Advanced Java

LESSON 8: SINGLE RESPONSIBILITY PRINCIPLE & OPEN/CLOSED PRINCIPLE & LISKOV SUBSTITUTION PRINCIPLE

Creative Commons

Attribution 4.0 International (CC BY 4.0)

This workforce product was funded by a grant awarded by the U.S. Department of Labor's Employment and Training Administration. The product was created by the grantee and does not necessarily reflect the official position of the U.S. Department of Labor. The U.S. Department of Labor makes no guarantees, warranties, or assurances of any kind, express or implied, with respect to such information, including any information on linked sites and including, but not limited to, accuracy of the information or its completeness, timeliness, usefulness, adequacy, continued availability, or ownership. This is an equal opportunity program. Assistive technologies are available upon request and include Voice/TTY (771 or 800-947-6644).

Agenda

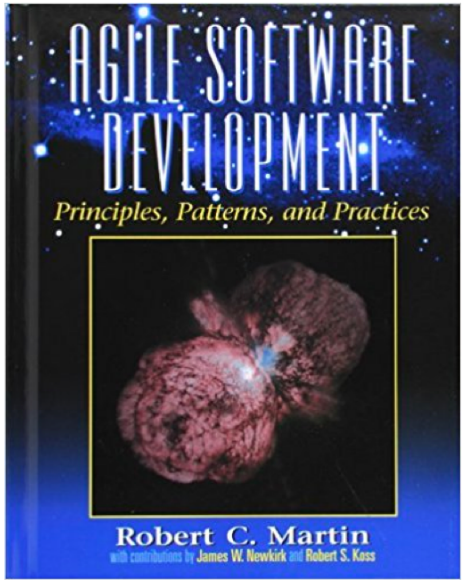
- Who is Uncle Bob?
- Why Concrete Inheritance is so Dangerous?
- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle

Who is Uncle Bob?

Who is Uncle Bob?

Robert Cecil Martin, Uncle Bob, is a leading software engineer and author.

- Co-author of the Agile Manifesto
- Primary promoter of SOLID



Why Concrete Inheritance is so Dangerous?

Why Concrete Inheritance is so Dangerous?

There are two types of Inheritance:

1. **Concrete:** when you inherit methods with a body (implementation). Technically this is called "implementation inheritance"
2. **Abstract:** when you inherit methods that are abstract (no body, no implementation). Technically this is called "interfaces inheritance" – not because it is limited to interfaces, but because abstract methods, whether in an abstract class or interface, represent ways to interface with objects.

Why Concrete Inheritance is so Dangerous?

Interface Inheritance

Does not mean you are inheriting from an Interface (although you might be, but it could also be an abstract class). It means you are inheriting abstract methods that represent only the interface (the abstract method) to a class, not the implementation.

Why Concrete Inheritance is so Dangerous?

Premise #1

You are most likely not the only developer that will use your code. In open source projects and when employed as a commercial programmer you are almost always a member of a team of programmers who share code with each other. Don’t assume they will understand or use your code correctly. Plus if you supply bad code, they will become victims of that malady.

Why Concrete Inheritance is so Dangerous?

Premise #2

Inheriting concrete methods (non-abstract) can be dangerous in many ways, however, this does not mean you should never take advantage of this feature. But you must at least understand the risks. Ideally you should prefer abstract inheritance if you want to write safe code.

Why Concrete Inheritance is so Dangerous?

Premise #3

Despite your abundant talent and best efforts you are not perfect; you will make mistakes. Don’t assume that the list of dangers below can be avoided due to the talent of you and your team.

Why Concrete Inheritance is so Dangerous?

Goals

Just a reminder – in this course we define high-quality code as code that is not rigid, not fragile and is very portable and flexible. Concrete inheritance makes these goals hard to achieve reliably.

Why Concrete Inheritance is so Dangerous?

A partial list of the dangers of concrete inheritance

1. Concrete methods are inherited invisibly, so you don't know what you are inheriting without spending a great deal of time researching how the super class works. If you don't know a feature exists, you can't use it. What else could go wrong?
2. The code you are inheriting may have bugs or be written poorly, affecting performance and memory utilization. How would you know? What must you do?

Why Concrete Inheritance is so Dangerous?

A partial list of the dangers of concrete inheritance

3. There's a chance you may override an inherited concrete method in a way not anticipated by the author of the original method in the super class. This may cause other code to break that depends on the original implementation.
4. With concrete inheritance you may be inheriting methods that you don't need or don't want. This adds to code bloat and consumes more memory and more effort on the part of the programmer to understand all of these unneeded features.

In-Class Activity

As a class, review "InheritanceDangers" sample project on Blackboard.

Single Responsibility Principle

Single Responsibility Principle

A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE.

What is a Responsibility?

“In the context of the Single Responsibility Principle (SRP) we define a responsibility to be ‘a reason for change.’ If you can think of more than one motive for changing a class, then that class has more than one responsibility. This is sometimes hard to see. “

Uncle Bob, SRP: The Single Responsibility Principle

Single Responsibility Principle

"This principle was described in the work of Tom DeMarco and Meilir Page-Jones. They called it **cohesion**. They defined **cohesion** as the functional relatedness of the elements of a module. In this chapter we'll shift that meaning a bit, and relate cohesion to the forces that cause a module, or a class, to change."

Uncle Bob, SRP: The Single Responsibility Principle

Single Responsibility Principle

"The SRP is one of the simplest of the principles, and one of the hardest to get right. Con-joining responsibilities is something that we do naturally. Finding and separating those responsibilities from one another is much of what software design is really about."

Uncle Bob, SRP: The Single Responsibility Principle

Open/Closed Principle

Open/Closed Principle

Classes should be open to extension, but closed to change.

- This means that you should be able to add new concrete methods to a class without breaking other people's code.
- Note that you cannot add new abstract methods because those require concrete declarations in sub classes, meaning change.
- You cannot change existing methods, properties or even the name of the class, without the risk that you might break other people's code.

Bertrand Meyer
Professor ETH Zürich

Open/Closed Principle

Rules of thumb:

- Make all concrete methods final unless you have good reasons to violate the Open/Closed Principle, and then only if you thoroughly document how to override such methods reliably.
- Never add abstract methods to an Interface or Abstract class after those are in production. Rather, create new, additional Interfaces or Abstract classes to deal with those additions.

Open/Closed Principle

Strategic Closure

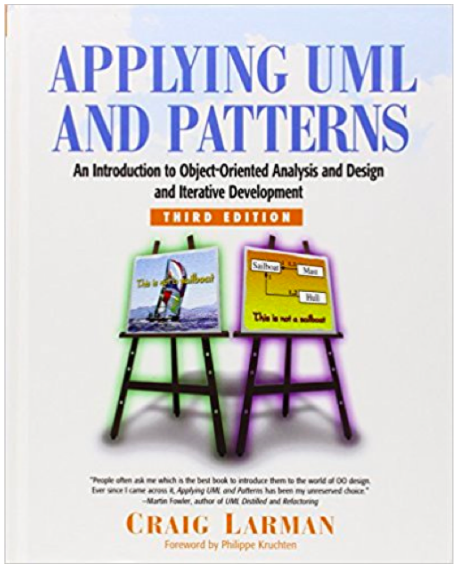
"Since closure cannot be complete, it must be strategic. That is, the designer must choose the kinds of changes against which to close his design. This takes a certain amount of prescience derived from experience. The experienced designer knows the users and the industry well enough to judge the probability of different kinds of changes."

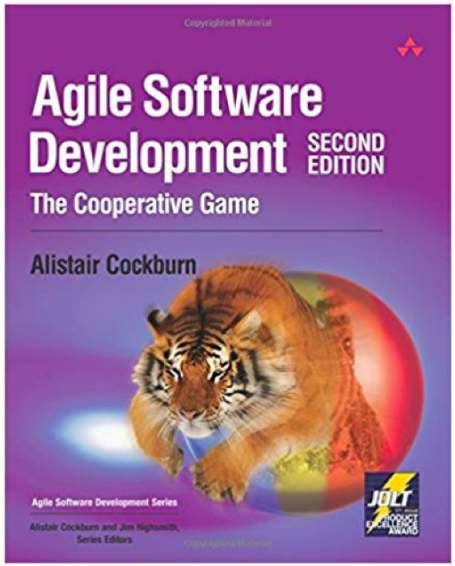
Uncle Bob, The Open-Closed Principle

Open/Closed Principle

According to Craig Larman, the open/closed principle is very similar to:

- **Protected Variations** – by Alistair Cockburn
- **Information Hiding** – by David Parnas





Liskov Substitution Principle

Liskov Substitution Principle

"What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."

Barbara Liskov
MIT Professor

Barbara Liskov, SRP: The Single Responsibility Principle

Liskov Substitution Principle

"It states that well designed code can be extended without modification; that in a well designed program new features are added by adding new code, rather than by changing old, already working, code."

Uncle Bob, SRP: The Single Responsibility Principle

Liskov Substitution Principle

A class property or method parameter or even just a regular variable should be type according to a super class so that the actual type of the object doesn't matter.

```
// Not compliant. Dog is not a super class
Dog dog = new Dog();
dog = new Cat();
// illegal, not flexible
// Compliant - assumes Dog and Cat inherit from Animal
Animal animal = new Dog();
animal = new Cat();
// legal and flexible
```

In-Class Activity

See the sample project named "InterfaceAbstractLab" on Blackboard. There are two packages named "lab1" and "lab2". Inside each package is a README.txt file. Instructions for completing these labs are in this file. Read the instructions carefully and completely. What you don't finish in class needs to be completed and posted to GITHub.