15. Software design

Ján Dugáček

February 11, 2019

Table of Contents

Motivation

- A program whose needed functionality and way of implementation is absolutely clear from the start exists only in the land of fairies and unicorns
- Real programs have to do tasks that are unclear in advance, it always turns out something cannot work the way it was planned and nobody knows what will they be used for 20 years later
- This lecture doesn't teach any code, only practices how to design a program, applicable also to other programming languages

Basic concepts of writing software well

- **Don't repeat yourself (DRY)** - do your best to avoid repetition of anything in the code, even if everyone has copy/paste and find/replace available
- **Code readability** - the code must be readable, function and variable names should explain everything to a reader who understands the topic, only nontrivial algorithms should require comments
- **Code maintainability** - the code will have to be changed, at unpredictable places, by you after forgetting what it did or by others who won't have time to study it in depth

Main ways to write software badly

- **We enjoy typing (WET)** - also known as *copy/paste-driven development*, repetitions make changes unbeliebably more difficult and error-prone to do and can cause a single error to appear on multiple locations
- **Spaghetti code** - the code flows almost unpredictably and it's not very visible where does it go from where (the design of most modern programming languages prevents this to some extent)
- **Inconsistency** - some names are in *snake_case*, others are in *camelCase*, some others are in *PascalCase* while they all name the same kind of constructs, indenting chaotically with spaces and tabs mixed, leaving random numbers of newlines between function, marking member variables differently, it all makes it hard to change the code using automatic tools (using different cases for functions and classes is good)
- **Cargo cult programming** - adhering to rules and following some techniques without understanding what they are good for, just because better programmers use them, this results in usage of these techniques in places where they don't make sense, disobeying essential rules to follow less essential rules or adhering to rules derived from some *alternative facts*

Motivation #2

- That was quite obvious, no?
- Well, it indeed was, but we are going to build on it
- When writing the contents of a function in any programming language with blocks, this breaks down just to using `for`s instead of code repetition and naming variables properly
- When writing a larger program with many objects and no perfectly detailed plan, adhering to these rules becomes more difficult
- Techniques were designed, books were written about ways to adhere to these standards
- Of course, it happens that there isn't a way to adhere to all these rules, so if no friend can help you, prefer to break the ones that seem less important in that context or to break them in a smaller scale

SOLID principles

SOLID principles are 5 guidelines that generally help keep the code clean. They are not equally important or equally obvious. They are:

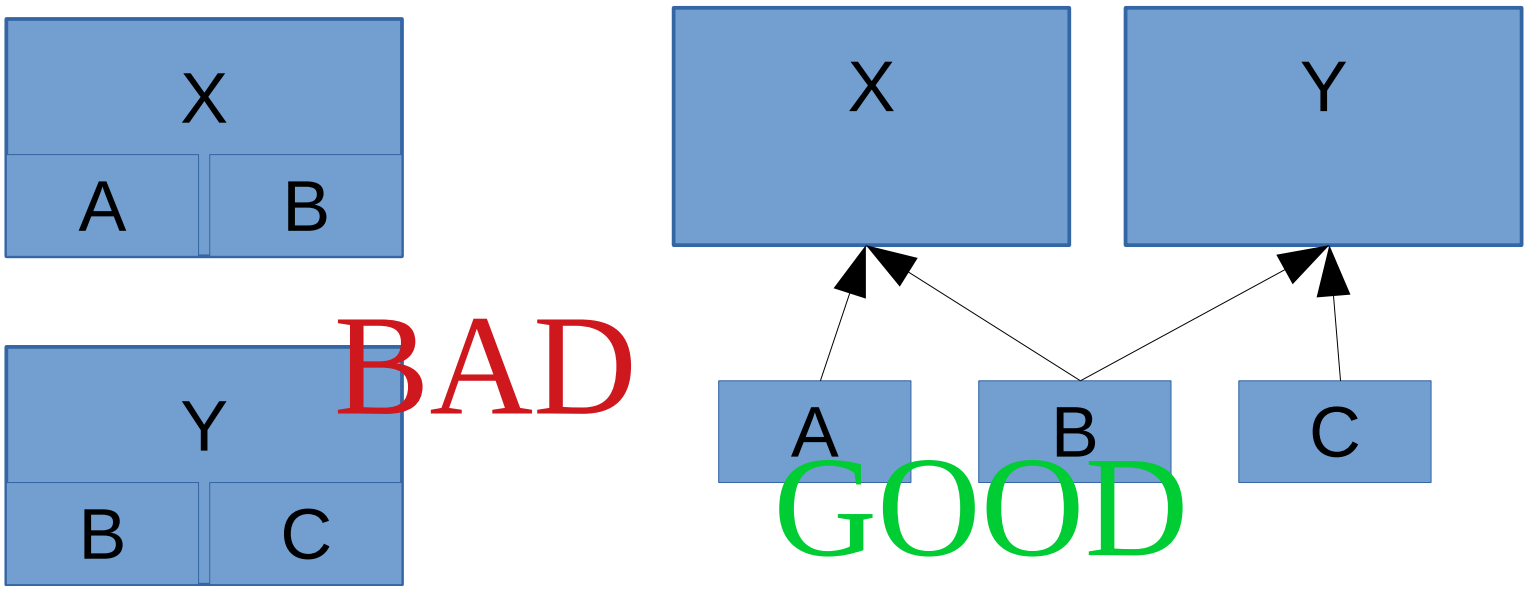
- **S**ingle responsibility principle
- **O**pen/closed principle
- **L**iskov substitution principle
- **I**nterface segregation principle
- **D**ependency inversion principle

Single Responsibility Principle

Every class represents exactly one logical object, every function does exactly one logical activity. Not more, not less.

- If class X represents functionalities A and B, it should be composed of classes for A and B, so that if you want to add class Y that has functionalities B and C, you won't have to write B again
- It's often not very visible that a class implements more than one functionality, so it's useful to occasionally check if a new class isn't going to share some of its functionality or inner workings or if more classes don't happen to share something in common
- Same as above applies to functions
- Big objects or long functions are signs that this rule is violated, but sometimes it's a false positive
- Functions also shouldn't do less than one thing, because then have to be called in groups, it is prone to human errors and repeating groups of functions cause problems related to We Enjoy Typing

Single Responsibility Principle #2



Open/Closed Principle

Classes should be open for extension, but closed to modification. There should be no way to drive an object into an inconsistent state by uninformed usage, but it should be possible to alter the class' functionallity by inheriting from it.

- This is very efficient at preventing human errors
- Using an object differently than intended by its creator means the person using it has made a mistake
- If the programmer using it actually wants to use it differently, a new class inheriting from it has to be created
- Classes' `public` methods and attributes should allow doing only what is intended to be done, `protected` methods and attributes are to be touched by new classes that are derived from it

Liskov Substitution Principle

Any derived class can replace its parent and the program will continue working correctly. It can do its work differently, it can do also other work, but it must behave in that context exactly like the parent would.

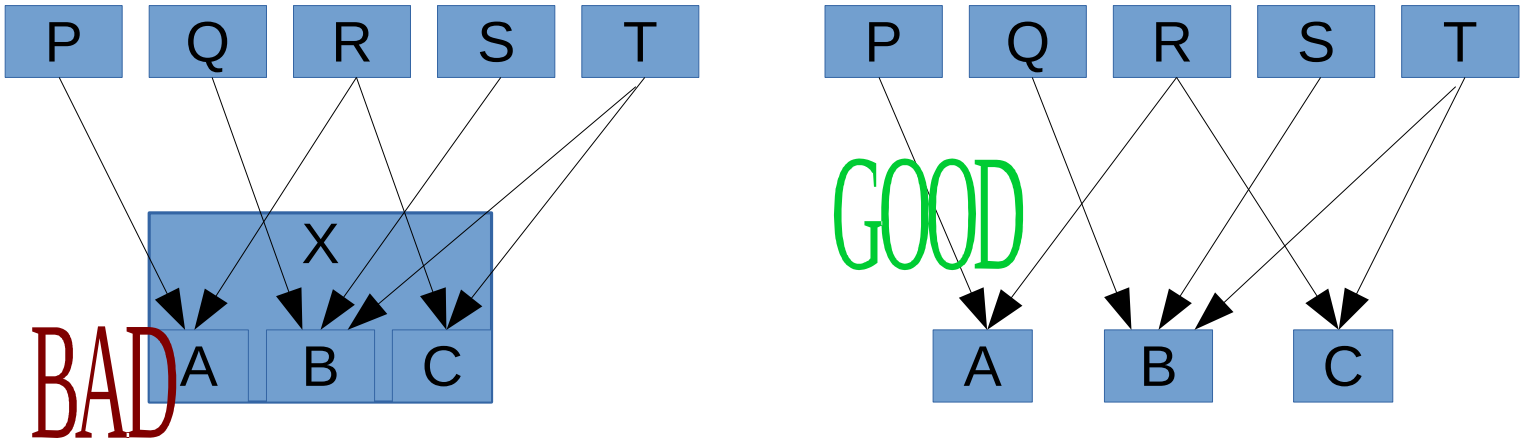
- This allows new classes to be used in the same code as old classes could be used
- It's essential when extending classes based on the Open/closed principle, because class B that can't be used as its parent A is even less useful than class C containing A
- Violations of this rule are quite obvious (or accidental)

Interface Segregation Principle

Each purpose should have its own interface, rather than one interface for many purposes.

- Various interfaces (or parent classes) should do unrelated things, one interface (or parent class) should do only closely related stuff
- If two interfaces serve two closely related purposes, it becomes difficult to use them separately and they don't really follow the Single responsibility principle with all the problems that descend from it
- Finding ways to properly split them can be a difficult task, often requiring to create lower level interfaces that are used by both

Interface Segregation Principle #2

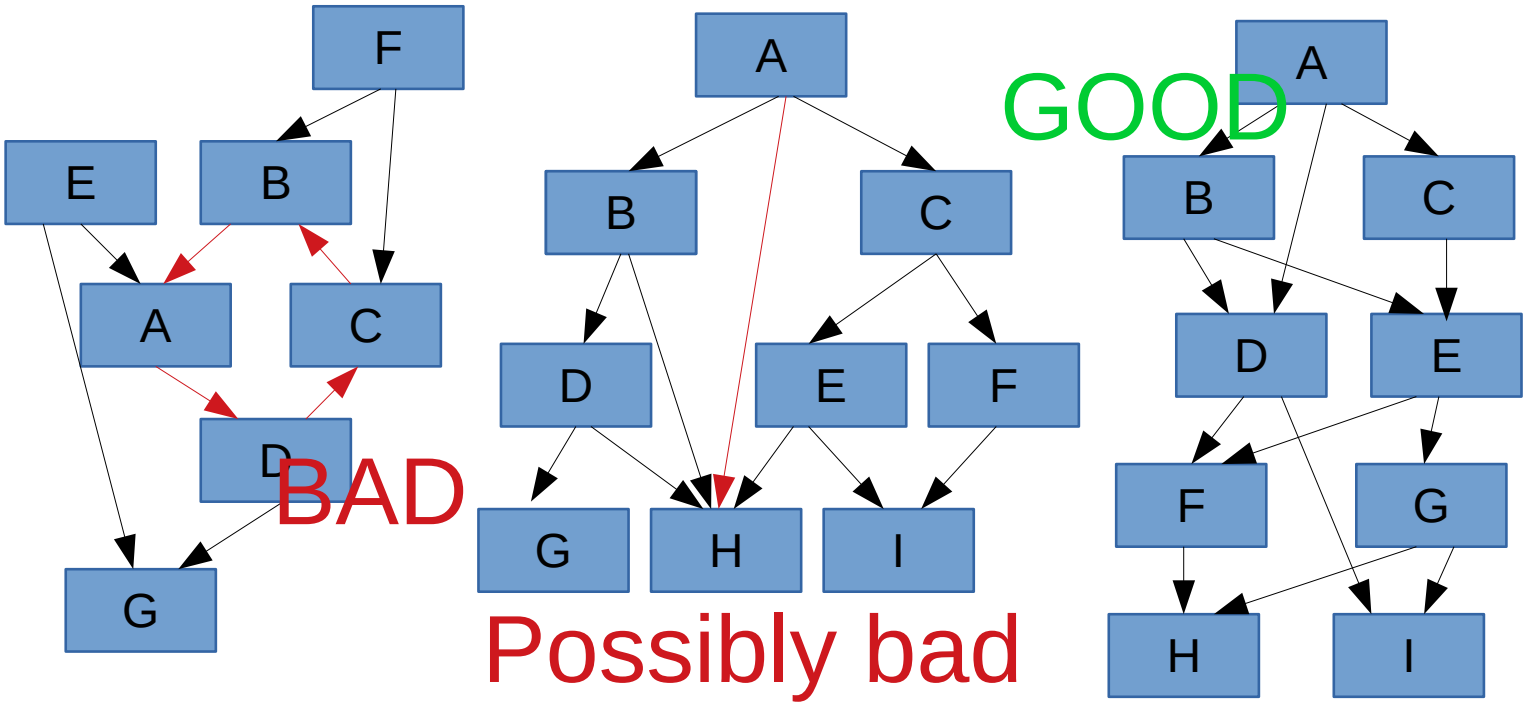


Dependency Inversion Principle

Lower level details should not depend on high level abstractions, higher level abstractions should not depend on details. There should be levels of abstraction, where lower levels don't call higher levels and higher levels don't call functionality from deep below.

- The program should have *layers*, it starts on the highest layer (*main*), general utilities and libraries are on the lowest, nothing of the higher layers should ever be mentioned in the code of lower layers (otherwise, these layers are joined, violating the Single responsibility principle)
- Also, no layer should touch layers too low below, because it makes it difficult to change anything on these layers (if a class is used by 100 classes above it, changing its interface is difficult)
- Overattachement to the second part can lead to *lasagna code*, where one change needs changing many layers, needless extra code lines with function calls; standard libraries are unlikely to ever change anyway

Dependency Inversion Principle #2



STUPID principles

STUPID principles are typical mistakes that are typically made when failing to write the code correctly. As before, those are rules of thumb, not immutable physical laws.

- **S**ingleton - a class that has only one instance possible, allowing to obtain it from any part of the code, allowing easy violation of Depedency inversion principle and can be lead to Untestability, but can help keep the DRY rule and improve efficiency, so it's controversial if it's really bad
- **T**ight coupling - classes are too closely related to each other, have to be used in groups and lead to We Enjoy Typing
- **U**ntestability - if a class is too connected to others, its testing becomes difficult without actually using the program and checking if it works
- **P**remature optimisation - making the code look like shit just to make it 2% faster or not using readability improvements that the compiler can easily change to the faster form anyway (4 statements instead of a `for` to avoid having to increment a variable), unless absolutely certain that it helps
  - Overattachement to avoiding premature optimisation may lead to *premature pessimisation*, when code is made slower without improving readability
- **I**ndescriptive naming - unobvious shortcuts used in an era where screens are wide enough
- **D**uplication = We Enjoy Typing

Design patterns

- Design patterns are techniques of making the code better
- They may or may not be useful depending on the situation
- Many of them are used to solve specific difficulties at adhering to SOLID principles
- Some of them are rather obvious and they are good to know only to know how to name them when describing code
- There's many of them, not all are mentioned
- Patterns can be used incorrectly
- Commonly used *bad* techniques are called *antipatterns*

Factory

- A factory is a class that makes classes of lower layer, whose constructors alone cannot appropriately construct them
- It may be used if the class' construction would require some data from a higher layer
- It may be used to create a load of classes from some cached data
- It may be used to create objects of various classes with common parent class from a file it reads and return them all as instances of the parent class

Publisher/Subscriber

- A class allows other classes to register something in it so that if it reacts to some action, it will perform some action they have set
- This allows classes of higher layers to have classes of lower layers call their methods while keeping the code clean
- It may be used to have a button alert some object that it was pressed by calling a method it has registered in it
- It may be used to have some communication class wait for external signals and call some functions of a class on a higher level
- A similar pattern is *Observer*, where one object allows other objects to register to be notified of its changes

RAII

- RAII means *Resource acquisition is initialisation*
- An object that is completely set up when created and completely frees everything when its existence ends
- It allows to keep something active (like an open file or access to some resource) until the end of block without having to deactivate it anywhere, making sure no one will accidentally leave it open
- This is very usual and well supported in C++

Facade

- A class that only calls methods of some other class
- It allows using multiple classes with incompatible interfaces by the same code
- It allows using a code with an ugly interface (whose authors have cared to write the code well) without using ugly code - there are libraries where using basic functionality requires 20 lines of code
- A similar pattern is *Adaptor* that serves to translate the output of one part of code so that it could be used by some other code, some sort of last resort solution when parts of code are incompatible

Flyweight

- An object that provides read-only data to other objects
- It improves performance by storing the parsed, computed or otherwise non-trivially obtained data for later use
- A typical use is to use it to store parsed configuration data there so that it would not ever be parsed again

Singleton

- A class that is instanced into only one object (usually using a private constructor accessible only to a function that returns it and checks it is created only once), allowing to obtain it from any part of program
- It saves other classes and methods from having to pass references to an object all around the program it's always clear what object is needed
- Unlike a global variable, it can control what is done with it and keep itself consistent
- A backslash is that altering the program to use more of these objects is difficult (but sometimes, there's no way more of them could be needed, if it is for example a Publisher for keybord input)
- Classes using it become hard to test because replacing it by a fake class for testing is an issue
- It is often used to hide other problems (like a class used by too many other classes) and can cause new problems (can be easily used by classes of layers below it, tangling the dependencies)
- Legitimate uses of singleton are rare, making its use quite controversial

Exercise

Don't write any code, actually coding this would take quite some time!

1. Design a simple *brick breaker* game
2. Draw a possible division to objects of a program that parses and numerically solves equations for x; a user uses a command line to write equations and receive solutions
3. Design a program that reads a file containing data, fits them using polynomials and draws graphs
4. Design a library that draws 3D graphs of 2D arrays
5. Design a program that watches a folder with subfolders for changes and synchronises the changes with a folder on a mobile that is running another part of the program

Homework

- Design a program that coordinates the work of devices of an astronomical telescope, composed of lenses, cameras, filters, engines and other parts typical for telescopes, offering a user interface for its operators
- Don't try to write it
- You have two weeks to do it