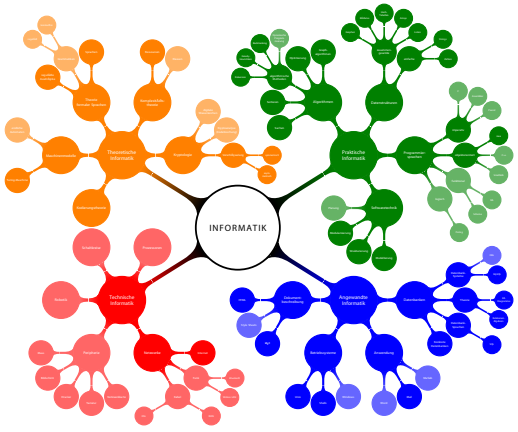
UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Kapitel 21

Modularisierung im Großen

Wie man große Probleme in genießbare Häppchen aufteilt

Vorlesung Einführung in die Informatik 1 vom 16. Januar 2014 von Till Tantau



Lernziele von Kapitel 21

1. Das Konzept des Top-Down-Entwurf verstehen und anwenden können
2. Das Konzept der Klassen- und Modulbildung verstehen

Gliederung von Kapitel 21

Wie baut man ein Haus?

Soll ein Haus gebaut werden, so hängt es von der Größe ab, wie viele Personen beteiligt sind:

Hundehütte	Zimmerer
Schuppen	Mutter und Tochter
Einfamilienhaus	Architekt, Maurerin, Elektriker, Zimmerer, Klempnerin, Fahrerin, Gehilfen, usw.
Hochhaus	Architekturbüro, Bauaufsicht, Bauingenieure, Statikerin, Maurer, Elektrikerin, Zimmerer, Klempnerin, Fahrer, viele Gehilfen, usw.

Wie baut man Software?

Bei Software hängt es ebenfalls von der Größe ab, wie viele Personen beteiligt sind:

Skalarprodukt	MLS-Student
Vier-Gewinnt	Programmiererin
Mail-Programm	Software-Architekt, Oberflächen-Designerin, Programmierer, Gehilfen, Pizza-Bote, usw.
Betriebssystem	(Software)-Architekten, Projektmanagerin, Oberflächen-Designer, Kernel-Programmiererin, Treiber-Programmierer, Oberflächen-Programmiererin, Gehilfen, viele Pizza-Boten, usw.

Probleme, die bei großer Software entstehen.

- Arbeiten an einem Softwareprojekt viele Beteiligte, so müssen sich diese *koordinieren*.
- Damit Beteiligte unabhängig arbeiten können, muss das Projekt in verschiedene *Module* aufgespalten werden.
- Programmiersprachen sollten diese Modularisierung *unterstützen*.

Ziel: Simulation des Immunsystems.

Eine komplexe Problemstellung

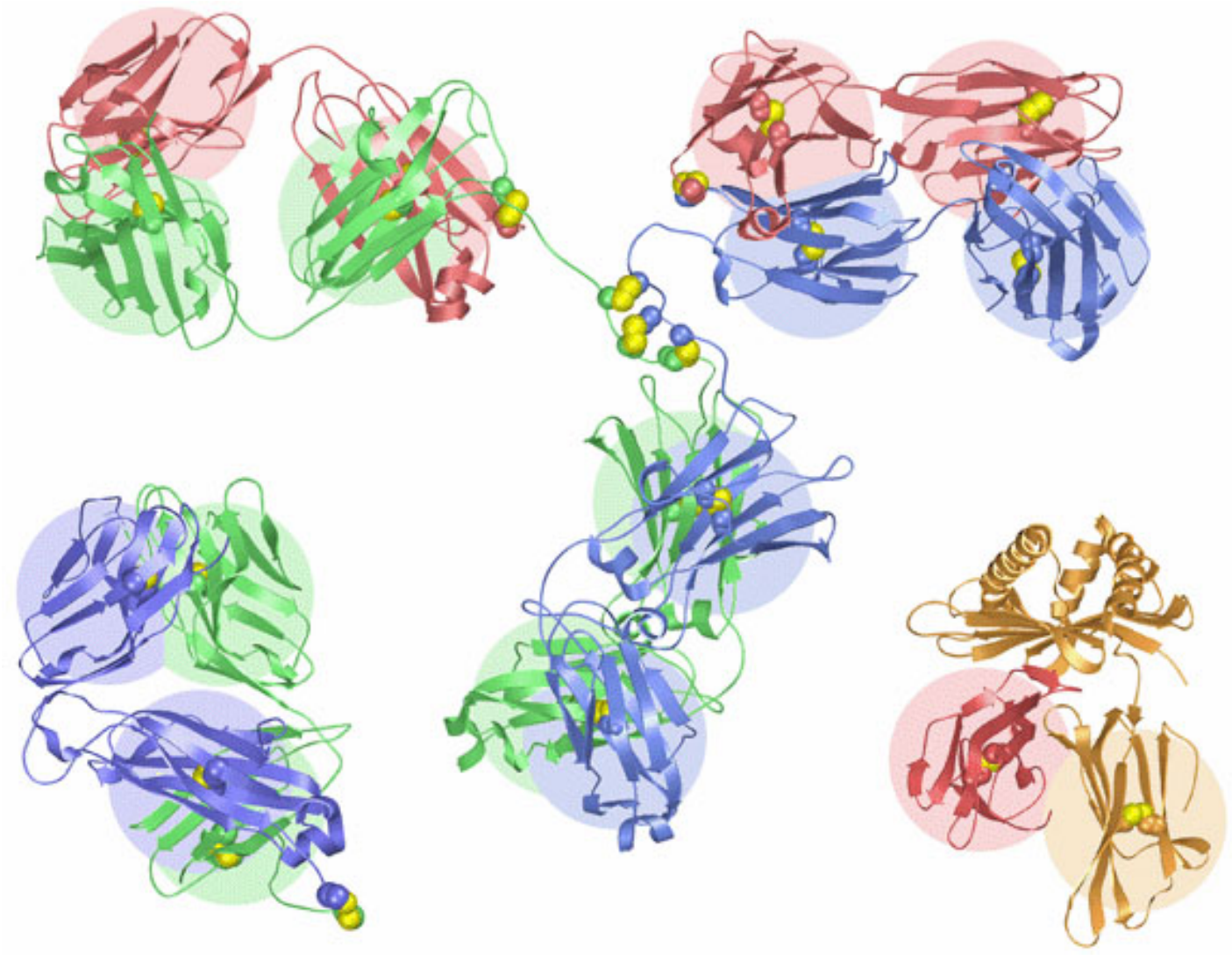
Mittels eines Computerprogramms soll *das Immunsystem simuliert werden*.

P. E. Seiden, F. Celada.
A model for simulating cognate recognition and response in the immune system.
Journal of Theoretical Biology, 158(3):329-57, 1992.

> *We have constructed a model of the immune system that focuses on the clonotypic cell types and their interactions with other cells, and with antigens and antibodies. [...] We propose using computer simulation as a tool for doing experiments in machine, in the computer, as an adjunct to the usual in vivo and in vitro techniques. [...] a model simulating areas of interest could be used for extensively testing ideas to help in the design of the critical biological experiments. [...]*

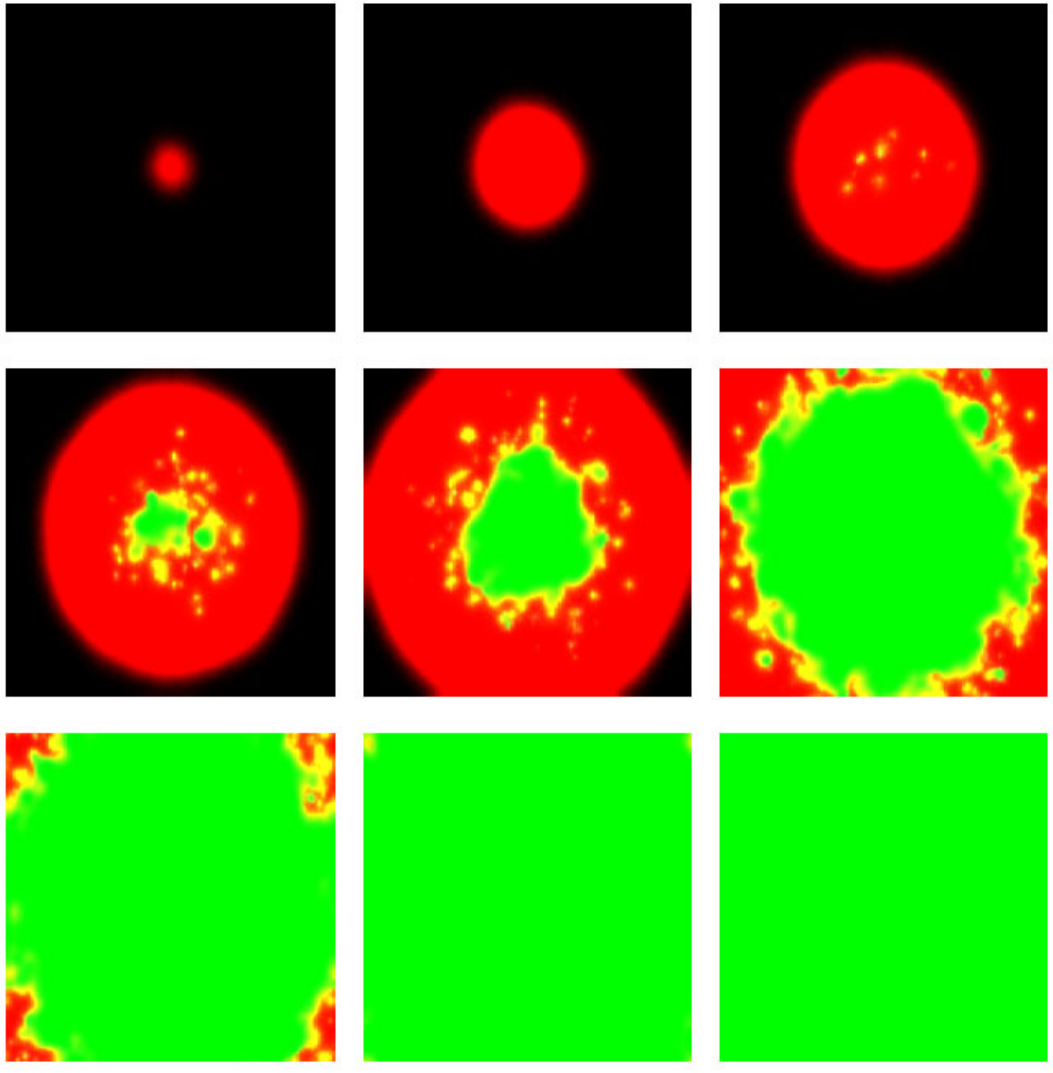
Einige interessante Teile des Immunsystems

T-Zellen Rezeptor, Antikörper, Major Histocompatibility Complex



Author David Goodsell, Scripps Research Institute, public domain

Ergebnis einer möglichen Simulation.



rot = Antigene, grün = Antikörper

Wiederholung:
Methoden erlauben die Strukturierung von Programmen.

- Programme werden in kleiner Unterprogramme, in Java *Methoden* genannt, aufgeteilt.
- Methoden sind in der Regel zwischen einer und etwa 30 Zeilen lang.
- Längere Methoden sollten in kleine Methoden aufgespalten werden.

Die Modularisierung mittels Methoden löst nicht alle Probleme.

- Wenn ein Programm aus 10.000.000 Zeilen Code besteht, dann braucht man grob *eine Million Methoden*.
- Offenbar muss hier *weiter modularisiert* werden.
- Dazu werden Methoden wiederum *hierarchisch* zusammengefasst.

Die Modularisierungshierarchie.

Instruktionsfolgen werden zu immer größeren Einheiten zusammengefasst:

1. *Blöcke* (Scopes) sammeln Anweisungen.
2. *Methoden* sammeln Blöcke.
3. *Klassen* sammeln Methoden.
4. *Pakete* sammeln Klassen.
5. *Pakete* sammeln Pakete.

Was ist Softwaretechnik?

- Bei einem umfangreichen Simulationsprogramm sollte man *nicht* »drauflosprogrammieren«.
- Vielmehr muss das Programm *geplant* werden.
- Einer der ersten Schritte ist die *Modularisierung*.
- Das Teilgebiet der Informatik, das sich mit der Planung und Programmierung großer Programme beschäftigt, heißt *Softwaretechnik*.

Die zwei Richtungen beim Softwareentwurf.

- Das *Bottom-Up-Prinzip*:
  - Bei der Planung stellt man fest, dass eine kleine Teilaufgabe oder -aspekt wiederholt vorkommt.
    Beispiel: Es müssen immer wieder Zellen bewegt werden.
    Beispiel: Es müssen immer wieder Reaktionen berechnet werden.
  - Teilaufgaben werden dann in *Methoden ausgelagert* oder in *Klassen gekapselt*.

  Mehr dazu im nächsten Kapitel.
- Das *Top-Down-Prinzip*:
  - Das Problem wird in große Teilbereiche aufgeteilt.
    Beispiel: Teilbereiche sind immunologische Modelle, Simulationsalgorithmen, graphische Darstellung, Datenkonvertierung usw.
  - Jeder Teilbereich wird seinerseits in kleinere Teilprobleme aufgeteilt; und so fort.
    Beispiel: Teilprobleme sind die Simulation einer T-Zelle, Simulation einer B-Zelle usw.

  Mehr dazu im Folgenden.

Was sind Klassen?

- *Klassen* sind ein Konzept in der objektorientierten Programmierung.
- Sie dienen zur *Modellierung*, da man mit ihnen neue *Datenstrukturen* erzeugt.
- Gleichzeitig dienen sie auch zur *Modularisierung*:
  1. Man identifiziert Teilbereiche des Problems und führt hierzu Klassen ein.
  2. Man implementiert Methoden, die für die Klassen nützlich sind.

Beispiel (Top-Down Modularisierung mit Klassen)

- Wir benötigen Methoden zum Lesen und Schreiben von Dateien. Weiter benötigen wir Methoden, die Reaktionen modellieren.
- Deshalb führen wir zwei Klassen `FileManagement` und `Reactions` ein und
- überlegen, welche konkreten Methoden nützlich wären.

Wie schreibt man die Klassen nun auf?

```
// In der Datei FileManagement.java
class FileManagement
{
  static void writeToFile(String s, String filename)
  {
    /* Code einer Methode, die den String s in die
    Datei filename schreibt. */
  }
  static String readFromFile(String filename)
  {
    /* Code einer Methode, die den Inhalt der Datei
    filename liest zurückgibt. */
  }
}

// In der Datei Reactions.java
class Reactions
{
  static double reactionLikelihood(String a, String b)
  {
    /* Diese Methode liefert die Wahrscheinlichkeit
    zurück, dass die durch die Strings a und b
    repräsentierten Oberflächen interagieren */
  }
}
```

Die Syntax von Klassen allgemein.

- Eine Klasse wird mittels `class` eingeleitet, gefolgt von ihrem Namen.
- *Der Name* ist wie immer ein Java-Bezeichner.
- Im Inneren der Klasse finden sich die *Methoden* der Klasse.

Eine Klasse darf auch Sub-Klassen haben; das brauchen wir aber nicht.

Wie benutzt man die Klassen nun?

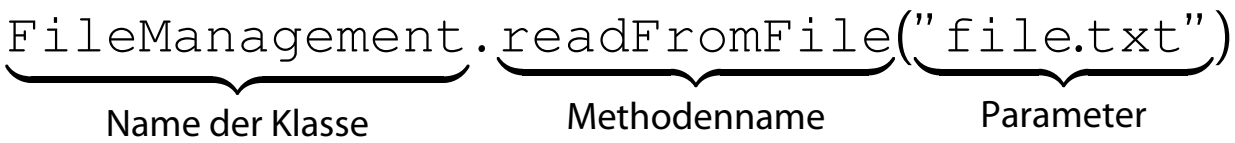
Jede Klasse ist ein eigener Scope, weshalb Methoden gleichen Namens in unterschiedlichen Klassen sich nicht stören.

Problem

Aber wie kann man dann Methoden benutzen, wenn sie doch außerhalb des Scopes nicht sichtbar sind?

Lösung

Es gibt eine spezielle Notation, mit der man *Scopes wieder öffnen kann* und dadurch Methoden *in anderen Scopes aufrufen* kann:



Eine Beispielbenutzung der Methoden aus `FileManagement`


```
// In der Datei FileManagement.java
class FileManagement {
  static void writeToFile(String s, String filename) {...}
  static String readFromFile(String filename)        {...}
}

// In der Datei Copy.java
class Copy
{
  public static void main (String[] args)
  {
    String file_contents =
      FileManagement.readFromFile("original.txt");
    FileManagement.writeToFile
      (file_contents, "copy␣of␣original.txt");
  }
}
```

5-Minuten-Aufgabe

Zur Übung

Die Klasse `Math` aus der System-Bibliothek stellt eine Funktion `exp` zur Verfügung zur Berechnung von e^x. Geben Sie an, wie die Deklaration von `exp` innerhalb von `Math` aussieht.

Das Prinzip der Kapselung.

- Methoden können *privat* oder *öffentlich* sein.
- Öffentliche Methoden kann man von außen mittels `SomeClass.method()` aufrufen.
  Um anzuzeigen, dass eine Methode öffentlich ist, stellt man ihr das Attribut `public` vor.
- Private Methoden kann man von außen nicht aufrufen.
  Um anzuzeigen, dass eine Methode privat ist, stellt man ihr das Attribut `private` vor.

Zur Diskussion

Welche Vorteile hat man durch die Unterscheidung in öffentliche und private Methoden?

5-Minuten-Aufgabe

Zur Übung

Nun soll eine Methode `vectorLength` in einer neuen Klasse `MathInThePlane` implementiert werden.
Es soll die Wurzelmethode `sqrt` aus der Klasse `Math` benutzt werden.
Wie lautet der Code (Parameter: Koordinaten eines Vektors)?

Die breiteste Hierarchieebene in Java: Pakete

- Java erlaubt es, mehrere Klassen zu einem *Paket* (`package`) zusammenzufassen.
- Mehrere Pakete können wiederum zu einem Paket zusammengefasst werden und so fort.
- Pakete können nur Klassen und Pakete enthalten, keine Methoden.

Java-Syntax für Pakete

Am Anfang einer Java-Programm-Datei kann man folgendes schreiben (man beachten: keine geschweiften Klammern):

```
package name_of_package;
```

Dies hat folgende Effekte:

- Alles, was nun folgt, wird dem Paket `name_of_package` hinzugefügt.
- Eine zweite Datei könnte ebenfalls so anfangen. Ihr Inhalt würde ebenfalls dem Paket hinzugeschlagen.
- Im *Gegensatz dazu* kann man einer *Klasse nicht* in einer zweiten Datei noch Methoden hinzufügen.

Das Paket `immuneSystemSimulation`.


```java
// In der Datei immuneSystemSimulation/FileManagement.java
package immuneSystemSimulation;

class FileManagement {
  static void writeToFile(String s, String filename) {...}
  static String readFromFile(String filename)        {...}
}

// In der Datei immuneSystemSimulation/Reactions.java
package immuneSystemSimulation;

class Reactions {
  public static double reactionLikelihood
    (String a, String b)
  {...}
}
```

Wie benutzt man die Pakete nun?

- Genau wie eine Klasse ist ein Paket ein eigener Scope.
- Deshalb hat man das gleiche Problem wie bei Klassen mit dem Zugriff,
- was aber auch genauso gelöst wird:
  - Möchte man auf `xyz` im Paket `somePackage` zugreifen, so schreibt man einfach *`somePackage.xyz`*.
  - Jede Klasse kann auch noch `public` oder `private` sein.

```
class Copy
{
  public static void main (String[] args)
  {
    String file_contents =
      immuneSystemSimulation.FileManagement.readFromFile
        ("original.txt");
    immuneSystemSimulation.FileManagement.writeToFile
      (file_contents, "copy␣of␣original.txt");
  }
}
```

Vereinfachung des Lebens mittels der Import-Anweisung.

Oft benutzte Klassen können »importiert« werden.

```
import immuneSystemSimulation.*;

class Copy
{
  public static void main (String[] args)
  {
    String file_contents =
      FileManagement.readFromFile("original.txt");
    FileManagement.writeToFile
      (file_contents, "copy␣of␣original.txt");
  }
}
```

Zusammenfassung

Hierarchieebenen in Java

1. *Blöcke* (Scopes) sammeln Anweisungen.
2. *Methoden* sammeln Blöcke.
3. *Klassen* sammeln Methoden.
4. *Pakete* sammeln Klassen.
5. *Pakete* sammeln Pakete.

Das »Öffnen benannter Scopes«

- Manche Scopes (Klassen, Pakete) haben einen Namen und lassen sich *von außen öffnen.*
- Die Syntax ist dabei immer `scope_name.ding_im_scope`.
- Dies ist allerdings nur möglich, wenn das »Ding im Scope« als **`public`** deklariert wurde.

Zusammenfassung

Syntax von Klassen

```
// Datei muss heißen wie die Klasse, also Example.java
// Klassennamen fangen mit Großbuchstaben an

class Example
{
  ... // Die Methoden
}
```

Syntax von Paketen

```
// Verzeichnis muss heißen wie das Paket, also myPackage
// und darin dann myPackage/Example.java
// Paketnamen fangen mit Kleinbuchstaben an

package myPackage;

class Example
{
  ... // Die Methoden
}
```