Agile Architectures – DRY and SOLID

15.6.2011

Andreas Schroeder, Annabelle Klarl

Bad Design

- Facing Bad Design is an unpleasant experience
- Even more, since most of the time, we are the authors of that design
- No one sets out to create bad design
- Bad design creeps into your code over time
- … but what is bad design?
  - **Rigid**: hard to change because changes affect large parts
  - **Fragile**: changes break unexpected parts of the system
  - **Immobile**: hard to reuse since it cannot be disentangled

Technical Debt

- Bad Design and bad code is like a financial debt
  - „We‘ll look at it later, place a TODO here“
- If you don‘t repay it swiftly, interest will build up
  - „I thought we had a document on this“
  - „I thought we had a tests on this“
  - „It‘s ok that these tests are failing, they were failing all the time“
- … and finally, interest will kill you
  - „we don‘t have time to fix this“
  - „we can‘t change this, it‘ll take too much time“

Overview

- Stay DRY
  - **D**on‘t
  - **R**epeat
  - **Y**ourself
- Create SOLID systems
  - **S**ingle Responsibility Principle (SRP)
  - **O**pen/Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation
  - **D**ependency Inversion

A word or warning

- All of the following principles are general guidelines to follow
- Overdoing them will lead to unmaintainable code as it will become extremely hard to understand and tiresome extend.

Why DRY (1/2)

- Non-DRY code is a maintenance nightmare
  - Bad code and bugs gets copied and need to be fixed everywhere
  - Imagine that:
    - A method fragment gets copy/pasted two times
    - The method that contains it gets copy/pasted two times
    - The class that contains the method gets copy/pasted two times
    - Grand total: seven copies (at least)
  - Avoid this ripple effect by all means

Why DRY (2/2)

- DRY is an architecture generating principle.
- O/R-Mapping Example:
  - SQL is a language with a lot of redundancy: the schema is implicitly repeated in every query
  - To stay DRY, query parts need to be extracted into separate methods
  - Congratulations! You‘ve just started to create your data access layer

Intermezzo!
Antipattern: String Typing

- Stringly Typed code (riff on “strongly typed”)
  - String method parameters where other types would fit
  - Repeated String serialization/parsing
  - Message passing with Strings
- … is very bad since:
  - it circumvents static type checking
  - it is hard to understand and check as type information is missing

[source: stackoverflow.com/questions/2349378]

Single Responsibility Principle

- Imagine that four classes are involved in the game filter functionality.
- Of these four classes, three are also involved in the players list functionality
- … now, if you change the filters functionality, how many classes do you have to look at?
- … what will happen with the players list functionality if you change the filters? Will it still work?

Single Responsibility Principle

- The **complexity** of code that do not follow SRP tend to **explode** as they evolve
- Making a design decision that doubles complexity of code **n times** makes the code quite complex:

$$\mathbf{2^n}\text{ times as complex}$$

- You will have to constantly **firefight** this complexity

Single Responsibility Principle

- Every object in your system should have a **single responsibility**, and all the object‘s services should be focused on carrying out that single responsibility

- Classes that follow SRP have **only one reason to change**
- They are therefore much easier to maintain and extend.
- … and they don‘t explode.

Open/Closed Principle

- Classes should be **open** for extension, but **closed** for modification

- Subclassing should allow to add behavior, but not to change the behavior of superclasses
- Also: Favor composition over inheritance – designs using composition are more flexible (think observer pattern)

Liskov Substitution Principle

- If B extends A, then objects of type A may be replaced by objects of type B.

- Of course, the Java type system lets you do that – but will the system still **behave** the same? If it doesn‘t, your code violates LSP.
- LSP gives your system behavioral stability in the face of change and extensions.
- LSP is less constraining than Open/Closed principle

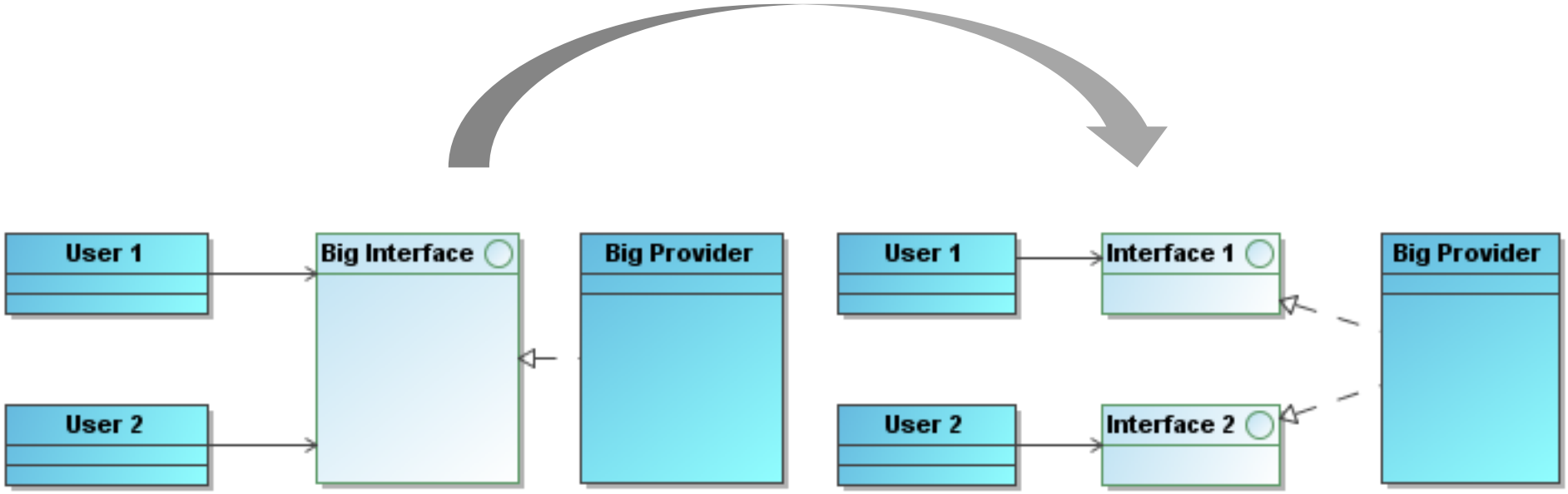
Interface Segregation Principle

- The dependency of one class to another one should depend on the smallest possible interface.

- Makes code easier to read
- Prevents introduction of invalid dependencies
- Prevents extensive re-compilation on changes that affect only parts of the clients

Interface Segregation Principle

- The dependency of one class to another one should depend on the smallest possible interface.



Depending on one big interface

Depending on small interfaces

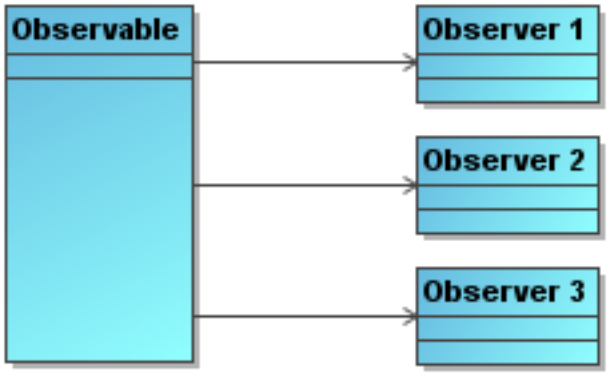
Dependency Inversion Principle

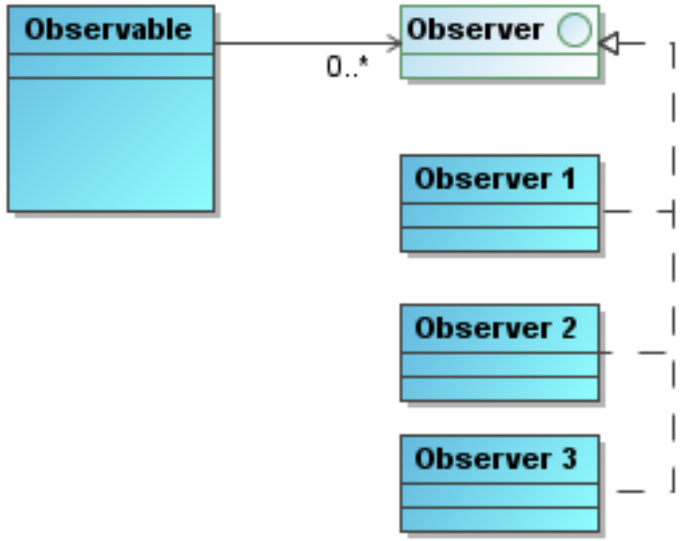
- Depend upon abstractions. Do not depend upon concrete classes.

- Depending on concrete classes makes it hard to exchange them
- Depending on concrete classes may break abstraction layers and prohibit re-use (e.g. a framework depending on a plug-in)

Dependency Inversion Principle

- Depend upon abstractions. Do not depend upon concrete classes.





Depending on concrete classes

Depending on abstraction

Wrap Up:

- We have talked about
  - Bad design
  - Technical debt
- We have discussed OO principles
  - DRY
  - SOLID
- We have seen one antipattern
  - Stringly typed code
  - There are many others!