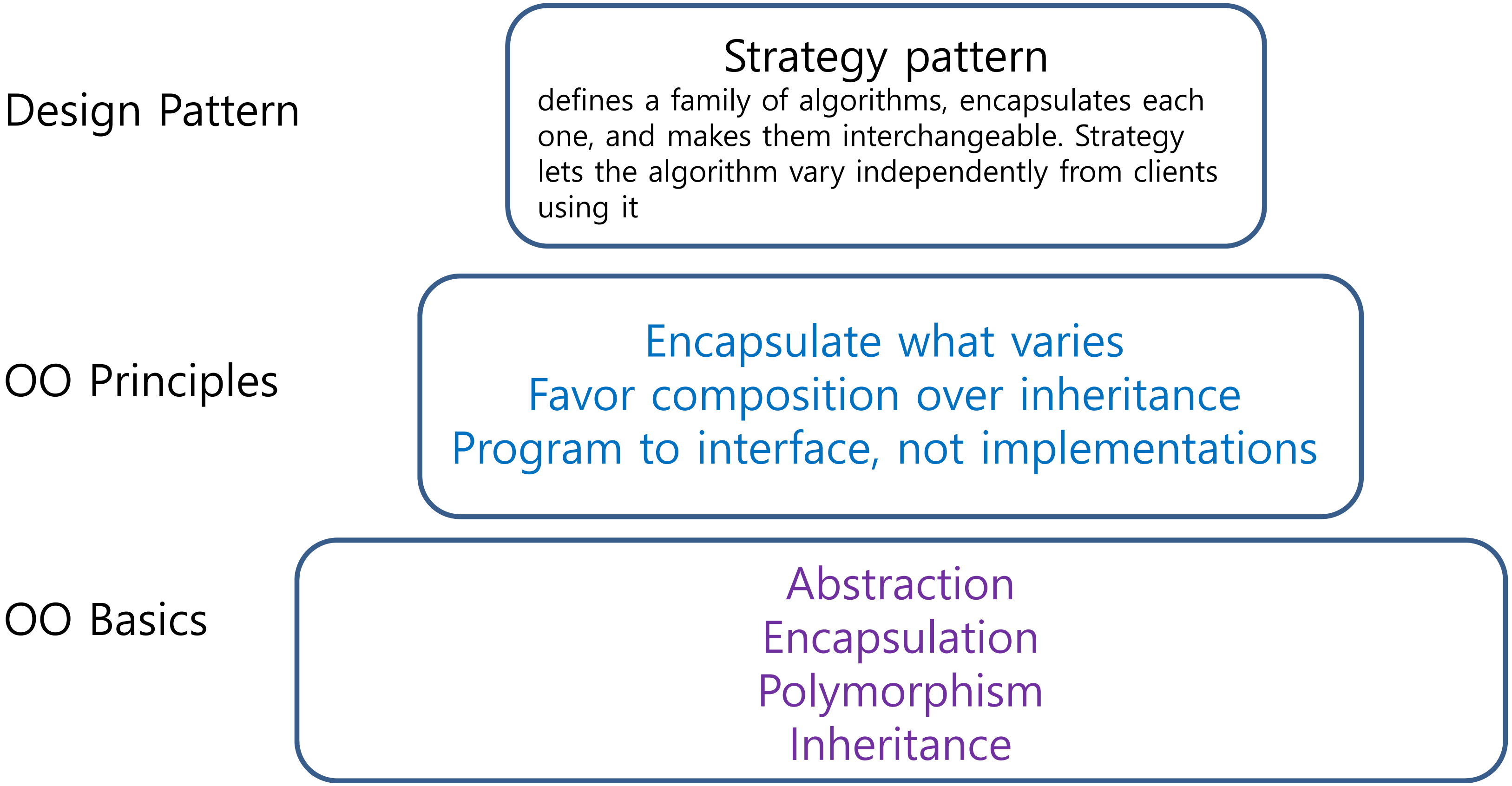
Design Principles

Contents

- Hierarchy of Pattern Knowledge
- OO Principles
- Dependency management
- RC Martin’s Software Design Principles (SOLID)

Hierarchy of Pattern Knowledge



Design Pattern

Strategy pattern

defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients using it

OO Principles

Encapsulate what varies
Favor composition over inheritance
Program to interface, not implementations

OO Basics

Abstraction
Encapsulation
Polymorphism
Inheritance

Figure from [HF]

Design Smells

Name	Symptoms
Rigidity(경직성)	The system is hard to change, because every time you change one thing, you have to change something else in a never ending succession of changes.
Fragility(취약성)	A change to one part of the system causes it to break in many other, completely unrelated parts.
Immobility(부동성)	It is hard to disentangle the system into components that can be reused in other systems.
Viscosity(점착성)	If it is easier to add a hack than it is to add code that fits into the design, then the system has high viscocity.
Needless Complexity (불필요한 복잡성)	There are lots of very clever code structures that aren't actually necessary right now, but could be very useful one day.
Needless Repetition (불필요한 반복)	The code looks like it was written by two programmers named Cut and Paste.
Opacity(불투명성)	Elucidation of the originator's intent presents certain difficulties related to convolution of expression

Table from [RC]

Dependency Management

- Design smells are resulted from mismanaged dependencies

- Mismanaged dependencies
  → Tangled mass of couplings (spaghetti code)

- OO languages provide tools helping managing dependencies
  - Interfaces: break or invert the direction of certain dependencies
  - Polymorphism: allows modules to invoke methods dynamically
  - Lots of power to shape the dependencies, indeed

- So, how do we want them shaped?

Object-Oriented Design Principles

- Program to interfaces, not implementations
- Favor object composition over class inheritance
- Encapsulate what varies
- Strive for loosely couple designs between objects that interact
- SOLID principles by R.C. Martin

R.C. Martin's Software design principles (SOLID)

- The Single-Responsibility Principle (SRP)
- The Open-Closed Principle (OCP)
- The Liskov Substitution Principle (LSP)
- The Interface Segregation Principle (ISP)
- The Dependency Inversion Principle (DIP)

Basically a set of principles for object-oriented design (with focus on designing the classes)

Single Responsibility Principle

Just because you can,
doesn't mean you should!

Single Responsibility Principle (SRP)

A class should have one, and only one, reason to change

The Single Responsibility Principle

- Responsibility
  - a reason to change
  - More responsibilities == More likelihood of change
  - The more a class changes, the more likely we will introduce bugs
  - Changes to one can impact the other
- Separate coupled responsibilities into separate classes
- Cohesion: how strongly-related and focused are the various responsibilities of a module

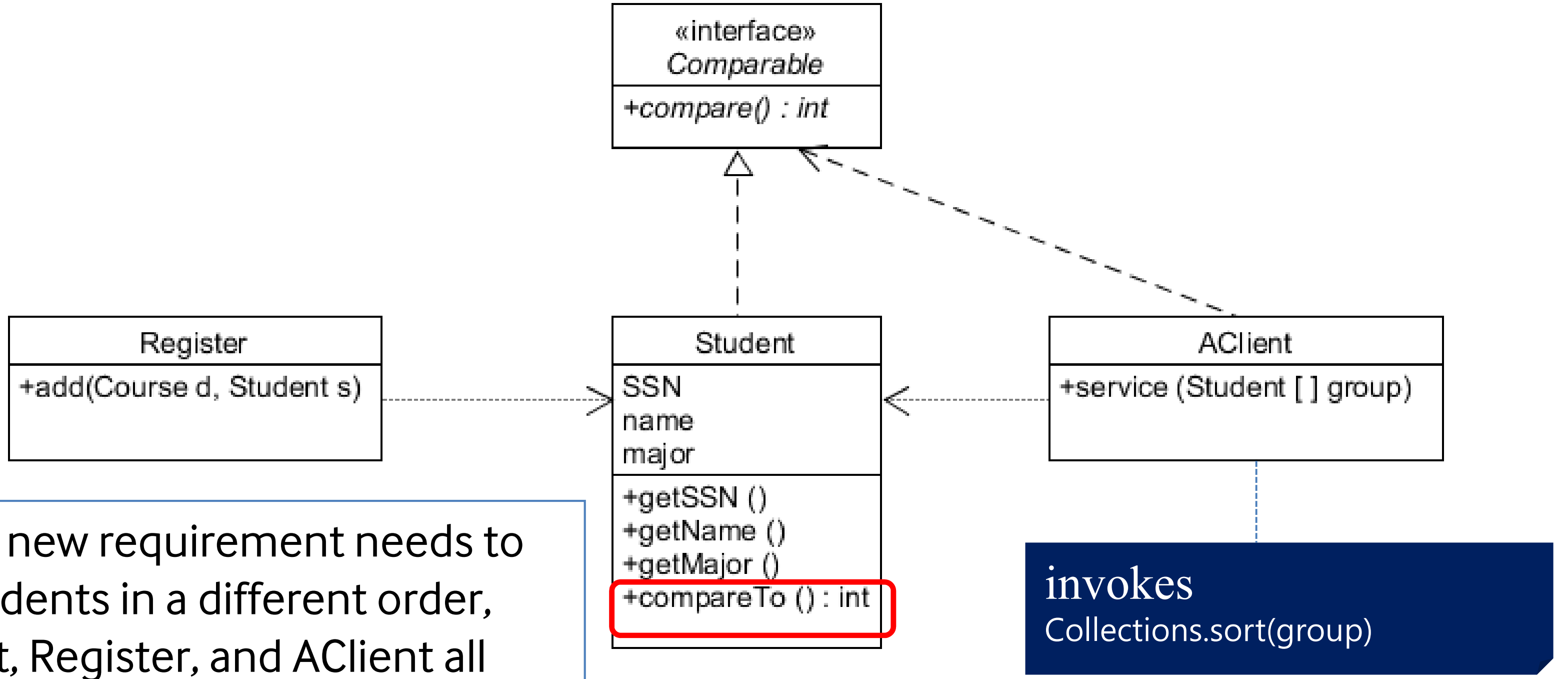
Example of SRP violation

- Often we need to sort students by their name, or SSN. So one may make Class Student implement the Java Comparable interface.

```
class Student implements Comparable {
      …
      int compareTo(Object o) { … }
      …
};
```

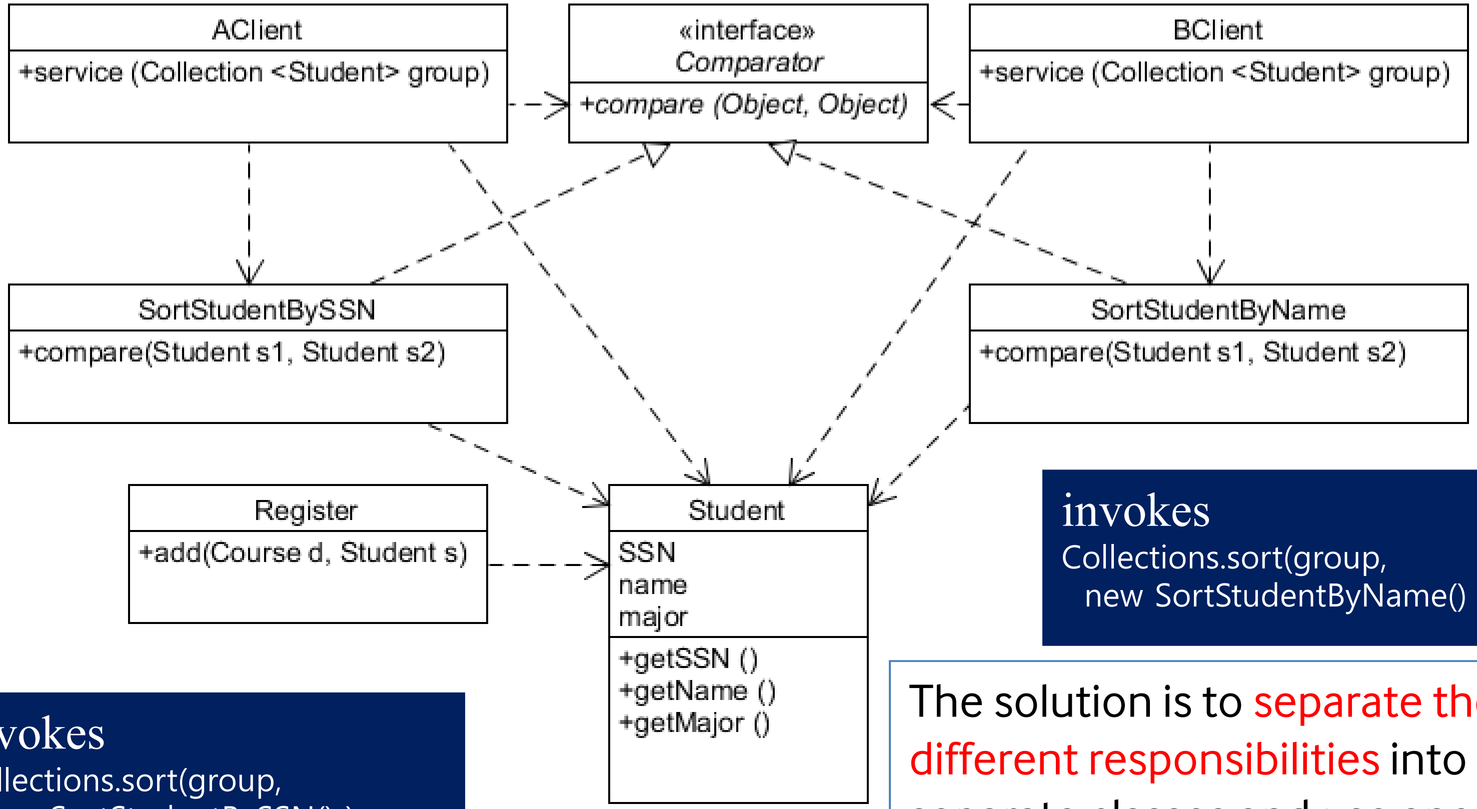
- Student is a business entity, it does not know in what order it should be sorted since the order of sorting is imposed by the client of Student.
- Worse: every time students need to be ordered differently, we have to recompile Student and all its client.
- Cause of the problems: we bundled two separate responsibilities (i.e., student as a business entity with ordering) into one class – a violation of SRP

Example of SRP violation: Case 1



When a new requirement needs to sort students in a different order, Student, Register, and AClient all need to be recompiled, even though Register has nothing to do with any ordering of Students.

Example of design following SRP



AClient
+service (Collection <Student> group)

«interface»
Comparator
+compare (Object, Object)

BClient
+service (Collection <Student> group)

SortStudentBySSN
+compare(Student s1, Student s2)

SortStudentByName
+compare(Student s1, Student s2)

Register
+add(Course d, Student s)

Student
SSN
name
major
+getSSN ()
+getName ()
+getMajor ()

invokes
Collections.sort(group,
new SortStudentByName())

invokes
Collections.sort(group,
new SortStudentBySSN())

The solution is to separate the different responsibilities into two separate classes and use another version of Collections.sort().

Example of SRP violation: Case 2

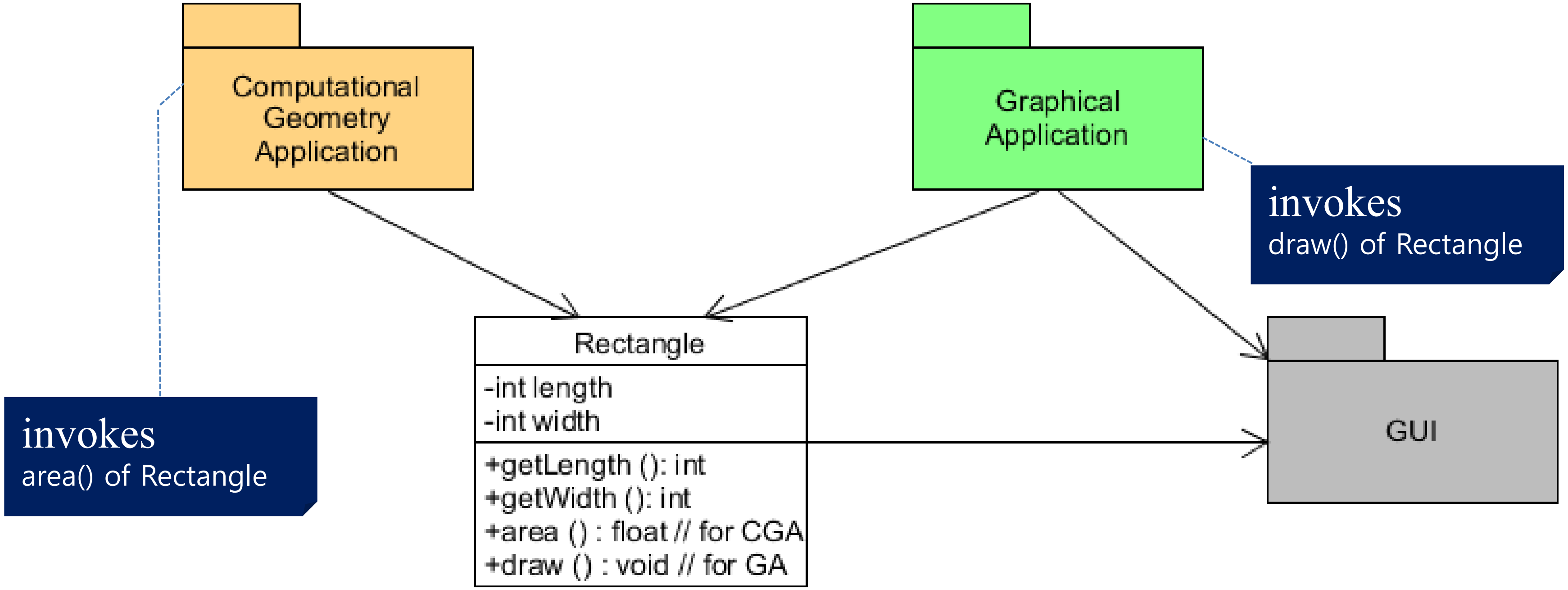


Figure from [RC]

Class Rectangle may be forced to make changes from two different unrelated sources. One is from the Computational Geometry Application (CGA). E.g., modifying area(). The other is from Graphical Application (GA). E.g., modifying draw() for different platforms. A change from either of the two source would still cause the other application to recompile.

Example of design following SRP (not enough yet)

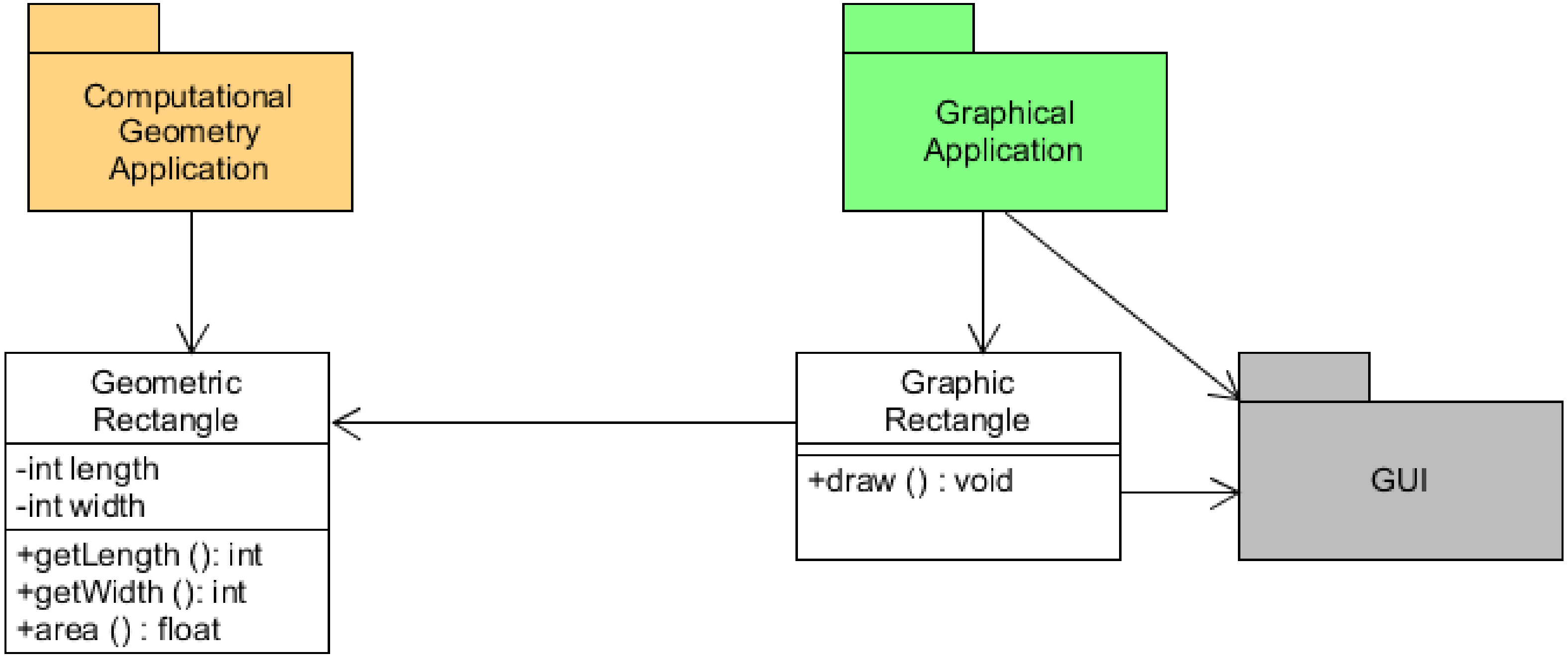


Figure from [RC]

- Package CGA is no longer dependent on graphical side of Rectangle and thus it becomes independent of package GUI. Any change caused by graphical application no longer requires CGA to be recompiled.
- However, any changes from the CGA side may cause GA to be recompiled.

Example of design following SRP

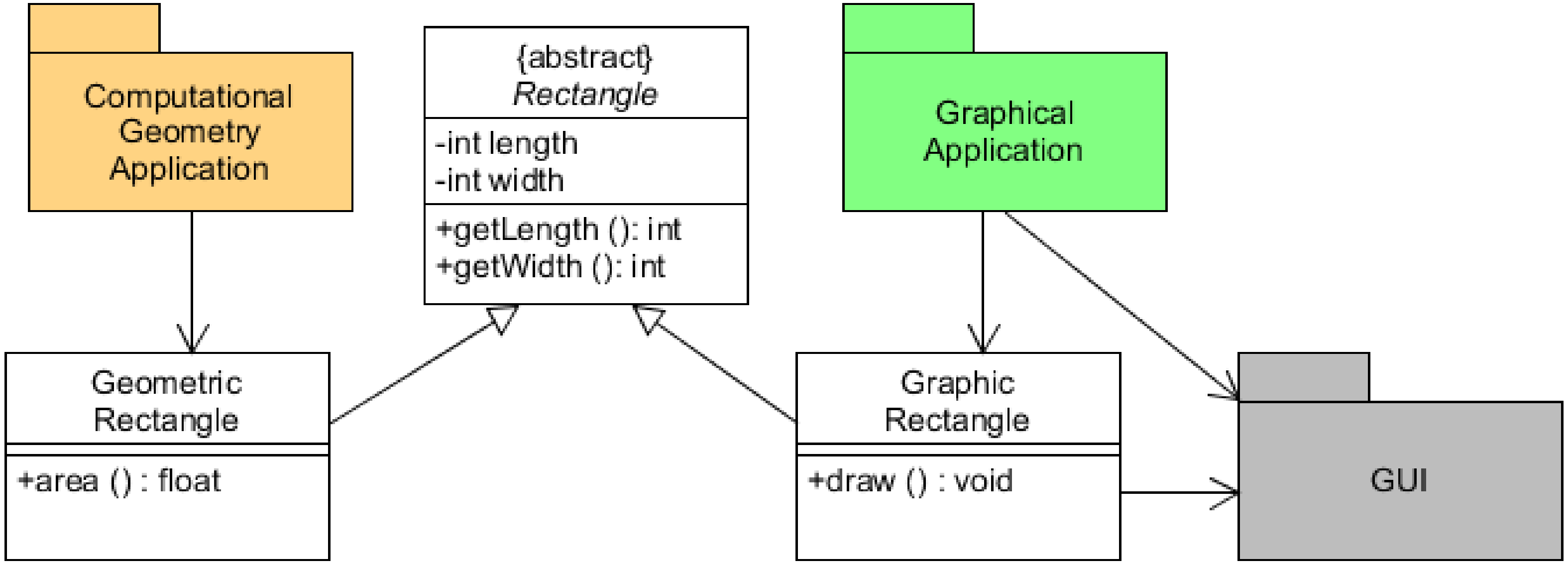


Figure from [RC]

Class Rectangle contains the most primitive attributes and operations of rectangles. Classes GeometricRectangle and GraphicRectangle are independent of each other. A change from either side of CGA or GA, it would not cause the other side to be recompiled.

Identifying Responsibilities

- Responsibility (in SRP)
  - A reason for change
  - Note: sometimes hard to see multiple responsibilities

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

Identifying Responsibilities

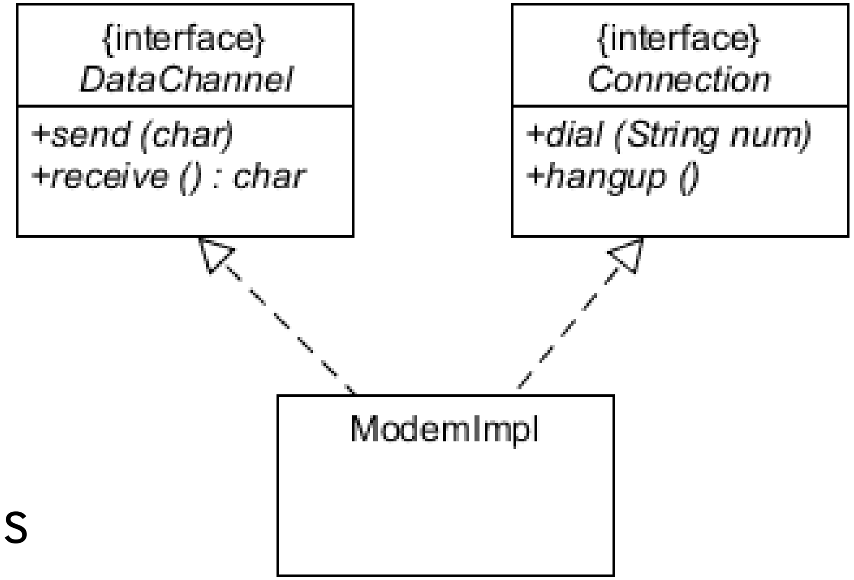


Figure from [RC]

- Two responsibilities
  - Connection management
  - Data communication
  - Note: It depends on how the application is changing

- Needless Complexity
  - If there is no symptom, it is not wise to apply the SRP or any other principle!

Open Closed Principle

Open chest surgery is NOT needed
when putting on a coat

Open Closed principle (OCP)

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

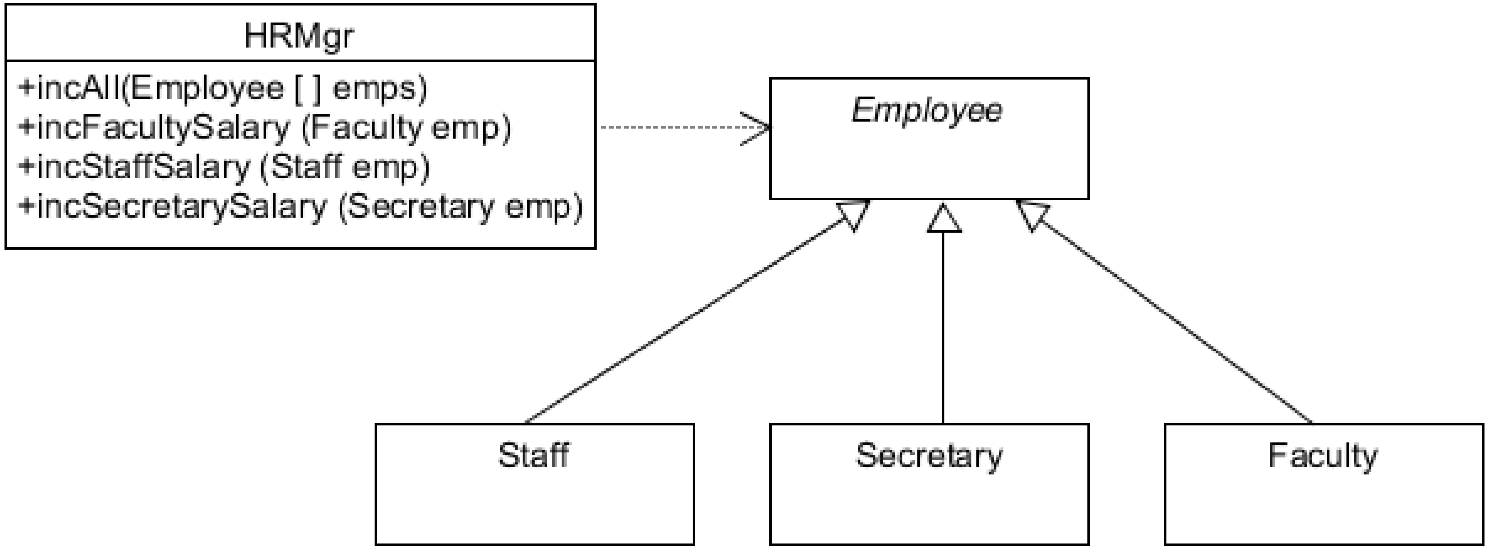
You should be able to extend a class's behavior, without modifying it.

Conforming to OCP

- Open for extension
  - Behavior of the module can be extended
  - We are able to change what the module does
- Closed for modification
  - Extending behavior does not result in excessive modification such as a rchitectural changes of the module

- Violation Indicator: Design Smell of Rigidity
  - A single change to a program results in a cascade of changes to dependent modules

Bad Design Example


HRMgr
+incAll(Employee [ ] emps)
+incFacultySalary (Faculty emp)
+incStaffSalary (Staff emp)
+incSecretarySalary (Secretary emp)
Employee
Staff
Secretary
Faculty

```
 void incAll(Employee[] emps) {
     for (int i = 0; i < emps.size(); i++) {
          if (emps[i].empType == FACULTY)
              incFacultySalary((Faculty)emps[i]);
          else if (emps[i].empType == STAFF)
              incStaffSalary((Staff)emps[i]);
          else if (emps[i].empType == SECRETARY)
              incSecretarySalary((Secretary)emps[i]);
     }
}
```

Problems of Bad Design

- Rigid
  - Adding new employee type requires significant changes
- Fragile
  - Many switch/case or if/else statements
  - Hard to find and understand

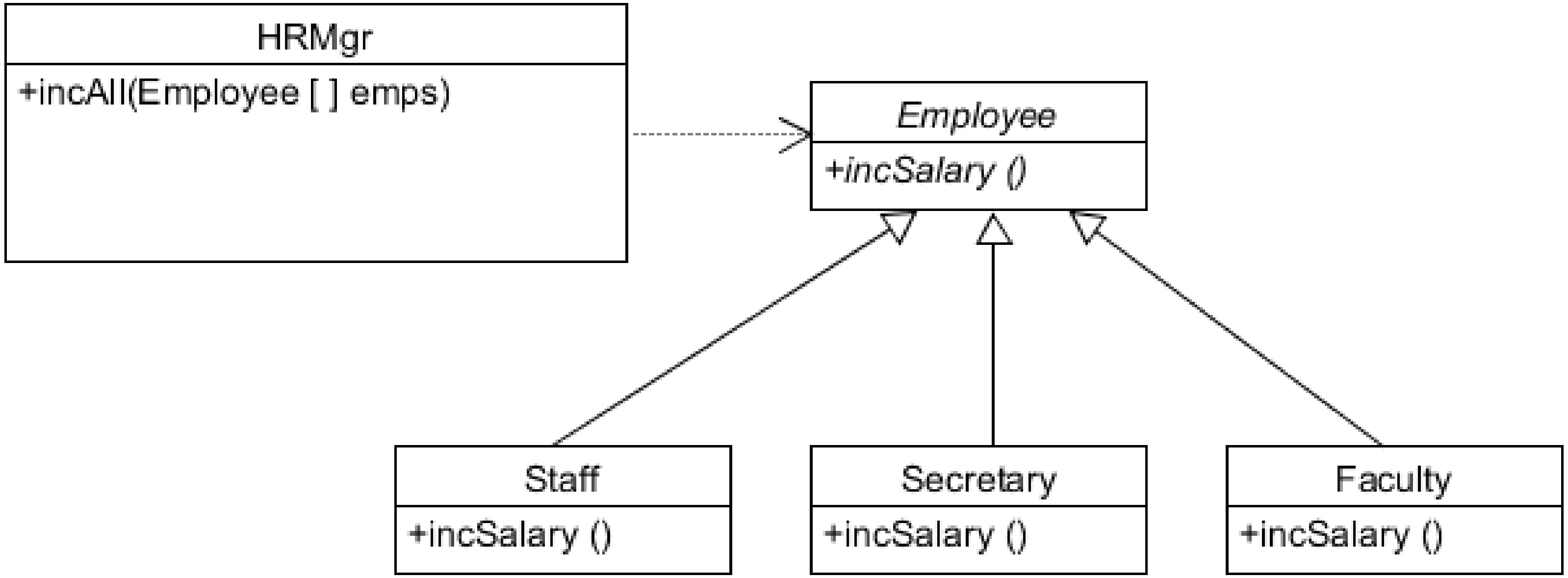
```
 void incAll(Employee[] emps) {
     for (int i = 0; i < emps.size(); i++) {
          if (emps[i].empType == FACULTY)
              incFacultySalary((Faculty)emps[i]);
          else if (emps[i].empType == STAFF)
              incStaffSalary((Staff)emps[i]);
          else if (emps[i].empType == SECRETARY)
              incSecretarySalary((Secretary)emps[i]);
          else if (emps[i].empType == ENGINEER)
              incEngineerSalary((Engineer)emps[i]);
     }
}
void incEngineerSalary (Engineer e) { ...  }
```

Problems of Bad Design

- Rigid
  - Adding new employee type requires significant changes
- Fragile
  - Many switch/case or if/else statements
  - Hard to find and understand
- Immobile
  - To reuse incAll() ---> we need Faculty, Staff, Secretary, too!
  - What if we need just Faculty and Staff only?

```
 void incAll(Employee[] emps) {
     for (int i = 0; i < emps.size(); i++) {
          if (emps[i].empType == FACULTY)
              incFacultySalary((Faculty)emps[i]);
          else if (emps[i].empType == STAFF)
              incStaffSalary((Staff)emps[i]);
          else if (emps[i].empType == SECRETARY)
              incSecretarySalary((Secretary)emps[i]);
     }
}
```

Better Design



```
void incAll(Employee[] emps) {
     for (int i = 0; i < emps.size(); i++)
          emps[i].incSalary();
}
```

- When Engineer is added, incAll() does not even need to recompile.
- This design is open to extension, closed for modification.

Abstraction is the Key!

- Abstractions
  - Fixed and yet represent an unbounded group of possible behaviors
  - Abstract base class : fixed
  - All the possible derived classes : unbounded group of possible behaviors

- Program the class
  - to interfaces (or abstract classes)
  - not to implementation (concrete classes)

Abstraction is the Key!

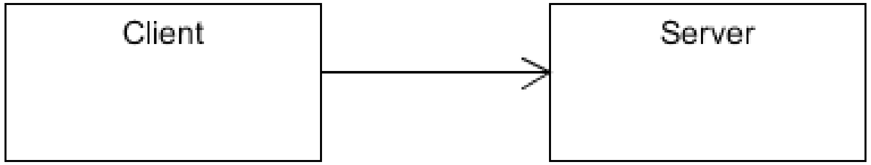


Figure from [RC]

What if we need to use other servers ?

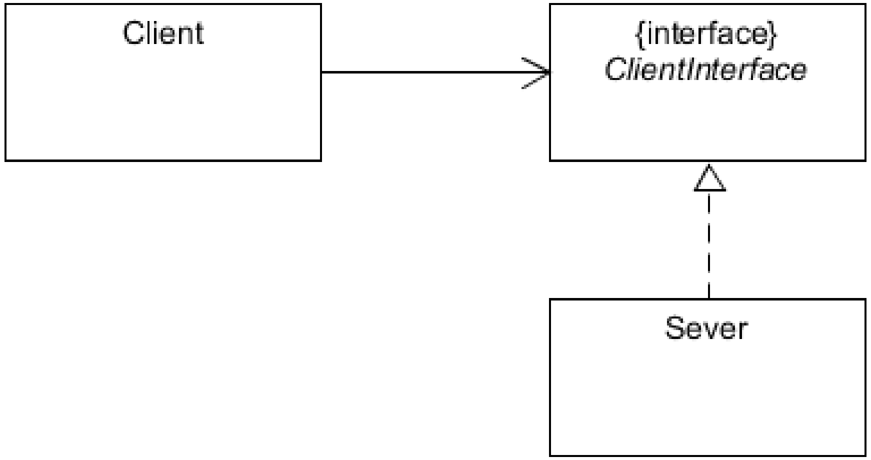


Figure from [RC]

Abstract classes (or interfaces) are more closely associated to their clients than to the classes that implement them

Anticipating Future Changes

- Strategy is needed
  - Choose the kinds of changes against which to close design
  - Guess the most likely kinds of changes, and then construct abstractions to protect him from those changes.
- Consider the cost!
  - Conforming to OCP is expensive
  - Time and effort to create appropriate abstractions
  - Abstractions also increase complexity

Anticipating Future Changes

- Do not put hooks in for changes that might happen
  - Instead, wait until the changes happen!

"Fool me once, shame on you. Fool me twice, shame on me."

- Initially write the code expecting it to not change.
- When a change occurs, implement the abstractions that protect from future changes of that kind.
  - It's better to take the first hit as early as possible.
    - We want to know what kind of changes are likely before going too far in the development.
  - Use TDD and listen to the tests. Develop in short cycles. Develop features before infrastructure. Develop the most important features first. Release early and often

Liskov Substitution Principle

If it looks like a duck, quacks like a duck, but need batteries

- You probably have the wrong abstraction

Liskov Substitution Principle (LSP)

Subtypes *must be* ***substitutable*** *for their* ***base types***

Derived classes must be substitutable for their base classes

Liskov Substitution Principle

- A rule that you want to **check when you decide to use inheritance or not**

- If ***C*** **is a subtype** of *P*, then objects of type *P* may be replaced with objects of type *C* **without** altering any of the desirable properties of the program (correctness, task performed, etc.)

Subtyping VS Inheritance

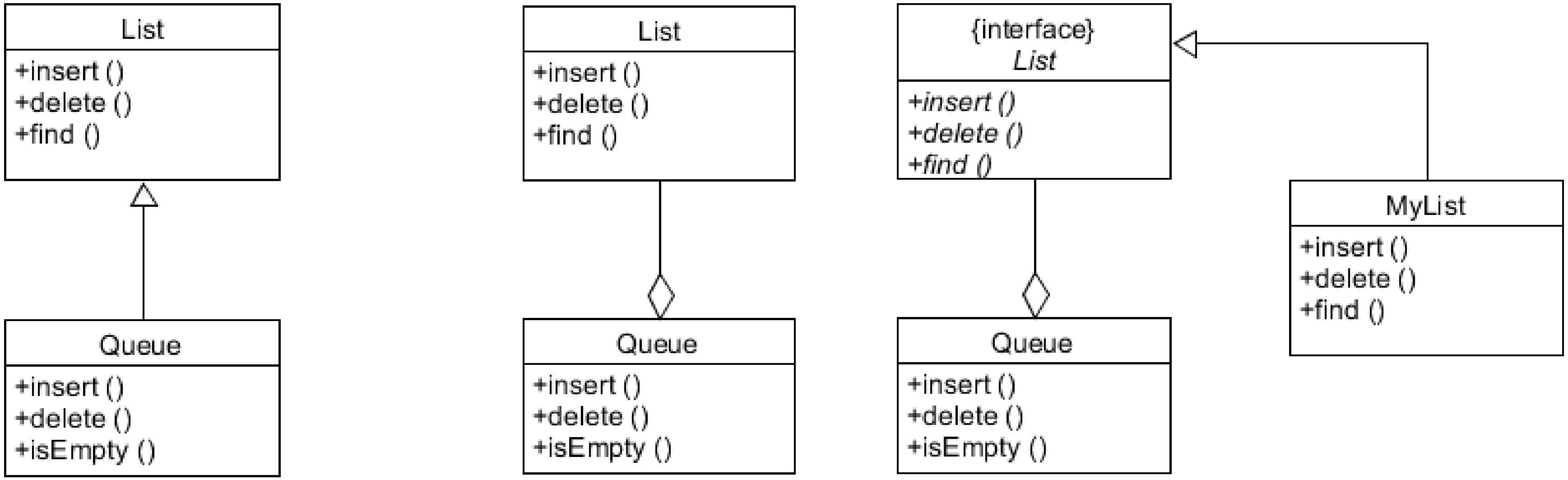
- In some languages inheritance and subtyping agree, whereas in others they differ

- Subtyping
  - establishes an IS_A relationship
  - also known as *interface inheritance*

- Inheritance
  - only reuses implementation and establishes a syntactic relationship not necessarily a semantic relationship
  - known as *implementation inheritance* or *code inheritance*

The Liskov Substitution Principle and Reuse

- Think twice when you decide to use Inheritance!
  - When you use List to implement Queue (in Java), use composition, not inheritance.
  - The intention is that you use only List’s implementation



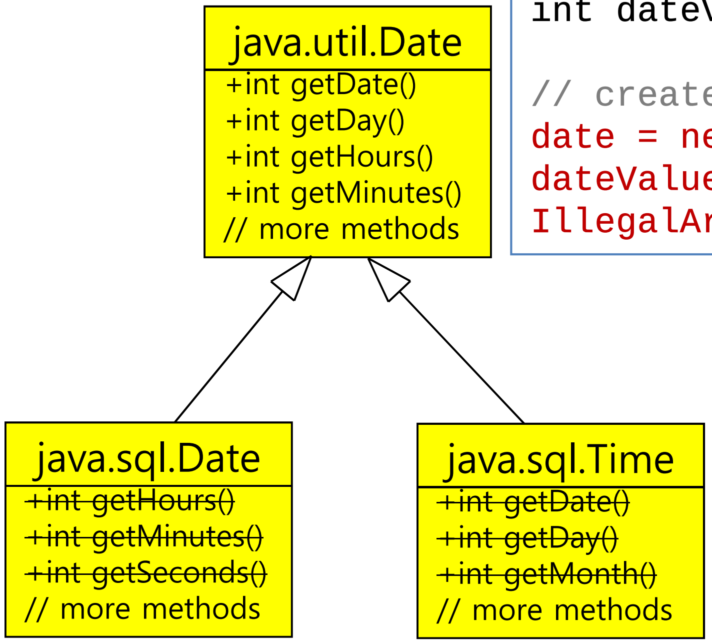
Violation of LSP!

Good

Better!

Figure from [RC]

Improper Inheritance



java.util.Date

+int getDate()
+int getDay()
+int getHours()
+int getMinutes()
// more methods

java.sql.Date

~~+int getHours()~~
~~+int getMinutes()~~
~~+int getSeconds()~~
// more methods

java.sql.Time

~~+int getDate()~~
~~+int getDay()~~
~~+int getMonth()~~
// more methods

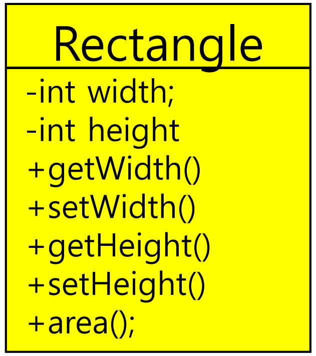
```
// create a Date object and reference it with Date type
java.util.Date date = new java.util.Date();
int dateValue = date.getDate(); // okay

// create Time object and with reference it with Date type
date = new java.sql.Time(10, 10, 10);
dateValue = date.getDate(); // throws
IllegalArgumentException
```

☹

Inheritance Decision can be subtle

```
class Rectangle {
    private int width;
    private int height;
    public void setHeight(int h) { height = h }
    public void setWidth(int w) { width = w }
}
```



Rectangle

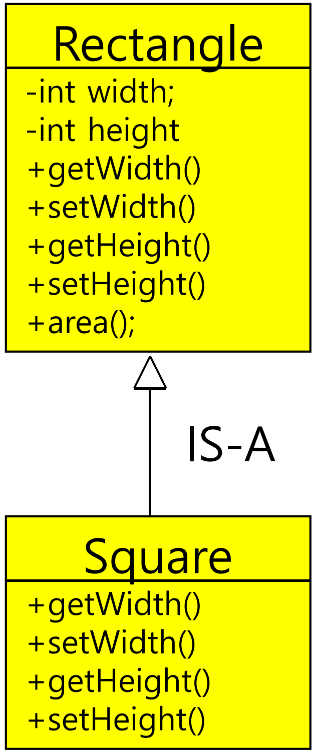
-int width;
-int height
+getWidth()
+setWidth()
+getHeight()
+setHeight()
+area();

Inheritance Decision can be subtle

```
class Square extends Rectangle {
   public void setWidth(int width) {
       super.setWidth(width);
       super.setHeight(width);
   }
   public void setHeight(int height) {
       super.setHeight(height);
       super.setWidth(height);
   }
}

void clientOfRectangle(Rectangle r) {
    r.setWidth(10);
    r.setHeight(20);
    assert(r.area() == 200);
}

Rectangle r = new Square();
clientOfRectangle(r); // what is output?
```



Rectangle

-int width;
-int height
+getWidth()
+setWidth()
+getHeight()
+setHeight()
+area();

IS-A

Square

+getWidth()
+setWidth()
+getHeight()
+setHeight()

IS_A Relationship

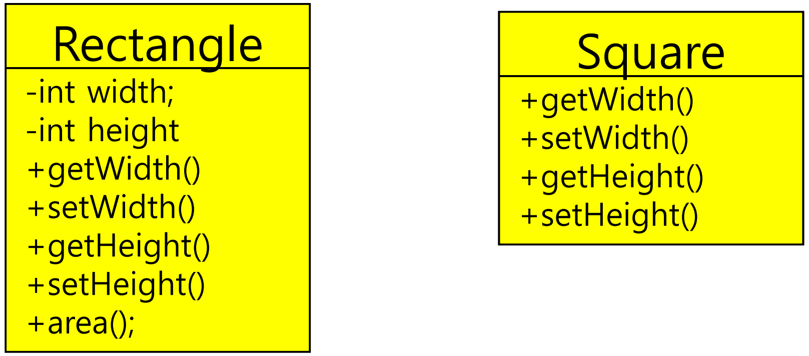
- Validity
  - A model, viewed in isolation, cannot be meaningfully validated
  - The validity of a model can only be expressed in terms of its clients
- IS_A relationship is about behavior
  - From the viewpoint of author of clientOfRectangle(), Square object is not a Rectangle object
  - Behavior of Square object is not consistent with the author's expectation of the behavior of Rectangle object

Design by Contract

- LSP
  - IS_A relationship pertains to behavior that can be reasonably assumed and that clients depends on
- How do you know what your clients will really expect?
  - Meyer (the author of Eiffel) proposed "Design by Contract"
    - A class explicitly states the contract for that class
    - Subclass can only
      - maintain or **weaken the pre-condition for superclass**
      - maintain or **strengthen the post-condition for superclass**

Example of DBC

- Post-condition of setWidth(double w)
  - Rectangle: width == w && height == old.height
  - Square: width == w && height == w
  - Hence, the post-condition for Square is not stronger than the post-condition for Rectangle nor maintain it
    - does not enforce the clause "height == old.height"



Rectangle
-int width;
-int height
+getWidth()
+setWidth()
+getHeight()
+setHeight()
+area();

Square
+getWidth()
+setWidth()
+getHeight()
+setHeight()

Violation of LSP can lead to another violations

```
void f(PType x) {
    ....
}

class CType extends PType {
    ...
}
```

- Assume that when *CType* is passed to *f()* instead of *PType*, it causes *f* to misbehave; Thus, *CType* violates the LSP

  → *CType* is fragile in the presence of *f*

- The owner of *f* may want to put test code for *CType*
  - This test violates OCP
  - *f* is **not closed** to all various **derivatives of *PType***

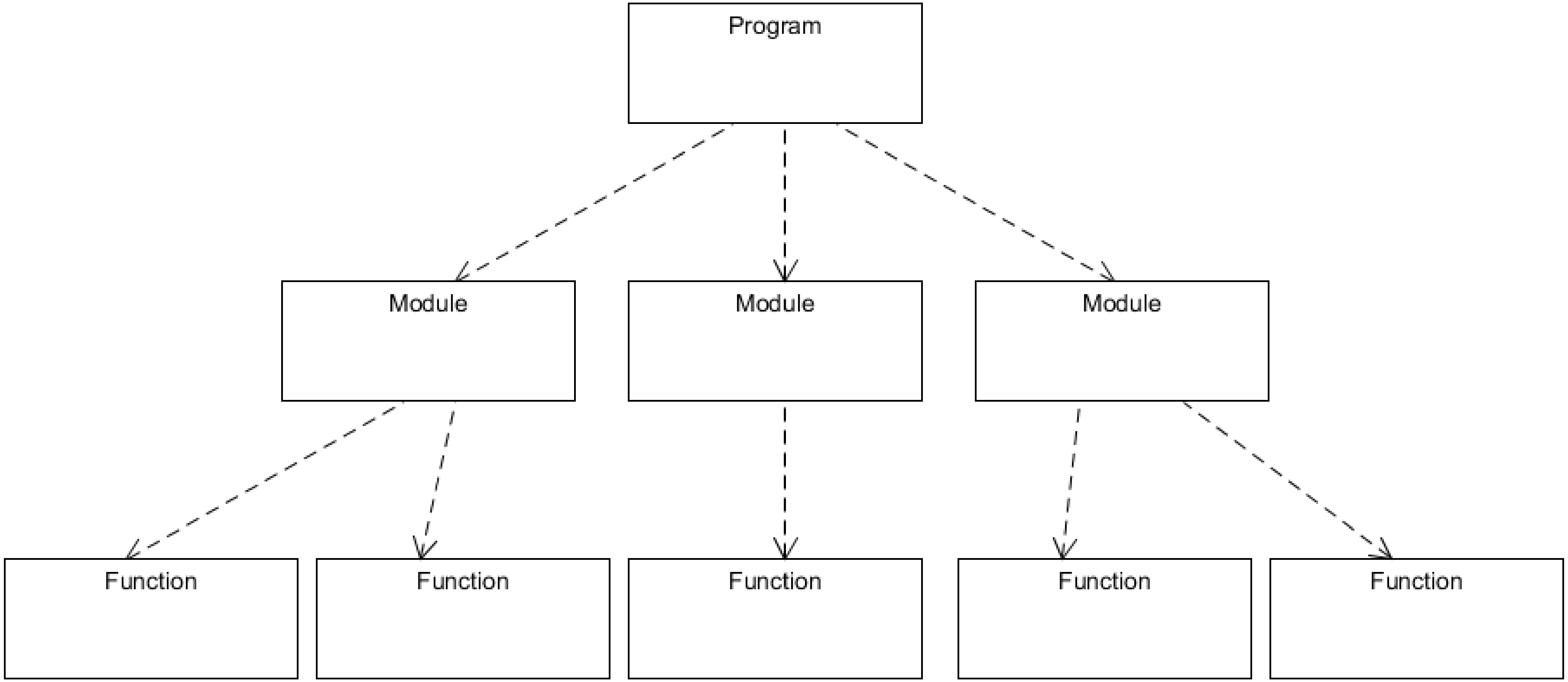
Dependency Inversion Principle

Would you solder a lamp directly to the electrical wiring in a wall?

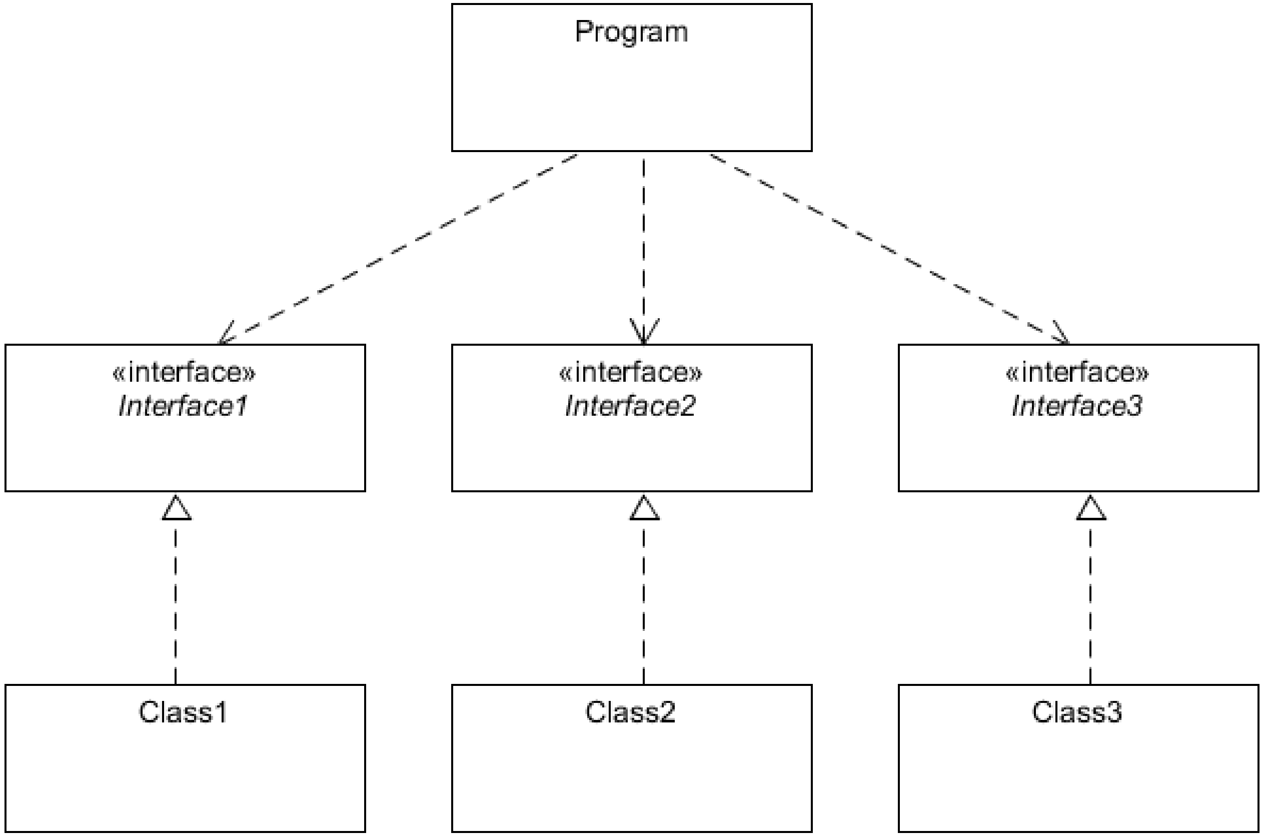
Dependency Inversion Principle (DIP)

- *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.***
- Why Inversion?
  - DIP attempts to “invert” the dependencies that result from a structured analysis and design approach

Typical in Structured Analysis & Design



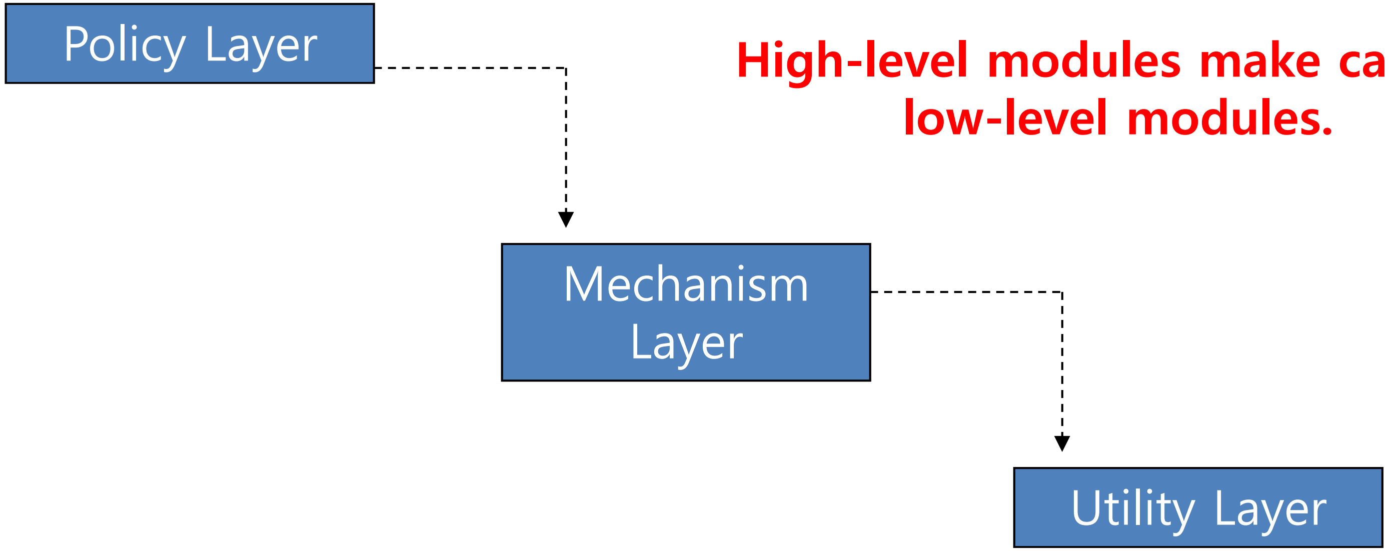
Dependency Inversion Principle



Inversion of Ownership

- Its not just an inversion of **dependency**, DIP also inverts **ownership**
  - Typically a service interface is “owned” or declared by the server, here the client is specifying what they want from the server
  - DIP asks the client to own the interface!

The dependency inversion principle

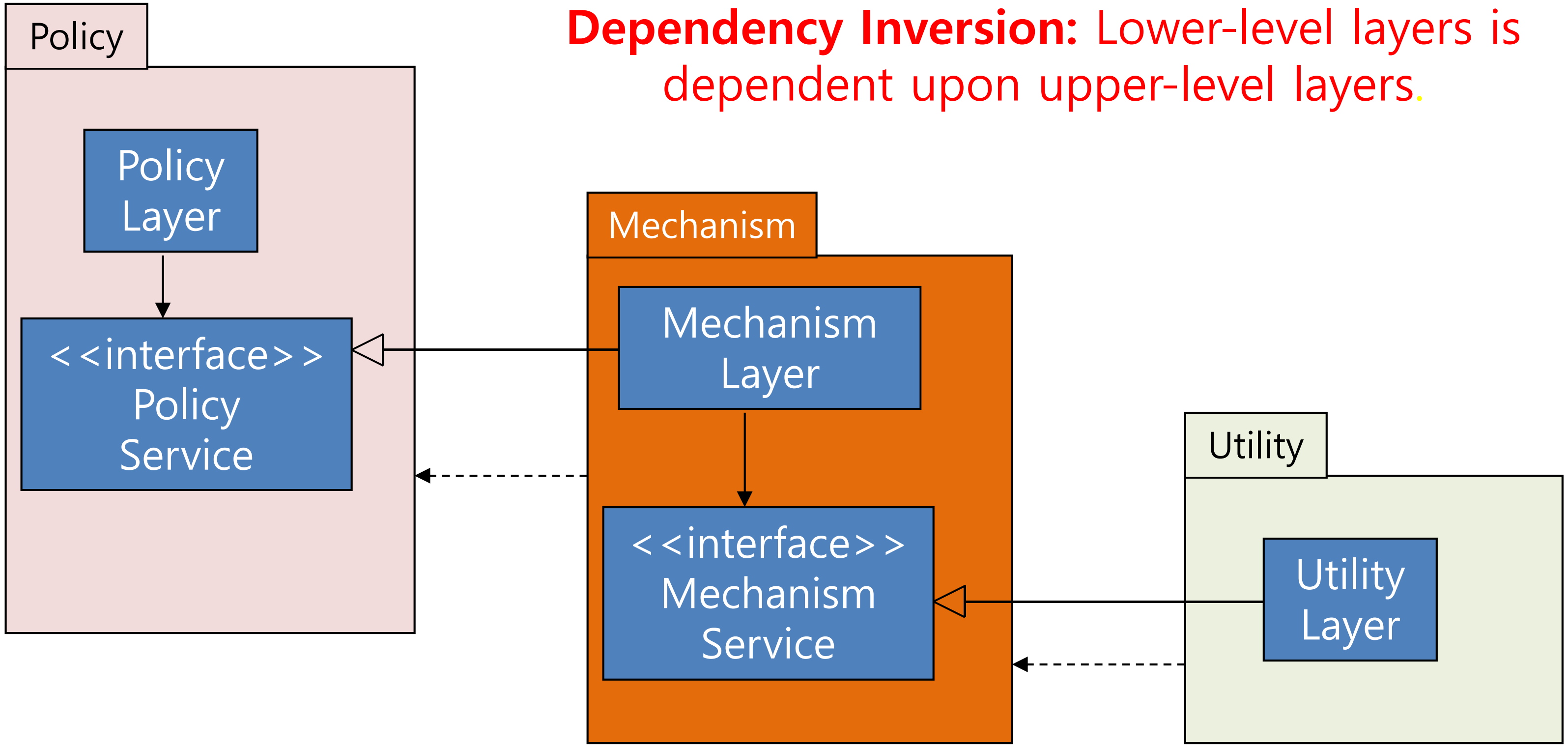


High-level modules make calls to low-level modules.

The upper-level layer is dependent upon lower-level layers.

Figure from [RC]

The dependency inversion principle



Dependency Inversion: Lower-level layers is dependent upon upper-level layers.

The client (upper-level layer) owns the interface, not the lower-level layers

Figure from [RC]

Interface Segregation Principle

You want me to plug this in, where?

Interface Segregation Principle (ISP)

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

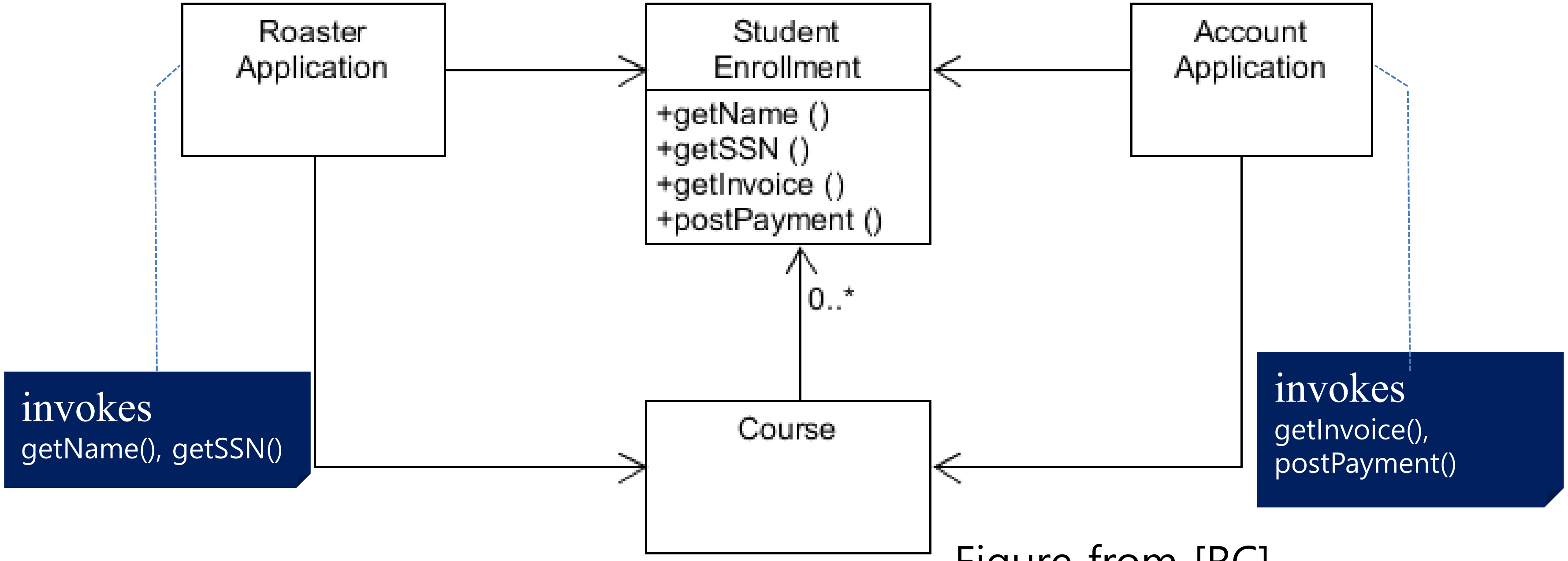
Make fine grained interfaces that are client specific

Fat interface

- When we bundle functions for different clients into one interface/class, we create unnecessary coupling among the clients.
    - When one client causes the interface to change, all other clients are forc ed to recompile.
    - The interfaces of the class can be broken up into groups of methods

- ISP solves non-cohesive interfaces
    - Clients should know only abstract base classes that have cohesive interfaces

ISP Example: Original Design



invokes
getName(), getSSN()

invokes
getInvoice(),
postPayment()

Figure from [RC]

- Suppose that RoasterApplication does not invoke methods getInvoice() or postPayment()
- Also suppose AccountApplication does not invoke the methods getName() or getSSN()
- Requirements change: add a new argument to the postPayment()
- This change force us to recompile and redeploy RoasterApplication, which does not care at all about the postPayment()

ISP Example: Better Design

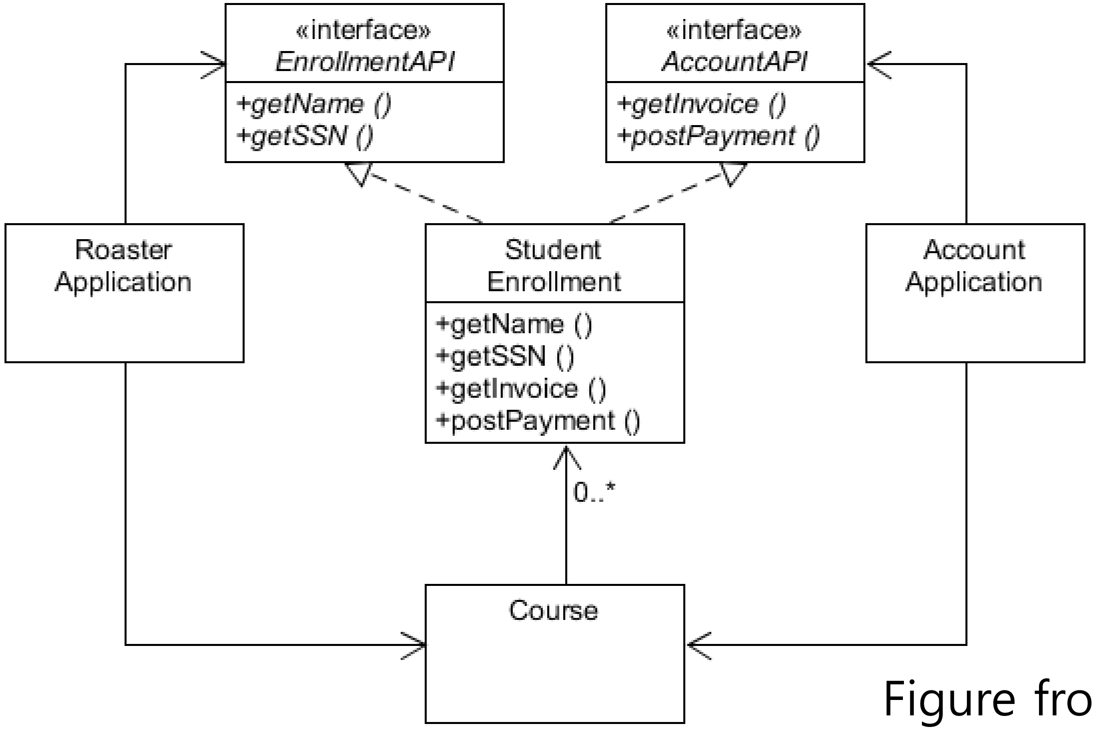


Figure from [RC]

- Now, each user of a StudentEnrollment object is given an interface that provides just the methods that it is interested in
- This protects the user from changes in methods that don't concern it
- It also protects the user from knowing too much about the implementation of the object it is using.

- SOLID
  - Single-Responsibility Principle
  - Open-Closed Principle
  - Liskov Substitution Principle
  - Dependency Inversion Principle
  - Interface Segregation Principle
- Design Principles
  - Help manage dependency
  - Improved maintainability, flexibility, robustness, and reusability
  - Abstraction is important

References

- [GOF] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design patterns: Elements of reusable object-oriented software. Addison-Wesley.
- [HF] Freeman, E., Robson, E., Sierra, K., & Bates, B. (2004). Head First design patterns. O'Reilly.
- [RC] Martin, Robert C. (2002). Agile Software Development: Principles, Patterns, and Practices. Pearson Education.