3ANHANG

3.1Programmierrichtlinien

Diese Richtlinien sollen helfen, gut lesbare, gut wartbare und möglichst robuste Programme zu erstellen.

3.1.1Zentrale Prinzipien

Alles so **konstant**, sinnvoll **initialisiert**, **lokal** und **redundanzfrei** wie möglich. Schachtelungsorientierte **Einrückung**! Erläuterungen dazu siehe Vortrag [Programmierstil].

Abbrechende Fehlerprüfung: Siehe Methode `lehrkraftbezogeneNachrichtErstellen` in der Klasse `LgLehrkraftnews` in der fb6-Java-Software:

https://www.assembla.com/code/lehrkraftnews/git/nodes/fb6/java/fb6/lehrkraftnews/lg/LgLehrkraftnews.java

3.1.1.1Zwischenergebnisse einfrieren

Aus der Anforderung, Variablen möglichst konstant und sinnvoll initialisiert zu deklarieren, ergibt sich, dass methodenlokale Zwischenergebnisse immer eingefroren werden sollen. Das Überschreiben von Variablenwerten durch eine Zuweisung soll nur für echten Zustand reserviert bleiben. Vorbildlich ist z.B. folgende Methode, in der eine Datenbanksuchanfrage schrittweise aufgebaut wird:

```
public LgBenutzerFilmBeziehung getBeziehung(
   final LgBenutzer benutzer, final LgFilm film, final LgSitzung session
){
   final CriteriaBuilder builder = session.getEm().getCriteriaBuilder();
   final CriteriaQuery<LgBenutzerFilmBeziehung> query
   = builder.createQuery(LgBenutzerFilmBeziehung.class);

   final Root<LgBenutzerFilmBeziehung> root
   = query.from(LgBenutzerFilmBeziehung.class);
   query.select(root);

   final Predicate pred1
   = builder.equal(root.<String> get(LgBenutzerFilmBeziehung.FILM), film);
   final Predicate pred2
   = builder.equal(root.<String> get(LgBenutzerFilmBeziehung.BENUTZER), benutzer);

   query.where(builder.and(pred1, pred2));

   return this.getSingleByCriteria(query, session);
}
```

3.1.2Redundanzfreie Verwendung von Literalen

Literale (Zahlen, **Bsp**. `5`, `3.1415` und Zeichenketten, Bsp. `"EUR"`) nie direkt im Anweisungsteil einer Operation benutzen, sondern immer nur zur einmaligen Deklaration einer ihrer Bedeutung entsprechend benannten Konstanten, deren Name dann in Anweisungen mehrfach benutzt werden darf.

Gründe:

a)Die meisten Literale haben eine bestimmte Bedeutung und sind nicht zufällig so gewählt. Für die Wartbarkeit ist es erforderlich, daß bei Änderung des Wertes diese nur an einer zentralen Stelle erfolgen

muss. Die zu obigen Literalen gehörigen Konstantendeklarationen könnten lauten:

Bsp.:
```
final byte plzLaenge = 5;
final float pi = 3.1415;
final String waehrungsKz = "EUR";
```

b)Eine Zeichenkette, aufgrund deren Wert eine Programmverzweigung vorgenommen wird, ist in Literalform besonders gefährlich, denn der Compiler kann die Existenz der entsprechenden Verarbeitung nicht überprüfen. Damit wird ein potentieller Fehler, z.B. bei Vertippen oder nach Umbenennung auf die Laufzeit, d.h. auf den Kunden abgewälzt. Man kann solche Fallunterscheidungen oft auch auf dem `Class`-Objekt der verarbeitenden Klasse statt auf einem String basieren lassen.

Bsp.: Statt wie in vielen Lehrbüchern bei Einsatz von JDBC zum Laden des Datenbanktreibers

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

ist die folgende Form sicherer, da bei Nichtvorhandensein der Treiber-Klasse der Compiler einen Fehler melden würde:

```
Class.forName(sun.jdbc.odbc.JdbcOdbcDriver.class.getName());
```

Bsp.: Statt in einer Swing-GUI eine button-abhängige Verzweigung durch Vergleich des Command-Strings mit einem String-Literal durchzuführen wie in:

```
void actionPerformed(final ActionEvent i_ev){
  final String command = i_ev.getActionCommand();
  if(command.equals("Drucken"){_drucken();
  }else if(command.equals("Speichern"){_speichern();
  }
```

ist es sicherer (und effizienter), die Verursacher-Objekte des Events zu vergleichen:

```
void actionPerformed(final ActionEvent i_ev){
  final Object command = i_ev.getSource();
  if(command == _druckenButton){_drucken();
  }else if(command == _speichernButton){_speichern();
  }else{throw new IllegalArgumentException(command.toString());
  }
  …
}
```

3.1.3Redundanzfreiheit zwischen externen und internen Größen

Die notwendige Konsistenz zwischen programmexternen Größen (z.B. Konfigurationsangaben, Nationaltexte) muss zur Übersetzungszeit gesichert werden. Zu den möglichen Mitteln zählt eine Konstantengenerierung.

Bsp.: Aus der Nationaltextdatei `texte.properties`:

loginUsername = Username
loginPassword = Password

kann mit wenig Aufwand folgende Klasse generiert werden:

```
class UiKeys {
    final String loginUsername = "loginUsername";
    final String loginPassword = "loginPassword";
}
```

Die Verwendung der Konstanten aus `UiKeys` garantiert dann die Existenz des entsprechenden Nationaltexts zur Übersetzungszeit:

```
usernameField.setLabel( getResource(UiKeys.loginUsername) );
```

3.1.4Namenskonventionen

Groß/Klein-Schreibung:

•Gliederung: Bezeichner werden durch eingestreute **Großbuchstaben** gegliedert. Bsp.: `zahlLesen`. Der Unterstrich ('_') ist nur für die weiter unten angegebenen Namenspräfixe reserviert.

•Typbezeichner: beginnt (nach eventuellem Namenspräfix) mit einem **Großbuchstaben**, Bsp.: `Punkt`. Alle anderen Bezeichner (Variablen, Konstanten, Methoden) beginnen mit einem **Kleinbuchstaben**., Bsp.: `final Punkt punkt, nachbarPunkt;`

Gründe: Wir müssen aufgrund der Sprachregeln Typ und Variable des Typs mit unterschiedlichen Bezeichnern versehen. Die Konvention verhindert, dass pausenlos Synonyme eingeführt werden müssen. Als **Negativbeispiel** sei gegeben:

```
class punkt { ... }
...
final punkt ort;
```

Viel besser ist:

```
class Punkt { ... }
...
final Punkt punkt;
```

Namenspräfixe für Parameterübergaberichtung

Die logische Parameterübergaberichtung (**in**, **out**, **in out**) jedes formalen Parameters soll dadurch ausgedrückt werden, daß jeder Parameterbezeichner mit einem der Präfixe '`i_`', '`o_`', '`io_`' versehen wird, ***Bsp***.:

```
/** Sortiert die ersten i_anzahl Elemente des Feldes io_feld */
static void sortieren(final long[] io_feld, final int i_anzahl){...}
```

Dabei ergibt sich, daß Parameter eines elementaren Typs nur den Präfix '`i_`' haben können, wohingegen bei Parametern eines Referenztyps nach der Übergaberichtung des Gezeigten (nicht der Referenz selbst) zu urteilen ist.

3.1.5Attributzugriff kenntlich machen

Jeder Zugriff auf ein Attribut der eigenen oder einer Oberklasse muss einheitlich erkennbar sein. Im Projekt müssen Sie sich festlegen, ob Sie dies mittels Namenspräfix _ für nichtöffentliche Attribute oder durch Qualifizierung jeden einzelnen Zugriffs mittels **`this`** oder **`super`** erreichen.

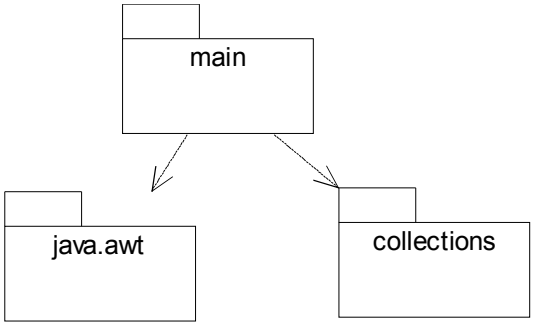
Bsp.:
```
class Person {
    private String name;
    String getName(){
        return this.name; //nur so erlaubt
    }
}
```

Gründe: Dies reduziert den globalen Namensraum. Am Namenspräfix **`this.`** kann der Definitionsort klassenweit statt methodenlokal erkannt werden.

Prüfung: Aktivieren Sie in Eclipse: Window > Preferences > Java > Compiler > Errors/Warnings > Unqualified access to instance field.

3.1.6Import-Deklarationen

Import-Deklaration: Der Globalimport `import paket.*;` ist verboten.



Gründe:

a) Ohne Globalimport erzwingt der Compiler, dass im Quellcode der Typname voll qualifiziert wird, d.h. mit Angabe des Paketnamens. Dies erhöht die Verständlichkeit des Codes und erleichtert das schnelle Auffinden der Typdefinition. Empfohlen wird statt Globalimport der Einzelimport, z.B.

```
import java.awt.List;
```

b) Der Globalimport mehrerer Pakete führt leicht zu verspäteten Namenskonflikten, ohne dass die Ursache dem Programmierer dann noch bewußt wäre. Bsp.: Im Paket `main` steht

```
import java.awt.*;
import collections.*;
```

und wird der Typ `List` aus `java.awt` benutzt. Es geht alles gut. Später kommt der Autor des Pakets `collections` auf die gute Idee, einen Typ `List` zu exportieren. Dann tritt in Paket `main` ein Namenskonflikt ein, ohne daß an `main` etwas geändert wurde.

3.1.7static-Variablen

`static`-Variablen sind verboten. Demgegenüber sollten Konstanten zwecks Speicherplatzersparnis sogar **`static`** sein.

Bsp.: Verboten sind:

```
static final LgPerson _aktuellerBenutzer = new LgPerson(); //änderbar
static String _letzteWebseite; //ersetzbar
```

Stattdessen sollen die benötigten Zustände/Abhängigkeiten als Objektattribute einer Kontextklasse gespeichert und möglichst im Konstruktor einfrierend initialisiert werden. Erlaubt sind daher:

```
private final LgPerson _aktuellerBenutzer = new LgPerson();
private String _letzteWebseite;
```

Gründe:

Um Komponentenorientierung (siehe Kapitel 1.9) durchzusetzen, muss feststehen, von welchen anderen Komponenten eine Komponente abhängig ist. Jede Komponente wird dabei als ein Objekt dargestellt, jede Abhängigkeit als eine Objektreferenz. Globale Objekte sind, wie früher globale Variablen, verpönt. Das **Singleton**-Muster gilt daher ausdrücklich als schädlich! Siehe:

http://www.c2.com/cgi/wiki?SingletonsAreEvil
http://archive.eiffel.com/doc/manuals/technology/bmarticles/joop/globals.html

Methoden:

`static`-Methoden sind unproblematisch, solange sie nicht auf **`static`**–Variablen zugreifen. Eine **`static`**-Methode darf daher nur auf Variablen zugreifen, die ihr als Parameter übergeben wurden. Dann ist sie eine reine Utility-Methode.

3.1.8Typspezialisierung (type cast)

Eine explizite Typspezialisierung mittels „type cast“ verschiebt die Typprüfung vom Übersetzungszeitpunkt auf den der Programmausführung (d.h. beim Kunden). ***Daher***:

- möglichst ersetzen durch Aufruf einer abstrakten Methode! ***Bsp***. `.toString()`
- unvermeidliche Typspezialisierung an Ort mit dem größten Wissen über Erfolg konzentrieren, d.i. meist in der Klasse, zu der man hinspezialisiert oder im Fall einer typsicheren generischen List in dieser.

Bsp.: `ELEMENT_List` in http://www.tfh-berlin.de/~knabe/java/lib/

3.1.9Steuerkonstrukte und geschweifte Klammern

Jeder Rumpf eines Steuerkonstruktes (**`if`**, **`else`**, **`while`**, **`for`**, **`do`**, **`switch`**) ist mit geschweiften Klammern zu versehen, auch wenn er aus nur einer Anweisung besteht.

Gründe: Die häufig verwendete eingerückte Schreibweise ohne geschweifte Klammern lässt nachträglich eingefügte Anweisungen fälschlicherweise als Teil des gesteuerten Rumpfes erscheinen. Ein korrektes Einfügen hingegen würde ein Einfügen an zwei auseinander liegenden Stellen erfordern. ***Bsp***.:

```
for(i = 0; i<N; i++)
    values[i]++;
    System.out.println(values[i]);
System.out.println("============================");
```

Hintergrund: In Sprachen wie Ada oder Modula-2 klammern die Schlüsselwörter **`if`** … **`end if`** von sich aus. In Perl mit seiner C-ähnlichen Syntax werden immer geschweifte Klammern verlangt.

3.1.10Steuerkonstrukte mit fehlendem else-Zweig

Alle mehrzweigigen Steuerkonstrukte (**`if`**, **`switch`**) müssen mit einem Restfall-Zweig versehen werden. Wenn der Restfall nicht erwartet wird, ist sein Betreten ein Fehler und sollte folgerichtig eine Ausnahme geworfen werden. ***Beispiele***:

```
final String command = … ;
if(command.equals(printCommand){
    print(filename);
}else if(command.equals(deleteCommand){
    delete(filename);
}else{
    throw new multex.Exc(
        "Kommando {0} unerwartet aufgetreten.", command
    );
}
```

```
final int command = … ;
switch(command){
    case printCommand:
        print(filename);
    break;
    case deleteCommand:
        delete(filename);
    break;
    default:
        throw new multex.Exc(
            "Kommando {0} unerwartet aufgetreten."
            , new Integer(command)
        );
    break;
}
```

3.1.11Entwicklungsdokumentation

Jede Einheit (Paket, Klasse, Methode, Attribut) muss mit einem Javadoc-Kommentar versehen sein. Bei einem Paket, einer Klasse oder einem Attribut beschreibt man den Zweck, bei einer Methode spezifiziert man ihren Effekt bzw. ihre Ausnahmen. Die Paket-Doku muss in einer Datei `package.html` im zugehörigen Verzeichnis stehen.

3.2Quellenangaben

[Ambler] Ambler, Scott W.: „Mapping Objects To Relational Databases“, Ronin International White Paper, http://www.AmbySoft.com/mappingObjects.pdf , 1998-2000.
Ausführliche Abwägung verschiedener Strategien des Object-Relational Mapping.
Ambler, Scott W.: „Design of a Robust Persistence Layer for Relational Databases“, Ronin International White Paper, http://www.AmbySoft.com/persistenceLayer.pdf , 1997-2000.
Entwurf einer Object-Relational-Mapping-Schicht und seine Begründung.

[Balzert99] Balzert, Heide: "Lehrbuch der Objektmodellierung. Analyse und Entwurf", Spektrum Akademischer Verlag, 1999, 573pp., ca. 50,- €.
Didaktisch sehr gut ausgearbeitet, dennoch erkennbar mit praktischer Erfahrung, deckt außer dem Bereich "Testen" alles Relevante für unser Software-Projekt ab, insbesondere Objektorientierte Analyse, Oberflächenentwurf, 3-Schichten-Architektur, Object-Relational Mapping.

[Bank3Tier] http://www.tfh-berlin.de/~knabe/java/bank3tier/:
Eine Demo-Java-Applikation aus dem Bankwesen (Kunde, Konto, überweisen) in 3 Schichten.

[cvshome] www.cvshome.org:
Übersicht über CVS und damit zusammenhängende Produkte.

[cvsgui] www.cvsgui.org:
Übersicht über GUI-Klienten für CVS.

[Database] www.javaskyline.com/database.html:
Übersicht über generische OR-Mapping-Tools.

[Fogel99] Fogel, Karl: „Open Source Development with CVS“, 446pp., Coriolis Group 1999 <en>, Bonn 2000 <de>.
Einführung in CVS und fortgeschrittene Benutzungsstrategien für Open-Source-Projekte.

[Gamma95] Gamma, Helm, Johnson, Vlissides: Entwurfsmuster, <de> Addison-Wesley, Bonn, 1996, <en> 1995
Das Standardwerk über Entwurfsmuster, aus dem die angeführten Muster entnommen sind.