SOLID
CSC207 Winter 2018
SOLID Principles of Object-Oriented Design

• How do we make decisions about what is better and what is worse design?

• Principles to aim for instead of rules.
  • e.g. there is no maximum or minimum number of classes you should have in your program. But if the number of classes violates a generally accepted principle, you should reconsider your class structure.

• The SOLID principles are useful and cover most major situations you are likely to encounter.
Software Design Goals

- A major goal when programming is to write a program that is:
  - easy-to-read
  - hard-to-break
  - maintainable
  - efficient

- Software design has you use a set of principles and techniques that help you do this.
- These help you make software design decisions and defend them.
Fundamental OOP Concepts

- **Abstraction** — the process of distilling a concept to a set of essential characteristics.
- **Encapsulation** — the process of binding together data with methods that manipulate that data, and hiding the internal representation.
- The result of applying abstraction and encapsulation is (often) a class with instance variables and methods that together model a concept from the real world.
- **Inheritance** — the concept that when a subclass is defined in terms another class, the features of that other class are inherited by the subclass.
- **Polymorphism** (“many forms”) — the ability of an expression (such as a method call) to be applied to objects of different types.
Fundamental OOD goals: low coupling, high cohesion

• **Coupling** — how much a class is directly linked to another class
  • High coupling means that changes to one class may lead to changes in several other classes.
  • Low coupling is, therefore a desired goal.

• **Cohesion** — how much the features of a class belong together.
  • Low cohesion means that methods in a class operate on unrelated tasks. This means the class does jobs that are unrelated.
  • High cohesion means that the methods have strongly-related functionality.
Fundamental OOD principles

SOLID: five basic principles of object-oriented design (Developed by Robert C. Martin, affectionately known as “Uncle Bob”.)

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle
Single Responsibility Principle
(“Each class does one thing and does it well”)

- Every class should have a single responsibility.
- Another way to view this is that a class should only have one reason to change.
- But who causes the change?
Single Responsibility Principle
(“Each class does one thing and does it well”)

• But who causes the change?
  • “This principle is about people. ... When you write a software module, you want to make sure that when changes are requested, those changes can only originate from a single person, or rather, a single tightly coupled group of people representing a single narrowly defined business function. You want to isolate your modules from the complexities of the organization as a whole, and design your systems such that each module is responsible (responds to) the needs of just that one business function.”

• “Gather together the things that change for the same reasons. Separate those things that change for different reasons.”

• [Uncle Bob, The Single Responsibility Principle]
Open/Closed Principle
(“Open for extension, closed for modification.”)

- Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.

- Add new features not by modifying the original class, but rather by extending it and adding new behaviours, or by adding **plugin** capabilities.

- “I’ve heard it said that the OCP is wrong, unworkable, impractical, and not for real programmers with real work to do. The rise of plugin architectures makes it plain that these views are utter nonsense. On the contrary, a strong plugin architecture is likely to be the most important aspect of future software systems.” [Uncle Bob, *The Open Closed Principle*]
Open/Closed Principle
(“Open for extension, closed for modification.”)

• An example, using inheritance:

  area calculates the area of all Rectangles in the input.

• What if we need to add more shapes?

<table>
<thead>
<tr>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>- width: double</td>
</tr>
<tr>
<td>- height: double</td>
</tr>
<tr>
<td>+ getWidth(): double</td>
</tr>
<tr>
<td>+ getHeight(): double</td>
</tr>
<tr>
<td>+ setWidth(w: double): void</td>
</tr>
<tr>
<td>+ setHeight(h: double): void</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AreaCalculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ area(shapes: Rectangle []): double</td>
</tr>
</tbody>
</table>
Open/Closed Principle (simplified)

With this design, we can add any number of shapes (open for extension) and we don't need to re-write the AreaCalculator class (closed for modification).
Liskov Substitution Principle
(“The child class must uphold the spirit of the parent class”)

- “Replacing a parent with the child should not break your code.”
- If S is a subtype of T, then objects of type S may be substituted for objects of type T, without altering any of the desired properties of the program.
- “S is a subtype of T”?
  - In Java, S is a child class of T, or S implements interface T.
  - For example, if C is a child class of P, then we should be able to substitute C for P in our code without breaking it.
Liskov Substitution Principle
(“The child class must uphold the spirit of the parent class”)

• A classic example of breaking this principle:
  • In OO programming and design, unlike in math, it is not the case that a Square is a Rectangle!
  • This is because a Rectangle has more behaviours than a Square, not less.
  • The LSP is related to the Open/Closed principle: the subclasses should only extend (add behaviours), not modify or remove them.
Interface Segregation Principle
(“Better to have many small interfaces than one big one.”)

- Here, interface means the public methods in a class. (In Java, these are often specified using a Java interface.)
- Context: a class that provides a service for other “client” programmers usually requires that the clients write code that has a particular set of features. The service provider says “your code needs to have this interface”.
- No client should be forced to implement irrelevant methods of an interface. Better to have lots of small, specific interfaces than fewer larger ones: easier to extend and modify the design.
- (Aside: “The interface keyword is harmful.” [Uncle Bob, 'Interface' Considered Harmful])
Dependency inversion principle
(“High-level modules should not depend on low-level modules; both should depend on abstractions”)

- “Abstractions should not depend on details; details should depend on abstractions”
- When building a complex system, programmers are often tempted to define “low-level” classes first and then build “higher-level” classes that use the low-level classes directly.
- But this approach is not flexible! What if we need to replace a low-level class? The logic in the high-level class will need to be replaced — an indication of high coupling.
- To avoid such problems, we introduce an abstraction layer between low-level classes and high-level classes.
Dependency inversion principle
(“High-level modules should not depend on low-level modules; both should depend on abstractions”)

• Goal:
  • You want to decouple your system so that you can change individual pieces without having to change anything more than the individual piece.

• Two aspects to the dependency inversion principle:
  • High-level modules should not depend on low-level modules. Both should depend on abstractions.
  • Abstractions should not depend upon details. Details should depend upon abstractions.
Dependency inversion principle
(“High-level modules should not depend on low-level modules; both should depend on abstractions”)

• Example: from Dependency Inversion Principle on OODesign
  • You have a large system, and part of it has Managers manage Workers. Let’s say that the company is restructuring and introducing new kinds of workers, and wants the code updated to reflect this.
  • Your code current has a Manager class and a Worker class, and the Manager class has several methods that have Worker parameters.
  • Now there’s a new kind of worker called SuperWorker, and their behaviour and features are separate from regular Workers.
  • Oh dear ...
Dependency inversion principle
(“High-level modules should not depend on low-level modules; both should depend on abstractions”)

- To make Manager work with SuperWorker, we would need to rewrite the code in Manager.

- Solution: create an IWorker interface and have Manager use it.
Dependency inversion principle
(“High-level modules should not depend on low-level modules; both should depend on abstractions”)

• In this design, Manager does not know anything about Worker, nor about SuperWorker. It can work with any IWorker, the code in Manager does not need rewriting.