XIII. Objektorientierte Programmierung – Entwurf, Modellierung und *Guidelines*

1. **UML als Werkzeug für den objektorientierten Entwurf – Grundlagen**
2. ***Styleguides* und Programmkonventionen**
3. **Objektorientierte Modellierung und Realisierung**

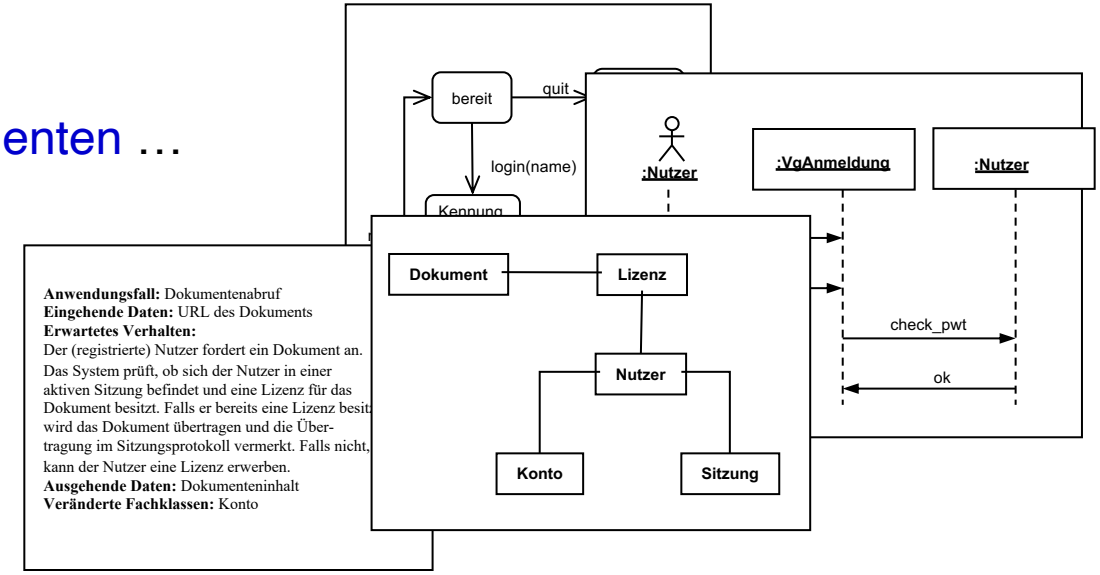
1. UML als Werkzeug für den objekt-orientierten Entwurf – Grundlagen

- Modellbegriff und objektorientierte Modellierung
- Klassendiagramme
- Klassenbeziehungen
- Sequenzdiagramme

Modellbegriff und objektorientierte Modellierung

Modelle – Wozu?

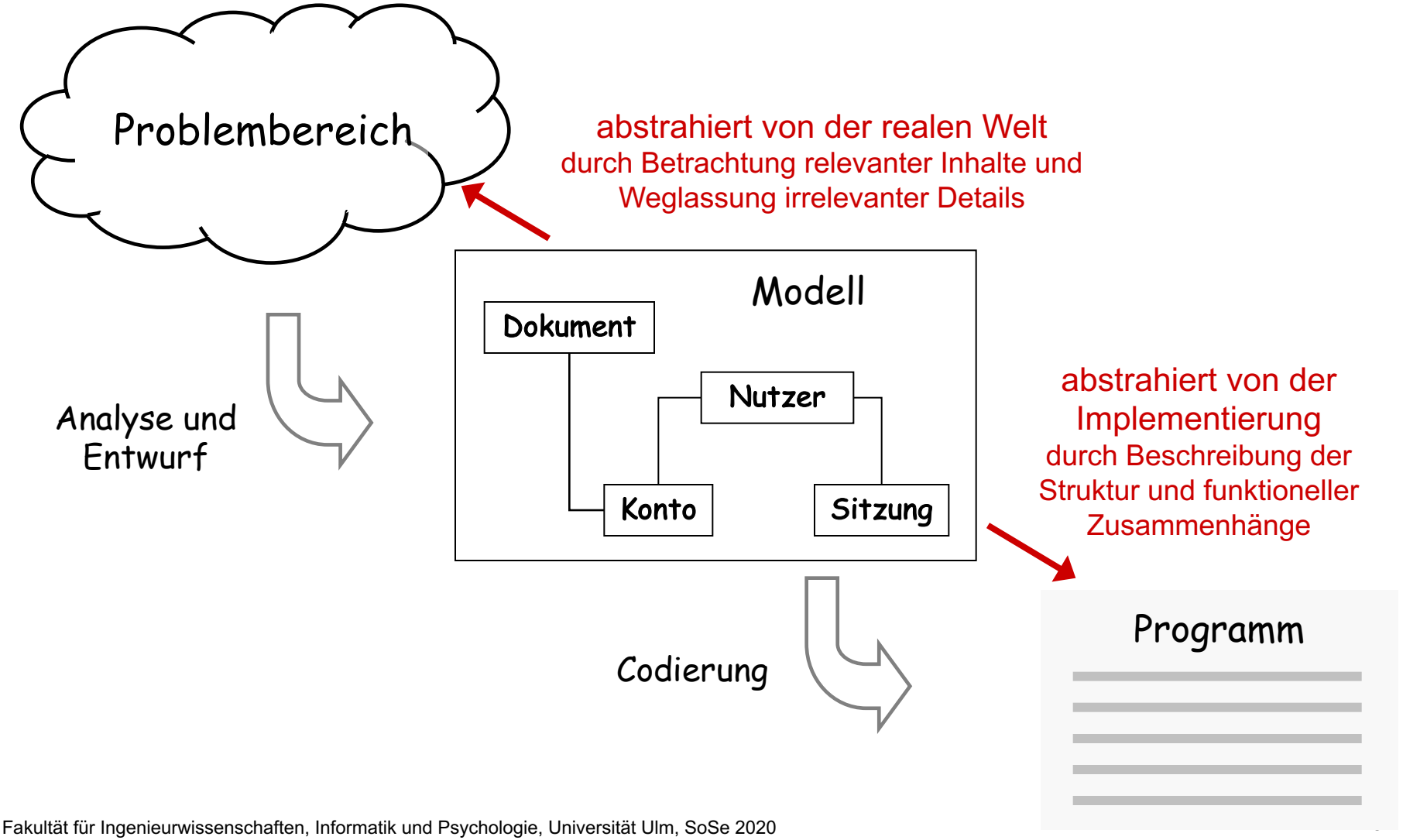
- Modelle für Softwarekomponenten …



- Ein Modell beschreibt die Eigenschaften eines realen Systems aus einer bestimmten Sichtweise heraus (z.B. Benutzersicht eines Systems) und für die Beschreibung bestimmter Eigenschaften und Zusammenhänge
- Modellierung impliziert Abstraktion und Reduktion durch Fokussierung auf die für die Aufgabenstellung (und ihre Lösung) relevanten Probleme
- Ein Modell für (objektorientierte) Software kann Text, Diagramme oder auch ausführbare Code-Fragmente enthalten

Verwendung von Modellen

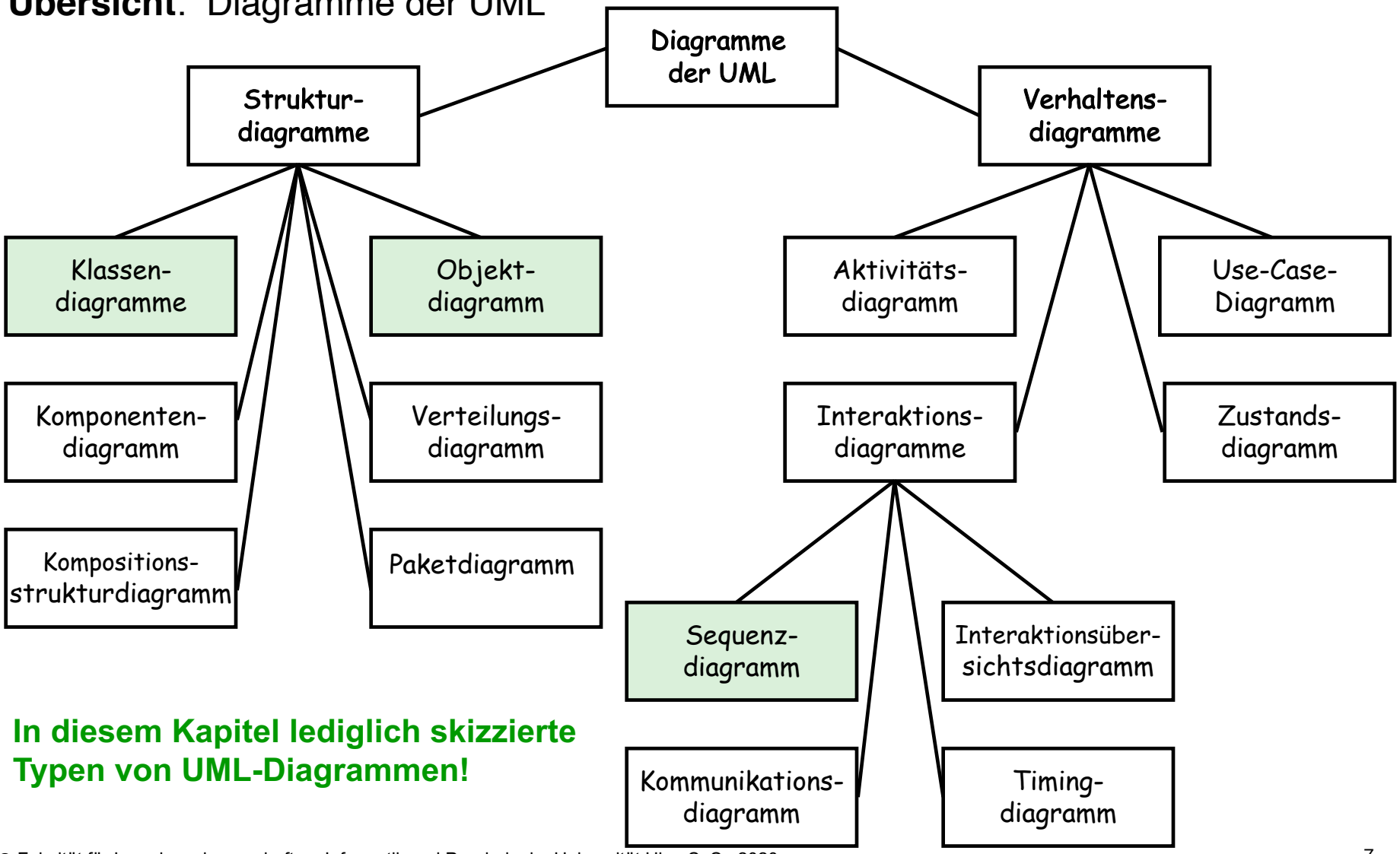
- Schritte beim (objektorientierten) Softwareentwurf (vgl. auch **Teil VII**)



- **Zielsetzung** der Modellierung bei der Softwareentwicklung
  - Vereinfachung der Kommunikation zwischen Softwareentwicklern untereinander und zwischen Entwicklern und Anwendern
  - Reduktion der Komplexität
  - Dokumentation des Systementwurfs auf angemessenem Abstraktionsniveau und Überprüfung der Vollständigkeit und Konsistenz
  - Modelle dienen als Hilfsmittel
    - für die Projektplanung
    - für den Auftraggeber zur Überprüfung des Funktionsumfangs
    - zur Konstruktion und Programmierung des Systems
- *Unified Modeling Language* (UML)
  - **Ziel**: Standardisierte Technik zur (abstrakten) Modellierung verschiedener Aspekte eines Systems, z.B. Komponenten und deren Interaktion, zeitliche Abläufe, etc.
  - Unabhängigkeit von Zielsprachen: UML definiert keine bzw. wenig programmiersprachliche Konzepte (z.B. Basistypen), die Spezifikation nutzt graphische Hilfsmittel
  - Kurze Historie: Standardisierte Sprache zur (objektorientierten) Modellierung von Systemen (Diagrammtypen, prädikative Sprache: *Object Constraint Language*, OCL)
  - Standardisierung durch *Object Management Group* (OMG; UML Version 1.1, 1997; UML Version 2.0, 2004)

Elemente der *Unified Modeling Language* (UML)

Übersicht: Diagramme der UML

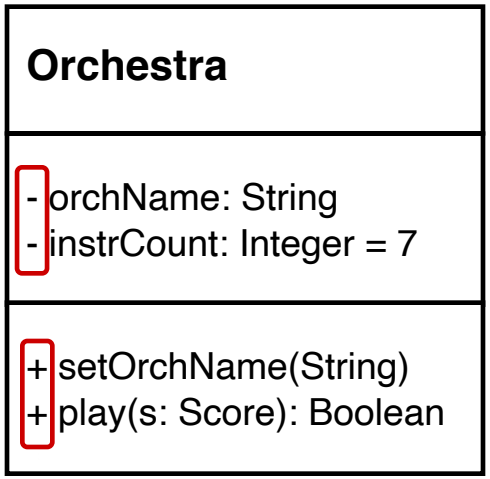


In diesem Kapitel lediglich skizzierte Typen von UML-Diagrammen!

Klassendiagramme

Verwendung und Abschnitte

- Klassendiagramme repräsentieren die statische Struktur eines Systems und zeigt Informationen zu Klassen und die Beziehungen zwischen ihnen an
- Struktur der Klassendiagramme
- Beispiel und Code-Ausschnitt



Klassenname

Attribute (*Member*-Felder)

Operationen (*Member*-Funktionen)

Sichbarkeitsindikatoren (s.S.10)

```
// Code-Ausschnitt in Java
class Orchestra {
    private String orchName;
    private int    instrCount = 7;

    public void setOrchName(String name) {
        ...
    }

    public boolean play(Score s) {
        ...
    }
}
```

Die einzelnen Abschnitte und ihre Notationen

Name

- Angabe des Klassennamens in Fettdruck
- Weitere optionale Angaben über die Eigenschaften der Klasse (in { … })

Attribute

- Optionales Feld mit der Angabe der Attribute (Felder), die den Zustand eines Objekts (Instanz der Klasse) bzw. permanente Klasseneigenschaften definieren
- Vollständige Definition:

```
Sichtbarkeit Name: Typ [Multiplizität] = Default-Wert
{Eigenschafts-String}
```

Hinweis: Üblicherweise werden nur Attribut-Namen und –typen sowie ihre Sichtbarkeit angegeben

Operationen (Methoden in Java)

- Optionales Feld mit der Definition der Funktionen, die das Systemverhalten repräsentieren

- Vollständige Definition:

```
Sichtbarkeit Name(Parameterliste): Rueckgabetyp-Ausdruck
{Eigenschafts-String}
```

Hinweis: Üblicherweise werden nur Operationsnamen und Parameter angegeben

- Operationale Sichtbarkeitsindikatoren für Felder und Funktionen

	Sichtbarkeits-Symbol			
UML	+	#	-	
Java	**public**	**protected**	**private**	(default)
sichtbar für: **gleiche Klasse**	ja	ja	ja	ja
andere Klasse, gleiches Paket	ja	ja/nein*	nein	ja
andere Klasse, anderes Paket	ja	nein	nein	nein
Unterklasse, gleiches Paket	ja	ja	nein	ja
Unterklasse, anderes Paket	ja	ja	nein	nein

(*) in UML und C++ "nein", in Java "ja"

Objektdiagramme

- Diagramme für die Darstellung von Instanzen aus den Klassen (Objekte) zur genaueren Detaillierung von Informationen

  Hinweis: Objektdiagramme werden eher selten verwendet

- Struktur: Text im Namensabschnitt wird unterstrichen

Objektnamen:	**: Klassenname**	(nur der Klassenname; anonymes Objekt)
	Objektname	(nur der Objektname)
	Objektname: Klassenname	(Objekt- und Klassenname)

- Beispiel:

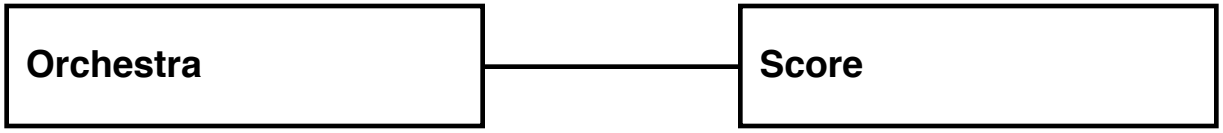
Objektname Klassenname

Attributnamen Attributwerte

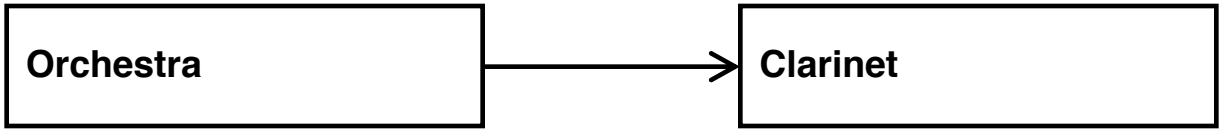
Klassenbeziehungen

Assoziationen

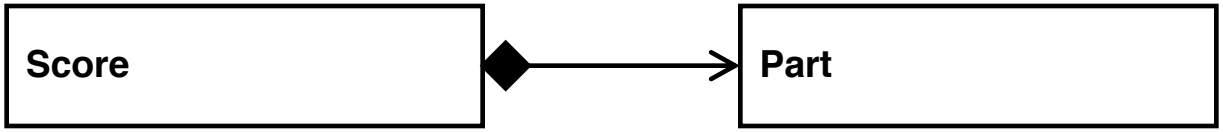
- Klassenbeziehungen werden durch Konnektoren (Verbindungslinien) und Klassendiagramme dargestellt
- Ungerichtete Assoziation: allgemeine Beziehung zwischen Klassen



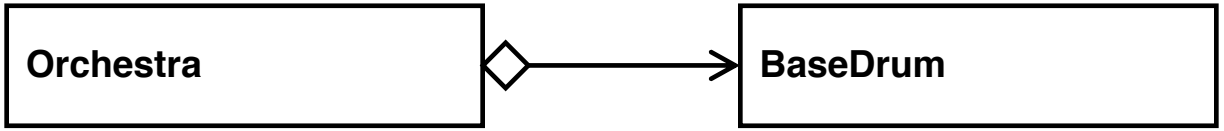
- Gerichtete (direkte) Assoziation: Navigierbarkeit, *has-a* (hat-eine) Beziehung



- Komposition: Containment (Enthalten-Sein), Modellierung von Teile-Ganzes Beziehungen (Teile können nicht ohne Ganzes existieren)

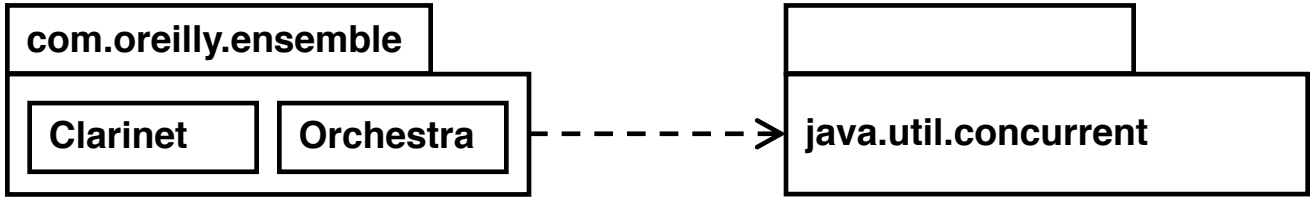


- Aggregation: Modellierung von Ganzes/Teil-Beziehungen
  (Teil steht auch für sich selbst)



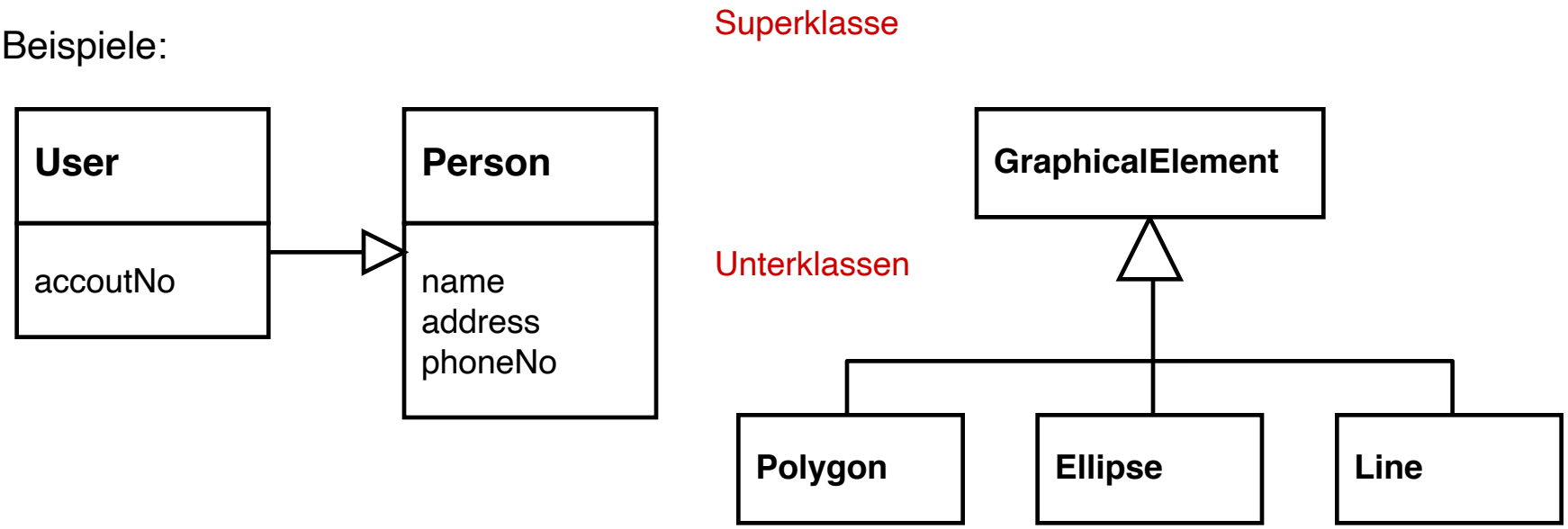
- Temporäre Assoziation (Abhängigkeit): *use-a* (benutzt-eine) Relation; auch für *Package* Abhängigkeiten

Packages werden durch Symbole ähnlich der Dateiordner symbolisiert (Namen werden in den größeren Abschnitt eingetragen, wenn dieser belegt ist, dann in den kleineren Abschnitt)



Generalisierung

- *is-a* Relation: eine speziellere Klasse (Unterklasse) **erbt** die Elemente von einer allgemeineren Klasse (Superklasse; vgl. **Teil XI**)
- Beispiele:



- Unterklassen besitzen alle Attribute, Assoziationen und Operationen ihrer Superklassen
- Ein Objekt einer Unterklasse ist auch Instanz der Superklasse (Polymorphismus)

Sequenzdiagramme

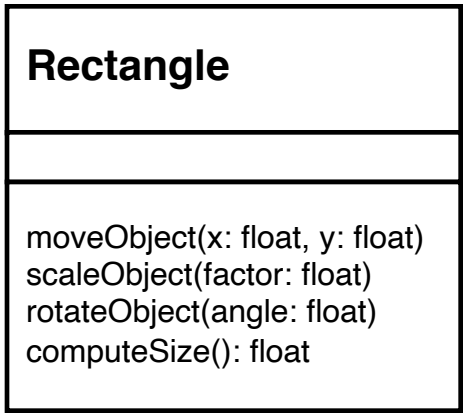
Einordnung

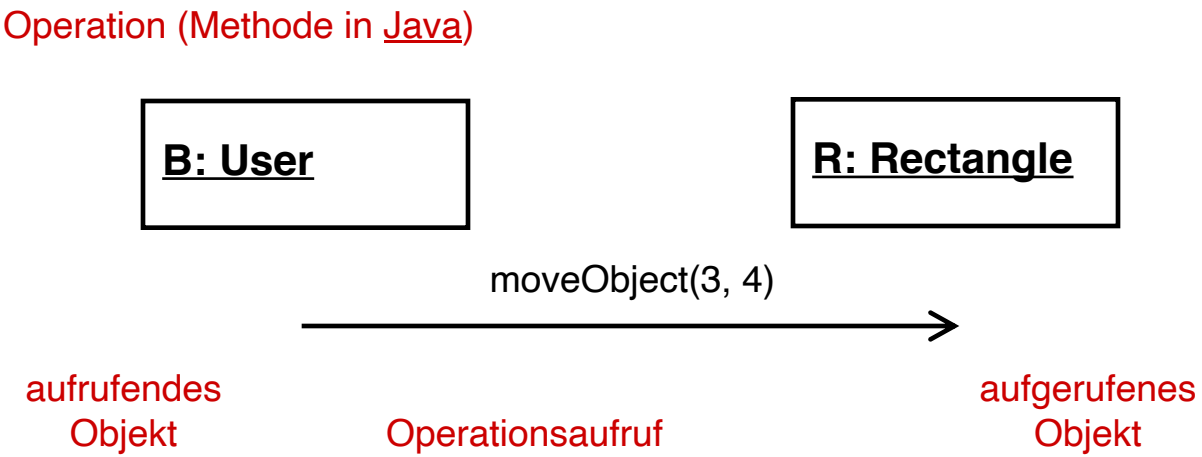
- Nachrichtenaustausch: Objekte kommunizieren miteinander durch Senden und Empfang von Nachrichten (*messages*)
- Ereignis (*event*): Eintreffen einer Nachricht bei einem Objekt
- Ein Ereignis löst Reaktionen bei dem betroffenen Objekt aus
  - Änderung von Attributwerten
  - Senden weiterer Nachrichten
- Nachrichten haben einen Namen und optionale Parameter
- Arten von Nachrichten:
  - Operationsaufrufe: scaleObject(3), size()
  - von Operationen zurück gegebene Werte: return(5)
  - externe Nachrichten: Zeitereignisse, z.B. „es ist Monatsende“
- Formen des Nachrichtenaustauschs: synchron (z.B. Telefongespräch), asynchron (z.B. senden einer e-mail)

Dynamische Interaktionen

Anmerkung: hier können für bestimmte Fragen auch anonyme Objekte verwendet werden, **:<Klasse>**

- Sequenzdiagramme zeigen dynamische Interaktionen zwischen Objekten

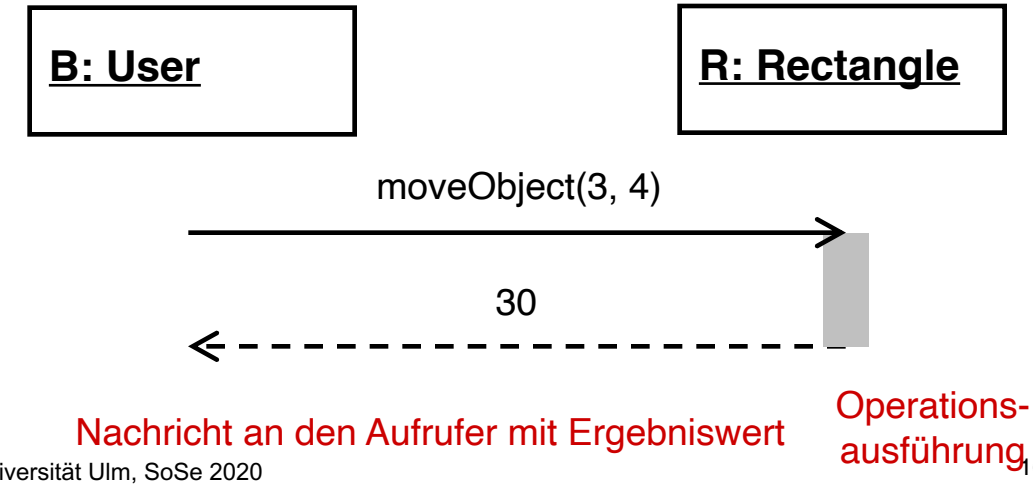




- Operationsausführung und Nachrichten an den Aufrufer

Operationsausführungen bestehen aus einer Folge von Nachrichten

- Operationsaufruf
- zurückgesendete Antwort (optional)
- evtl. andere aufgerufene Operation



2. *Styleguides* und Programmkonventionen

- Allgemeine Bemerkungen zu *Styleguides*
- *Styleguides* und Konventionen für Java-Programme

Allgemeine Namenskonventionen

- Pakete (**`package`**): Bezeichner in Kleinbuchstaben (**1**)

  `mypackage`, `com.company.application.ui`

- Klassen (**`class`**) und neue Datentypen: Bezeichner in Kleinbuchstaben, beginnend mit Großbuchstaben; Elemente zusammengesetzt (**1**)

  `Line`, `String`, `Bicycle[]`, `MyClass`

- Methoden: Name soll eine Aktivität bezeichnen (Verb zu Beginn); in Kleinbuchstaben, in zusammengesetzten Worten wird das jeweils neue Wort mit einem Großbuchstaben begonnen (**1**)

  `getName()`, `computeTotalWidth()`, `printState()`, `readPosInt()`

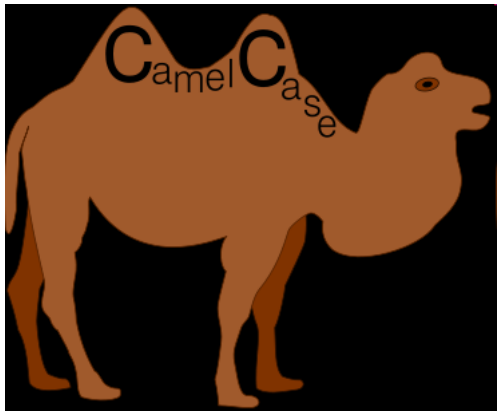
- Variablen: Bezeichner in Kleinbuchstaben, in zusammengesetzten Worten wird das jeweils neue Wort mit einem Großbuchstaben begonnen (**1**)

  `gamesWon`, `otherTrack`, `computersNumber`, `startValue`

- Konstanten (**`final`**): Bezeichner in Großbuchstaben, zusammengesetzte Worte werden durch Unterstrich getrennt (**1**)

  `MAX_GUESSES`, `LEVEL_MAX`, `MAX_ITERATIONS`

Anmerkungen zum Format der Bezeichner



http://en.wikipedia.org/wiki/File:CamelCase.svg

- Die Konvention für die Wahl von Bezeichnern in Java bei **Methoden** und **Variablen** (sowie auch bei **Klassen**) wird als so genanntes *CamelCase* bezeichnet
- Als *CamelCase* wird die Praxis bezeichnet, zusammengesetzte Worte zu schreiben, in denen die Einzelworte jeweils durch einen Großbuchstaben kenntlich gemacht werden

  Anmerkung: Die strikte *CamelCase*-Konvention geht davon aus, dass das erste Zeichen eines Bezeichners grundsätzlich klein geschrieben wird (Klassen in Java würden damit nicht als *CamelCase* bezeichnet – daher der Absatz oben!)

- In **anderen Programmiersprachen** werden andere Konventionen verwendet:
  - *Pascal Case*: Bezeichner konstituierender Wörter werden jeweils groß geschrieben (Beispiele: `UserName`, `DeleteStudentAccount`, ...)
  - Bezeichner konstituierender Wörter werden durch Unterstrich "_" getrennt (Beispiele: `user_name`, `delete_student_account`, ...)
  - Trennung konstituierender Wörter durch Bindestrich "-", z.B. in COBOL, Scheme (LISP) (Beispiele: `USER-NAME`, `fak-iter`, `sum-with`)

Anweisungen

- Typkonvertierungen müssen immer explizit angegeben werden (**1**)

  ```
  doubleValue = (double)intValue
  ```
  (obwohl in dieser Richtung eine implizite Konvertierung durchgeführt wird; vgl. **Teil III**)

- Variablen sollten nur solange erhalten bleiben, wie sie benötigt werden (**2**)

  Der gezielte Einsatz von Blöcken macht die Kontrolle der Auswirkungen einer Variablen und deren mögliche Seiteneffekte einfacher

- Bei Zählschleifen dürfen ausschließlich Steuervariablen in die Kontrollanweisung integriert werden (**1**) – vgl. Motivation der Blockstrukturen und Wiederholungen

  ```
  sum = 0;
  for (i = 0; i < 100; i++) {
    sum = sum + value[i];
  }
  ```

- Zählvariablen einer Wiederholungsschleife sollten direkt vor Beginn der Schleife initialisiert werden (**2**)

  ```
  isDone = false;
  while (!isDone) {
    :
  }
  ```

- Bedingungen in bedingten Anweisungen oder Auswahlanweisungen sollten in separaten Zeilen geschrieben werden (**2**)

```
if (isDone)
  doCleanup();
```

im Gegensatz zu

```
if (isDone) doCleanup();
```

- Variablen können in ihren Deklarationen linksbündig formatiert werden (**3**)

```
TextFile file;
int      nPoints;
double   x,
         y;
```

- Anweisungen sollten bündig formatiert werden, wenn es der Lesbarkeit dient (**2**)

```
value = (potential        * oilDensity)    / constant1 +
        (depth            * waterDensity)  / constant2 +
        (zCoordinateValue * gasDensity)    / constant3;

minPosition     = computeDistance(min,     x, y, z);
averagePosition = computeDistance(average, x, y, z);
```

- Alle Kommentare sollten in Englisch geschrieben werden (**2**)

  Kommentar: in den Programmbeispielen des Skripts wurden die Kommentare und Ausgaben in Deutsch angegeben, um den sprachlichen Einstieg im ersten Semester niedrig zu halten; alle Bezeichner sind schon jetzt konsistent in Englisch

3. Objektorientierte Modellierung und Realisierung – am Beispiel des Spiels 'Game of Life'

- Problemstellung
- Modellierung
- Implementierung

Problemstellung

John Conway
(brit. Mathematiker,
* 1937, † 2020,
Bild: Wikipedia)

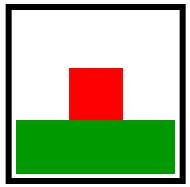
Modellierungsgegenstand

- Basierend auf dem 'Spiel des Lebens' (*Game of Life*) des Mathematikers John Horton Conway simulieren wir die Entstehung und das Sterben von Zellen
- Wir entwickeln und implementieren das Programm zuerst in konventioneller Technik und im Anschluss daran mit einem objektorientierten Ansatz
- Die Spielregeln: (Quelle: http://de.wikipedia.org/wiki/Conways_Spiel_des_Lebens)
  - Die Folgegeneration wird für alle Zellen gleichzeitig berechnet und ersetzt die aktuelle Generation.
  - Der Zustand einer Zelle, lebendig oder tot, in der Folgegeneration hängt nur vom Zustand der acht Nachbarzellen dieser Zelle in der aktuellen Generation ab.
  - Es gelten die folgenden Regeln:

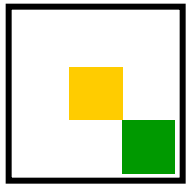
Die Spielregeln:

Die Folgegeneration wird für alle Zellen gleichzeitig berechnet und ersetzt die aktuelle Generation. Der Zustand einer Zelle (lebendig oder tot) in der Folgegeneration hängt nur vom Zustand der acht Nachbarzellen dieser Zelle in der aktuellen Generation ab.

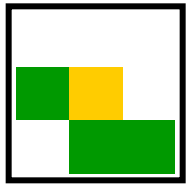
- Eine tote Zelle mit genau drei lebenden Nachbarn wird in der Folgegeneration neu geboren.



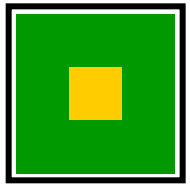
- Lebende Zellen mit weniger als zwei lebenden Nachbarn sterben in der Folgegeneration an Einsamkeit.



- Eine lebende Zelle mit zwei oder drei lebenden Nachbarn bleibt in der Folgegeneration lebend.



- Lebende Zellen mit mehr als drei lebenden Nachbarn sterben in der Folgegeneration an Überbevölkerung.



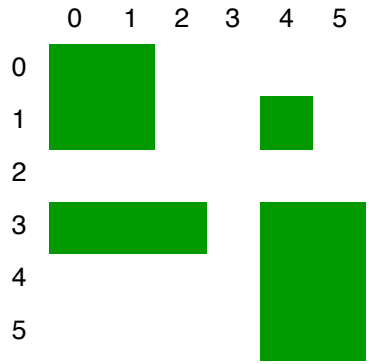
Tote Zelle, die in der nächsten Generation geboren wird

Lebende Nachbarn der Zelle

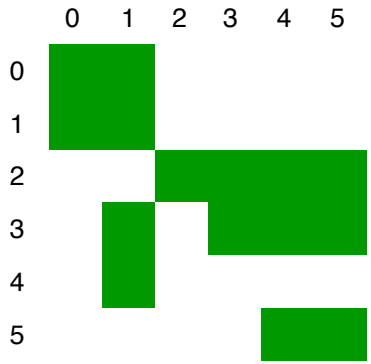
Lebende Zelle, die betrachtet wird

Konventionelle Lösung

- Es bietet sich an, das Spielfeld als 2D-Matrix (`xDim`, `yDim`) vom Typ Boolean zu realisieren, um die beiden Zustände „lebendig“ und „tot“ darstellen zu können
- Da die Nachfolgegeneration für alle Zellen gleichzeitig berechnet wird, benötigen wir die Spielfeld-Matrix zweifach. Es bietet sich an, beide Spielfeld-Matrizen in einer dreidimensionalen **`alive`**-Matrix vom Typ **`boolean [xDim][yDim][2]`** speichern



`alive [][][0]`



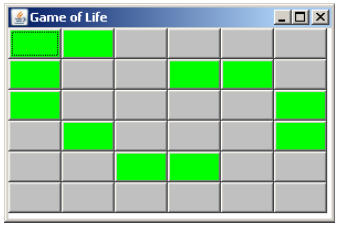
`alive [][][1]`

	0	1	2	3	4	5
0	**3**	**3**	**2**	**1**	**1**	**1**
1	**3**	**3**	**2**	**1**	**0**	**1**
2	**4**	**5**	**3**	**3**	**3**	**3**
3	**1**	**2**	**1**	**3**	**3**	**3**
4	**2**	**3**	**2**	**4**	**5**	**5**
5	**0**	**0**	**0**	**2**	**3**	**3**

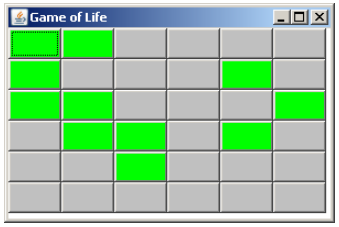
Anzahl lebende Nachbarzellen

Regeln in Kurzform:

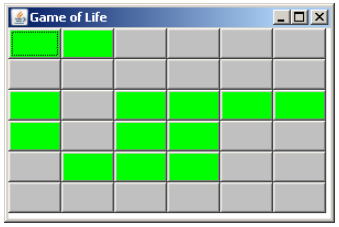
- Eine **lebende Zelle** überlebt, wenn sie 2 oder 3 lebende Nachbarzellen hat, ansonsten stirbt sie
- Eine **tote Zelle** wird neu geboren, wenn sie genau 3 lebende Nachbarzellen hat



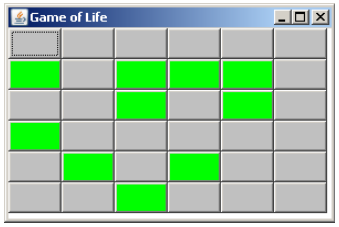
2. Iteration



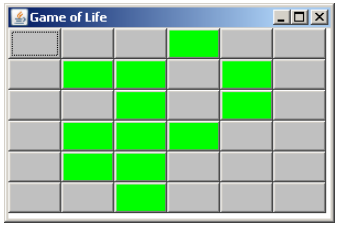
3. Iteration



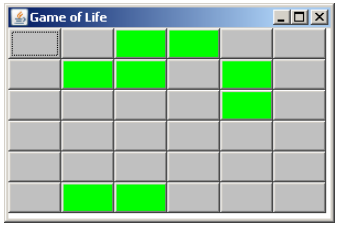
4. Iteration



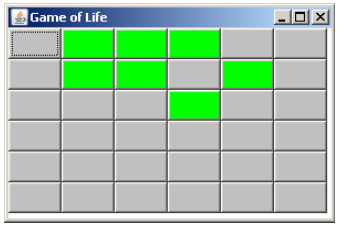
5. Iteration



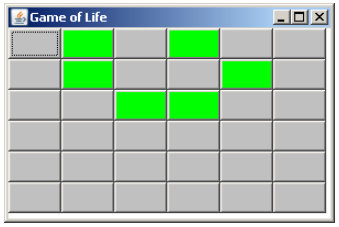
6. Iteration



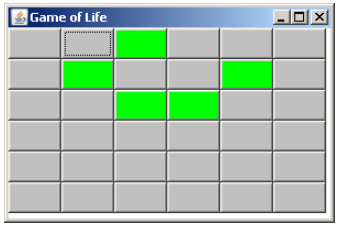
7. Iteration



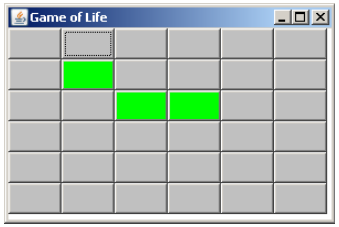
8. Iteration



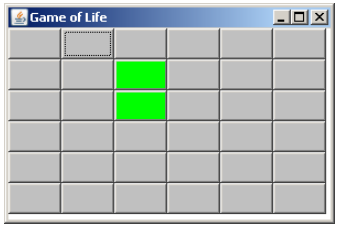
9. Iteration



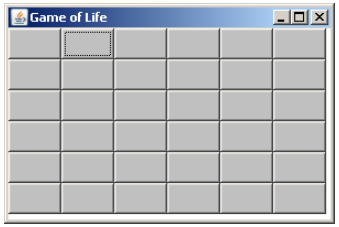
10. Iteration



11. Iteration

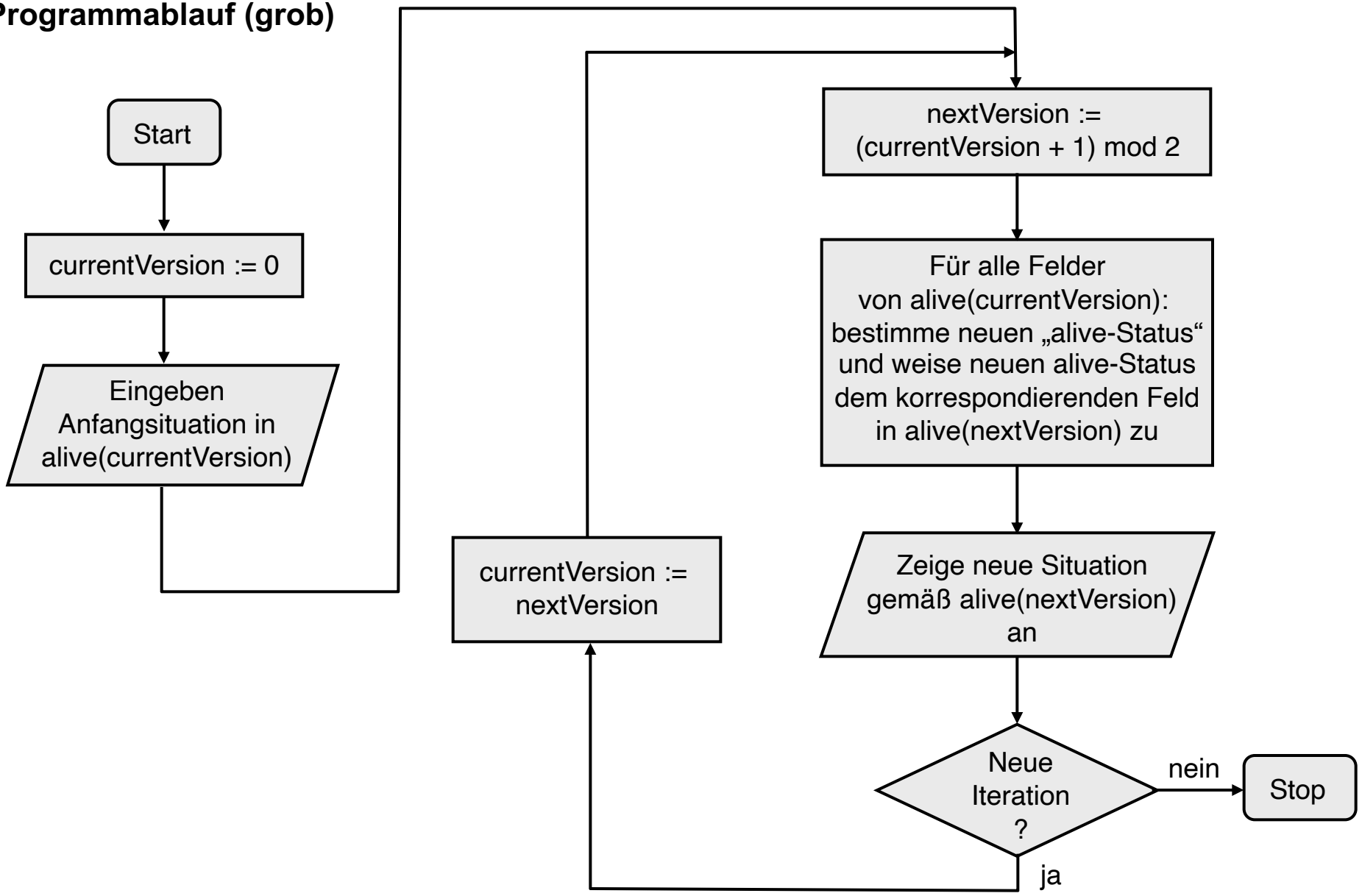


12. Iteration



13. Iteration

Programmablauf (grob)



Konventionelle Lösung (Forts.)

- Wir verwenden zwei `int`-Variablen **`currentVersion`** und **`nextVersion`**, in denen wir jeweils hinterlegen, ob **`alive[][][0]`** oder **`alive[][][1]`** die aktuelle bzw. die nächste Version ist
- Das „Umschalten“ zwischen `currentVersion` und `nextVersion` nach Neuberechnung der nächsten Generation zwischen vollziehen wir mittels:

```
nextVersion = CurrentVersion;
currentVersion = (currentVersion + 1) % 2
```

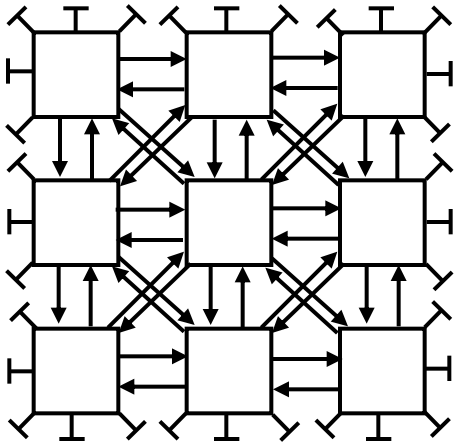
Objektorientierte Lösung

- Vorüberlegungen
  - Was sind geeignete Objekte?
  - Welche Werte/Informationen sollen Objekte speichern? (⇒ „Instanzvariablen“)
  - Welches Verhalten sollen Objekte aufweisen? (⇒ „Objektmethoden“)
  - Welches Objekt benötigt von welchem anderen Objekt welche Information?

- Anmerkungen
  - Meist gibt es zu einer Problemstellung verschiedene Alternativen für die Realisierung; das gilt auch für den objektorientierten Ansatz
    - Was macht man zu „Objekten“?
    - Welche Werte speichern die Objekte / welche Methoden definiert man?
    - Lässt man Zugriff auf Instanzvariablen zu oder nur über Methoden?
  - Es bietet sich daher an, verschiedene Alternativen in Betracht zu ziehen und jeweils bis zu einer gewissen Tiefe gedanklich durchzuspielen

Entwurfsüberlegungen für „Game of Life“

- Verzicht auf globale „alive“-Matrix möglich und sinnvoll?
- Können Entscheidungen, ob Zelle überlebt, stirbt oder geboren wird (sinnvoll) auf Zellebene getroffen werden?
- **Überlegung: Zelle als Objekttyp?**
  - Zelle könnte selbst speichern und Auskunft geben, ob sie lebt oder ob sie tot ist
  - Zelle könnte speichern, wie ihr Zustand in der nächsten Generation sein wird
  - Zelle könnte mit ihren Nachbarzellen kommunizieren und damit ihren Folgezustand dann sogar selbst bestimmen
  - Aber: Wie „weiß“ eine Zelle, welches ihre Nachbarzellen sind?
  - **Überlegung 1: Können wir die Zelle irgendwie mit ihren Nachbarzellen „Verzeigern“**



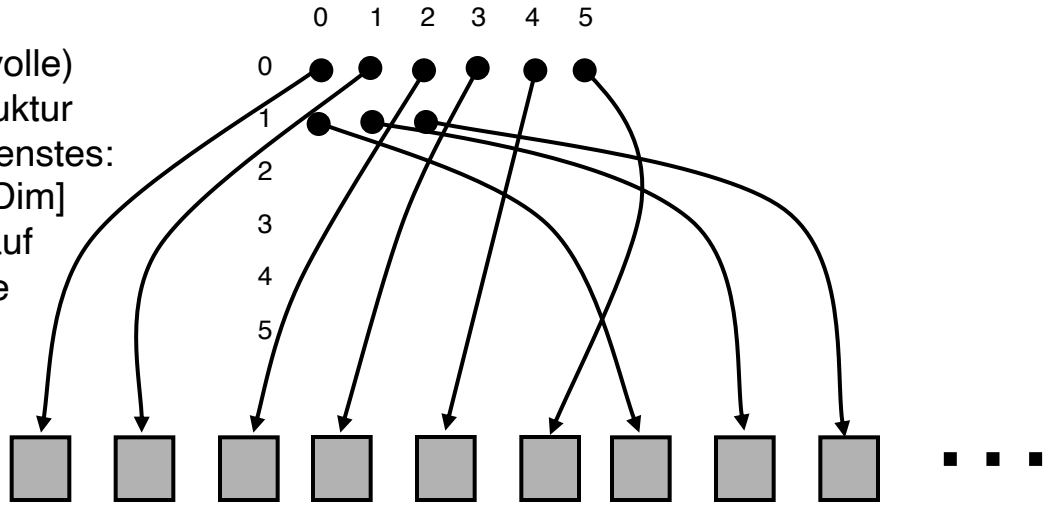
Bewertung:

- **Ist technisch im Prinzip machbar**
  (ist auch nur ein Graph)
- **Ist aber ziemlich aufwendig**

- **Überlegung: Zelle als Objekttyp? (Forts.)**
  - **Überlegung 2: Könnte jemand anders über die Nachbarschaftsbeziehungen „Buch führen“ und einer Zelle bei Bedarf Auskunft geben?**

- **Überlegung: Zell-Verzeichnisdienst als Objekttyp?**
  - Dieser „Dienst“ könnte die Referenzen auf die Zell-Objekte in einer Matrix-Struktur speichern und auf Nachfrage die Referenzen auf die Nachbarzellen zurückgeben
  - Außerdem könnte er die Zellen auffordern, die nächste Generation zu erzeugen

Mögliche (sinnvolle) Verwaltungsstruktur des Verzeichnisdienstes: Matrix[xDim][yDim] mit Zeigern auf Zell-Objekte



- **Überlegung: Graphische Darstellung als Objekttyp kapseln?**
  - Es soll „irgendwie“ ein Spielfeld mit den vorgegebenen Dimensionen in Matrix-Form erzeugt werden (die Details der Realisierung sind für das Hauptprogramm nicht wichtig)
  - Jede „Zelle“ im graphischen Spielfeld ist eine graphische Darstellung des Zustands (lebendig oder tot) eines bestimmten Zell-Objekts
  - Da die interne Realisierung verborgen bleiben soll, müssen „nach außen“ Methoden bereitgestellt werden, um den Zustand einer „Spielfeld-Zelle“ auf „lebendig“ oder auf „tot“setzen zu können
  - **Überlegung: Wie stellen wir den Bezug zwischen „Spielfeld-Zelle“ und „Zell-Objekt“ her?**
    - ➢ Möglichkeit 1:
      - Wir könnten die „Spielfeld-Zellen“ auch als Objekte realisieren, dann könnte jedes Zellobjekt den Verweis auf „sein“ „Spielfeld-Zellen-Objekt“ speichern
      - In diesem Fall müssen wir uns überlegen, wie die Zelle an die Objektreferenz „ihrer“ Spielfeld-Zelle kommt
    - ➢ Möglichkeit 2:
      - Die Zell-Objekte haben sich bei ihrer Erzeugung gemerkt, welche xy-Koordinaten sie im Zell-Verzeichnis haben
      - Dies sind dieselben Koordinaten wie im graphischen Spielfeld
      - Damit können die Zellen für die „Spielfeld-Methoden“ zum Setzen des Zustandes sehr einfach den Bezug zu den Spielfeld-Zellen herstellen

- **Entscheidung für Realisierung von „Game of Life“**
  - **Objekttypen**
    - **Cell** (*Zelle*)
    - **CellDirectory** (*Zellverzeichnis*)
    - **GameOfLifeGrid** (*Graphische Darstellung des Spielfeldes)*
  - **Entwurfsentscheidung**
    - Wir realisieren alle Objekttypen „streng gekapselt“,
    - d.h. es gibt keinen direkten Zugriff „von außen“ auf Instanzvariablen von Objekten
    - alle Zugriffe werden über Methoden realisiert („gekapselt“)

- **Cell**
  - **Instanzvariablen (alle „private“)**
    - `int x` `// x-Koordinate`
    - `int y` `// y-Koordinate`
    - `boolean[] aliveStatus` `// lebendig oder tot`
      `// aktuelle/nächste Generation`
  - **„Public“ Methoden**
    - `Cell(int x, int y)` `// Konstruktor`
    - `int getXcoordinate()`
    - `int getYcoordinate()`
    - `boolean alive(int version)`
    - `void setStatus(int version, boolean status)`
    - `validateAndSetNextState(int currentVersion, CellDirectory cd, GameOfLifeGrid g)`

- **CellDirectory**
  - **Instanzvariablen (alle „private“)**
    - `Cell[][] cellMatrix`
    - `int xDim    // Matrix-Dimension x-Richtung`
    - `int yDim    // Matrix-Dimension y-Richtung`
  - **„Public“ Methoden**
    - `CellDirectory(int xDim, int yDim)  // Konstruktor`
    - `Cell[] getNeighbourCells(Cell c)  // gibt Array von`
      `// Zellen zurück`
    - `Cell cellObj(int x, int y)  // liefert Cell-`
      `// Objekt an xy-Pos.`
    - `void generateNextVersion( int currentVersion,`
      `GameOfLifeGrid g)`

- **GameOfLifeGrid**
  - **Instanzvariablen (alle „private“)**
    - `Button[][] buttonArray`
  - **„Public“ Methoden**
    - `GameOfLifeGrid(String Title, // Konstruktor`
      `int xDim,`
      `int yDim)`
    - `void setStatus(int x, int y, boolean aliveStatus)`
  - **Anmerkung**
    - Das graphische Spielfeld gibt es nur in einer Version
    - Es wird stets der aktuelle Zustand angezeigt

- **Klassendiagramme in UML**

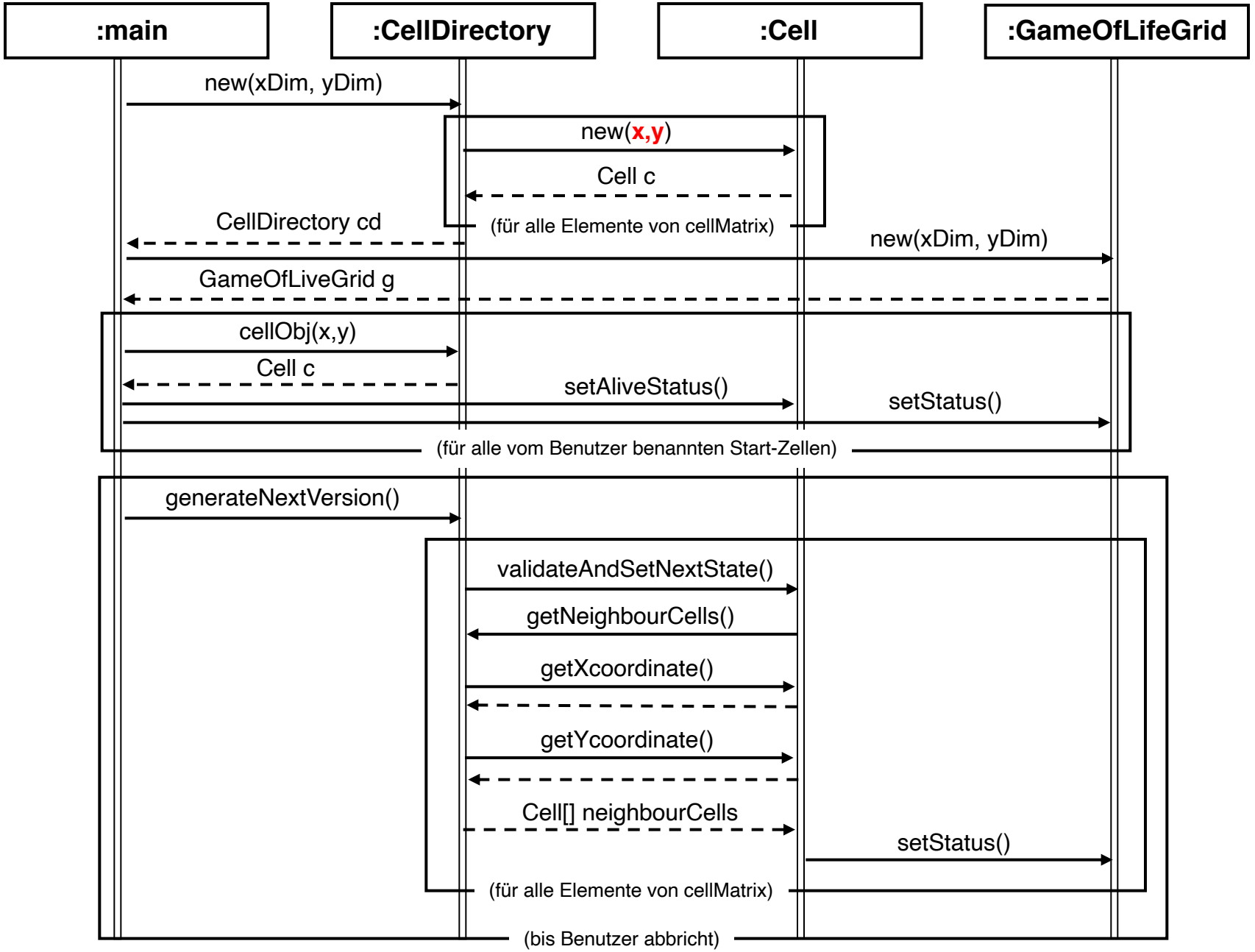
CellDirectory
- Cell [][] cellMatrix - int xDim - int yDim
+ Cell[] getNeighbourCells(Cell c) + Cell cellObj(int x, int y) + generateNextVersion(int currentVersion, GameOfLifeGrid g)

Cell
- int x - int y - boolean [] alive
+ int getXcoordinate() + int getYcoordinate() + boolean getAliveStatus(int version) + setAliveStatus(int version, boolean status) + void validateAndSetNextState(int currentVersion, CellDirectory cd, GameOfLifeGrid g)

GameOfLifeGrid
- Button [][] buttonArray
+ setStatus(int x, int y, boolean aliveStatus)

(„Hilfs-Klasse“ ScannerInputMethoden hier weggelassen – ebenso die „Hilfsmethoden“ der Klassen)

Sequenzdiagramm in UML



Ende der Vorlesung …

… viel Erfolg und vor allem Spaß
bei den Prüfungen und
im weiteren Studium!