Java Programming
Module
Object-oriented Design

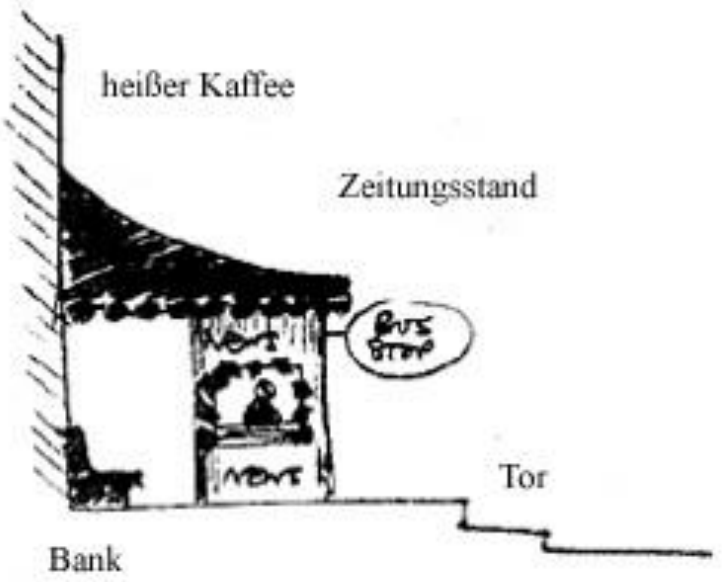
DHBW Mannheim
Duale Hochschule
Baden-Württemberg

Prof. Dr. Holger D. Hofmann

Design Patterns

- Developed by the architect Christopher Alexander during the 70s
- Goal: Creation of Design Components in Architecture
- Each Design Component represents a vocabulary of a Design Language

- Example Bus Stop
  - Food Stands
  - Path Shape
  - A Place To Wait
  - Main Gateway
  - Public Outdoor Room
  - Seat Spots



Design Patterns in Computer Science

Erich Gamma et al: Design Patterns: Elements of Reusable Object-Oriented Software, 1995

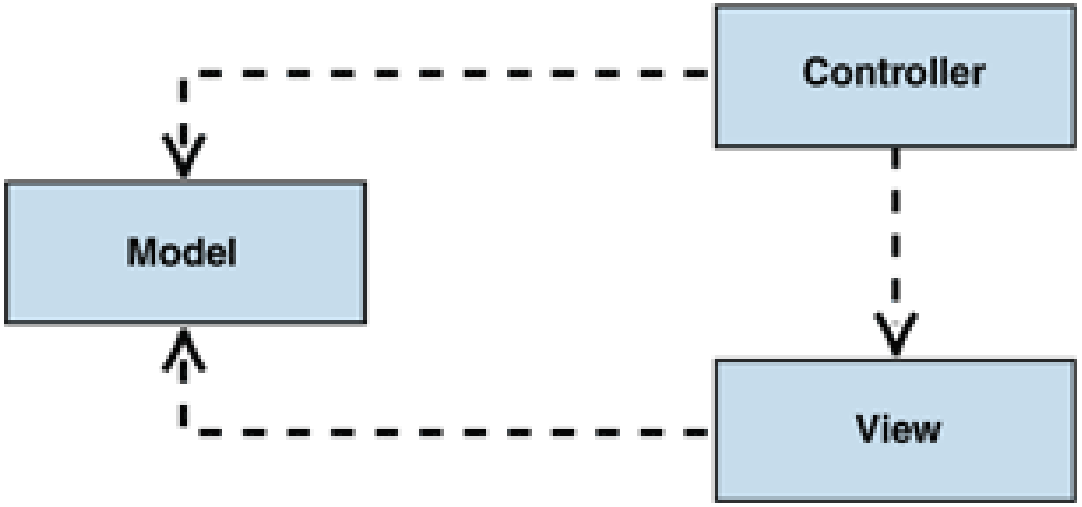
- Structural patterns
  - **Adapter** (wraps interfaces to provide class collaboration)
  - **Bridge** (decouples interface from implementation to enable evolution)
  - **Composite** (Aggregates multiple objects so that they can be managed together)
  - **Decorator** (Changes/Adds Functionality to existing Methods)
  - **Facade** (Simple Interface to complex Sub-System)
  - **Flyweight** (Creation of huge numbers of similar objects)
  - **Proxy** (Object Placeholder)
- Creational patterns
  - **Abstract Factory** (Grouping of Object Factories)
  - **Builder** (Construction of complex objects by separating construction and representation)
  - **Factory Method** (Object Creation)
  - **Prototype** (Creation of Objects by Cloning)
  - **Singleton** (Creation of only one Instance)

Design Patterns in Computer Science

- Behavioral patterns
  - **Chain of responsibility** (Delegation of Events)
  - **Command** (Object representing actions and associated parameters)
  - **Interpreter** (Interpretion of a Programming Language)
  - **Iterator** (Sequential Access to multiple Elements)
  - **Mediator** (Louse Coupling between Objects by hiding Implementation Details
  - **Memento** (Undo for Objects)
  - **Observer** (Publish/Subscribe Event Mechanism)
  - **State** (Change of Object Behavior based on Object State)
  - **Strategy** (Selection of algorithm on-the-fly)
  - **Template Method** (Provision of Algorithm Skeleton by Abstract Base Classes)
  - **Visitor** (Separation of Algorithm from Object Structure)

Design Pattern "Model-View-Controller"

- Achieves De-Coupling of data management, program logic, and presentation
- Model manages Data
- Controller works and Requests and Data
- View represents Data and provides User Interaction



Design by Contract

- A concept to improve software quality
- Term DBC created by Bertrand Meyer in connection with his programming lanuage Eiffel (www.eiffel.com), Bertrand Meyer: *Object-Oriented Software Construction*, Prentice Hall, 1988
- DBC is based on the concept that within the code, there are conditions which have to be met
  - For clients using objects
  - For objects during their lifetime
  - For loops being executed
  - For arbitraty code sections

Design by Contract

- Preconditions
  - Conditions which have to be met before code execution
  - E.g., before filling a tank, it should not be full
- Postcondition
  - Conditions which have to be met after code execution
  - After adding an element to an array, the array should have <number_of_elements_before> + 1 elements
- Class/Object Invariants
  - Conditions which have to be met during object lifetime
  - E.g., the number of elements on a stack should be greater than 0 and smaller than <max_capacity>
- Loop Invariants
  - Conditions which have to be met during loop execution

DBC in Java

- Can be realised using DBC frameworks
- Can be realised by Assertions (> JDK 1.4)
  - Syntax:
    - **`assert expression [ : expression ];`**
    - If assertion is evaluation to true, nothing happens
    - If assertion is evaluated to false, an exception is raised
  - Advantages of assert in contrast to, e.g., if blocks:
    - Shorter code
    - Differentiation between program logic and checks for correctness
  - Assertions can be enabled or disabled during runtime (no overhead when disabled)

```
java [ -enableassertions | -ea  ] [:PackageName... | :
  ClassName ]
java [ -disableassertions | -da  ] [:Package... | :ClassName ]
```

Application of Assertions: SimpleIntList.java

```
001 public class SimpleIntList
002 {
003   private int[] data;
004   private int   len;
005
006   public SimpleIntList(int size)
007   {
008     this.data = new int[size];
009     this.len  = 0;
010   }
011
012   public void add(int value)
013   {
014     //Precondition als RuntimeException
015     if (full()) {
016       throw new RuntimeException("Liste voll");
017     }
018     //Implementierung
019     data[len++] = value;
020     //Postcondition
021     assert !empty();
022   }
```

Application of Assertions: SimpleIntList.java

```
023
024   public void bubblesort()
025   {
026     if (!empty()) {
027       int cnt = 0;
028       while (true) {
029         //Schleifeninvariante
030         assert cnt++ < len: "Zu viele Iterationen";
031         //Implementierung...
032         boolean sorted = true;
033         for (int i = 1; i < len; ++i) {
034           if (sortTwoElements(i - 1, i)) {
035             sorted = false;
036           }
037         }
038         if (sorted) {
039           break;
040         }
041       }
042     }
043   }
044
```

Application of Assertions: SimpleIntList.java

```
045   public boolean empty()
046   {
047     return len <= 0;
048   }
049
050   public boolean full()
051   {
052     return len >= data.length;
053   }  // 054 empty line
055   private boolean sortTwoElements(int pos1, int pos2)
056   {
057     //Private Preconditions
058     assert (pos1 >= 0 && pos1 < len);
059     assert (pos2 >= 0 && pos2 < len);
060     //Implementierung...
061     boolean ret = false;
062     if (data[pos1] > data[pos2]) {
063       int tmp = data[pos1];
064       data[pos1] = data[pos2];
065       data[pos2] = tmp;
066       ret = true;
067     }
068     //Postcondition
069     assert data[pos1] <= data[pos2] : "Sortierfehler";
070     return ret;
071   }
072 }
```