SOLID principles

Overview

- The first five principles of Object Oriented Design (OOD)
- Defined by Robert C. Martin in the early 2000s
- Creating a system that is easy to maintain and extend over time

The principles

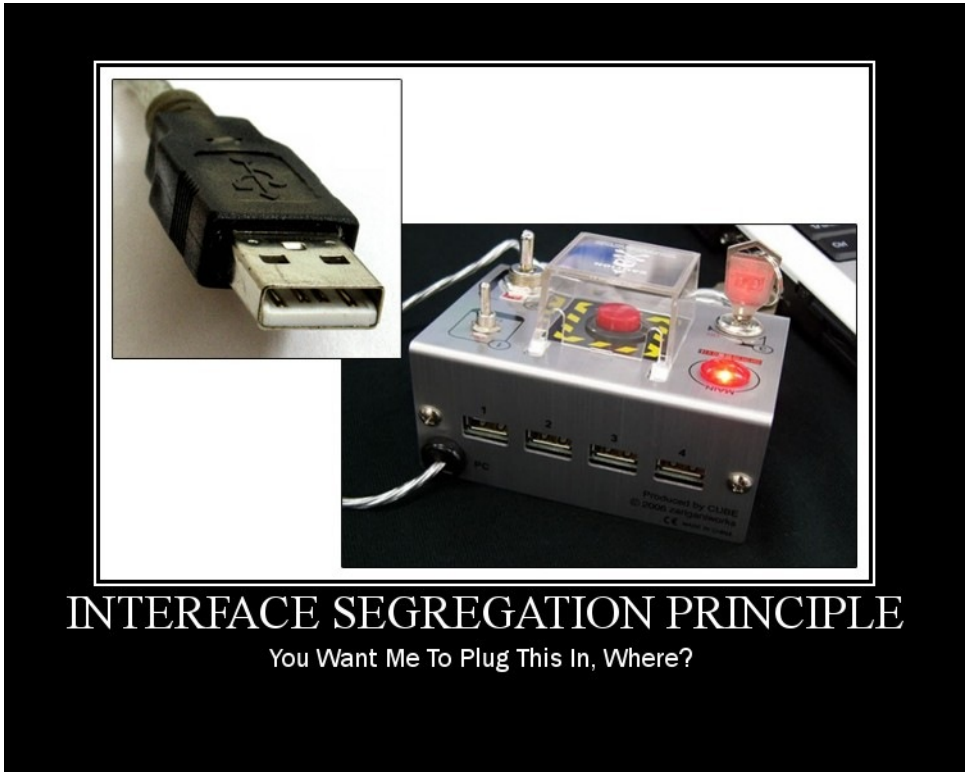
- **(S) SRP - Single Responsibility Principle**
- (O) OCP - Open/Closed Principle
- (L) LSP - Liskov Substitution Principle
- **(I) ISP - Interface Segregation Principle**
- **(D) DIP - Dependency Inversion Principle**

Interface Segregation Principle (ISP)

The common approach

- Low-level module is used by many different high-level modules (clients), all of them having a specific way of working the low-level module.
- This leads to problems:
  - **Interface pollution** - an interface is being polluted with methods that it doesn't require making it "fat“ and non-cohesive.
  - **Degenerate method** implementations in implementing classes.

The problem illustrated



ISP defined

Clients should not be forced to depend on methods they do not use.

Example: Door

Exercises

Exercise: Factory

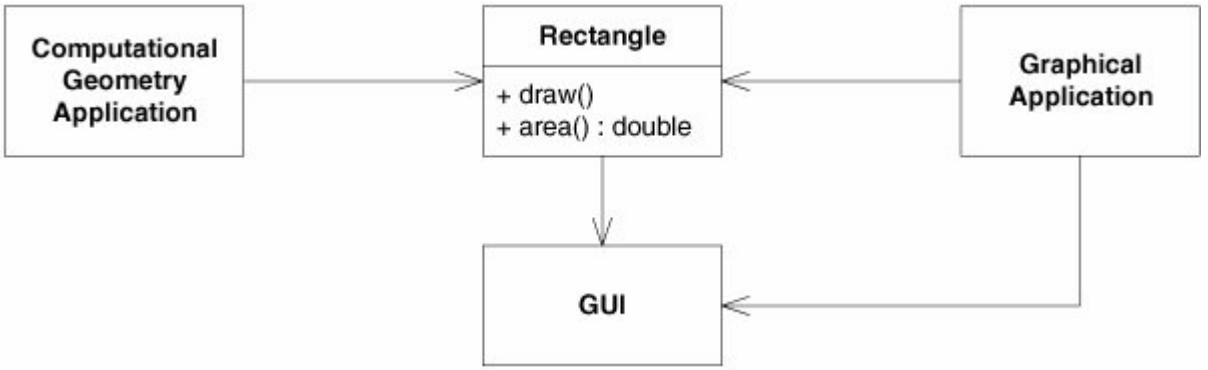
Refactor the Factory application to be compliant with the interface segregation principle.

Single Responsibility Principle (SRP)

The common approach

- Classes tend to have many responsibilities. In a class that has more than one responsibility, the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class's ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed.
- A responsibility is defined as an axis of change. A class has multiple responsibilities if it has more than one reason for change.

Example

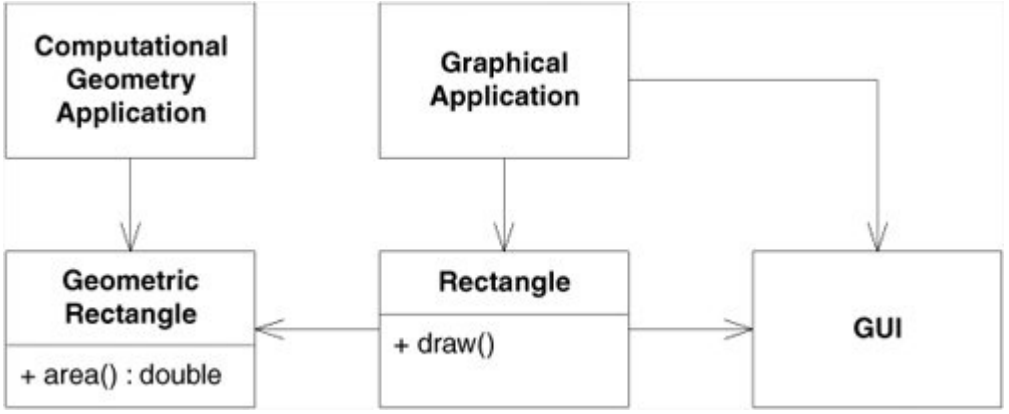


Problems:

- The GUI library needs to be included in the Computational Geometry Application.
- Changes to the Graphical Application that require changes to the Rectangle class will force changes in the Computational Geometry Application as well forcing the application to be rebuilt, retested and redeployed.

SRP defined

A class should have only one reason to change.

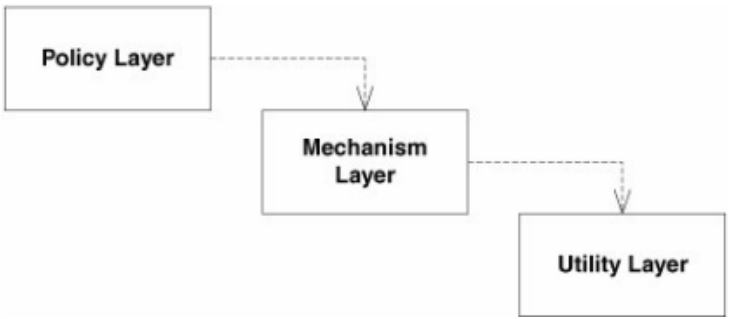


Example: Rectangle

Dependency Inversion Principle (DIP)

The common approach

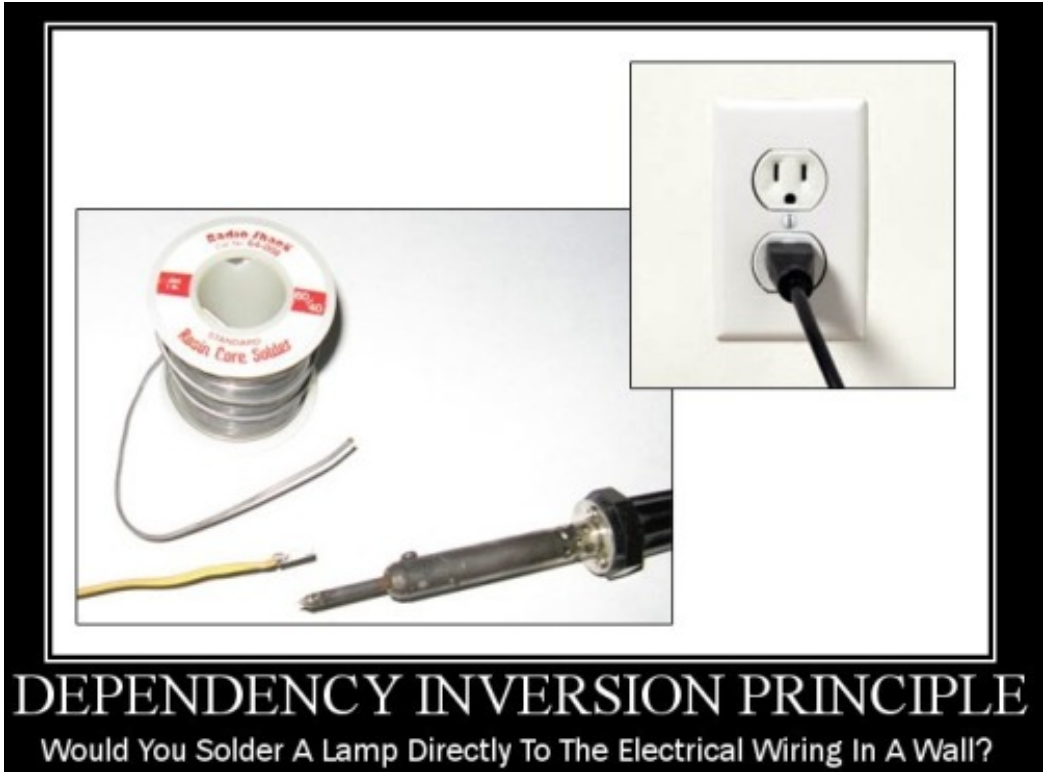
- Traditionally high-level modules have been dependent on low-level modules.
- Examples:
  - User management module would depend on a DAO module that persists the user data in a relational database.
  - Checkout module would depend on a payment gateway such as PayPal.
  - The password management module would depend on a SMTP server for sending the "forgot password" email.



Hence the problems

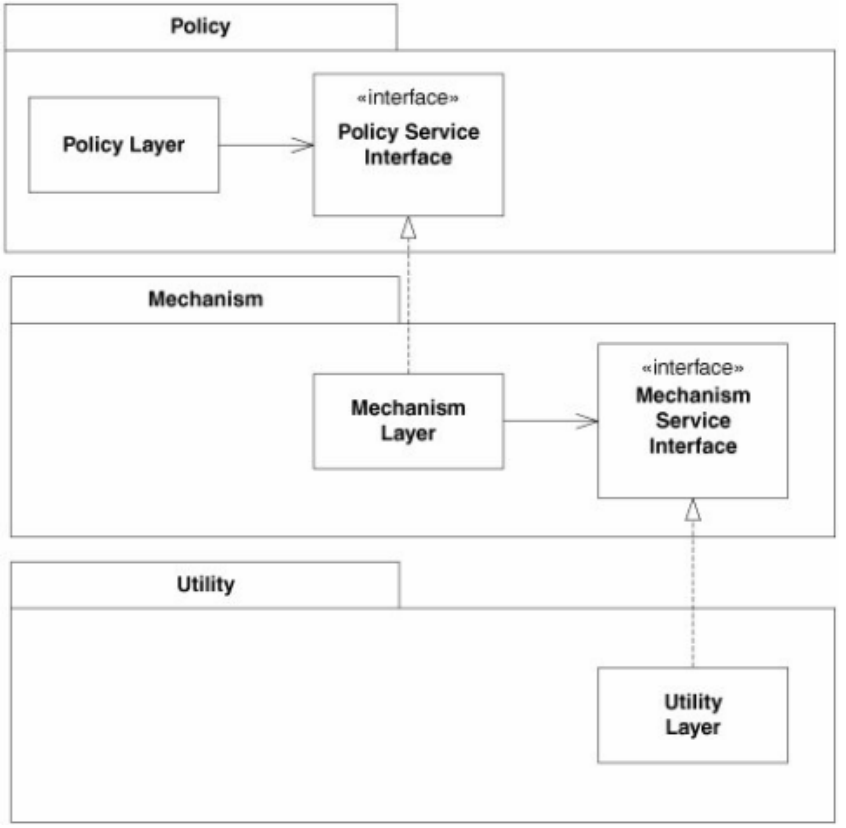
- Changes to low-level modules have direct effects on the high-level modules.
  - Change to the API of a low-level module would require changes in the code of a high-level module that depends on it.
- The tight coupling between modules makes it very hard to replace low-level modules with alternatives. The same is problem when it comes to reusing high-level modules in different contexts.
  - Replacing the DAO module that stores the data in a relational database with one that stores it in a NoSQL database.
  - Replacing PayPal with Google Wallet.
  - Opting for an SMS "forgot password" notification rather than an email based one.

The problem illustrated



DIP defined

- High-level modules should not depend on low-level modules. Both should depend on abstractions that draw the behavior of high-level modules.
- Abstractions should not depend upon details. Details should depend upon abstractions.



Example: ElectricalSystem

Exercises

Exercise: MotionDetector

- Create a motion detector component which once started takes images (byte array) from an image capturing device every one (1) second. In case it detects changes between two subsequent images, it needs to trigger alarm on a number of previously registered alarm channels.
- For testing/simulation purposes, create an image capturing device which simulates image capturing by reading text from the console (System.in) and a single alarm channel which when used writes the “Alarm” text to the console (System.out).
- Group component related classes and interfaces in distinct packages.

Exercise: WaterHeater

- A large industry manufacturer Ventoelectrics has asked your company ACME to provide two thermoregulator components, **standard** and **efficient**, which they intent to use in their water heater production line. They have their own production for the other components, the power switch, the heater, and the thermometer.
- The thermoregulator components should run on its own taking a read from the thermometer, the standard version every three (3) seconds and the efficient every one (1) second. Both enable and disable the heater in respect to the following rules:
  - The heater is disabled if the temperature raises above a configured value.
  - The heater is enabled if the temperature drops below a configured value.
- Group component related classes and interfaces in distinct packages following the “com.<company>.<component>” package organization schema.
- Make sure you’re able to offer your thermoregulator product line to other vendors in the future - your thermoregulator components should not depend on Ventoelectrics’ components (classes and interfaces).

Exercise: WaterHeater – Starting model