Principles of Software Design
Principles of O-O design

Robert Lukoťka
lukotka@dcs.fmph.uniba.sk
www.dcs.fmph.uniba.sk/~lukotka

M-255

Modeling and design

Common levels of abstractions:

- Classes, most important attributes, relations between classes.
- + **most of the methods, however, some aspects are omitted for model simplicity**
- Full implementation model, can be used as a template for implementation

Aspects that may be omitted early in the design process typically include concurrency and persistence.

- However, you should have your general approach settled as these are typically crosscutting concerns.
- You just do not need to decide exactly where each lock is, etc.

Mid-level design preconditions

Given the right decision was made at the architecture level your system is modularized enough that a subsystem you are dealing with

- Is well defined (defined scope and interfaces)
- Is small enough to be comprehensible by a single person.
- You understand what are you designing (i.e. you have most of the requirements affecting this module)

Mid-level design goals

Goals for the design at this level:

- Modularization
- Abstraction
- Information hiding
- Separating interface and implementation
- Low Coupling (few dependencies between parts)
- High Cohesion (parts do just one thing)
- Sufficiency, Completeness, Simplicity, Flexibility ...

You want most of these on architecture level too, but there are subtle differences.

Mid-level design goals

Modularization

- Focused mostly to modularize implementation and verification (at architectural level we are more focused on modularizing requirements, documentation, etc.)
- You want to separate aspects that has different properties.
  - Create modules that allow consistent coding/testing style (see High Cohesion).
    - Separate stuff that requires state/does not require state.
    - Separate stuff with different testing requirements (e.g. for some trivial code it does not make sense to write unit tests)
    - Units allow for consistent coding style.

Mid-level design goals

Abstraction

- Focused on stuff like simplicity
  - Making various arbitrary data in the domain irrelevant for the implementation.
  - Not only are such data unnecesarily complex, but they are also prone to changes.
- Avoiding repeated code.

Mid-level design goals

Information hiding, Separating interface and implementation

- to preserve modularization over time
- not necessarily that strict for tightly related parts

Low Coupling, High Cohesion

- Coupling - degree to which parts depend on each other.
- Cohesion - degree to which a part is uniform.

What are our tools

- Abstraction (techniques from domain modeling are applicable to the modeling of the software system)
  - This includes stuff like abstraction of types, relationships. etc.
  - it is good idea to perform these simplifications before refining the model
- Don't repeat yourself, Rule of 3
- You ain't gonna need it
- ...

For object-oriented design:

- SOLID
- Dependency injection

Design verification

You can verify design:

- Can you really perform all the function prescribed by the interfaces you have to implement?

It is much harder to validate the design, example:

- We expect aspect A of the system to vary, thus our design makes it easy to change aspect A
- Validation means confirming that this assumption is correct.
- This can be done e.g. by evaluating past changes (even before the development of our system started).
- Often it is hard to predict future changes - the flexibility of design is important (however one needs to balance the flexibility and the added complexity).

YAGNI

You ain't gonna need it.

- Relevant also on the architecture level
- Unless the evidence says otherwise, you should prefer simplicity over
  - additional features
  - excessive flexibility
  - performance
  - etc.

This does not mean that you should not apply basic principles while creating the design. However, you should avoid doing complex non-standard stuff unless you have really good evidence you need to.

DRY

Don't repeat yourself

- Repeated code is
  - longer
  - harder to maintain
  - error-prone
- You should avoid repeating the code.
- It is always possible using the right technique.
- Tradeoff: sometimes avoiding the code repetition creates some quite hard to grasp abstraction (e.g. template method pattern with many parameters) - goes against simplicity

Rule of 3

If in doubt, apply (rule of thumb) Rule of 3:

- If similar piece of code appear three times, change the design so that the code is not repeated.
- The idea is, if the code appears for the third time, it is likely that it will happen even more, in which case the change is utterly necessary.
- If you have code that is repeated twice document this very carefully (e.g. comments in the code referring to the other part and vice versa).
- Similar reasoning can be applied elsewhere (e.g. if some aspect of the system changes three times, you should change your design so that the change is easy to do)

Low Coupling

Coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules [0].

- A change in one module usually forces a ripple effect of changes in other modules.
- Assembly of modules might require more effort and/or time due to the increased inter-module dependency.
- A particular module might be harder to reuse and/or test because dependent modules must be included.

Low Coupling

- Information expert principle (rule of thumb for low coupling)
  - A Responsibility belongs to a class which has most data to perform the task
- Coupling strength, Coupling distance
  - It is more acceptable to have high coupling between classes with small distance

High cohesion

Cohesion - cohesion refers to the degree to which the elements inside a module belong together [0].

- Example metrics LCOM4:
  - Create a graph, vertices - attributes and methods, edges method calls a method or uses an attribute
  - LCOM4 = number of components
  - If LCOM4 $\neq 1$, the class is not very cohesive.
- Similar testing requirements.
- Similar cross-cutting concerns (e.g. db, logging).
- Similar expected coding style.

Encapsulate what varies

If some functionality (e.g. method of a class) changes a lot we should encapsulate this functionality

- Create a new class whose concern is exactly this behavior
- Hide behind an interface (strategy pattern)
- This protects the rest of the class from the frequent changes.
  - protects the original class against newly introduced errors.
  - allows to narrow the scope of the change making it easier.

Alternative approach: abstraction - design things so, that the changes just affect input data (e.g. a change in a configuration file)

Object oriented design

Information hiding, Separation of interface and implementation

- Encapsulation does some information hiding, however, interfaces are a more robust tool to do this.
- Implementation depends on the interface and not vice versa
  - This is general principle valid also outside OOP.
  - OOP separates interface from implementation using polymorphism.
  - The caller (who requires interface) needs not to be aware of who exactly the callee (implements interface) is - polymorphic dispatch.
  - The caller code should not need to recompile upon a change in the callee code.
  - Private/public implicitly define an interface, however, this tool may not be sufficient.

Anemic domain model

Anemic domain model - classes have little or no behavior.

M. Fowler, 2003:

> *"The fundamental horror of this anti-pattern is that it's so contrary to the basic idea of object-oriented designing; which is to combine data and process them together. The anemic domain model is just a procedural style design, exactly the kind of thing that object bigots like me ... have been fighting since our early days in Smalltalk. What's worse, many people think that anemic objects are real objects, and thus completely miss the point of what object-oriented design is all about."*

Anemic domain model

- Suggest insufficient abstraction
- Not really an O-O design
- While this is an antipattern in O-O design, it may be natural part of other approaches (e.g. procedural, functional design).

SOLID

- Single responsibility principle
- Open-closed principle
- Liskov substitution principle (Liskovej substitučný princíp)
- Interface segregation principle
- Dependency inversion principle

Single responsibility principle

A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.

This implies:

- high cohesion, LCOM4$=1$
- class has a responsibility

How to separate other responsibilities? E.g.. strategy pattern.

Open-closed principle

Software entities should be open for extension, but closed for modification.

How to change a class without modifying it?:

- Inheritance
- Composition (preferred, *"Composition over inheritance"*)

How to compose objects without creating an explicit dependency?

- Dependency injection (via constructor/ method, described in the next lecture)

Liskov substitution principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

- Inheritance is not as IS-A relationship
- e.g. Square is a rectangle, but square should not be a subclass of a rectangle.

Interface segregation principle

Many client-specific interfaces are better than one general-purpose interface.

- Segregates interface from implementation
- Makes the interface easier to use in the client code
- Limits the impact of the interface change
- You can use Adapter pattern to adjust the class to the required interface.

Dependency inversion principle [0]

One should "depend upon abstractions, not concretions

- High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g. interfaces).
- Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.
- By dictating that both high-level and low-level objects must depend on the same abstraction, this design principle inverts the way some people may think about object-oriented programming
- If you apply this principle 100% you should have an interface between each cooperating classes. This may not always be most practical, however, more distant parts of the system should be separated by interfaces and adhere this principle.

Bob Martin - SOLID Principles of O-O and Agile Design

Recommended video on the topic: Bob Martin - SOLID Principles of OO and Agile Design (12:30-34:45, 37:45-52:30)

Dependency injection

- This is an important design concept that allows modularization of testing.
- Do not confuse it with dependency inversion

Dependency injection and dependency inversion

- A class should not create instances of other classes (if we do sociable testing it might make some exceptions of this rule for closely related classes) - dependency injection / dependency injection of a factory.
- A class should not depend on the implementation of the collaborator, just on the interface - dependency inversion

Dependency injection is a way to implement **dependency inversion**.

OO design and object dependence

All collaborators (we want to separate) should be separated by an interface.

- calls method, modifies, . . . - we give the collaborator in the constructor or as a method parameter
- creates, destroys - we give a factory (implementing a factory interface) as an argument in the constructor of the object

We can inject the dependency either via constructor or as a method argument.

- It is like aplying strategy pattern all the time, even if you have just one strategy.
- One of the less obvious reasons to do this is the ability to separate testing by replacing actual implementation object with test doubles

Automated dependency injection tools, e.g. dependency injector.

UML Sequence diagrams

- UML class diagram with several additional comments in text form is often sufficient to explain design.
- In some more complex cases one needs to provide detailed dynamic description of how the classes accomplish a goal.
- UML Sequence diagrams provides dynamic view of the described system.

UML Sequence diagrams

- UML class diagram with several additional comments in text form is often sufficient to explain design.
- In some more complex cases one needs to provide detailed dynamic description of how the classes accomplish a goal.
- UML Sequence diagrams provides dynamic view of the described system.
- Have a quick look at the examples and the reference.
- Note that synchronous and asynchronous calls are distinguished.

Resources I

- M. Fowler - YAGNI
- Wikipedia - SOLID
- Wikipedia - Dependency inversion principle
- Bob Martin on SOLID (especially 12:30-34:45, 37:45-52:30)
- uml-diagrams.org: Sequence diagram examples
- uml-diagrams.org: Sequence diagram reference

References I

 Wikipedia - Coupling

 Wikipedia - Cohesion

 Wikipedia - SOLID

 Wikipedia - Dependency inversion principle