VII: Objektorientiertes Design

- Zerlegung einer Problemstellung in Klassen
- Schnittstellen
- Packages

Fallstudie: Game of Life

- Erfunden 1970 von John Horton Conway.
- Spielplan: unendliches 2D-Gitter (für uns: $\infty = 100$ o.ä.)
- Eine Zelle ist entweder lebendig oder tot.
- Zu Beginn sind manche Zellen lebendig andere nicht.
- In jeder Runde ändern sich die Zustände wie folgt:
  - Hat eine lebendige Zelle nur einen oder gar keine lebendigen Nachbarn, so "stirbt" sie in der nächsten Runde.
  - Hat eine lebendige Zelle vier oder mehr lebendige Nachbarn so "stirbt" sie ebenso in der nächsten Runde.
  - Hat eine tote Zelle genau drei lebendige Nachbarn, so wird sie in der nächsten Runde lebendig.
  - Ansonsten ändert sich der Zustand der Zelle nicht.

Aufgabe: Simulation des Spiels.

Benutzerschnittstelle

- Das Spielfeld wird als $m \times n$ Gitter im Grafikfenster dargestellt.
- Lebendige Zellen werden rot ausgefüllt, tote Zellen weiß.
- Zu Beginn werden die lebendigen Zellen mit der Maus angewählt.
- Es werden 1000 Runden simuliert; der zeitliche Abstand der Runden kann eingestellt werden.

Grundlegende Designprinzipien

- Mögliche Kandidaten für Klassen sind die Substantive der Problembeschreibung.
- Mögliche Kandidaten für Methoden sind die Verben der Problembeschreibung.
- Klassen sollen nicht zu eng gekoppelt sein (nicht “uses”-Beziehungen zwischen **je** zwei Klassen)
- Klassen sollen nicht zu groß werden; meist nicht mehr als 1-2 Bildschirmseiten.
- Großzügig Klassen einführen, keine falsche Sparsamkeit.
- Auf bekannte Entwurfsmuster zurückgreifen.
- Design immer wieder verbessern; Architektur mutig umkrempeln (code smells $\rightarrow$ refactoring)

Architektur der Fallstudie

- Eine Klasse, die die Parameter (Geometrie, Schnelligkeit, Anzahl der Runden) definiert. Enthält nur statische Instanzvariablen.
- Eine Klasse für Positionen (nicht veränderbar, *immutable*):
  - Instanzvariablen x, y.
  - Methoden: x, y abrufen; Nachbarpositionen berechnen.
- Eine Klasse für Zellen (nicht veränderbar, *immutable*):
  - Instanzvariablen: Position, Zustand (tot/lebendig)
  - Methoden: Zustand und Position abfragen.
- Eine Klasse für das Spielfeld:
  - Instanzvariablen: 2D Array von Zellen
  - Methoden: Zelle an einer Position abrufen; Zelle setzen.
- Eine Klasse für den Spieler (“die Natur”):
  - Instanzvariablen: Spielfeld, Ansicht.

Architektur der Fallstudie

  - Methoden: eine Runde durchführen:

- Eine Klasse für die Ansicht:
  - Instanzvariablen: GraphicsWindow
  - Methoden: Spielfeld zeichnen, Zelle zeichnen, Position durch Mausklick.

Entwurfsprinzip MVC (Model-View-Controller)

Die Aufgaben sind auf drei Komponenten verteilt:

- Modell, hier das Spielfeld. Verantwortlich für die Datenhaltung.
- View, hier die Ansicht. Verantwortlich für die (grafische) Darstellung der Daten.
- Controller, hier der Spieler. Verantwortlich fuer das Verarbeiten von Aktionen. Verändert das Modell entsprechend und veranlasst Darstellung der Änderungen bei der Ansicht.

Dieses Architekturprinzip hat sich sehr bewährt; nicht davon abweichen!

Glaubt man, die spezielle Problemstellung erfordere eine andere Architektur, so liegt oft ein Mangel an Erfahrung vor.

Umgang mit Zellen am Rand des Spielfelds

- zur Ermittlung des Folgezustands einer Zelle müssen alle acht Nachbarn (N, NO, O, SO, S, SW, W, NW) besucht werden.
- Zellen am Rand des Spielfelds haben aber nur 3 Nachbarn, noch dazu jeweils unterschiedliche.
- Vergisst man das, so greift man auf nicht existierende Arrayfächer zu. Programmabsturz.
- Explizite Behandlung der Randfälle mit Fallunterscheidungen ist lästig.

  Aus der Spieleprogrammierung sind zwei Standardlösungen des Problems bekannt.

Berechnung der Nachbarn

Entweder:

Die Randzellen bleiben immer tot; sie werden auch nicht angezeigt und ihr Folgezustand wird nicht neu berechnet. Sie fungieren nur als fiktive Nachbarn des Randes des sichtbaren Felds.

Oder:

Der linke Nachbar einer Zelle am linken Rand ist per Definition der rechts gegenüberliegende. Mit anderen Worten werden linker und rechter Rand verklebt, sowie auch der obere und untere. Man spielt sozusagen auf einem Autoreifen (Torus).

Implementierung der Fallstudie

Wir implementieren die Fallstudie während der Vorlesung.

Die endgültige Version wird anschließend auf der Vorlesungsseite bereitgestellt.

Packages

Man kann mehrere Klassen in eine *package* zusammenfassen.

- Diese müssen dann in einem Unterverzeichnis liegen, dessen Name der Packagename ist.
- Jede Datei der Package muss mit `package <name>;` beginnen, wobei `<name>` der Packagename ist.
- Auf Klassen der Package greift man durch Vorschalten von `<name>.` zu.
- Alternativ kann man `import <name>.*;` verwenden.
- Ist eine Klasse oder Methode oder Instanzvariable weder `public` noch `private`, so ist sie nur innerhalb ihrer Package sichtbar.
- Packages vom übergeordneten Verzeichnis kompilieren und ausführen.

Kompilieren von Packages

- Die Package `name` befindet sich im Unterverzeichnis `name` des aktuellen Verzeichnisses (folder, directory, Ordner).
- Man kompiliert die Datei `datei.java` in der Package `name` mit dem Befehl

  ```
  javac name/datei.java
  ```

  (Windows: `/` durch `\` ersetzen)
- Es genügt meistens, die Datei mit der `main()`-Methode zu kompilieren, da benutzte Dateien automatisch mitkompiliert werden.
- Ausführen des Programms durch den Befehl `java name.Klasse` wobei `Klasse` die Klasse mit der `main`-Methode ist.

Schnittstellen

Eine Schnittstelle (*interface*) besteht aus Methodensignaturen, d.h. Methodendefinitionen ohne den `{...}` Block.

Syntax:

```
public interface NamederSchnittstelle {
   MethodenSignatur1;
   MethodenSignatur2;
   ...
}
```

Mit einer Schnittstelle lassen sich Objekte, die ähnliche, gleichnamige Methoden implementieren, zusammenfassen.

`NamederSchnittstelle` ist jetzt ein Typ, der überall dort verwendet werden kann, wo Typen vorkommen (Deklaration von Variablen, Rückgabewerten).

Beispiel: Measurable

Beispiel:

```
public interface Measurable{
   double getMeasure();
}
```

Die Schnittstelle `Measurable` fasst Objekte zusammen, die ein Double-wertiges “Maß” haben.

Wir können jetzt die Klasse `DataSet` aus Kapitel V so implementieren, dass sie beliebige Objekte vom Typ `Measurable` verarbeiten kann.

Implementierung

Durch die Klausel `implements NamederSchnittstelle` in einer Klassendefinition wird angezeigt, dass Objekte dieser Klasse die Schnittstelle implementieren.

In der Klasse müssen dann alle Methoden der Schnittstelle implementiert werden.

Objekte der Klasse haben dann automatisch auch den Typ `NamederSchnittstelle`.

Implementiert eine Klasse die Methoden einer Schnittstelle, fehlt aber die `implements`-Klausel, so haben Objekte der Klasse *nicht* den Typ der Schnittstelle.

In UML zeichnet man von der Klasse zur Schnittstelle einen Pfeil mit kreisrunder, hohler Spitze `-----o`.

Beispiel

```
public class RectangleWithArea implements Measure {
   private Rectangle rectangle;
   double getMeasure() {
        return rectangle.width * rectangle*height;
   }
   public RectangleWithArea(Rectangle rectangle) {
        this.rectangle = rectangle;
   }
}
```

Sprites

Anderes Beispiel:

```
public interface Sprite {
    /** Zeichnen des Sprite in ein gegebenes
        Rechteck */
    public void zeichnen(GraphicsWindow g, Rectangle r);
}
```

Die Schnittstelle `Sprite` fasst Objekte zusammen, die “sich” in ein gegebenes Rechteck zeichnen können.

Engl.: *sprite* = kleiner Waldgeist.

Bezeichnet kleine Figuren, die im Rahmen eines Computerspiels oder einer Animation sich auf dem Bildschirm umherbewegen.

Spezifikation von Methoden: Vor- und Nachbedingungen

Man kann das Verhalten von Methoden durch eine Vor- und Nachbedingung (wie in der Hoare-Logik) beschreiben.

Dazu fügt man in der javadoc Dokumentation an geeigneter Stelle eine *Vorbedingung* und eine *Nachbedingung* ein.

Die Vorbedingung bezieht sich auf die Werte der Parameter der Methode und die Werte der Instanzvariablen vor Ausführung der Methode.

Die Nachbedingung bezieht sich auf die Werte der Parameter (vor Ausführung der Methode), die Instanzvariablen nach Ausführung der Methode und den Rückgabewert. Auf die Werte der Instanzvariablen vor Ausführung der Methode kann man auch Bezug nehmen durch entsprechende Kennzeichnung, etwa durch `_alt`.

Vor- und Nachbedingungen sind für uns *informell*.

Es gibt Werkzeuge wie JML, in denen Vor- und Nachbedingungen formalen Status haben und überprüft werden.

Spezifikation von Methoden: Vor- und Nachbedingungen

Beispiel:

```
/** Einzahlen.
   @param betrag der eizuzahlende Betrag.
   Vorbedingung: betrag >= 0.
   Nachbedingung: kontostand = kontostand_alt + betrag
*/
public void einzahlen(double betrag) {
    kontostand = kontostand + betrag;
}
```

Spezifikation von Methoden und Klassen: Invarianten

Oft soll eine bestimmte Bedingung an die Instanzvariablen von jeder Methode erhalten werden.

Statt diese in jede Vor- und Nachbedingung aufzunehmen kann man solch eine Bedingung als *Invariante* der Klasse spezifizieren.

Jede Methode muss dann diese Invariante nach ihrer Ausführung garantieren; im Gegenzug darf die Invariante vor Ausführung jeder Methode angenommen werden.

```
/** Klasse fuer Bankkonten mit Ueberziehungsmoeglichkeit.
    (Invariante: kontostand >= -limit).
*/
public class Bankkonto {
    /** Der Kontostand. */
    private double kontostand;
    /** Ueberziehungslimit. */
    private double limit;
```

Spezifikation von Klassen und Methoden: Zusammenfassung

Man kann die Beschreibung des Verhaltens von Methoden und Klassen in Vor- und Nachbedingungen, sowie Invarianten gliedern.

Die Vorbedingung einer Methode legt Bedingungen an ihren Aufrufkontext fest. Die Methode ist immer so aufzurufen, dass die Vorbedingung erfüllt ist.

Die Nachbedingung einer Methode spezifiziert den Zustand des Objekts nach Aufruf der Methode. Die Methode ist so zu implementieren, dass die Nachbedingung immer erfüllt ist, vorausgesetzt die Vorbedingung war erfüllt.

Die Invariante einer Klasse ist eine implizite Vor- und Nachbedingung für alle Methoden einer Klasse: Alle Methoden (einschließlich der Konstruktoren) sind so zu implementieren, dass die Invariante erhalten wird.

Verwender der Klasse können dann voraussetzen, dass alle Objekte der Klasse die Invariante erfüllen.

Spezifikation von Klassen und Methoden: Zusammenfassung

Vor- und Nachbedingungen sind für uns informell. Will man sie formalisieren, so treten nichttriviale Schwierigkeiten auf (exakter Geltungsbereich von Invarianten, Formulierung von Bedingungen ohne Bezugnahme auf private Instanzvariablen, . . .).

Es existieren solche Formalisierungen, z.B.: JML, die zudem das Erfülltsein von Bedingungen teilweise automatisch überprüfen.