DigiPen
INSTITUTE OF TECHNOLOGY
SINGAPORE

[CS 225] Advanced C/C++

Lecture 7: S.O.L.I.D. Principles

Sławomir "Swavek" Włodkowski
(Summer 2020)

Agenda

- Software development
- Goals of software design
- Sources of good practices
- Code smells
- S.O.L.I.D. Principles
  - SRP
  - OCP
  - LSP
  - ISP
  - DIP

Software development

Being a software developer does not mean just developing the code. The software development life cycle (SDLC) includes other stages too:

- Inception (triggered by new business needs),
- Requirements analysis
- Design,
- Development (implementation),
- Testing,
- Deployment,
- Maintenance.

Goals of software design

Being able to design the code before it is implemented gives us measurable benefits:

- Let's us build a consistent picture of the software being built between all team members.
- Let's us avoid poor architectural structure (***architecture*** is the part of a product that is difficult to change).
- Let's us deal with software's ***complexity*** (feature rich software has a lot of parts).
- Let's us deal with the challenges of the software development process and the code itself being ***difficult*** (unfamiliar, as new code, or someone else's code, always is).

Goals of software design

We want development of ***complex*** software to be ***easy***.

Goals of software design

Other benefits:

- Increases reuse.
- Reduces size of code base.
- Reduces potential of mistakes.
- Introduces clear boundaries between parts of a system.
- Enables places for extension.
- Increases security and fault-tolerance.
- Increases functionality and performance of a system.

Sources of good practices

- Following suggestions of other practitioners.
- Asking software development communities for advice.
- Researching literature and other media.
- Following idioms and design patterns.
- Applying accepted policies and principles.

Some software engineers gained recognition for promoting sound and helpful practices, among them Robert C. Martin (a.k.a. "Uncle Bob").

Sources of good practices

Robert C. Martin, “Uncle Bob” is a renown software engineer, instructor and advocate of good software development practices. Author of books (*“Clean code”*), articles, and technical talks.

Best known for coauthoring the ***Agile Manifesto*** and promoting software design principles, with the main five referred to as the **S.O.L.I.D. principles**.

Code smells

Often referred to by Robert C. Martin and other authors related to agile design and design practices:

> “Design smells are the odours of rotting software”

A **code smell** is a characteristic of the source code that possibly indicates a deeper design problem.

Code smells

Rigidity – a program is hard to change, every change forces further changes to other parts of the system.

Fragility – a program tends to break easily seemingly as a result of making unrelated changes.

Immobility – parts of a program are easier to rewrite than to reuse in similar places in the code.

Viscosity – changes to the code impact the design; hacking is easier than following the design principles.

Code smells

Needless complexity – the design includes parts that are currently not useful (for example, foreseen for future use).

Needless repetition – parts of a program could be reused but are not.

Opacity – code is difficult to understand, not clear, not implemented in a direct, sound manner.

S.O.L.I.D. Principles

Established by Robert C. Martin's paper "*Design Principles and Design Patterns*" (2000), although introduced earlier.

The principles aim to promote design that makes software:

- Understandable
- Flexible
- Maintainable
- Free of code smells

S.O.L.I.D. Principles

SRP
Single-responsibility principle

For a class (also a module, function, etc.), there should be only one reason to change; only changes to one part of the specification should affect the design of the class.

S.O.L.I.D. Principles

SRP deals both with the **separation of concerns** (unrelated operations should be implemented in different places), as well as with **decomposition** of variant functionalities into their individual classes.

Implementation classes do not have to represent real-world entities; they can represent abstract notions, such as an action, a style, or even implementation concepts, such as a proxy, a selector, etc.

S.O.L.I.D. Principles

OCP
Open/close principle

A class (also a module, function, etc.) should be:

- open for extension,
- closed for modification.

S.O.L.I.D. Principles

According to **OCP**, code should be **closed for modification**, meaning it should be given a well-defined role through its programming interface and non-virtual behaviour.

If the code represents a non-interface class, encapsulation, `const` and `final` keywords are some of the tools that help developers protect its state from modification.

S.O.L.I.D. Principles

According to **OCP**, code should be also **open for extension**, such that adding new members and functionality should be possible as long as it does not contradict or invalidate the original design of the code.

Inheritance and polymorphism are often applied to achieve these effects in C++.

S.O.L.I.D. Principles

LSP
Liskov substitution principle

Code that is valid for an object of a base type, should be valid for an object of a derived type.

S.O.L.I.D. Principles

In line with **LSP**, a base class or an interface establishes a contract that lets a user make certain assumptions about any derived class.

A derived class should not implement this inherited interface changing its meaning.

A derived class should also not implement this interface if its operations lead to invalidating the object's state.

S.O.L.I.D. Principles

Example of **LSP**: a square is not a rectangle!

S.O.L.I.D. Principles

ISP
Interface segregation principle

Code should rely on the smallest possible interface related to a given functionality.

S.O.L.I.D. Principles

ISP informs us to break do-all interfaces into smaller interface segregating unrelated functionality.

Avoid "fat“, “do-all” interfaces (abstract or not) and replace them with role-specific interfaces (preferably abstract interface classes).

S.O.L.I.D. Principles

DIP
Dependency inversion principles

High-level code should not limit low-level code; both should depend on abstracts, not specific types.

S.O.L.I.D. Principles

If we follow **DIP**, concrete implementations should depend on abstractions so that the specific implementation may come later and be ignored for the most part of the design. Design should be based on the abstracts (typically, interface classes).

This "inverts" typical object-oriented class organization, where objects reference and create other objects; hence the name of the principle.