Zurich University of Applied Sciences

School of Engineering
InIT Institute of Applied Information Technology

PROG1 Clean-Code-Regeln

Über dieses Dokument

Dieses Dokument listet die in PROG1 einzuhaltenden Clean-Code Regeln.

Es ist eine gekürzte Fassung des in PSIT2 verwendeten Materials, siehe OLAT -> Kurs «PSIT1/2» für Ihr Semester -> PSIT2 -> Code Review -> handbuch.html

Diese Clean-Code Regeln sind für das ganze Studium an der ZHAW gültig und werden, sofern nicht anders vermerkt, konsequent eingefordert. Bei den Code Reviews in PSIT2 werden Sie diese Regeln verwenden, ergänzt um einige zusätzliche Regeln, die wir hier vorerst ausgelassen haben.

Die vorhandenen Beispiele sollen die Clean-Code-Regeln illustrieren. Sie dienen **nicht** dazu, neuen Stoff über den in der Vorlesung behandelten einzuführen.

Zurich University of Applied Sciences

School of Engineering
InIT Institute of Applied Information Technology

Inhaltsverzeichnis

Über dieses Dokument 1

Review-Kriterien im Detail 3

- Lesbarkeit 3
  - Java-Namenskonventionen werden eingehalten (PROG1 Kapitel 6: Bibliotheksklassen) 3
  - Code ist aufgeräumt (PROG1 Kapitel 3: Objektinteraktion, Kapitel 6: Bibliotheksklassen) 3
  - Sinnvolle Namensgebung (PROG1 Kapitel 1: Objekte und Klassen) 5
  - Methodengestaltung (PROG1 Kapitel 3: Objektinteraktion, Kapitel 6, Bibliotheksklassen) 6
- Klassengestaltung 7
  - Klassen haben Verhalten (PROG1 Kapitel 3: Objektinteraktion) 7
  - Klassenoberfläche ist minimal (PROG1 Kapitel 3: Objektinteraktion, Kapitel 10: Vererbung) 7
  - Klassen stehen für sich allein (PROG1 Kapitel 8: Klassenentwurf) 7
  - Sinnvolle Abstraktionsebene (PROG1 Kapitel 12: Abstrakte Klassen und Interfaces) 8
- Testing 9
  - Ein sinnvoller Unit-Test ist vorhanden (PROG1 Kapitel 9: Fehlervermeidung) 9
  - Logik mit Tests abgedeckt (PROG1 Kapitel 9: Fehlervermeidung) 9
  - Mock-Objekte statt externe Kollaborateure (PROG1 Exkurs: Isoliertes Testen mit Stubs) 9
- Wartbarkeit 11
  - Klassen, ENUMs statt primitive Typen (PROG1 Kapitel 8: Klassenentwurf) 11
  - Vernünftiger Einsatz von Feldern (PROG1 Kapitel 3: Objektinteraktion) 12
  - Interfaces statt konkrete Typen (PROG1 Kapitel 12: Abstrakte Klassen und Interfaces) 12
  - Kollaborateure als Konstruktor-Argumente (PROG1 Exkurs: Isoliertes Testen mit Stubs) 13
  - Javadoc (PROG1 Kapitel 2: Klassendefinitionen) 14
- Korrektheit 16
  - Null-Objekte anstelle von `null` (PROG1 Kapitel 4: Sammlungen) 16
  - Eingabeüberprüfung (PROG1 Kapitel 9: Fehlervermeidung) 16
  - Object-Klassenvertrag wird eingehalten (PROG1 Kapitel 11: Mehr über Vererbung) 18
- Varia 19
  - Angemessene Verwendung von Collections (PROG1 Kapitel 4: Sammlungen) 19

Review-Kriterien im Detail

Lesbarkeit

Java-Namenskonventionen werden eingehalten (PROG1 Kapitel 6: Bibliotheksklassen)

- Klassennamen sind Upper Camel Case (`MySpecialClass`)
- Methodennamen, Variablennamen und Feldnamen sind Lower Camel Case (`someMethod()`)
- Konstanten sind Grossbuchstaben (`ABSOLUTE_NULL`)

Java-Klasse mit eingehaltenen Namenskonventionen ☺

```
public class Temperature {

    private static final BigDecimal KELVIN_CELSIUS_DIFF = BigDecimal.valueOf(273.15);

    private final BigDecimal value;

    public Temperature(BigDecimal degreesKelvin) {
        this.value = degreesKelvin;
    }

    public BigDecimal toCelsius() {
        return value.add(KELVIN_CELSIUS_DIFF);
    }
}
```

Code ist aufgeräumt (PROG1 Kapitel 3: Objektinteraktion, Kapitel 6: Bibliotheksklassen)

Kein toter Code ist vorhanden

Entfernen Sie auskommentierten Code umgehend. Er reduziert die Lesbarkeit des Codes und wird spätestens nach der nächsten Umbenennung einer Variablen nutzlos. Falls Sie fürchten, die alte Implementierung noch einmal zu brauchen: Dafür ist die Versionskontrolle da.

Klasse mit totem Code ☹

```
public class Temperature {
    public static int toCelsius(int degreesKelvin) {
       // int degreeCelsius = value + 272 //(1)
        // return defreeCelsius;
        return degreesKelvin + 273;
    }
}
```

1. Toter Code. Ist `value` dasselbe wie `degreesKelvin`?

Sinnvolle Kommentare

Redundanter Kommentar ☹

```
public class Temperature {
    public static int toCelsius(int degreesKelvin) {
        // Add 273.15 to the value in Kelvin to get the temperature in celsius! //(1)
        return degreesKelvin + 273;
    }
}
```

1. Erzählt nur, was der Code in der nächsten Zeile tut. Aber warum tut er, was er tut?

Anstatt zu beschreiben, was der Code tut, fügen Sie nützliche Informationen hinzu, die Ihrem zukünftigen Ich beim Verständnis der Passage helfen:

- Berechnungsformeln
- Links zu Online-Referenzen
- Begründungen

Weiterführender Kommentar ☺

```
public class Temperature {
    public int toCelsius(int degreesKelvin) {
        // https://en.wikipedia.org/wiki/Conversion_of_units_of_temperature
        return degreesKelvin + 273;
    }
}
```

1. Schreiben Sie nicht bei jeder RuntimeException oder einer Ableitung davon den Kommentar "Unexpected!" hin. Dies ist in diesem Beispiel nur aufgrund des Kontexts angebracht.

Importieren Sie nur, was Sie brauchen

Importieren Sie nur die Namen, die Sie in der aktuellen Datei tatsächlich brauchen. Unnötige Imports (und dazu gehören auch die sogenannten Star Imports wie `import java.util.*;`) sorgen zwar nur für einen kleinen Performance-Verlust und das auch nur beim Kompilieren, können Ihnen aber Wartungsprobleme bereiten. Folgender Code lief in Java 1.1 problemlos. Beim Upgrade auf Java 1.2 versagt er, auch wenn der Code nicht angefasst wurde. Warum?

Drohendes Unglück ☹

```
import java.math.BigDecimal;
import java.util.*;
import java.awt.*;

public class Test {

    public static void main(String[] args) {
        List list = new List();
    }
}
```

Der Typ List ist `java.awt.List`, in Java 1.1 der einzig verfügbare Listen-Typ. Mit Java 1.2 kommt mit `java.util.List` plötzlich ein zweiter Typ List dazu, womit der Code mehrdeutig wird.

Verzichten Sie auf Copy & Paste

Bevor Sie Code kopieren und an einer anderen Stelle einfügen, überlegen Sie sich, ob Sie nicht eine Methode schreiben und diese referenzieren können.

Besonders geeignete Kandidaten:

- Berechnungen bzw. Zusammensetzung von Zeichenketten (z.B. Vor- und Nachname)
- Objekte erstellen aus SQL-ResultSet und umgekehrt
- Hilfsmethoden, z.B. zum Ausgeben eines Datums

Ziehen Sie Konstanten Magic Numbers vor

Magic Numbers sind (meist) Zahlen, die plötzlich im Code auftauchen — und niemand weiss, was sie bedeuten und warum gerade der angegebene Wert der Richtige ist.

Temperaturkonversion mit Magic Number ☹

```
public class Temperature {
    public static BigDecimal toCelsius(BigDecimal degreesKelvin) {
        return degreesKelvin.add(BigDecimal.valueOf(273.15)); //(1)
    }
}
```

1. 273.15 ist eine Magic Number

Erklären Sie die Magic Numbers mit einem Kommentar oder — meist besser — durch eine Konstante, da Sie dann immer denselben Wert referenzieren und ihn so nur einmal zu ändern brauchen.

Magic Number als Konstante ausgedrückt ☺

```
public class Temperature {

    private static final BigDecimal KELVIN_CELSIUS_DIFF = BigDecimal.valueOf(273.15);
//(1)

    public static BigDecimal toCelsius(BigDecimal degreesKelvin) {
        return degreesKelvin.add(KELVIN_CELSIUS_DIFF);
    }
}
```

1. Benannte Konstante

Formatieren Sie Code konsistent

Ob Sie Tabs oder Leerschläge zum Einrücken des Codes verwenden, spielt keine grosse Rolle. Ebenso wenig, ob Sie Blöcke auf derselben Zeile mit `if`, `for` usw. beginnen oder in der Zeile darunter. Nur tun Sie es konsistent. Dies verbessert die Lesbarkeit und sorgt für minimalen "Lärm" durch Formatierungsänderungen beim Code Review.

Tip	Verwenden Sie den Eclipse Formatter, um den Code automatisch formatieren zu lassen. Auf Windows drücken Sie dazu Ctrl + ⇧ + F und auf Mac ⌘ + ⇧ + F. Sie müssen im ganzen Team dieselben Einstellungen bezüglich Tabulatoren und Leerzeichen haben. Wenn nach dem Einsatz des Formatters alle Zeilen als "verändert" angezeigt werden, müssen Sie zuerst diese Einstellungen kontrollieren.

Sinnvolle Namensgebung (PROG1 Kapitel 1: Objekte und Klassen)

Achten Sie bei der Benennung von Variablen, Klassen und Methoden darauf, möglichst beschreibende Namen zu verwenden. Dies hilft, Ihren Code nachzuvollziehen.

Unklare Variablen- und Methodennamen ☹

```
public class Temperature {
    public static BigDecimal calculate(BigDecimal value) { //(1) (2)
        return value.add(273.15);
    }
}
```

Was macht calculate genau?

Was muss man bei `value` angeben? Celsius? Fahrenheit?

Methoden- und Variablennamen kommunizieren Zweck ☺

```
public class Temperature {
    public static BigDecimal toCelsius(BigDecimal degreesKelvin) { //(1) (2)
        return degreesKelvin.add(273.15);
    }
}
```

1. Mit dem Methodennamen `toCelsius` wird kommuniziert, dass der übergebene Wert in Grad Celsius konvertiert wird.
2. Dank dem Variablennamen `degreesKelvin` weiss der Programmierer, dass die Gradangabe in Kelvin zu erfolgen hat.

Verzichten Sie ausserdem auf die Einbettung von Typen in Variablennamen, Klassen oder Interfaces ("ungarische Notation").

Ungarische Notation ☹

```
public class MathClass { //(1)
    private static final double S_PI = 3.141; //(2)

    public static double circleArea(double dRadius) { //(3)
        return S_PI * dRadius * dRadius;
    }
}
```

1. Aufgrund des Suffix `Class` wissen Sie, dass es sich bei `MathClass` um eine Klasse handelt.
2. Das Prefix `S_` kommuniziert, dass es sich um eine statische Variable handelt.
3. Das `d` signalisiert, dass in `dRadius` ein `double` gespeichert ist.

Methodengestaltung (PROG1 Kapitel 3: Objektinteraktion, Kapitel 6, Bibliotheksklassen)

Methoden sind maximal 40 Zeilen lang

Beschränken Sie die Länge einer Methode auf maximal 40 Zeilen. Reichen die 40 Zeilen nicht aus für die Logik, die Sie unterbringen wollen, machen Sie nicht einfach die Zeilen länger. Lagern Sie stattdessen sinnvolle Teile der Logik (z.B. Bedingungen, Teilberechnungen) in weitere Methoden aus.

Klassengestaltung

Klassen haben Verhalten (PROG1 Kapitel 3: Objektinteraktion)

Achten Sie darauf, dass Ihre Klassen nicht nur Daten speichern, sondern auch Verhalten (d.h. Methoden jenseits von Getter und Setter) aufweisen. Denn das ist der Kern der objektorientierten Programmierung.

Klasse, die nur Daten speichert ☹

```
public class Circle {

    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }
}
```

Klasse mit Verhalten ☺

```
public class Circle {

    private final double radius;

    public Circle(double radius) {
        this.radius = radius;
    }

    public double getRadius() {
        return radius;
    }

    public double getArea() {
        return radius * radius * Math.PI;
    }

    public double getCircumference() {
        return 2 * Math.PI * radius;
    }
}
```

Klassenoberfläche ist minimal (PROG1 Kapitel 3: Objektinteraktion, Kapitel 10: Vererbung)

Machen Sie jede Klasse, ihre Eigenschaften und Methoden so wenig zugänglich wie möglich. Dies bedeutet: Können Sie ein Feld oder eine Methode private deklarieren, deklarieren Sie sie private. Verwenden Sie weniger restriktive Zugangsbeschränkungen (`protected` oder gar `public`) nur, wenn es unbedingt nötig ist.

Instanzfelder sollten nur in Ausnahmefällen `public` sein. Generieren Sie auch nicht für jedes Feld einen Getter oder Setter, nur weil die IDE dies Ihnen anbietet.

All diese Massnahmen reduzieren die Abhängigkeiten zwischen den verschiedenen Klassen und machen die Wartung des Codes einfacher.

Weiterführende Informationen

- Westphal, Ralf; Lieser, Stefan: Clean Code Developer. Information Hiding Principle [Online]. URL: http://clean-code-developer.de/die-grade/gelber-grad/#Information_Hiding_Principle [Stand: 05.03.2018]

Klassen stehen für sich allein (PROG1 Kapitel 8: Klassenentwurf)

Achten Sie darauf, dass Ihre Klassen für sich alleine stehen. Symptome, dass Ihre Klassen nicht für sich alleine stehen:

- Eine Klasse und ihre Methoden müssen andauernd andere Klassen nach ihrem Zustand fragen.

- Eine Klasse und ihre Methoden müssen mehrere Methoden einer anderen Klasse aufrufen, um etwas zu erledigen.
- Eine Klasse und ihre Methoden müssen Methoden einer anderen Klasse in einer bestimmten Reihenfolge aufrufen.
- Eine Klasse und ihre Methoden müssen über mehrere Objekte hinweg nach Daten fragen (z.B. `this.getOther().getAnother().getData()`)

In diesem Fall ist es angebracht, die Funktionalität in die anderen Klassen zu verschieben respektive von dort in die aktuelle Klasse zu bewegen, um die Abhängigkeiten zu reduzieren.

Weiterführende Informationen

- Westphal, Ralf; Lieser, Stefan: Clean Code Developer. Tell, don't ask [Online]. URL: http://clean-code-developer.de/die-grade/gruener-grad/#Tell_dont_ask [Stand: 05.03.2018]
- Westphal, Ralf; Lieser, Stefan: Clean Code Developer. Law of Demeter [Online]. URL: http://clean-code-developer.de/die-grade/gruener-grad/#Law_of_Demeter [Stand: 05.03.2018]

Sinnvolle Abstraktionsebene (PROG1 Kapitel 12: Abstrakte Klassen und Interfaces)

Interfaces statt abstrakte Klassen

Ziehen Sie Interfaces abstrakten Klassen vor (Composition over Inheritance). Abstrakte Klassen geben ein enges Korsett vor: Abgeleitete Klassen müssen eine strikte Hierarchie bilden (Ist-ein-Beziehung) und das Liskovsche Substitutionsprinzip einhalten. Ausserdem kann in Java jeweils nur von einer Klasse geerbt werden. Interfaces sind im Vergleich viel flexibler: Existierende Klassen können jederzeit zusätzliche Interfaces implementieren und nichthierarchische Typsysteme bilden. Ausserdem können existierende Klassen nachträglich mit zusätzlicher Funktionalität (sogenannten Mixins) ausgerüstet werden, ohne die Klassen selber anpassen zu müssen.

Keine unnötige Abstraktion

Verzichten Sie darauf, auf Vorrat abstrakte Klassen und Methoden sowie Interfaces zu erstellen für den Fall, dass man sie in Zukunft *vielleicht* einmal brauchen könnte. Existiert für ein Interface oder eine abstrakte Klasse nur eine einzelne Implementierung, verzichten Sie darauf. Gleiches gilt für ungenutzte Methoden oder Klassen, die fast keine Aufgaben haben.

Weiterführende Informationen

- Bloch, Jushua: Effective Java. Pearson Education, 2017, S. 99-103.
- Westphal, Ralf; Lieser, Stefan: Clean Code Developer. Liskov Substitution Principle [Online]. URL: http://clean-code-developer.de/die-grade/gelber-grad/#Liskov_Substitution_Principle [Stand: 05.04.2018]

Testing

Ein sinnvoller Unit-Test ist vorhanden (PROG1 Kapitel 9: Fehlervermeidung)

In Ihrem gesamten Projekt sollte sich zumindest ein sinnvoller Unit Test befinden, der ein Mindestmass an Logik (= ein Ausdruck, der etwas berechnet) testet (= mindestens eine Assertion ist vorhanden).

Ein sinnvoller Test ☺

```
class EmployeeTest {

    @Test
    void fullNameComposedOfFirstAndLastName() {
        Employee employee = new Employee();
        employee.setName("Diana");
        employee.setLastName("Prince");
        assertEquals("Diana Prince", employee.getFullName()); //(1)
    }
}
```

1. Mindestens eine Assertion, die mehr als assertTrue(true) testet, ist gefordert!

Logik mit Tests abgedeckt (PROG1 Kapitel 9: Fehlervermeidung)

Die wesentliche Logik Ihrer Software ist mit automatisierten Tests abgedeckt. "Wesentlich" bedeutet, dass Sie weder Getter noch Setter testen brauchen, solange sie nichts anderes machen, als Felder zu lesen beziehungsweise zu setzen. Dasselbe gilt für Hilfsmethoden wie `toString()`. Bei allem anderen sollte die korrekte Funktionsweise durch einen automatisierten Test mit geeigneten Assertions sichergestellt sein.

Mock-Objekte statt externe Kollaborateure (PROG1 Exkurs: Isoliertes Testen mit Stubs)

Komponenten (wie ein Data Access Object) verwenden in der Regel andere Komponenten (wie eine Datenbankverbindung), um eine Aufgabe zu erledigen. Um eine Komponente sinnvoll zu testen, ist es ratsam, diese von den anderen Komponenten zu isolieren. In Kollaborateure als Konstruktor-Argumente (L3) wird der erste Schritt erklärt, um eine Komponente zu isolieren, nämlich die Übergabe von Kollaborateuren als Konstruktorargument. Der zweite Schritt ist, anstelle des eigentlichen Kollaborateurs eine Testattrappe oder Mock-Objekt zu übergeben. Mit Hilfe einer Testattrappe können Sie in einem Test verifizieren, wie die zu testende Komponente mit dem eigentlichen Kollaborateur interagieren würde und ob sie es richtig tut.

Zu testen: Wird die Buchbewertung korrekt hinzugefügt?

```
class BookService {

    private final BookRepository bookRepository;

    public BookService(BookRepository bookRepository) {
        this.bookRepository = bookRepository;
    }

    public void reviewBook(ISBN isbn, Customer reviewer, String review) {
        Book book = this.bookRepository.getBook(isbn);
        book.addReview(reviewer, review);
    }
}

class BookRepository {

    private final DataSource dataSource;

    public BookRepository(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public Book getBook(ISBN isbn) {
        // call database, return book
    }
}
```

Wie lässt sich testen, ob die Bewertung zum richtigen Buch hinzugefügt würde, ohne die Datenbank zu bemühen? Indem man das BookRepository durch eine Testattrappe ersetzt, die ohne Datenbank auskommt.

Test des BookService in Isolation ☺

```
class BookServiceTest {

    @Test
    void reviewAdded() {
        Book testBook = new Book();
        TestBookRepository bookRepository = new TestBookRepository(testBook);
        BookService bookService = new BookService(bookRepository);

        bookService.reviewBook(new ISBN("123"), new Customer(), "Great book");

        assertEquals(bookRepository.getLastUsedIsbn(), new ISBN("123"));
        assertEquals(1, testBook.getReviews().size());
    }

    class TestBookRepository extends BookRepository { //(1)

        private final Book book;
        private ISBN lastUsedIsbn;

        TestBookRepository(Book book) { //(2)
            this.book = book;
        }

        @Override
        public Book getBook(ISBN isbn) { //(3)
            this.lastUsedIsbn = isbn; //(4)
            return book;
        }

        ISBN getLastUsedIsbn() {
            return lastUsedIsbn;
        }
    }
}
```

1. Indem TestBookRepository von BookRepository erbt, kann es anstelle des BookRepository, das eine Datenbankverbindung bräuchte, an den BookService übergeben werden.
2. Die Methode Book `getBook(ISBN)` muss ein Buch zurückgeben. Dieses wird über den Konstruktor des TestBookRepository für die spätere Rückgabe bereitgelegt.
3. Durch das Überschreiben der Methode Book `getBook(ISBN)` wird die Datenbankverbindung der ursprünglichen Implementierung nicht mehr benötigt. Die restliche Logik in der ursprünglichen Implementierung kann ebenfalls ignoriert werden, da man sich nur noch für die Interaktion von BookService mit BookRepository interessiert: Wird die richtige ISBN übergeben und das Buch korrekt weiterverwendet?
4. Damit man überprüfen kann, ob die richtige ISBN übergeben wurde, muss das `TestBookRepository` sie sich merken und über einen separaten Getter zugänglich machen.

Wartbarkeit

Klassen, ENUMs statt primitive Typen oder String (PROG1 Kapitel 8: Klassenentwurf)

Verwenden Sie Klassen und `ENUM`s anstelle von primitiven Typen oder `String`. Denn solange Sie Klassen oder `ENUM`s verwenden, hilft Ihnen der Compiler dabei, keine Tippfehler zu machen oder Wertebereiche zu beschränken. Verwenden Sie dagegen primitive Typen oder String, sind Sie auf sich allein gestellt.

String zum Typenvergleich ☹

```
public class Card {

    private final String type = "DIAMOND";

    public void display() {
        switch(type) {
            case "DIAMOND": //(1)
                System.out.println("♦");
                break;
            // etc.
        }
    }
}
```

1. Damit der Vergleich funktioniert, müssen Sie immer exakt DIAMOND schreiben, nicht diamond oder eine andere Variante.

Typsichere Vergleiche dank ENUM ☺

```
enum CardType {
    DIAMOND, CLUB, HEART, SPADE
}

public class Card {

    private final CardType type = CardType.DIAMOND;

    public void display() {
        switch(type) {
            case DIAMOND:
                System.out.println("♦");
                break;
            // etc.
        }
    }
}
```

Tip	Wenn Sie tatsächlich einmal nicht darum herumkommen, einen `ENUM`-Wert als `String` entgegen zu nehmen (z.B. in einer Webapplikation), konvertieren Sie ihn frühzeitig zurück zum `ENUM` mittels `valueOf(String)`, z.B. `CardType.valueOf("DIAMOND")`.

Verwenden Sie ausserdem Klassen, um komplexe Konzepte zu modellieren, die z.B. aus Wert plus Einheit bestehen (Geld, Temperatur, …), aus mehreren Werten (Adresse, …) oder zu denen spezielle (Formatierungs-)Regeln gehören (wie bei Telefonnummern).

Produktpreis modelliert durch primitiven Typ ☹

```
public class Product {
    private double price; (1) (2)
}
```

1. Wenn eine exakte Repräsentation gefragt ist, verwenden Sie nie einen double, sondern einen BigDecimal.
2. Dem double mangelt es an Ausdrucksstärke: Weder kann er eine Einheit wie CHF transportieren noch sich selber validieren.

Produktpreis modelliert durch eine Klasse ☺

```
public class Product {
    private Price price;
}

public class Price {
    private BigDecimal amount;
    private Currency currency;
}

private Currency {
    private String name; // like CHF
}
```

Vernünftiger Einsatz von Feldern (PROG1 Kapitel 3: Objektinteraktion)

Speichern Sie nur in Feldern, was sie für einen längeren Zeitraum (= länger als der aktuelle Methodenaufruf) brauchen. Für alles andere reicht eine lokale Variable.

Interfaces statt konkrete Typen (PROG1 Kapitel 12: Abstrakte Klassen und Interfaces)

Beziehen Sie sich beim Referenzieren von Parametern, Rückgabewerten, Variablen und Feldern auf Interfaces (wie `List`) anstatt auf konkrete Typen (wie `ArrayList`), wenn ein geeignetes Interface existiert. Verwenden Sie konkrete Typen nur beim Erstellen des Typs. Dies macht Ihr Programm viel flexibler. Wenn Sie feststellen, dass eine andere Implementierung eines Interfaces geeigneter ist, brauchen Sie nur den Konstruktoraufruf auszutauschen und Ihr Programm funktioniert weiter.

Interface wird anstelle des konkreten Typs verwendet ☺

```
SortedSet<String> aSet = new TreeSet<String>();
```

Wenn Sie mehrere Interfaces zur Auswahl haben, verwenden Sie dasjenige, das Ihnen die Funktionalität bietet, die Sie benötigen.

Warning	Achtung vor versteckten Anforderungen Achten Sie darauf, das Interface spezifisch genug zu wählen, um nicht über versteckte Anforderungen zu stolpern. Folgende Deklaration könnte gefährlich werden: `Set<String> aSet = new TreeSet<String>();` Wenn Sie ein TreeSet verwenden, werden die im Set gespeicherten Strings automatisch sortiert, selbst wenn Sie das Set nur über das Interface Set ansprechen. Wenn Sie eines Tages das TreeSet aus Performance-Gründen in ein HashSet verwandeln, geht die Eigenschaft der automatischen Sortierung verloren. Wenn es für das restliche Programm wichtig ist, dass die Werte sortiert sind, wählen Sie als Interface SortedSet, da dieses die Sortierung garantiert. Sonst verzichten Sie am besten gleich auf TreeSet und wählen von Beginn weg HashSet.

Weiterführende Informationen

- Bloch, Jushua: Effective Java. Pearson Education, 2017, S. 280-281.
- Open Web Application Security Project: Servlet spec - web.xml [Online]. URL: https://www.owasp.org/index.php?title=Servlet_spec_-_web.xml&oldid=235287 [Stand: 05.03.2018]
- Oracle, Inc.: Handling JSP Page Errors [Online]. URL: https://docs.oracle.com/cd/E19575-01/819-3669/bnahi/index.html [Stand: 05.03.2018]

Kollaborateure als Konstruktor-Argumente (PROG1 Exkurs: Isoliertes Testen mit Stubs)

Wenn Ihre Klasse (nennen wir sie A) eine andere Klasse (B) verwendet, um eine Aufgabe zu erledigen, ist B ein Kollaborateur oder eine Abhängigkeit (Dependency) von A. Übergeben Sie B immer als Konstruktor-Argument an A, statt B im Konstruktor von A mit new zu erstellen. Das macht es einfacher, die Klasse zu testen.

Abhängigkeiten werden im Konstruktor erstellt ☹

```
public class SomeEntityService {

    private final SomeEntityDao someEntityDao;

    public SomeEntityService() {
        this.someEntityDao = new SomeEntityDao();
    }
}
```

Wenn Sie in einem Test `SomeEntityDao` durch einen Mock ersetzen wollen, sind Sie aufgeschmissen. Übergeben Sie stattdessen `SomeEntityDao` als Konstruktorargument und Sie können im Rahmen eines Tests eine andere Instanz übergeben.

Abhängigkeiten werden als Konstruktorargument übergeben ☺

```
public class SomeEntityService {

    private final SomeEntityDao someEntityDao;

    public SomeEntityService(SomeEntityDao dao) { //(1)
        this.someEntityDao = dao;
    }
}
```

1. Potentielle Verbesserung: Extrahieren Sie ein Interface aus `SomeEntityDao`, dann brauchen Sie nicht einmal eine Mocking-Bibliothek, um einen Mock zu erstellen.

Unfreundlich zum Testen sind auch statische Methoden, da Sie diese ebenfalls nicht ersetzen können:

Statische Methoden sind untestbar ☹

```
public class SomeEntityService {

    private final SomeEntityDao someEntityDao;

    public SomeEntityService(SomeEntityDao dao) {
        this.someEntityDao = dao;
    }

    public void createSomeEntity(String name) {
        entity = new SomeEntity();
        entity.setName(name);
        entity.dateCreated(Instant.now()) (1)
        this.someEntityDao.save(entity);
    }
}
```

1. In einem Test wissen Sie nicht, welche Zeit gesetzt wird. Sie können sich in Assertions bestenfalls annähern und einen Hack wie `assertTrue(dateCreated.isAfter(Instant.now().minusSeconds(2)))` in Kombination mit `assertTrue(dateCreated.isBefore(Instant.now()))` verwenden.

Abhilfe schafft, keine statischen Methoden zu verwenden oder sie mit Argumenten zu modifizieren. `java.time` bietet eine elegante Möglichkeit mit der abstrakten Clock, für die verschiedene Implementierungen existieren — unter anderem die Systemuhr, die jeweils die aktuelle Zeit zurückgibt, und eine Variante, die immer dieselbe vorgegebene Zeit zurückgibt (`Clock.fixed(Instant.now(), ZoneOffset.UTC)`) und damit ideal für Tests ist.

Parameterisierte statische Methode erhöht Testbarkeit ☺

```
public class SomeEntityService {

    private final SomeEntityDao someEntityDao;

    public SomeEntityService(SomeEntityDao dao) { //(1)
        this.someEntityDao = dao;
    }

    public void createSomeEntity(String name, Clock clock) {
        entity = new SomeEntity();
        entity.setName(name);
        entity.dateCreated(Instant.now(clock)) //(2)
        this.someEntityDao.save(entity);
    }
}
```

1. Übergeben Sie `Clock` als Konstruktorargument. Dies erlaubt Ihnen später, die Uhr für Tests auszutauschen.
2. Übergeben Sie die Clock zur Bestimmung der aktuellen Zeit.

Weiterführende Informationen

- Bloch, Jushua: Effective Java. Pearson Education, 2017, S. 20-21.
- Westphal, Ralf; Lieser, Stefan: Clean Code Developer. Dependency Inversion Principle [Online]. URL: http://clean-code-developer.de/die-grade/gelber-grad/#Dependency_Inversion_Principle [Stand: 05.03.2018]

Javadoc (PROG1 Kapitel 2: Klassendefinitionen)

> Bemerkung PROG1: In PROG1 müssen Sie nur Javadoc für die von Ihnen geschriebenen Klassen schreiben. Deren öffentliche Methoden müssen Sie bereits nicht mehr per Javadoc dokumentieren. In der Projektschiene wird dies anders aussehen. Dort wird wie in der Praxis draussen auch, nachfolgende Regelung gelten.

Beschreiben Sie in Javadoc den Zweck der von Ihnen erstellten Klassen und Interfaces.

Dokumentieren Sie ausserdem Methoden in Interfaces, abstrakte Methoden sowie öffentliche Methoden von Klassen, die die Rolle von internen Schnittstellen einnehmen wie zum Beispiel Data Access Objects. Für Methoden ist es wichtig, dass der Javadoc-Kommentar den Vertrag zwischen Methode und anwendendem Code beschreibt:

- Es ist erklärt, *was* die Methode macht.
- Es sind sämtliche Vorbedingungen aufgelistet (z.B., dass Argument nicht `null` sein darf).
- Es sind sämtliche Nachbedingungen aufgelistet (z.B., dass retournierte Collection nie `null` sein kann).
- Es ist jede Exception aufgelistet, die auftreten kann und in welcher Situation das passiert.
- Jeder Parameter ist beschrieben.
- Sofern sinnvoll ist der Rückgabewert beschrieben.

Für abstrakte Methoden ist ausserdem wichtig, dass Sie beschreiben, welche Erwartungen an eine Implementierung gestellt werden.

Javadoc für eine abstrakte Methode aus Spring ☺

```
public abstract class AbstractBeanFactory extends FactoryBeanRegistrySupport
implements ConfigurableBeanFactory {
        /**
         * Return the bean definition for the given bean name. (1)
     *
         * Subclasses should normally implement caching, as this method is invoked (2)
         * by this class every time bean definition metadata is needed.
         *
     * <p>Depending on the nature of the concrete bean factory implementation,
         * this operation might be expensive (for example, because of directory
lookups
         * in external registries). However, for listable bean factories, this usually
         * just amounts to a local hash lookup: The operation is therefore part of the
         * public interface there. The same implementation can serve for both this
         * template method and the public interface method in that case.
         *
     * @param beanName the name of the bean to find a definition for (3)
         * @return the BeanDefinition for this prototype name (never {@code null}) (4)
         * @throws org.springframework.beans.factory.NoSuchBeanDefinitionException (5)
         * if the bean definition cannot be resolved
         * @throws BeansException in case of errors
         *
     * @see RootBeanDefinition
         * @see ChildBeanDefinition
         * @see
org.springframework.beans.factory.config.ConfigurableListableBeanFactory#getBeanDefini
tion
         */
}
```

1. Kurzbeschrieb der Methode. Erscheint in der Methodenübersicht im Javadoc.
2. Anweisungen für die Implementierung.
3. Beschreibung der Parameter
4. Beschreibung des Rückgabewerts inklusive Nachbedingungen
5. Beschreibung der möglicherweise auftretenden Exceptions inklusive Erklärung, wann sie auftreten können.

Weiterführende Informationen

- Bloch, Jushua: Effective Java. Pearson Education, 2017, S. 254-260.

Korrektheit

Null-Objekte anstelle von `null` (PROG1 Kapitel 4: Sammlungen)

Geben Sie als Rückgabewerte von Methoden nach Möglichkeit sogenannte Null-Objekte zurück und nicht `null` an sich. Damit verhindern Sie eine lästige `NullPointerException`.

Grosse Gefahr für `NullPointerException` im aufrufenden Code ☹

```
public class MenuPlan {

    private List<Menu> menus;

    public List<String> getMenuNames() {
        if (menus == null) {
            return null;
        }

        return menus.stream().map(Menu::getName).collect(Collectors.toList());
    }
}
```

Gefahrlose Methodennutzung dank Null-Objekt ☺

```
class MenuPlan {

    private List<Menu> menus;

    public List<String> getMenuNames() {
        if (menus == null) {
            return Collections.emptyList();
        }

        return menus.stream().map(Menu::getName).collect(Collectors.toList());
    }
}
```

Obige Lösung kann noch verbessert werden, indem Sie verunmöglichen, dass das Feld menus jemals null sein kann.

Eingabeüberprüfung (PROG1 Kapitel 9: Fehlervermeidung)

Argumentüberprüfung

Überprüfen Sie Argumente, bevor Sie sie weiterverarbeiten. Zum Beispiel dürfen Index-Angaben für Arrays nicht negativ sein oder eine Division durch Null ist nicht erlaubt. Verhindern Sie diese Probleme nicht frühzeitig, können unverständliche Exceptions auftreten oder stillschweigend ungültige Werte berechnet werden. Indem Sie die Argumente frühzeitig validieren und Exceptions werfen, können Sie Ihren Code robuster gestalten, Sicherheitsprobleme und Exceptions verhindern.

Modulus wird frühzeitig validiert (Ausschnitt aus JDK 8) ☺

```
/**
 * Returns a BigInteger whose value is {@code (this mod m}).  This method
 * differs from {@code remainder} in that it always returns a
 * <i>non-negative</i> BigInteger.
 *
 * @param  m the modulus.
 * @return {@code this mod m}
 * @throws ArithmeticException {@code m} &le; 0 (3)
 * @see    #remainder
 */
public BigInteger mod(BigInteger m) {
    if (m.signum <= 0) { (1)
        throw new ArithmeticException("BigInteger: modulus not positive"); (2)
    }
    // Actual computation
}
```

1. Bevor die eigentliche Berechnung ausgeführt wird, wird geprüft, ob der Modulus negativ ist.

2. Es wird eine Exception geworfen, bevor der Code während der Berechnung in einen möglicherweise ungültigen Zustand kommt.
3. Dokumentieren Sie die Exception und in welchen Fällen sie geworfen wird im Javadoc!

Man kann sich streiten, ob vor dem Zugriff auf `m.signum` geprüft werden sollte, ob `m` nicht eventuell `null` ist. Weil `m` direkt verwendet wird, kann man sich das sparen, weil man weiss, dass Java eine `NullPointerException` werfen wird. Würde `m` dagegen erst später verwendet werden, wäre dies dagegen nötig, wie im folgenden Konstruktor:

Argumentüberprüfung warnt frühzeitig vor ungültigem Argument ☺

```
public class SomeEntityDao {

    private final DataSource dataSource;

    public SomeEntityDao(DataSource dataSource) {
        Objects.requireNonNull(dataSource, "DataSource may not be null"); //(1)
        this.dataSource = dataSource;
    }

    public List<SomeEntity> getAll() {
        try (Connection connection = this.dataSource.getConnection(); //(2)
             Statement stmt = connection.createStatement()) {
            // query database
        } catch (SQLException e) {
            throw new RuntimeException(e);
        }
    }
}
```

1. Ist `dataSource null`, wird eine `NullPointerException` geworfen.
2. Würde `dataSource` nicht im Konstruktor validiert, würde die `NullPointerException` erst hier auftreten und das könnte auch zeitlich viel später sein. Je nach Architektur Ihrer Software wird die fehlende `DataSource` mit der Validierung schon beim Applikationsstart bemerkt und nicht erst, wenn eine Webseite besucht wird, auf der `getAll()` verwendet wird.

Prepared Statements

Verwenden Sie Prepared Statements, um SQL-Injection-Angriffe zu verhindern. Bei SQL-Injection-Angriffen ist es Angreifern möglich, durch geschickte Abänderung von URL-Parametern oder Eingabefeldern in Formularen SQL-Abfragen zu manipulieren, um an zusätzliche Daten zu kommen, sie zu verändern oder zu löschen.

Zusammengesetzte SQL-Abfrage, die anfällig ist für SQL Injection ☹

```
String query = "SELECT balance FROM account WHERE user_name = '" +
request.getParameter("userName") + "'"; //(1)
```

1. Durch das direkte Anhängen des Inhalts des Parameters `userName` kann die SQL-Abfrage modifiziert werden. Gibt man für den Parameter zum Beispiel `' OR '1'='1` ein, entsteht `SELECT balance FROM account WHERE user_name = '' OR '1'='1'`, womit die Einschränkung durch die `WHERE`-Klausel ausgehebelt wird und der Kontostand sämtlicher Kunden abgefragt werden kann.

Bei Prepared Statements werden die SQL-Abfragen nicht zusammengesetzt, sondern Platzhalter hinterlegt und dann Werte für die Platzhalter angegeben. Auf diese Weise funktioniert der SQL-Injection-Angriff nicht mehr, weil die SQL-Abfrage unveränderlich ist.

SQL-Abfrage mit Platzhalter, immun gegen SQL Injection ☺

```
String userName = request.getParameter("userName"); //(1)
String query = "SELECT balance FROM account WHERE user_name = ?";
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setString(1, userName);
```

1. Vergessen Sie nicht, den Benutzernamen zusätzlich zu validieren, z.B. mit einem regulären Ausdruck, bevor Sie ihn weiterverarbeiten. Nicht nur SQL ist anfällig für manipulierte Eingabewerte. Indem Sie möglichst früh Eingabewerte validieren, reduzieren Sie das Risiko für erfolgreiche Angriffe.

Note	Schnellere Datenbankabfragen dank Prepared Statements Der eigentliche Zweck von Prepared Statements ist, Datenbankoperationen zu beschleunigen. Vor allem im Kontext von Batch-Operationen, bei denen immer wieder dieselbe Operation mit unterschiedlichen Werten ausgeführt wird, können Sie dem Datenbankserver viel Arbeit ersparen, da er bei Prepared Statements nicht immer wieder dasselbe SQL Statement aufbereiten muss. Aber selbst ausserhalb von Batch-Operationen ist die Nutzung von Prepared Statements vorteilhaft, da moderne Datenbankserver über Execution Plan Caches verfügen. Mit diesen merken sich Datenbankserver über längere Zeit hinweg, wie sich SQL-Operationen optimal ausführen lassen. Die Voraussetzung dafür ist aber, dass Sie dem Datenbankserver helfen, die Operationen wiederzuerkennen, indem Sie Prepared Statements verwenden, da die Abfragen damit selbst bei variierenden Parametern unveränderlich bleiben. Mehr dazu erfahren Sie im Blog-Beitrag der JOOQ-Entwickler Why SQL Bind Variables are Important for Performance.

Weiterführende Informationen

- Bloch, Jushua: Effective Java. Pearson Education, 2017, S. 227-230.
- Munroe, Randall: XKCD, Exploits of a Mom [Online]. URL: https://www.xkcd.com/327/ [Stand: 05.03.2018]
- Open Web Application Security Project: SQL Injection Prevention Cheat Sheet [Online]. URL: https://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet [Stand: 05.03.2018]

Object-Klassenvertrag wird eingehalten (PROG1 Kapitel 11: Mehr über Vererbung)

Vermeiden Sie nach Möglichkeit, die Methoden `equals(Object)` und `hashCode()` der Klasse `Object` zu überschreiben. Falls Sie es doch tun, beachten Sie folgende Kriterien:

- Die Klasse muss ein Value Type, also unveränderlich, sein.
- Wenn `equals(Object)` überschrieben wird, muss gleichzeitig auch `hashCode()` überschrieben werden und umgekehrt.
- Der Klassenvertrag, also die Javadoc-Beschreibung, muss unbedingt eingehalten werden.

Halten Sie sich unbedingt an diese Regeln, da es ansonsten zu überraschenden Fehlern kommen kann, beispielsweise dass Objekte in sortierten Collections nicht am richtigen Ort erscheinen oder nicht gefunden werden. Wie Sie die Anforderungen richtig umsetzen, ist detailliert im Buch *Effective Java* beschrieben (siehe Referenzen).

Weiterführende Informationen

- Bloch, Jushua: Effective Java. Pearson Education, 2017, S. 37-54.

Varia

Angemessene Verwendung von Collections (PROG1 Kapitel 4: Sammlungen)

Kommentar PROG1: Verwenden Sie den jeweils zur Problemstellung passenden Sammlungstyp (z.B. HashMap, HashSet, ArrayList).

Wenn Sie Objekte in Collections (Array, List, …) ablegen, achten Sie darauf, die geeignete Collection zu wählen. Die Auswahl ist nämlich gross und Sie können sich dadurch eine Menge Arbeit sparen. Ein `SortedSet` oder eine `SortedMap` sind zum Beispiel implizit sortiert, bei einer `ArrayList` müssen Sie sich selbst darum kümmern.

- Wenn Sie spezielle Anforderungen haben, studieren Sie die Übersicht über das Java Collections Framework, um eine passende Collection zu finden.
- Verwenden Sie `ArrayList` anstelle von Arrays. `ArrayList` ist typsicher während der Übersetzung, d.h. der Compiler hält Sie davon ab, ungültige Datentypen in einer `ArrayList` zu speichern. Arrays sind dagegen nur zur Laufzeit typsicher. `ArrayList` ist ausserdem flexibler bezüglich Iteration, unterscheidet zwischen belegten Elementen und seiner tatsächlichen Länge und seine Länge kann dynamisch verändert werden.
- Die Hilfsklasse `Collections` bietet eine grosse Auswahl nützlicher Methoden an, u.a. zum Kopieren und Sortieren von Collections. Wenn Sie `Collections` mit keinem oder nur einem Element benötigen, verwenden Sie `Collections.emptyList()` und Konsorten beziehungsweise `Collections.singletonList(T o)` und so weiter. Diese sind effizienter, als wenn Sie selber die entsprechenden Collections erstellen.
- Verzichten Sie auf `java.util.Vector`.

Tip	Studieren Sie das Javadoc der verschiedenen Collections und insbesondere deren Konstruktoren. Eine `LinkedBlockingQueue` kann man zum Beispiel mit den richtigen Konstruktor-Argumenten in ihrer Grösse begrenzen oder aus einer `LinkedHashMap` durch überschreiben einer Methode einen Cache machen.