert m1 mit m2.
    return first * second;
    // return second * second;
    // return m1 * m1 * m3;
}
```

Keine Fehlerbehandlung beim Parsen von String zu Integer

- Mit Integer.parseInt(String) kann ein String in einen Integer konvertiert werden
- Aber Vorsicht:
  - NutzerIn kann unsinnige Eingaben machen
    - int number = Integer.parseInt("hallo"); → NumberFormatException
  - Integer beinhaltet nur Zahlen zwischen -2147483648 und 2147483647
    - int number = Integer.parseInt("2147483648"); → NumberFormatException
  - ParseInt akzeptiert auch Eingaben wie "-0", "+42" oder "07"
- Lösung: Try-Catch und Regex

Keine Fehlerbehandlung beim Parsen von String zu Integer

- Folgendes Beispiel akzeptiert darstellbare Integer ohne führende Nullen oder redundanten Vorzeichen

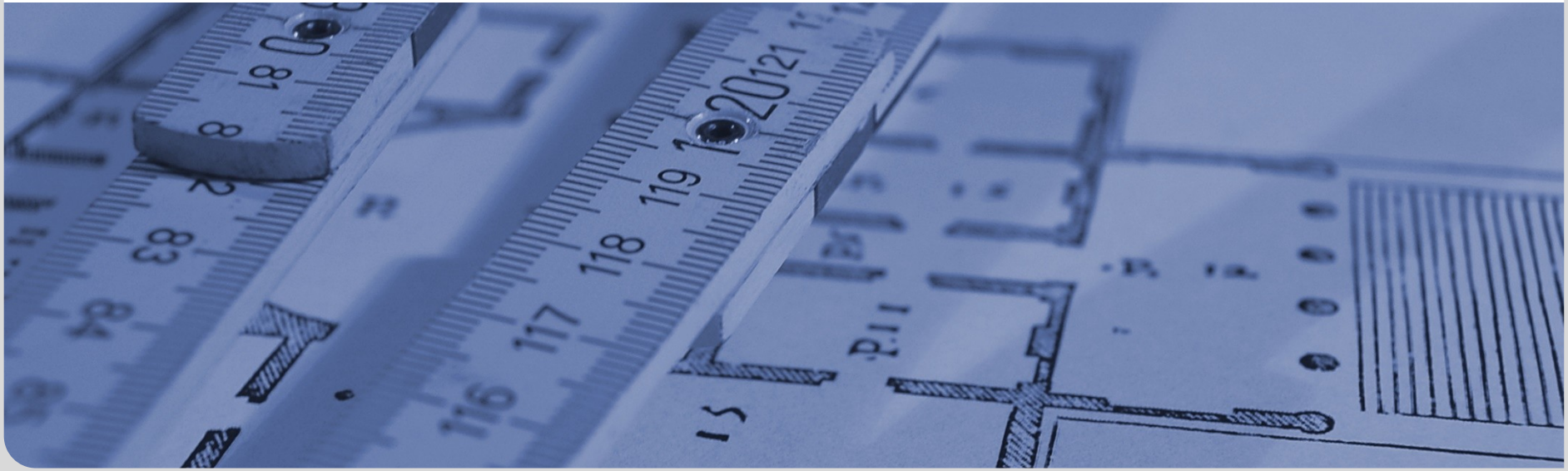
```
int number = 0;
if (input.matches("0|-?[1-9]\\d*")) {
    try {
        number = Integer.parseInt(input);
    } catch (NumberFormatException exception) {
        System.err.println("Invalid input format!");
        return;
    }
} else {
    System.err.println("Invalid input format!");
    return;
}
```

Karlsruher Institut für Technologie

Bewertungsrichtlinien

Runtime Exceptions

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Runtime Exceptions

- *Runtime Exceptions* sind Programmierprobleme und somit nicht sinnvoll behandelbar und nicht zur Kontrollflusssteuerung gedacht
- Niemals *ArrayIndexOutOfBoundsException* oder *NullPointerException* fangen
- Ausnahme bei *NumberFormatException* und *Integer.parseInt(…)*
- Beispiel:
  - ```
    try { isNull.doSomething(); }
    catch (NullPointerException e) { }
    ```
  - ```
    try { for (int i = 0; true; i++) doSomething(array[i], i); }
    catch (ArrayIndexOutOfBoundsException e) { }
    ```

  Stattdessen:

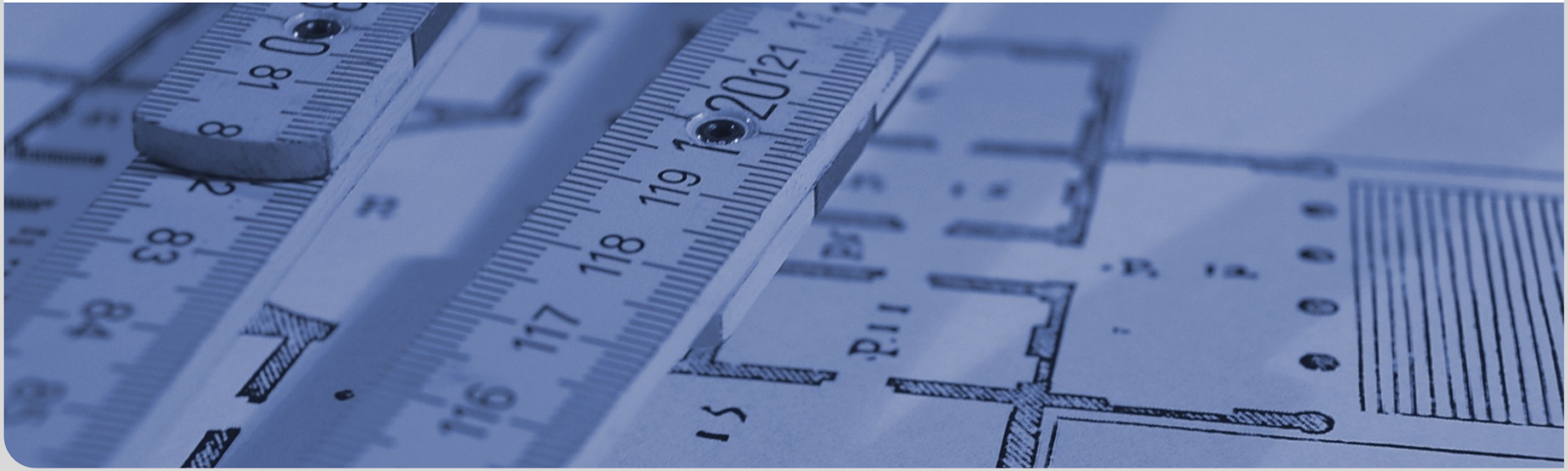
  - ```
    if (isNull == null) { throw new IllegalArgumentException() }
    isNull.doSomething();
    ```
  - ```
    for (int i = 0; i < array.length; i++) {
        doSomething(array[i], i);
    }
    ```

Karlsruher Institut für Technologie

Bewertungsrichtlinien

Sichtbarkeit

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Sichtbarkeit

- Attribute und Methoden sind vor externen Zugriffen zu schützen
- Alles sollte einen überlegten Modifikator haben
  - default-Modifikator mit ein Kommentar versehen (nicht übersehen o.ä.)
  - *private* für Attribute, Hilfsmethoden und private Konstruktoren
  - *public* definiert Schnittstellen zu anderen Klassen
  - *protected* falls notwendig
- Beginne immer mit *private* und lockere die Sichtbarkeit nach Bedarf
- Beispiel:

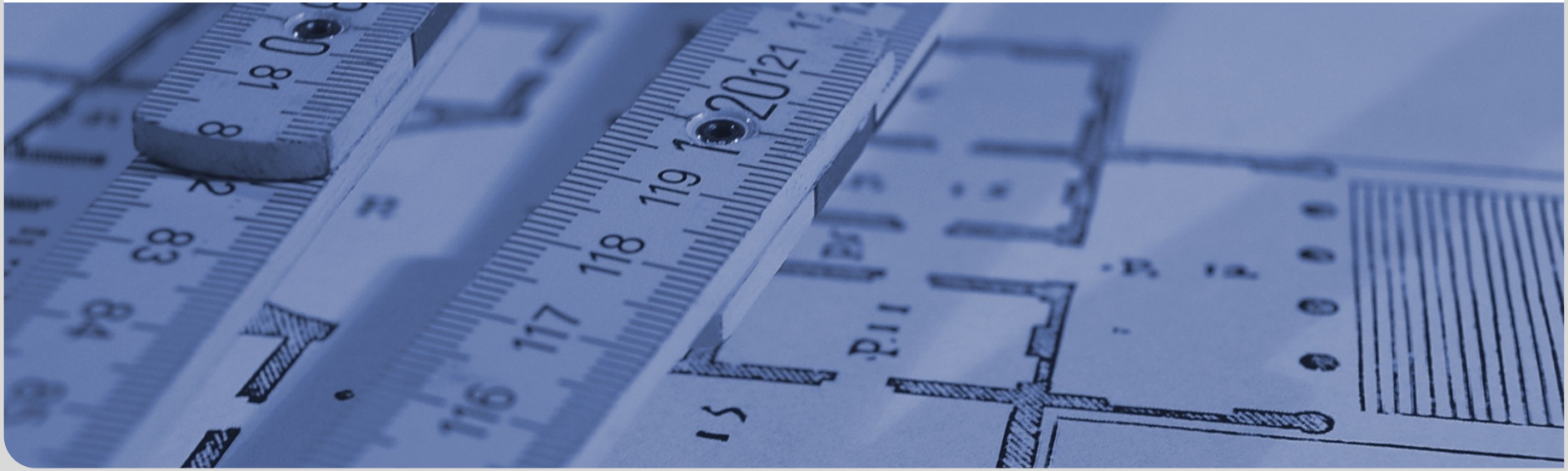
```
public class Book {
    // Öffentlich, also durch alle veränderbar und lesbar
    // Besser beides private setzen
    public String title;
    public long isbn;
}
```

Karlsruher Institut für Technologie

Bewertungsrichtlinien

Statische Methode

SOFTWARE DESIGN AND QUALITY, INSTITUT FÜR PROGRAMMSTRUKTUREN UND DATENORGANISATION, KIT-FAKULTÄT FÜR INFORMATIK



Statische Methode

- Statische Hilfsmethoden, die von Attributen abhängig sind, sollen diese auch nutzen und Instanzmethoden sein, statt sie unnötigerweise zu übergeben
- Beispiel:

```
public class Receipt {
    private List<Product> products;

    // Attribut übergeben ist überflüssig
    // Folglich: Parameter und static entfernen
    private static double getTotalPrice(List<Product> products) {
        double total;

        for (Product p : products) {
            total += p.getPrice();
        }

        return total;
    }
}
```