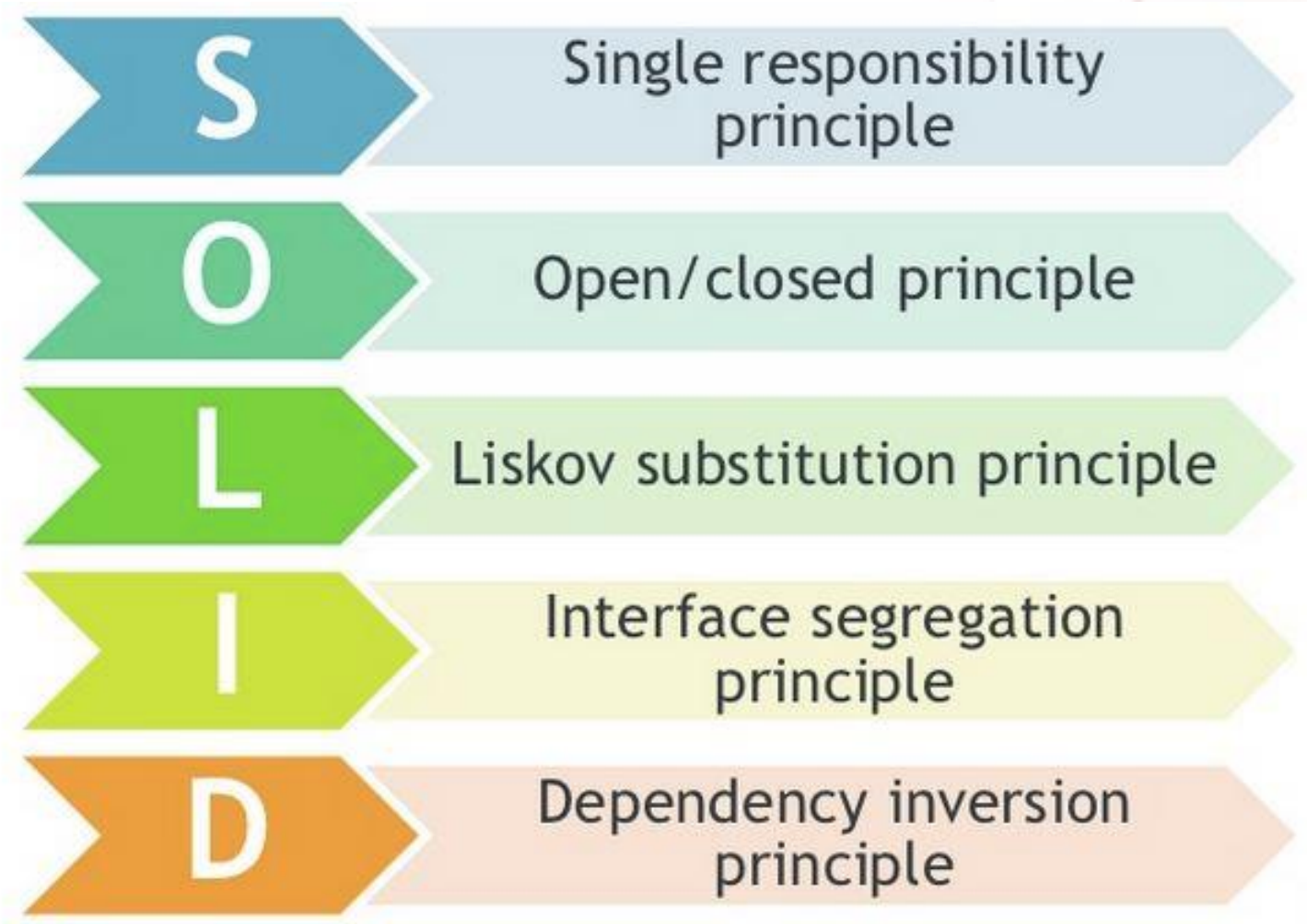
STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

SSW 322: Software Engineering Design VI

SOLID Design Principles
2020 Spring

Prof. Lu Xiao
lxiao6@stevens.edu
Office Hour: Monday/Wednesday 2 to 4 pm
https://stevens.zoom.us/j/632866976
Software Engineering
School of Systems and Enterprises

Today's Topic - SOLID



About SOLID

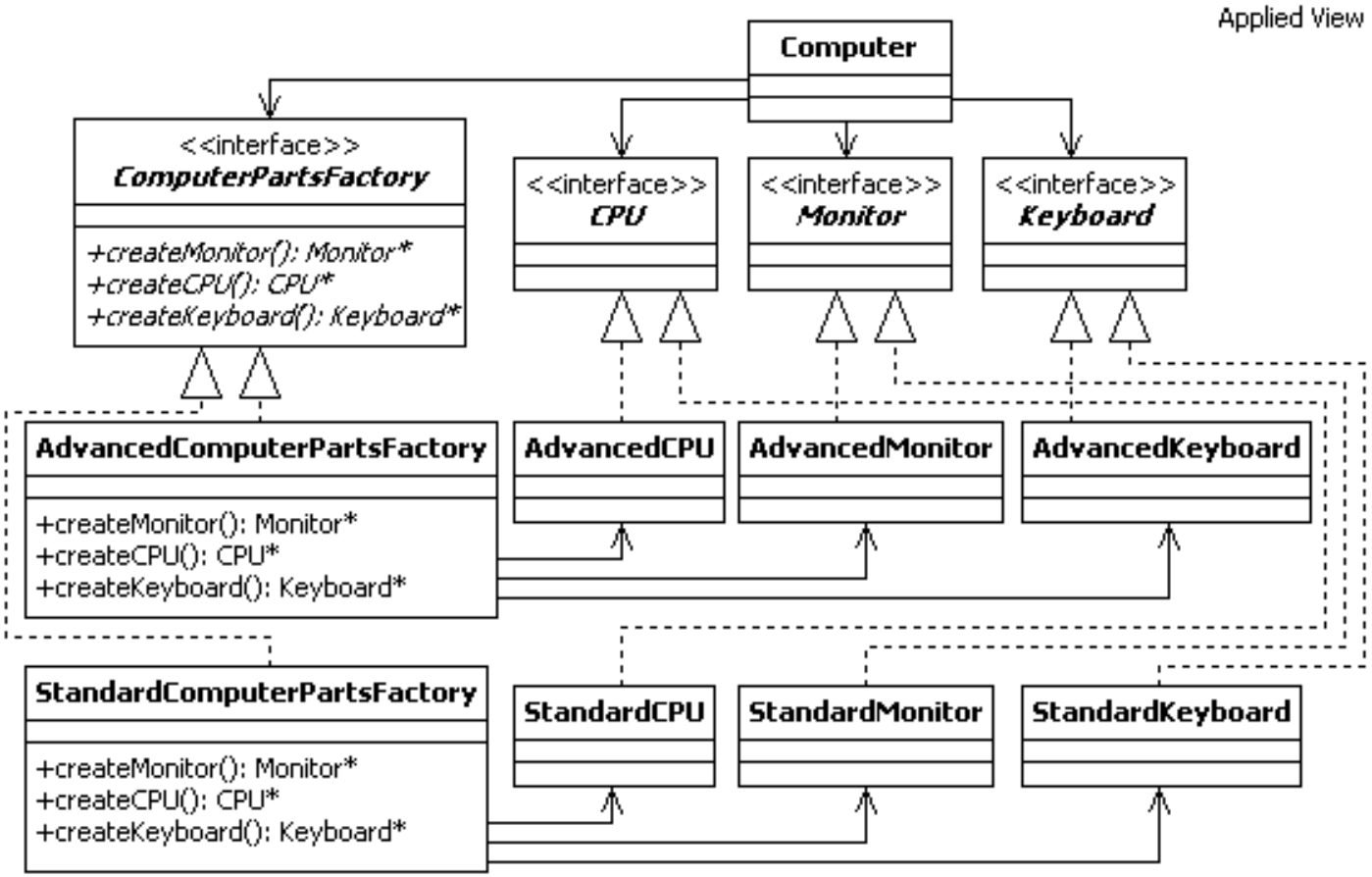
- In object-oriented programming, the term **SOLID** is a mnemonic acronym for five design principles intended to make software designs more understandable, flexible and maintainable.
- Though they apply to any object-oriented design, the SOLID principles can also form a core philosophy for methodologies such as agile development or adaptive software development.
- The SOLID acronym was introduced by Michael Feathers. They are a subset of many principles promoted by Robert C. Martin.

- **Single Responsibility**
  ---Do one thing and do one thing well

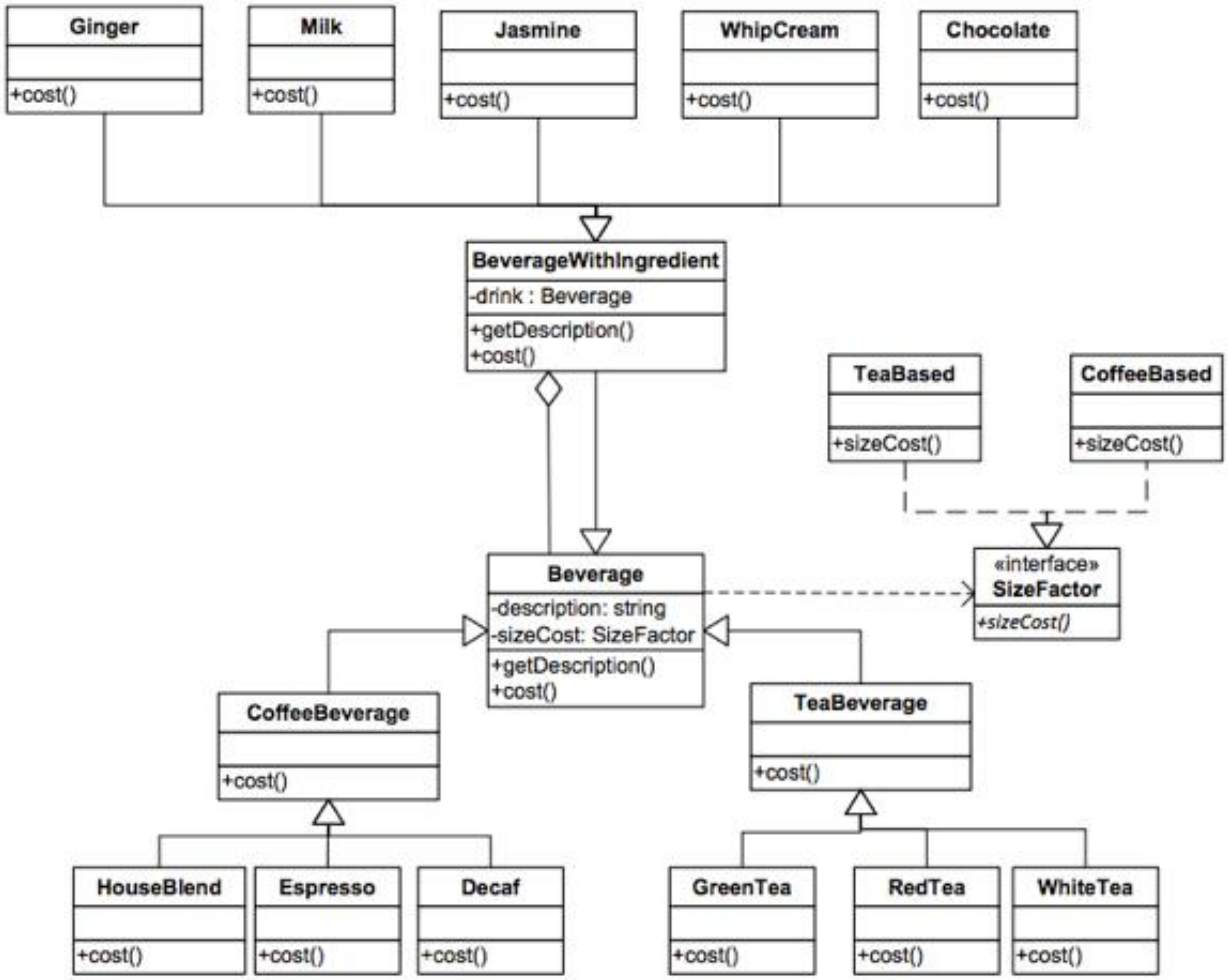
Single Responsibility

- A class (method) should have only a single responsibility (i.e. changes to only one part of the software's specification should be able to affect the specification of the class).
- Robert C. Martin expresses the principle as, "A class should have only one reason to change.“
- It is also related to **separation of concerns** (**SoC**): separating a computer program into distinct sections, such that each section addresses a separate concern.

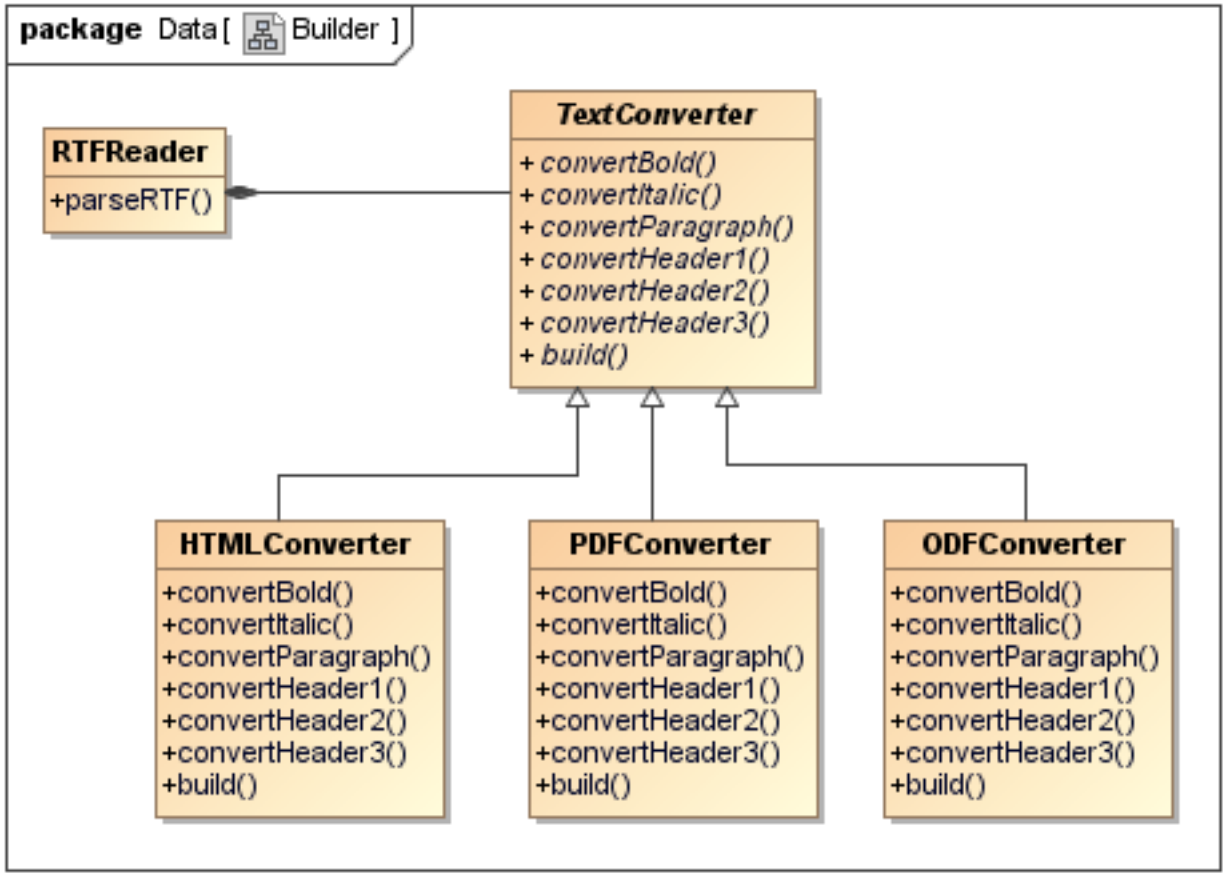
Single Responsibility-Applied (1)



Single Responsibility-Applied (2)



Single Responsibility-Applied (3)



Measure of Single Responsibility

Coupling

Cohesion

Measure of Single Responsibility

Coupling

Coupling refers to the interdependencies between modules.

Cohesion

Cohesion describes how related the functions within a single module are.

Measure of Single Responsibility

Coupling

Coupling refers to the interdependencies between modules,

Cohesion

Cohesion describes how related the functions within a single module are.

Low cohesion implies that a given module performs tasks which are not very related to each other and hence can create problems as the module becomes large.

- **Open Closed Principle**
  ---Open to extension and closed to modification

Open Closed Principle

- Software entities should be open for ?, but closed for ?

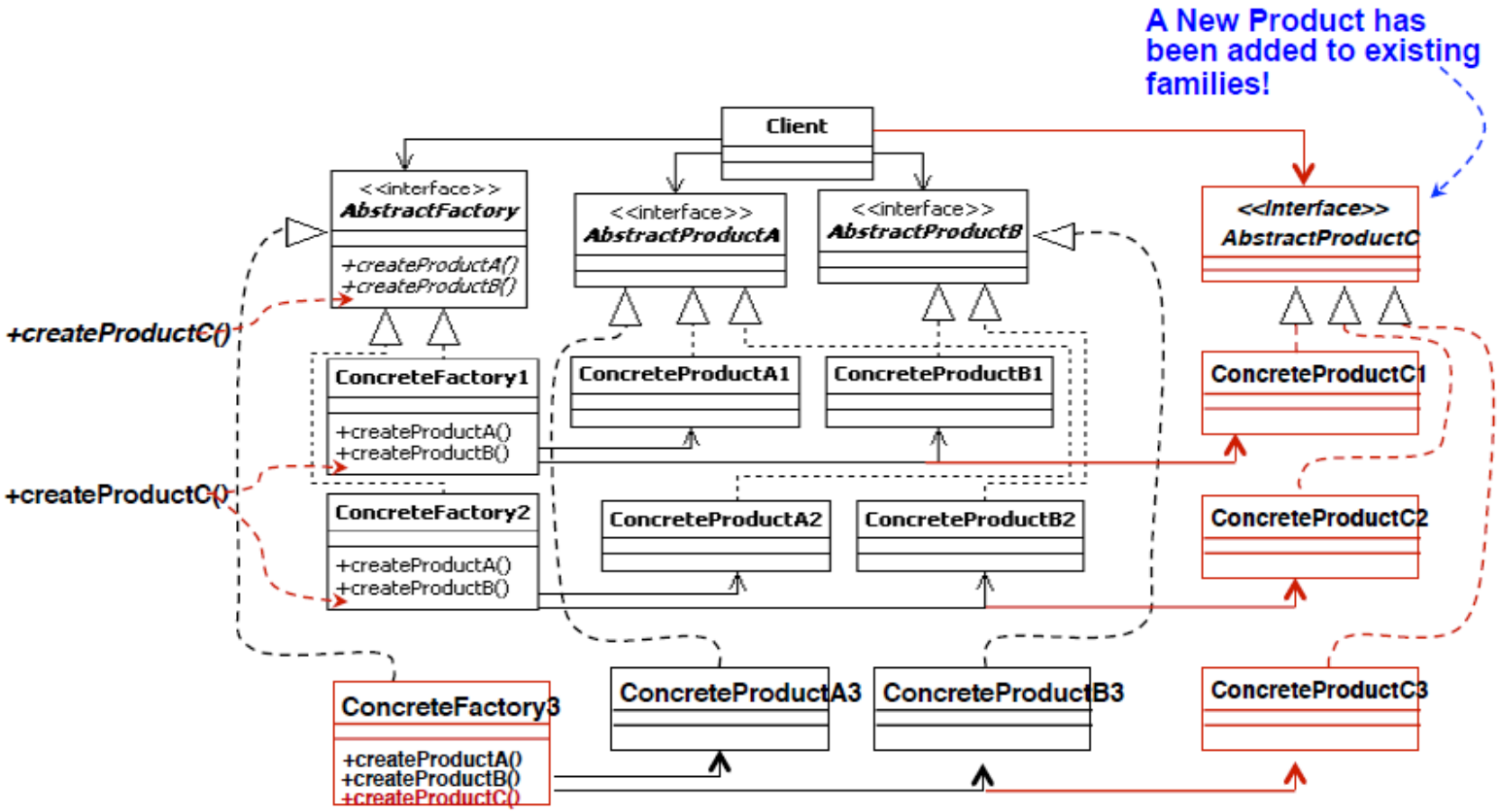
Open Closed Principle

- Software entities should be open for ?, but closed for ?

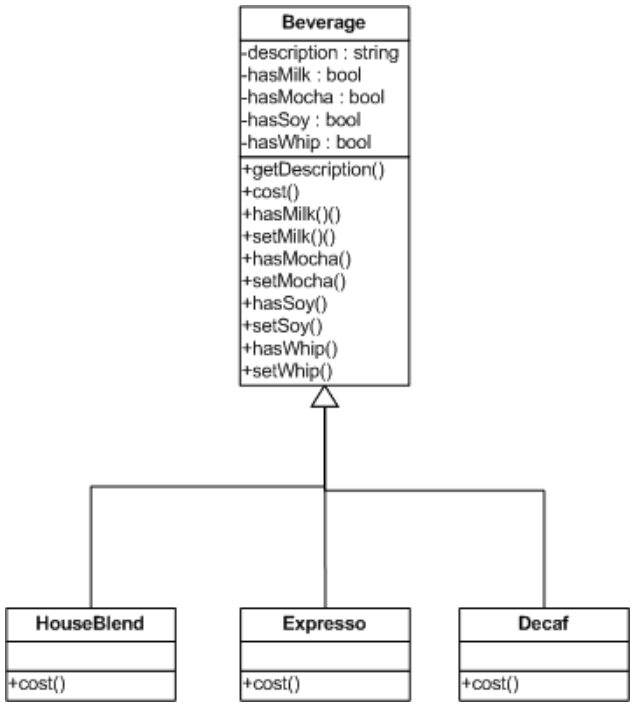
Extension

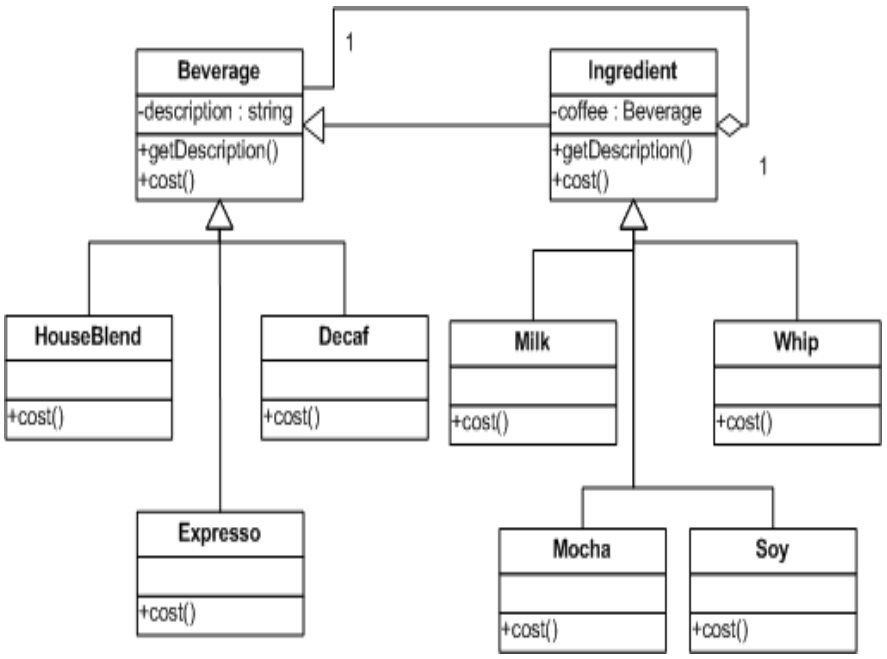
Modification

Open Closed Principle – Applied (1)



Open Closed Principle – Applied (2)

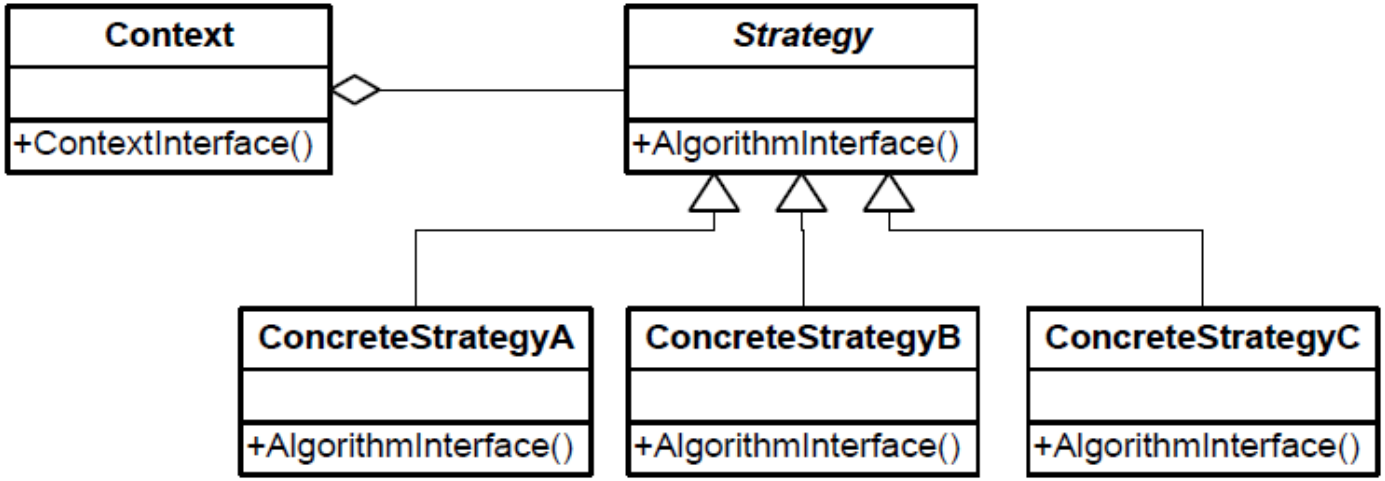




Modification

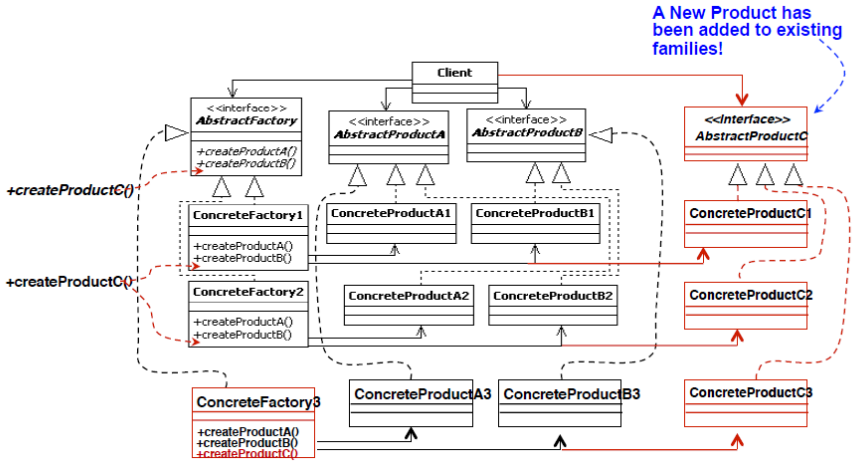
Extension

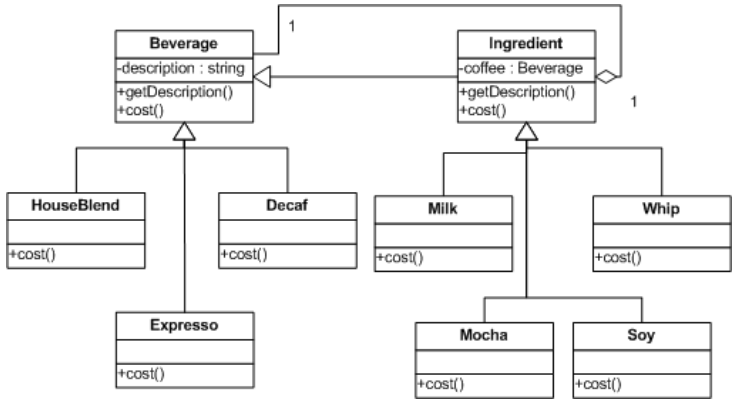
Open Closed Principle – Applied (3)



Extension

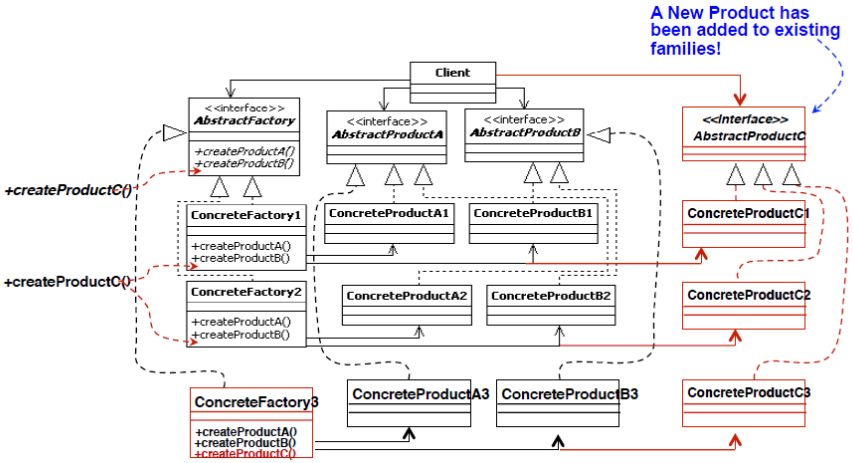
What are the two different ways of extending existing system ?

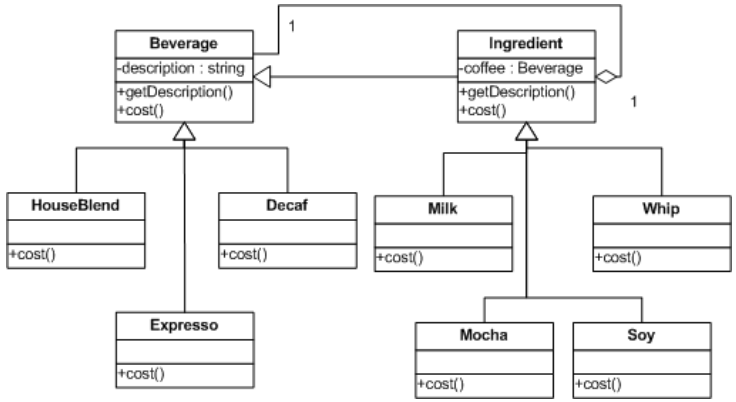




Extension

What are the two different ways of extending existing system ?





Inheritance

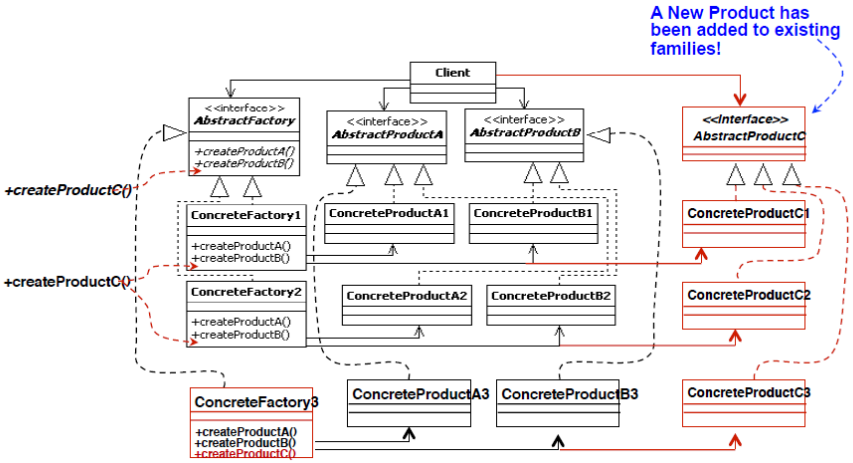
Static - compile time

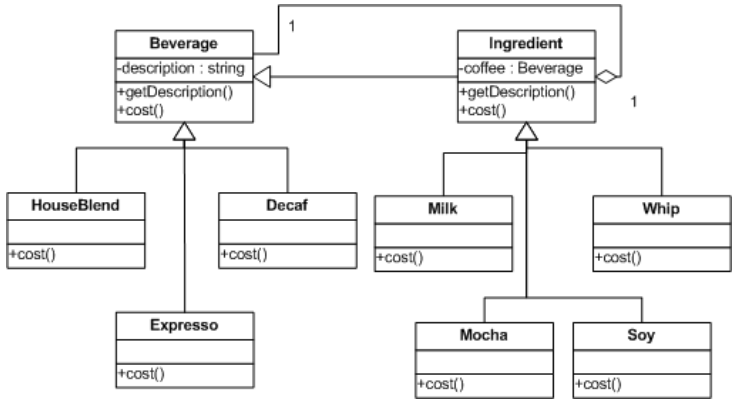
Composition

Dynamic - run time

Extension

- Favor ? over ?
- And why?





Inheritance

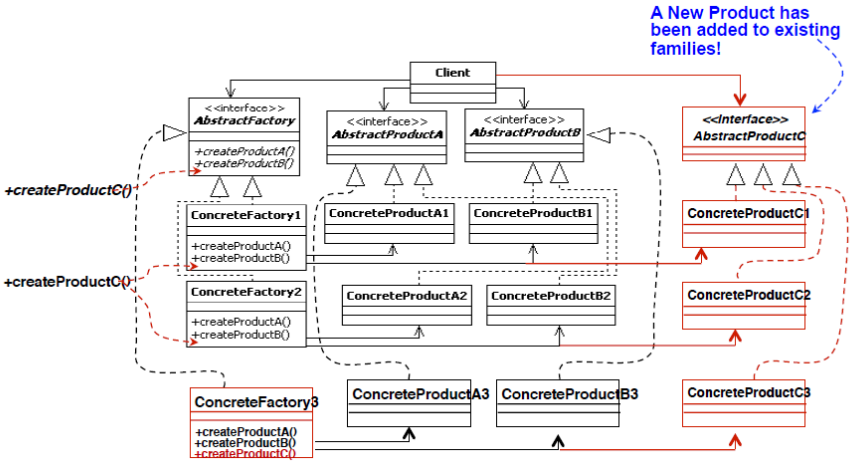
Static - compile time

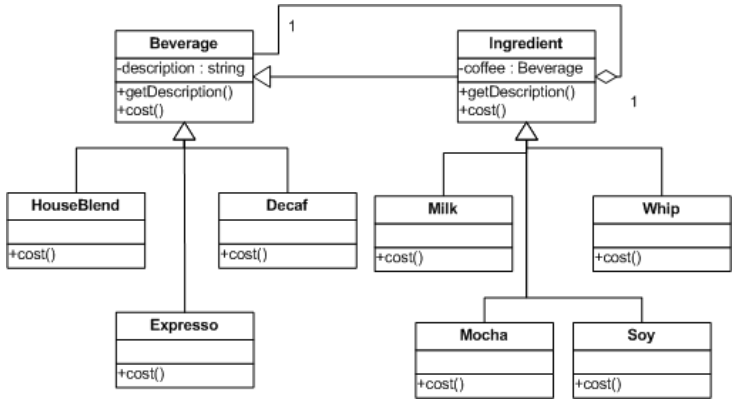
Composition

Dynamic - run time

Extension

- Favor Composition over Inheritance
- Higher reusability and flexibility





Inheritance

Static - compile time

Composition

Dynamic - run time

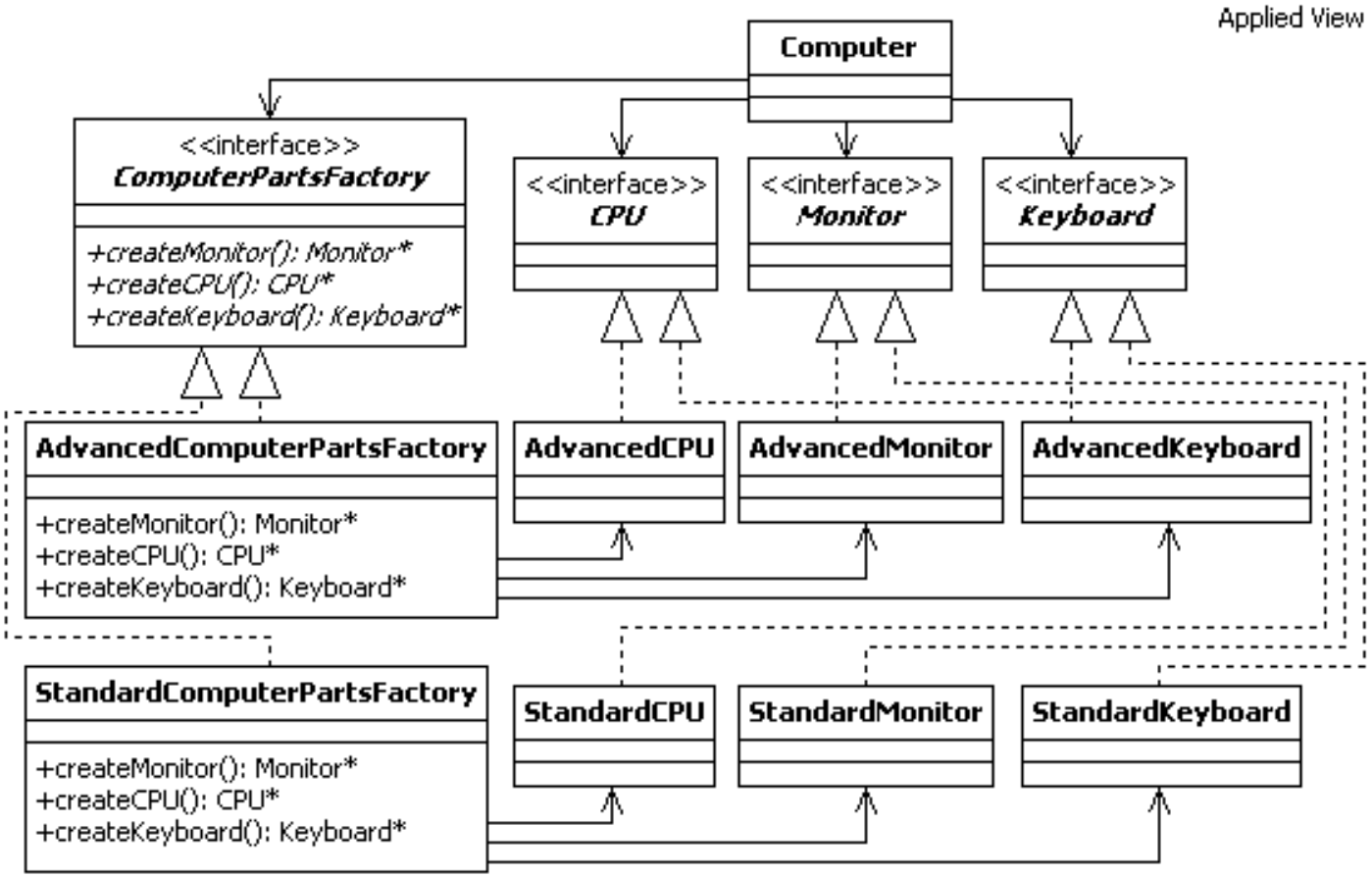
- **Liksov Substitution ---substitutable by subtypes**

Liksov Substitution

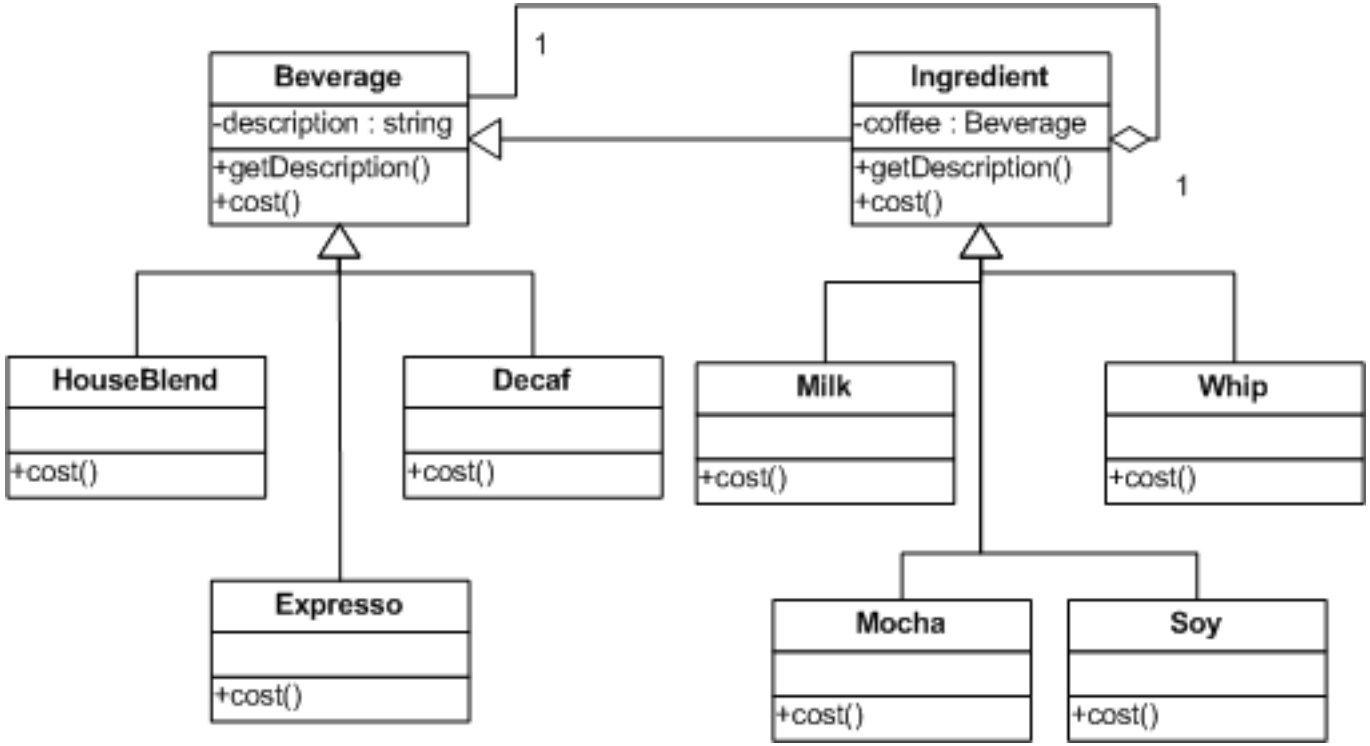
- Objects in a program should be replaceable with instances of their subtypes without altering the (semantic not syntactic) correctness of that program.

- Design by Contract (DbC): It prescribes that software designers should define formal, precise and verifiable interface specifications for software components.

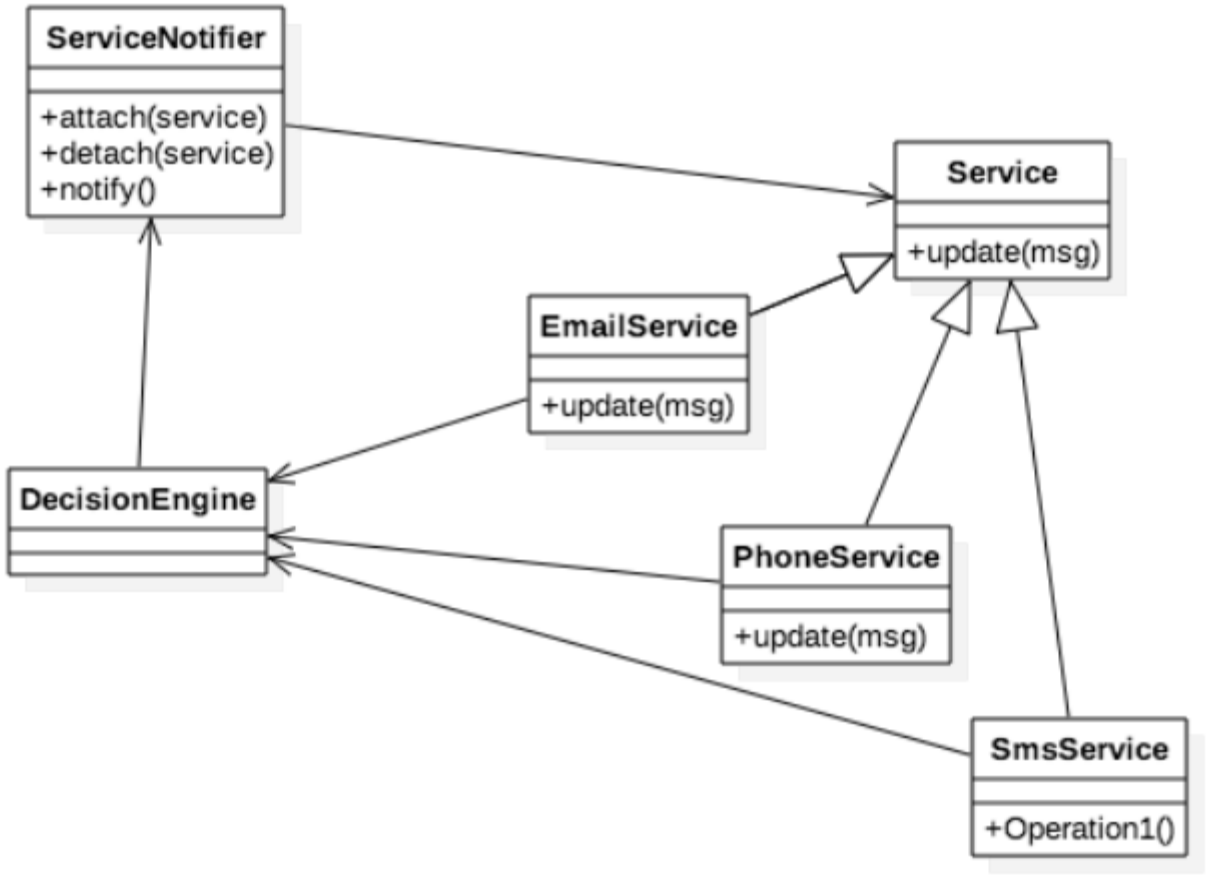
Liksov Substitution – Applied (1)



Liksov Substitution – Applied (2)



Liksov Substitution – Applied (3)



Violation of Liksov Substitution

- Write a new subclass and insert it into existing, working, tested code.
  - If the subclasses don't adhere properly to the interface of the abstract base class, you have to go through the existing code and account for the special cases involving the delinquent subclasses.
  - This is a violation of the Liksov Substitution.
- This is also a blatant violation of the Open Closed Principle.

- **Interface Segregation**
  ---Provide specific interfaces

Interface Segregation

- No client should be forced to depend on methods it does not use.
- Splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them.
- It is intended to keep a system decoupled and thus easier to refactor, change, and redeploy.

Interface Segregation – Real life example

Vs.

Interface Segregation – Real life example

Vs.

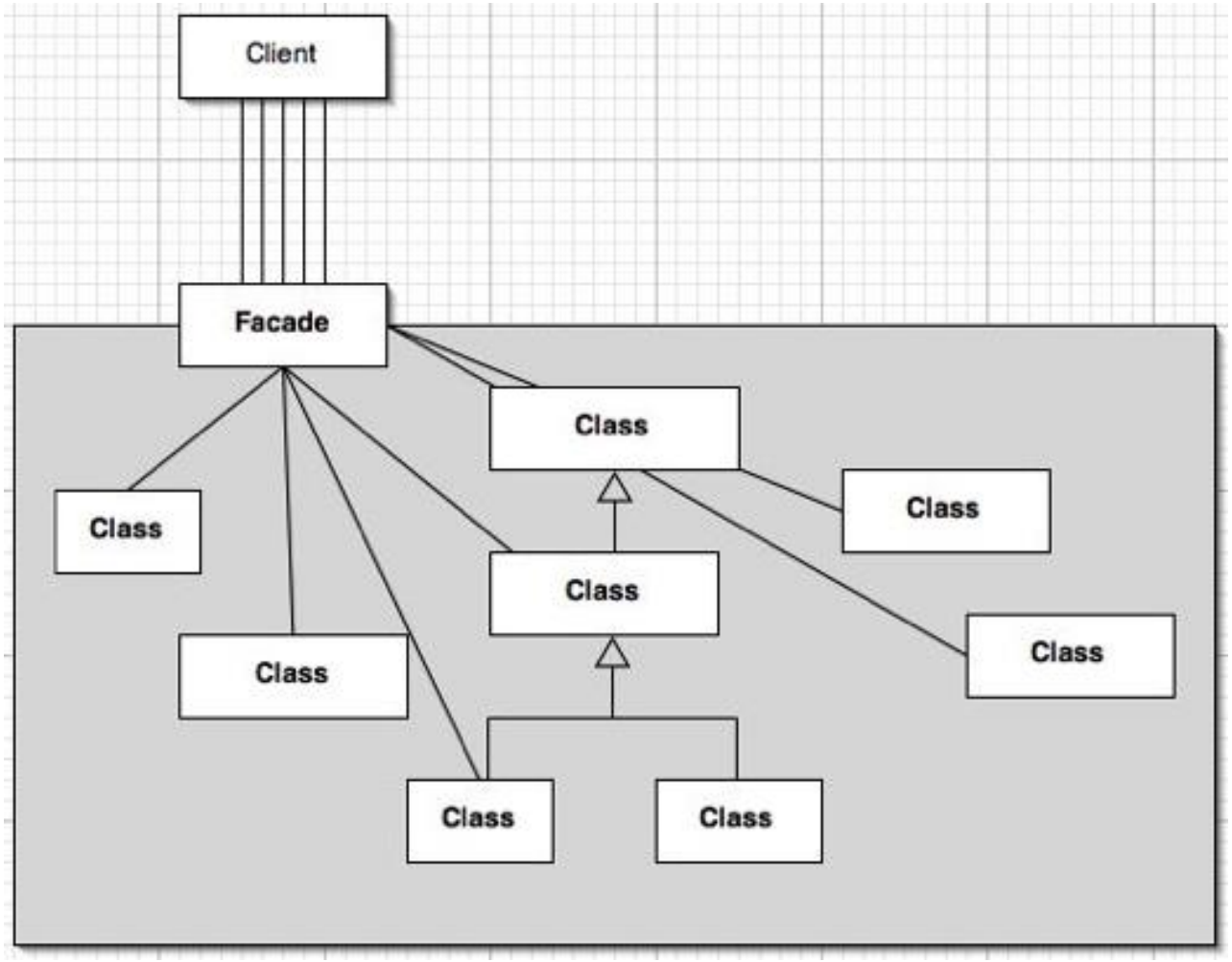
Interface Segregation – Real life example

Vs.

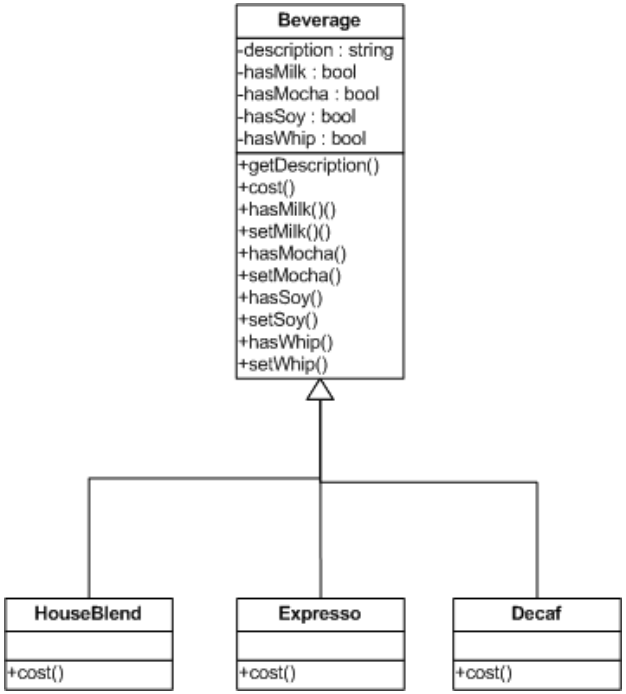
Interface Segregation – Real life example

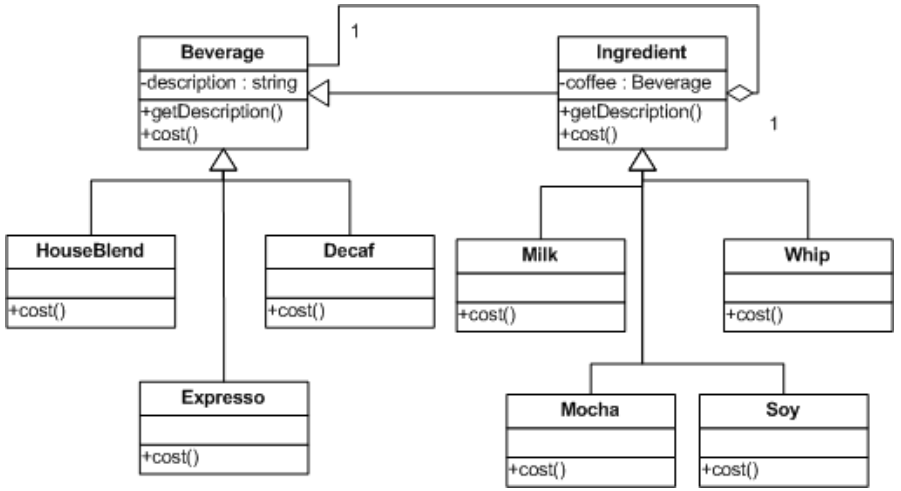
Vs.

Interface Segregation – Applied (1)



Interface Segregation – Applied (2)





- **Dependency Inversion ---Abstractions should not depend on details**

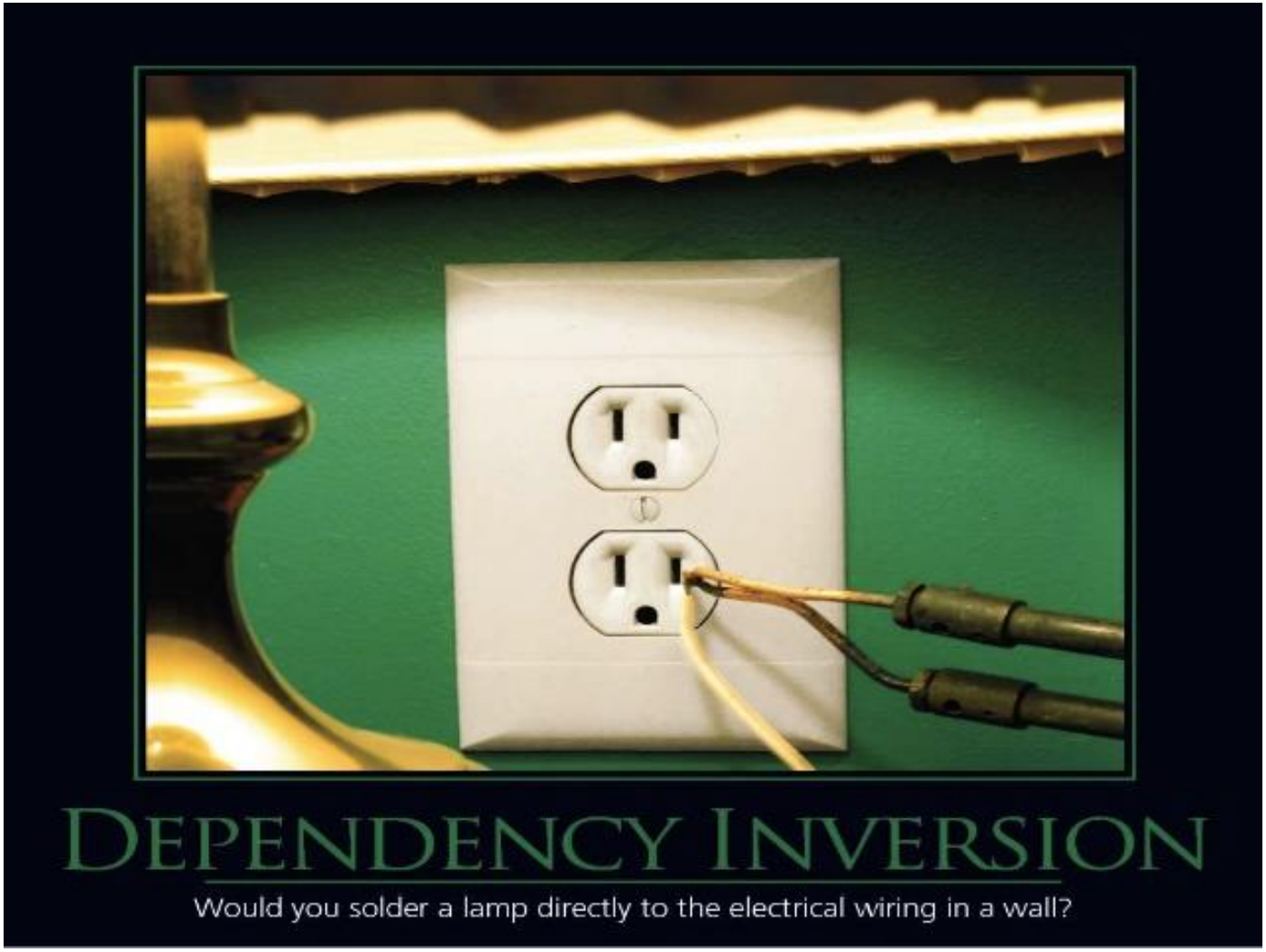
Dependency Inversion

- High-level modules should not depend on low level modules. Both should depend on abstractions.

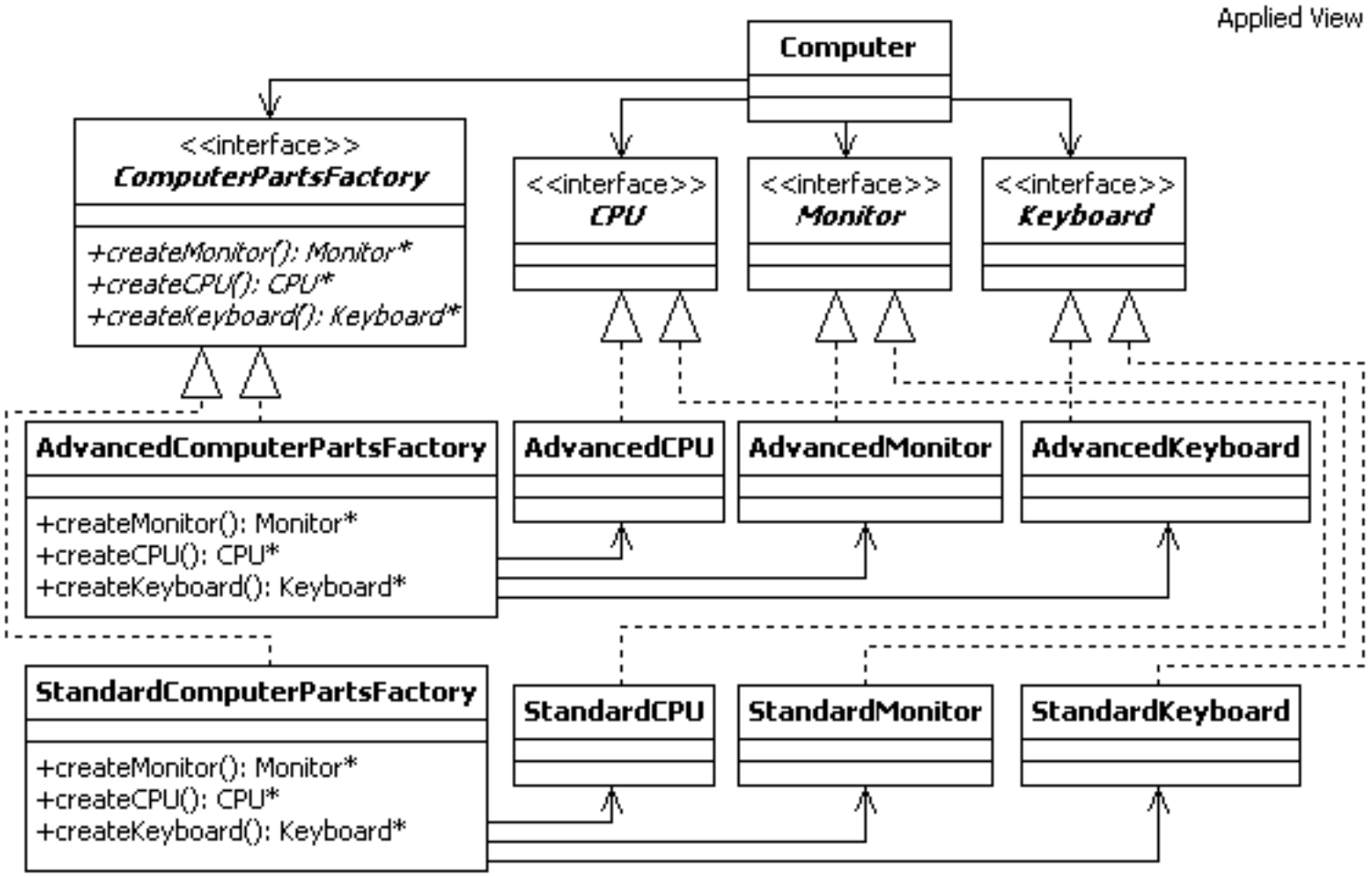
- Abstractions should not depend on details. Details should depend on abstractions.

- Whenever a high-level policy changes, the low-level details must adapt. However, when the low-level details change, the high-level policies remain untouched.

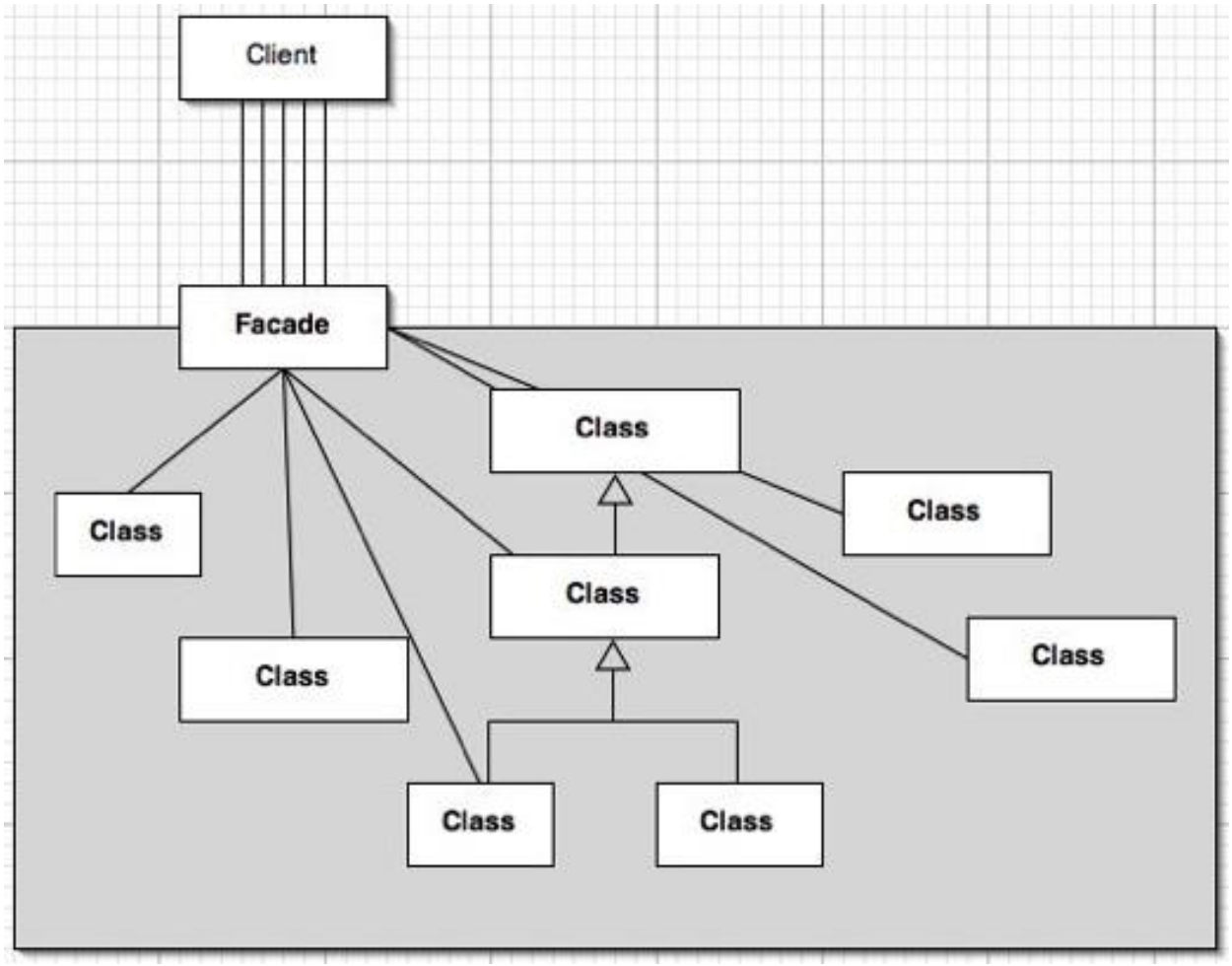
Dependency Inversion



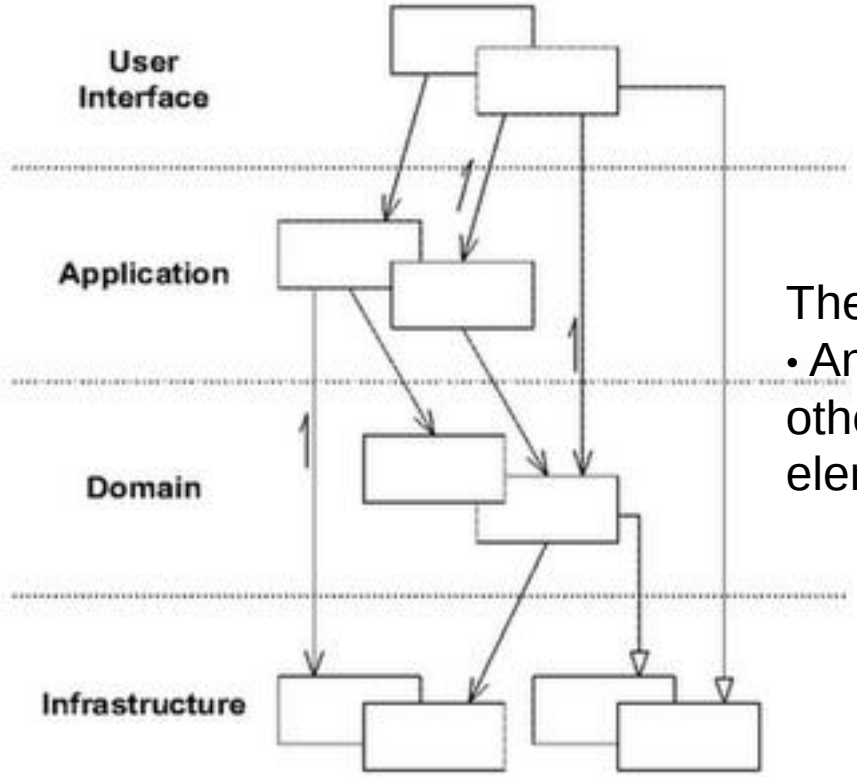
Dependency Inversion – Applied (1)



Dependency Inversion – Applied (2)



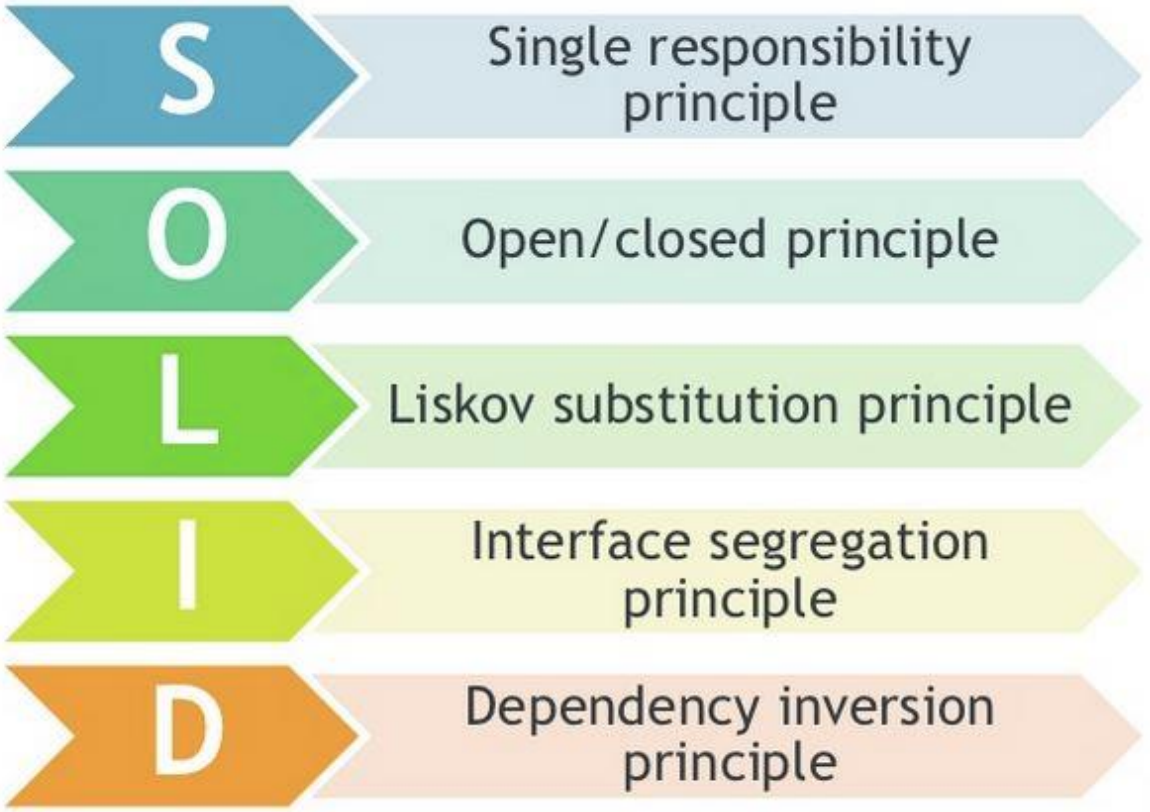
Dependency Inversion – Applied (3)



The essential principle:

- Any element of a layer depends only on other elements in the *same layer* or elements of the *layers "beneath" it*.

Summary - SOLID



These concept are very simple, but extremely important for effective OO design. Take your time to fully understand each principle in practice.


thank you