Software Engineering

Introduction to Design Patterns
A pattern describes...

- a problem which occurs over and over again in our environment,
- the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

(Christopher Alexander)
On Patterns...

- Patterns are proven
- Proven software practice
- Piece of literature
- Building block, with various abstraction levels:
  - Idiom (Coplien, 1991)
  - Design Pattern (Gamma et al., 1995)
  - Architectural Pattern (Buschmann et al., 1996)

"Aggressive disregard for originality."
Idioms

... are not (OO-) Design Patterns
An **idiom** is a low-level pattern (typically specific to a programming language).

- String copy in C
  (s and d are char arrays)

  ```c
  while (*d++=*s++);
  ```
An **idiom** is a low-level pattern (typically specific to a programming language).

- Lazy instantiation of Singletons in Java (Double-checked Locking Idiom)
  ```java
  private static Device device = null;
  public static Device instance() {
      if (device == null) {
          synchronized (Device.class) {
              if (device == null) {
                  device = new Device();
              }
          }
      }
      return device;
  }
  ```

  Requires Java 6 or newer to work correctly!
Template Method

A first Design Pattern
Design Goal

- We want to implement an algorithm such that certain (specific) parts can be adapted / changed later on.
The Template Method Pattern

• Define a skeleton of an algorithm in an operation, but defer some steps to subclasses

• Often found in frameworks and APIs

```java
OutputStream
{abstract}

write(byte[] b)  
write(byte[] b, int off, int len)  
write(int)

«method»
{
    for (byte i : b) {
        write(i);
    }
}

FileOutputStream

write(int)
```
The Template Method Pattern

- Use the Template Method Pattern to ....
  - separate variant and invariant parts
  - avoid code duplication in subclasses; the common behavior is factored and localized in a common class
  - control subclass extensions

```
AbstractClass
{abstract}

templateMethod() { 
  opA(); 
  opB(); 
}

ConcreteClass

opA() 
opB()
```

The template method is the method that defines the
Besides, abstract operations (must be overridden) it is possible to define
Design Patterns - Motivation

• Designing reusable, extensible software is hard
• Novices are overwhelmed
• Experts draw from experience
• Some design solutions reoccur

• Understanding reoccurring solutions has several facets:
  • Know when to apply
  • Know how to establish it in a generic way
  • Know the consequence (trade-offs)
Architectural Patterns

... are not Design Patterns
Architectural patterns help to specify the fundamental structure of a software system, or important parts of it.

- Architectural patterns have an important impact on the appearance of concrete software architectures
- Define a system’s global properties, such as …
  - how distributed components cooperate and exchange data
  - boundaries for subsystems
- The selection of an architectural pattern is a fundamental design decision; it governs “every” development activity that follows
Architectural patterns help to specify the fundamental structure of a software system, or important parts of it.

**Architectural Patterns**

- Pipes and Filters
- Broker Pattern
- MVC
- Broker
- ...

Often, it is not sufficient to choose just one architectural pattern; instead it is necessary to combine several architectural patterns.
Example: Model-View Controller (MVC)

The MVC pattern describes a fundamental structural organization for interactive software systems

- The model contains the core functionality and data
  The model is independent of output representations or input behavior.

- The user interface is comprised of:
  - Views that display information to the user
    The view obtains the data from the model.
  - Controllers that handle user input
    Each view has a controller. A controller receives input. The events are then translated to service requests for the model or the view. All interaction goes through a controller.
Example: Model-View Controller (MVC) Change Propagation

• A change propagation mechanism ensures consistency between the user interface and the model. (The change-propagation mechanism is usually implemented using the Observer pattern / the Publisher-Subscriber pattern.)

Basic Idea:
A view registers itself with the model.
If the behavior of a controller depends on the state of the model, the controller registers itself with the change propagation mechanism.
Example: Model-View Controller (MVC)

Change Propagation

Use the MVC pattern for building interactive applications with a flexible human-computer interface. When...

• the same information should be presented differently (in different windows...)
• the display and behavior of the application must reflect data manipulations immediately
• porting the UI (or changing the L&F) should not affect code in the core of the application
Example: Model-View Controller (MVC)

Structure

While the Controller and the View are directly coupled with the Model, the Model is not directly coupled with the Controller or the View.
Example: Model-View Controller (MVC)

Liabilities

- Increased complexity
  Using separate view and controller components can increase complexity without gaining much flexibility
- Potential for excessive number of updates
  Not all views are always interested in all changes.
- Intimate connection between view and controller

(Liabilities = dt. Verantwortlichkeiten / Verbindlichkeiten)
Architectural Patterns

Recommended Resources

- **Pattern-Oriented Software Architecture - A System of Patterns**; Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal; Wiley 1996

- **Design Patterns**; Gamma et al.

- **Patterns of Enterprise Application Architecture**; Martin Fowler; Addison Wesley 2003
Properties of (Design) Patterns
Design Patterns - Benefits

• Systematic (software-)development:
  • Documenting expert knowledge
  • Use of generic solutions
  • **Raising the abstraction level**
Design Patterns - Essentials

- a pattern has a name
- the problem has to reoccur to make the solution relevant in situations outside the immediate one
- it has to be possible to tailor the solution to a variant of the problem

A Design Pattern describes a solution for a problem in a context.

(to tailor = dt. anpassen)
Essential Parts of Patterns

1. **Pattern Name**
   A short mnemonic to increase your design vocabulary.

2. **Problem**
   Description when to apply the pattern (conditions that have to be met before it makes sense to apply the pattern).

3. **Solution**
   The elements that make up the design, their relationships, responsibilities and collaborations.

4. **Consequences**
   Costs and benefits of applying the pattern. Language and implementation issues as well as impact on system flexibility, extensibility, or portability.
   The goal is to help understand and evaluate a pattern.
Template for Design Patterns
(For Design Patterns as described by Gamma et al., 1995)

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
</table>
| 1. | ▶ Name  
    | ▶ Intent |
| 2. | ▶ Motivation  
    | ▶ Applicability |
| 3. | ▶ Structure  
    | ▶ Participants  
    | ▶ Collaboration  
    | ▶ Implementation |
| 4. | ▶ Consequences |
| 5. | ▶ Known Uses  
    | ▶ Related Patterns |
To document a used design pattern use the participant names of the pattern to specify a class’ role in the implementation of patterns.

**Template Method Pattern**

- **AbstractClass**
  - templateMethod()
  - opA()
  - opB()

- **ConcreteClass**
  - opA()
  - opB()

**Use of the Template Method Pattern in Java**

- **OutputStream**
  - abstract
  - write(byte[] b)
  - write(byte[] b, int off, int len)
  - write(int)

- **FileOutputStream**
  - write(int)
Levels of Consciousness for a Design Pattern

1. Innocence
2. Known tricks
3. Competent trick application
4. Applicability & consequences known
5. Wide knowledge of patterns & their interaction
6. **Capable of capturing knowledge into literate form**
## Design Patterns Serve Multiple Purposes

<table>
<thead>
<tr>
<th>Elements of Reusable Software</th>
<th>patterns foster reusability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reuse of Design</td>
<td>rather than code</td>
</tr>
<tr>
<td>Communication</td>
<td>design vocabulary</td>
</tr>
<tr>
<td>Documentation</td>
<td>information chunks</td>
</tr>
<tr>
<td>Language Design</td>
<td>high level languages</td>
</tr>
<tr>
<td>Teaching</td>
<td>passing on culture</td>
</tr>
</tbody>
</table>
Patterns enable the construction of high-quality software architectures.
A **software design pattern** describes...

- a commonly recurring structure of interacting software components
- that solve a general software design problem within a particular context.
## Design Patterns - Occurrences

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chess</td>
<td>from rules to expertise</td>
</tr>
<tr>
<td>literature</td>
<td>oldest reference</td>
</tr>
<tr>
<td>agriculture</td>
<td>wisdom vs. science</td>
</tr>
<tr>
<td>architecture</td>
<td>pioneering work</td>
</tr>
<tr>
<td>software design</td>
<td></td>
</tr>
</tbody>
</table>

### Patterns in Architecture

- **Light from two sides**
  - Place at Window
  - Deep terrace

© T. Kühne
Main Focus
(Content relevant for the exam!)

(Design Patterns = dt. Entwurfsmuster)
Summary
The goal of this lecture is to enable you to systematically carry out small(er) software projects that produce quality software.

- Idioms, Design Patterns and Architectural Patterns help you to solve recurring problems (at different abstraction levels) and to immediately understand the benefits and tradeoffs.
- Patterns enable you to talk about the design of your application at a higher abstraction level.