Design Patterns

Prof. Dr. Ralf Lämmel
Universität Koblenz-Landau
Software Languages Team
Reusable solutions to recurrent problems
Selected patterns
(Pattern names with intents)

- **Composite** -- provide uniform interface on part-whole hierarchies
- **Command** -- encapsulate the execution of functionality; enable undo
- **Visitor** -- represent operations on object structure as objects
- **Observer** -- provide change notifications to objects depending on state
- **MVC** -- decouple model, view, control in for interactive applications
- **Proxy** -- refine or replace behavior of a given object
- **Object Adapter** -- provide a different interface for an existing object
- **Template Method** -- capture general structure of an algorithm
- ...

(Intent = short combo of problem + solution)

(C) 2010-2017 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Key elements of a design pattern

- **Name and intent**

- The *problem* describes when to apply the pattern.

- The *solution* describes elements of the design, their relationships, responsibilities, and collaborations.

- The *consequences* are the results and trade-offs of applying the pattern.
“All” elements of a design pattern

- Pattern Name
- Classification
- Intent
- Also Known As
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

We do not go into these details in this course.

(C) 2010-2017 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Classification of design patterns

- Aspects
  - Creational patterns
  - Structural patterns
  - Behavioral patterns

- Scope
  - Class
  - Object
## Design patterns -- Overview

<table>
<thead>
<tr>
<th>Scope</th>
<th>Class</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Creational</strong></td>
</tr>
<tr>
<td></td>
<td>Factory Method (107)</td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td>Abstract Factory (87)</td>
<td>Adapter (object) (139)</td>
</tr>
<tr>
<td></td>
<td>Builder (97)</td>
<td>Adapter (class) (139)</td>
</tr>
<tr>
<td></td>
<td>Prototype (117)</td>
<td>Structural</td>
</tr>
<tr>
<td></td>
<td>Singleton (127)</td>
<td>Interpreter (243)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Template Method (325)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Behavioral</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Chain of Responsibility (223)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Command (233)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Iterator (257)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mediator (273)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memento (283)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Observer (293)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>State (305)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Strategy (315)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Visitor (331)</td>
</tr>
</tbody>
</table>

[Gamma et al., 1995, S. 10]
101 contributions using patterns

https://101wiki.softlang.org/Contribution:javaVisitor

https://101wiki.softlang.org/Contribution:javaTemplate

https://101wiki.softlang.org/Contribution:javaExorcism
Patterns to be discussed

- Abstract Factory
- Adapter
- Command
- Composite
- Decorator
- Observer
- Proxy
- Singleton
- Template
- Visitor
Abstract factory pattern
https://101companies.org/Abstract_factory_pattern

**Problem**: Consider two similar libraries (e.g., for GUIs). How to abstract from the choice of a specific library?

**101**: POJOs versus “beans”

(C) 2010-2017 Prof. Dr. Ralf Lämmel, Universität Koblenz-Landau (where applicable)
Abstract factory pattern
https://101companies.org/Abstract_factory_pattern
Problem: What if an expected interface is not supported by an implementation, but only a similar one? Also, we cannot or do not want to migrate to actual interface.

101:
Class adapter: add observability of list interface.
Object adapter: downgrade java.util.List to simple list.
Class Adapter pattern

https://101companies.org/Adapter_pattern

GilHamilton, Public Domain
Command pattern

https://101companies.org/Command_pattern

Problem: What if a certain action needs to be executed later (or an event needs to be triggered later), but all the relevant information is readily available now?

101: Schedule but otherwise defer cut operation.
Command pattern

https://101companies.org/Command_pattern

Notation: UML

Client

Caller

Command

execute()

ConcreteCommand

state

execute()

receiver.action()

Receiver

action()
**Problem:** When facing tree-structured data, how to distinguish (or not) behavior for composites versus leafs? How to avoid that code is littered with cases for composites versus leafs?

**101:**
Companies as collections of departments.
Departments as recursive collections of departments.
Departments also as collections of employees.
Composite pattern
https://101companies.org/Composite_pattern

Component
+ operation()

Leaf
+ operation()

Composite
+ operation()
+ add()
+ remove()
+ getChild()

Trashtoy, Aaron Rotenberg, Public Domain
Decorator pattern
https://101companies.org/Decorator_pattern

Problem: What if an object needs to vary its behavior at runtime?

101: Add observability (change tracking) to lists.
Observer pattern

https://101companies.org/Observer_pattern

Problem: What if one object’s consistency depends on awareness of another object’s state?

101:
Use observer to provide logging.
Use observer to enforce data constraint.
Observer pattern
https://101companies.org/Observer_pattern

Diagram:
- **Subject**
  - `+observerCollection`
  - `+registerObserver(observer)`
  - `+unregisterObserver(observer)`
  - `+notifyObservers()`

- **Observer**
  - `+notify()`

- **ConcreteObserverA**
  - `+notify()`

- **ConcreteObserverB**
  - `+notify()`

Model:
- `notifyObservers()`
- for observer in observerCollection
- call observer.notify()
Proxy pattern

https://101companies.org/Proxy_pattern

**Problem:** What if we cannot or do not want (yet) to expose a specific object without constraints to other objects?

**Scenarios:**
Additional access behavior for objects.
Access management for remote objects.
Access management for expensive objects.

**101:**
Use security proxy to protect salary access.
There is an interesting issue of proxy deployment.
Proxy pattern

https://101companies.org/Proxy_pattern

https://en.wikipedia.org/wiki/Proxy_pattern#/media/File:Proxy_pattern_diagram.svg
Traced by User:Stannered, created by en:User:TravisHein
CC BY-SA 3.0
Singleton pattern
https://101companies.org/Singleton_pattern

Problem: What if we need to ensure (for consistency or efficiency) that there is exactly one instance of a given class?

101: Reuse functor objects.
## Singleton pattern

https://101companies.org/Singleton_pattern

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
</table>
| - `singleton : Singleton`
| - `Singleton()` |
| + `getInstance() : Singleton` |
Problem: What if some behaviors (algorithms) follow a common scheme? How to reuse the scheme? How to parameterize the scheme for instantiation?

101:
Queries such as 101feature:Total
Transformations such as 101feature:Cut
Template Method pattern
https://101companies.org/Template_method_pattern

```java
// ...
doSomething();
// ...
PrimitiveOperation1();
// ...
PrimitiveOperation1();
// ...
doAbsolutelyThis();
// ...

AbstractClass

PrimitiveOperation1()
PrimitiveOperation2()
TemplateMethod()

doAbsolutelyThis()
doSomething()

ConcreteClass

PrimitiveOperation1()
PrimitiveOperation2()
doSomething()

https://en.wikipedia.org/wiki/Proxy_pattern#/media/File:Proxy_pattern_diagram.svg
Giacomo Ritucci, CC BY-SA 3.0
Visitor pattern
https://101companies.org/Visitor_pattern

See
previous
lecture.
A more complex pattern:

Model View Controller

Copyright notice: The MVC slide decks are derived from text parts, figures and source fragments from the following online article: © Robert Eckstein: Java SE Application Design With MVC, Sun Developer Network (SDN), http://java.sun.com/developer/technicalArticles/javase/mvc/, March 2007. If you derive any work from the present slide deck, you must keep the copyright notice in tact. All remaining copyright, if any, is with Ralf Lämmel.

Complete MVC:
https://github.com/101companies/101repo/tree/master/technologies/Java_platform/samples/sunMvcSample
Elevator speech

If you do business programming with GUIs on the web or not, you need the model-view-controller (MVC) design; it helps you to decouple data access and business logic from the manner in which it is displayed to the user.
Motivation: an editor

- Data is stored in objects.
- One view may change data so that other view needs update.
Straightforward implementation

- Design
  - Use *GUI listeners* to access data.
  - Use *Observer pattern* to notify editor of changes.

- Consequences
  - GUI is locked into the data access layer.
  - Restrictions of user actions hard-coded into GUI.
  - Multiple views are hard to sync efficiently.
Model View Controller

MVC is an important architectural or design pattern for decoupling data model (data access), business logic in an application and presentation in the user interface.

- The view submits gestures to controller.
- The view registers with model for data changes.
The elements of MVC

- **Model** - The model represents data and the rules that govern access to and updates of this data. In enterprise software, a model often serves as a software approximation of a real-world process.

- **View** - The view renders the contents of a model. It specifies exactly how the model data should be presented. If the model data changes, the view must update its presentation as needed. This can be achieved by using a *push model*, in which the view registers itself with the model for change notifications, or a *pull model*, in which the view is responsible for calling the model when it needs to retrieve the most current data.

- **Controller** - The controller translates the user's interactions with the view into actions that the model will perform. In a stand-alone GUI client, user interactions could be button clicks or menu selections, whereas in an enterprise web application, they appear as GET and POST HTTP requests. Depending on the context, a controller may also select a new view -- for example, a web page of results -- to present back to the user.
Two views that use the same model
Model-view controller
Implementation

- DocumentModel
- TextElementModel
- AbstractModel
- DefaultController
- AbstractController
- DisplayViewPanel
- PropertiesViewPanel
- AbstractViewPanel


Maintains collections for registered models and views that are registered. Registers itself as a property change listener on any model. Uses reflection to broadcast changes.
Summary

The use of design patterns is a basic element of software development. If you wouldn’t use them, your designs and implementations are unnecessarily unstandardized, idiosyncratic, incomprehensible, unmaintainable, etc. That is, if you are interested in a software development career, or you plan to talk to software developers at an interesting level of detail, you have to aggregate substantial knowledge of design patterns.