

# Principles of Software Design

## Design patterns, Code smells, Refactoring

Robert Lukočka

lukotka@dcs.fmph.uniba.sk

www.dcs.fmph.uniba.sk/~lukotka

M-255

# Design pattern

Design pattern:

*Repeatable solution to a commonly occurring problem in software design.*

- It is not about reusing code.
- Design patterns are mostly about typical class structures present in designs.

# Design patterns

- Are reusable solutions for design.
- Give terminology to ease speaking about design.
- Known solution is easier to understand.
- Provide inspiration even if the case is not covered by a design pattern.
- They may indicate missing features in the programming languages.
- Some of the patterns are included into languages (e.g. Decorator in Python).
- Duck-typing languages need far less elements to attain the same goal. Even if some elements (e.g. interfaces) are not in the code explicitly, they are still there implicitly.

## Example - Decorator pattern

**Decorator pattern** - There are other viable solutions to this problem, but

- many of the other solution are less flexible,
- other solutions are much harder to explain - if you use the pattern a single word “Decorator” is enough to describe several classes.

# Types of design patterns

Design pattern Gang of Four types:

- Creational
- Structural
- Behavioral

But we have also

- Concurrency patterns
- Domain-specific patterns
- ...

# Creational design patterns

- **Factory method** - a method (which may be static but is not a constructor) of a class, which returns new instances of some other class.
  - The method may return instances of more than one class depending upon the parameters received.
  - The caller does not need to know the exact class of the instance (just an explicit or implicit interface it has to satisfy).
  - A class containing a factory method can be injected into a class calling it e.g. in constructor. We can replace the factory if we need to produce instances of a different class.
- **Abstract factory** - An interface containing several related factory methods.
  - Generally, there exists more implementations of the interface.

# Factory method - Example

- A graph may be represented with adjacency matrix or adjacency list
- We want to create graphs in our class A, but which representation is desirable is outside the scope of the class.
- We need the following
  - GraphFactory - interface defining the factory method
  - SparseGraphFactory and DenseGraphFactory implement GraphFactory
  - Graph - interface that contain some graph methods (so we can actually do something with the graph even if we do not know the implementation).
  - SparseGraph and DenseGraph implement Graph.
  - Our class A that creates graphs takes GraphFactory in its constructor.
  - According to the implementation of Graphfactory we provide class A either creates SparseGraph or DenseGraph instances.

# Creational design patterns

- **Builder** - A class that is used to incrementally create or modify instances of other class..
  - If creating or modifying an object is complex we could end up with two sets of methods, one for the building / modifying phase and second for the actual usage phase. This violates single responsibility principle.
  - Distinct data structures may be needed in each phase.
- Example: Java StringBuilder



## Builder - example

- Graph has methods `addEdge(...)`, `removeEdge(...)`, `addVertex(...)` (represented by adjacency lists)
- We want to replace the edge by two consecutive edges incident to a new vertex.
- We can: remove edge, add vertex, add two edges.
  - We have to manipulate the lists a lot.
  - This is especially bad if we do many such operations
- `GraphBuilder` - we add the new vertex and write its new adjacency list. Note that now we do not have a graph now, thus it is good that we have a distinct class in this situation. We correct the entries in the adjacency lists of incident vertices, We have a proper graph again and we can convert the result to `Graph`.

# Creational design patterns

- **Object pool** - Instead of creating / destroying a class we just take an instance from / return an instance to a pool of objects.
  - Useful if it is hard to create an instance (threads, connections, etc.)
  - We can control the resources by providing limits on the number of instances available.
  - typical use: ThreadPool, ConnectionPool
- **Prototype** - New instances are being created by copying a prototype.
- **Singleton** - A class that is guaranteed to have only one instance.
- ...

# Structural patterns

- **Decorator**
- **Composite** - Treelike structure to compose objects.
- **Facade** - Class giving an easy access to whole subsystem.
- **Adapter** - Class that modifies the methods of another class to satisfy an interface.
- **Proxy** - An object representing another object.
  - Access proxy, remote proxy, virtual proxy, ...
- **Flyweight** - We divide a class into a part that is common for many instances and a part that is specific for each instance.

# Behavioral patterns

- **Iterator**
- **Observer** - Notify objects of about changes.
- **Strategy** - Encapsulation of an algorithm.
- **Template method** - Method in an abstract class using other abstract methods.
- **Null object** - Sometimes reasonable “default” exists which can be returned in case of failure.
- **Memento** - Keep and reconstructs a state of an object
- **Visitor** - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- ...

# Code smells

Code smell is

*"a code smell is a surface indication that usually corresponds to a deeper problem in the system" - - M.Fowler - -*

- Code smell is by definition not a bug.
- May indicate technical debt.

**Refactoring is a process of restructuring the source code without changing its external behavior.**

# Software entropy

Ivar Jacobson et al.:

*The second law of thermodynamics, in principle, states that a closed system's disorder cannot be reduced, it can only remain unchanged or increase. A measure of this disorder is entropy. This law also seems plausible for software systems; as a system is modified, its disorder, or entropy, tends to increase. This is known as software entropy.*

M.M.Lehman, L.A.Belady:

- 1 *A computer program that is used will be modified*
- 2 *When a program is modified, its complexity will increase, provided that one does not actively work against this.*

# Incorporating refactoring into software development process

- Refactoring should be a incorporated into our software development process

Example: Test-driven development

- 1 Write a test.
- 2 Check if the test fails (this is, according to my experience, unexpectedly useful)
- 3 Write code
- 4 Check if the test passes
- 5 Refactor
- 6 Check if the test passes

Note the division between adding functionality and refactoring, this is important.

# Code smells - examples

## Code smells - Sourcemaking

- Long class
- Too many arguments in a method
- Switch statement
- Parallel inheritance hierarchies
- Repeating code
- Too many comments (even in the case they seem useful)
- ...



# Refactoring - examples

## Code smells - Sourcemaking

- Decompose complex conditional
- Extract Method
- Extract Variable
- Replace Nested Conditional with Guard Clauses
- Introduce Parameter Object
- Form Template Method
- ...

# Resources |

- [Sourcemaking](#)
- [oodesign.com](#)

# References |



Sourcemaking



oodesign.com