Design Principles I

Ergude Bao

Beijing Jiaotong University

Contents

- Symptoms of Poor Design
- Single-Responsibility Principle (SRP)

Symptoms of Poor Design

Symptoms of Poor Design

- Design smells/odors of rotting software
  - Rigidity
  - Fragility
  - Immobility
  - Viscosity
  - Needless complexity
  - Needless repetition
  - Opacity

Rigidity Smell

- A design is rigid if a single change causes a cascade of subsequent changes in dependent modules
  - The more modules to change, the more rigid the design is
- Root causes are high coupling and low cohesion

Coupling

- Coupling describes how dependent one module is on another module (that it uses)
  - Modules that are loosely coupled can be changed quite radically without impacting each other
  - The slightest change to modules that are tightly coupled can cause a host of problems

Coupling

- class A {
- int x;
- ...
- }
- class B extends A {
- void b() {
- x = 5;
- }
- }

Problems with High Coupling

- Change in one module forces changes in other modules
  - Rigidity and fragility
- Modules are difficult to be reused or tested because dependent modules must be included
  - Immobility
- Modules are difficult to be understood in isolation
  - Opacity

Example Solutions

- "Favor object composition over class inheritance" and "Program to an interface, not an implementation"
  - Erich Gamma (GoF): http://c2.com/cgi/wiki?GangOfFour
  - The emphasis on inheritance and classes can result in some types of undesirable coupling

Cohesion

- Cohesion is a measure of how well the lines of source code within a module work together to provide a specific piece of functionality
- Cohesion should be high (or strong)
- Cohesion is low (or weak) if
  - The responsibilities (methods) of a class have little in common, or
  - Methods carry out many varied activities, often using unrelated sets of data

Problems with Low Cohesion

- Increased difficulty in maintaining a system
  - Rigidity and fragility
- Increased difficulty in reusing a module because most applications will not need the random set of operations provided by a module
  - Immobility
- Increased difficulty in understanding modules
  - Opacity

Fragility Smell

- Changes cause the system to break in places that have no conceptual relationship to the part that was changed
- Same root causes with rigidity smell
- Fragility tends to get worse, and the software gets impossible to maintain
  - Managers (and now also developers) will fear change

Immobility Smell

- It is hard to disentangle the system into components that can be reused in other systems
- Direct cause is modules are not designed for reuse, e.g. when modules depend on infrastructure or when modules are too specialized
- Same root causes with rigidity smell
- The consequence is that software is written from scratch

Viscosity Smell

- If making changes that preserves the design is harder to do than doing “hacks”, the viscosity of the design is high
  - Hacks make the code even more rigid, fragile, immobile…
- Two forms: Viscosity of the design or viscosity of the environment
  - Viscosity of environment comes about when the development environment is slow and inefficient

Needless Complexity Smell

- The design contains infrastructure that adds no direct benefit
- This frequently happen when architect or developers anticipate changes to the requirements, and put in facilities for those potential changes
  - The design will carry the weight of all the unused design elements, and possibly make other changes difficult

Needless Repetition Smell

- The design contains repeating structures that could be unified under a single abstraction
- Makes the software difficult to maintain
  - Any duplication is bad
  - Semi-duplication, code that is almost the same, is even worse

Opacity Smell

- The code is hard to be read and understood. It does not express its intent well
- Reasons
  - Not following a coding standard
  - Bad or inconsistent naming
  - Bad or lacking commenting
  - High coupling and low cohesion
- Some kind of code review should be done to avoid opaque code

What Stimulates the Software to Rot?

- Poor design!
- Short-term-thinking!
- Requirements always change!

Single-Responsibility Principle (SRP)

Design Principles

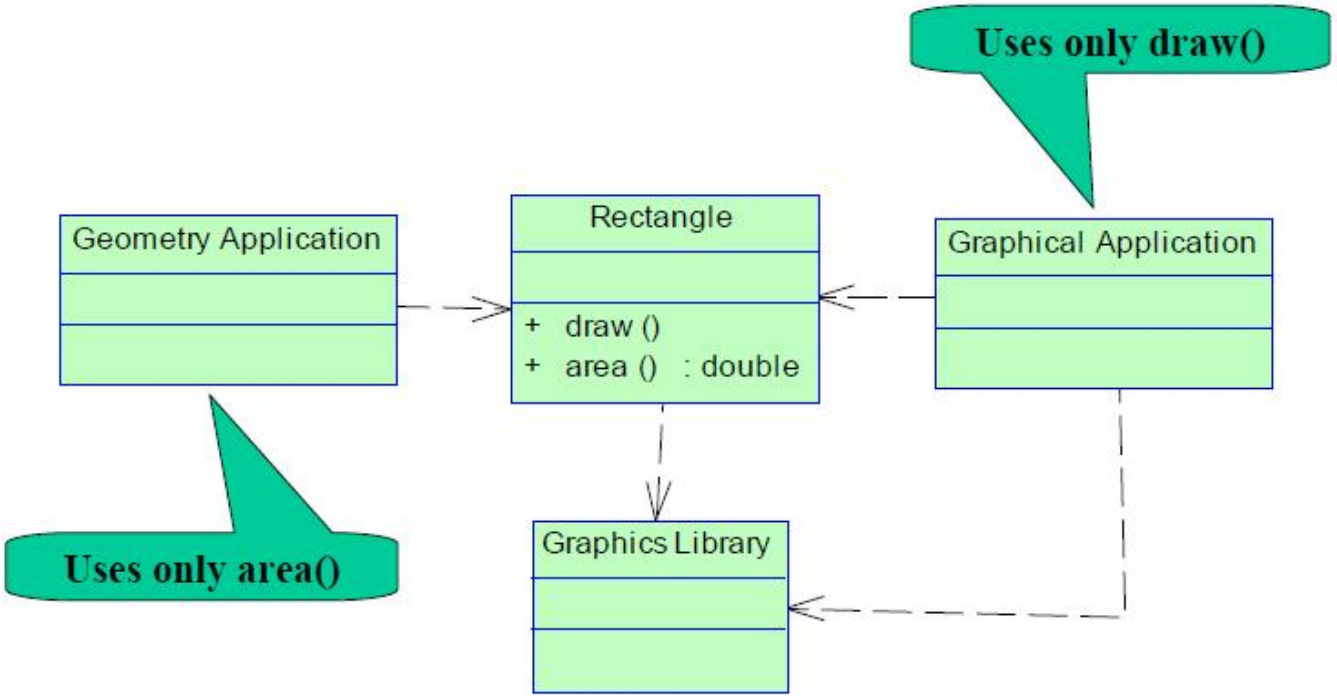
- Software design principles represent a set of guidelines that helps us to avoid having a bad design
- Five design principles
  - Single Responsibility Principle (SRP)
  - Open-Closed Principle (OCP)
  - Liskov Substitution Principle (LSP)
  - Dependency-Inversion Principle (DIP)
  - Interface-Segregation Principle (ISP)

Single Responsibility Principle (SRP)

- A class should have only one reason to change
  - Responsibility = “a reason to change”
  - If a class has more than one responsibility, then the responsibilities become coupled

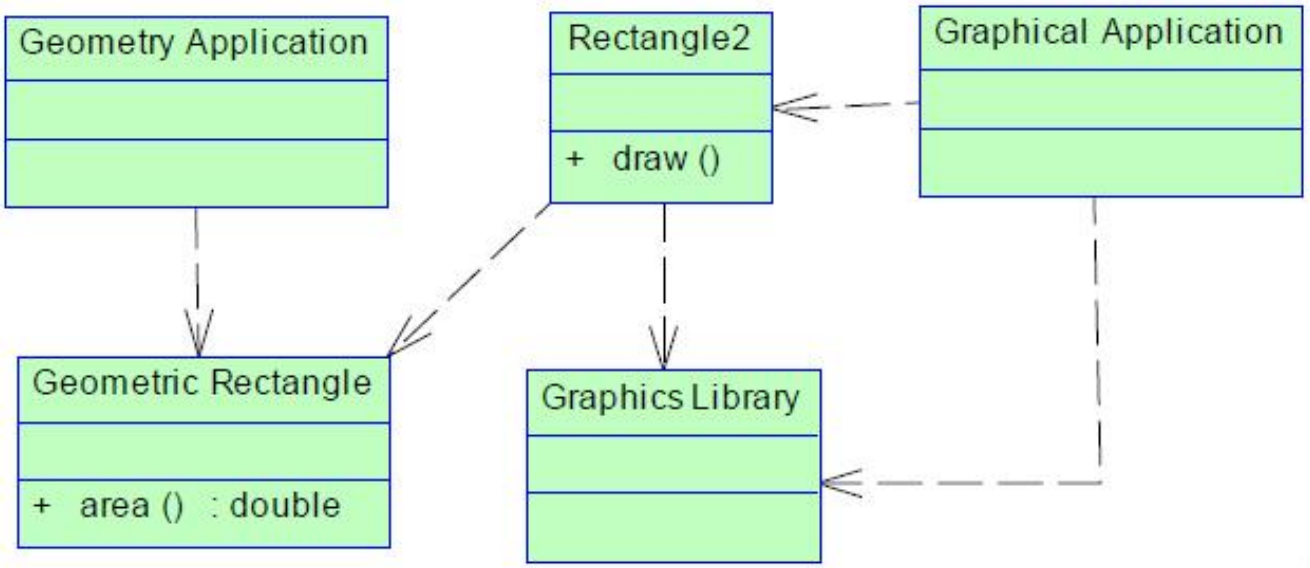
Example I

- Rectangle is a class with two responsibilities
  - Geometrical calculations
  - Drawing an object



Example I

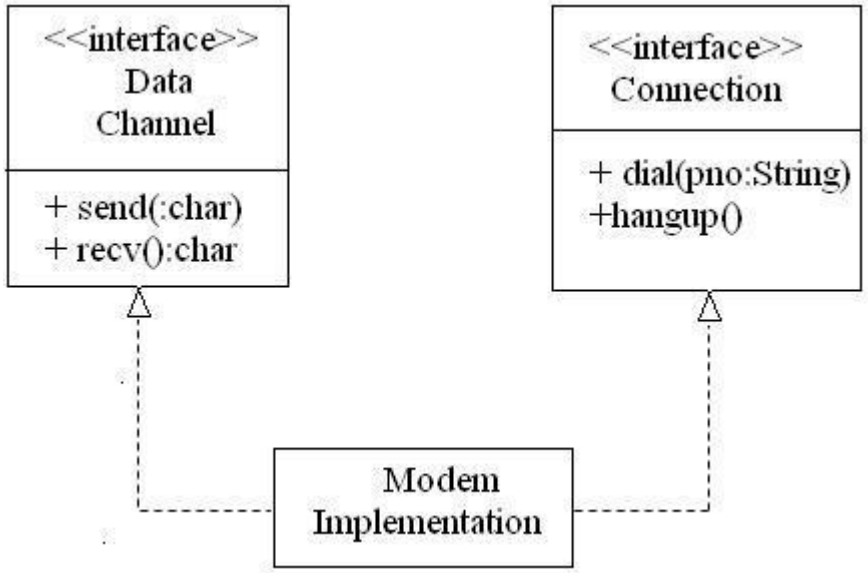
- Separate the responsibility into 2 classes
  - Geometric Rectangle for geometrical calculations
  - Rectangle2 for drawing an object



Example II

```
interface Modem
{
   public void dial(String pno);
   public void hangup();
   public void send(char c);
   public char recv();
}
```

Example II


<<interface>>
Data
Channel
+ send(:char)
+ recv():char
<<interface>>
Connection
+ dial(pno:String)
+hangup()
Modem
Implementation

Note

- Do not separate responsibilities if it is unlikely to have independent changes. Otherwise, the codes will have the needless complexity smell

A broader perspective

- The principle of high cohesion can be applied at different levels. The examples we have seen so far focus primarily on class-cohesion
- We can also talk about cohesion in methods, packages and subsystems
  - An example of method cohesion, see: http://www.javaworld.com/jw-05-1998/jw-05-techniques.html

Questions

- Please give an example that violates SRP and explain why? How to modify it to conform to SRP?