Good Programming

Not hacking!

Liskov Substitution Principle

How do we know if saying *B* is a subtype of *A* is safe?

Substitution Principle: If *B* is a subtype of *A*, everywhere the code expects an *A*, a *B* can be used instead *and* the program still satisfies its specification

Subtype Condition: Signature Rule

We can use a subtype method where a supertype methods is expected:

- Subtype must implement all of the supertype methods
- Argument types must not be more restrictive
- Result type must be at least as restrictive
- Subtype method must not throw exceptions that are not subtypes of exceptions thrown by supertype

LSP and syntactic interfaces

```
class T {
        Object a() { … }
}

class S extends T {
        @Override String a() { … } √
}
```

More specific classes may have more specific return types

Simple Example

```
class Bird {
  public void fly(){}
  public void eat(){}
}

class Crow extends Bird {}

class Ostrich extends Bird{
  fly(){
    throw new UnsupportedOperationException();
  }
}
```

```
public BirdTest{
  public static void main(String[] args){
    List<Bird> birdList = new ArrayList<Bird>();
    birdList.add(new Bird());
    birdList.add(new Crow());
    birdList.add(new Ostrich());
    letTheBirdsFly ( birdList );
  }
  static void letTheBirdsFly ( List<Bird> birdList ){
    for ( Bird b : birdList ) {
      b.fly();
    }
  }
}
```

What do you think would happen when this code is executed? As soon as an Ostrich instance is passed, it blows up!!! Here the sub type is not replaceable for the super type.

How do we fix such issues?

Factoring out the common features into a separate class can help in creating a hierarchy that confirms to LSP.

In the above scenario we can factor out the fly feature into- Flight and NonFlight birds.

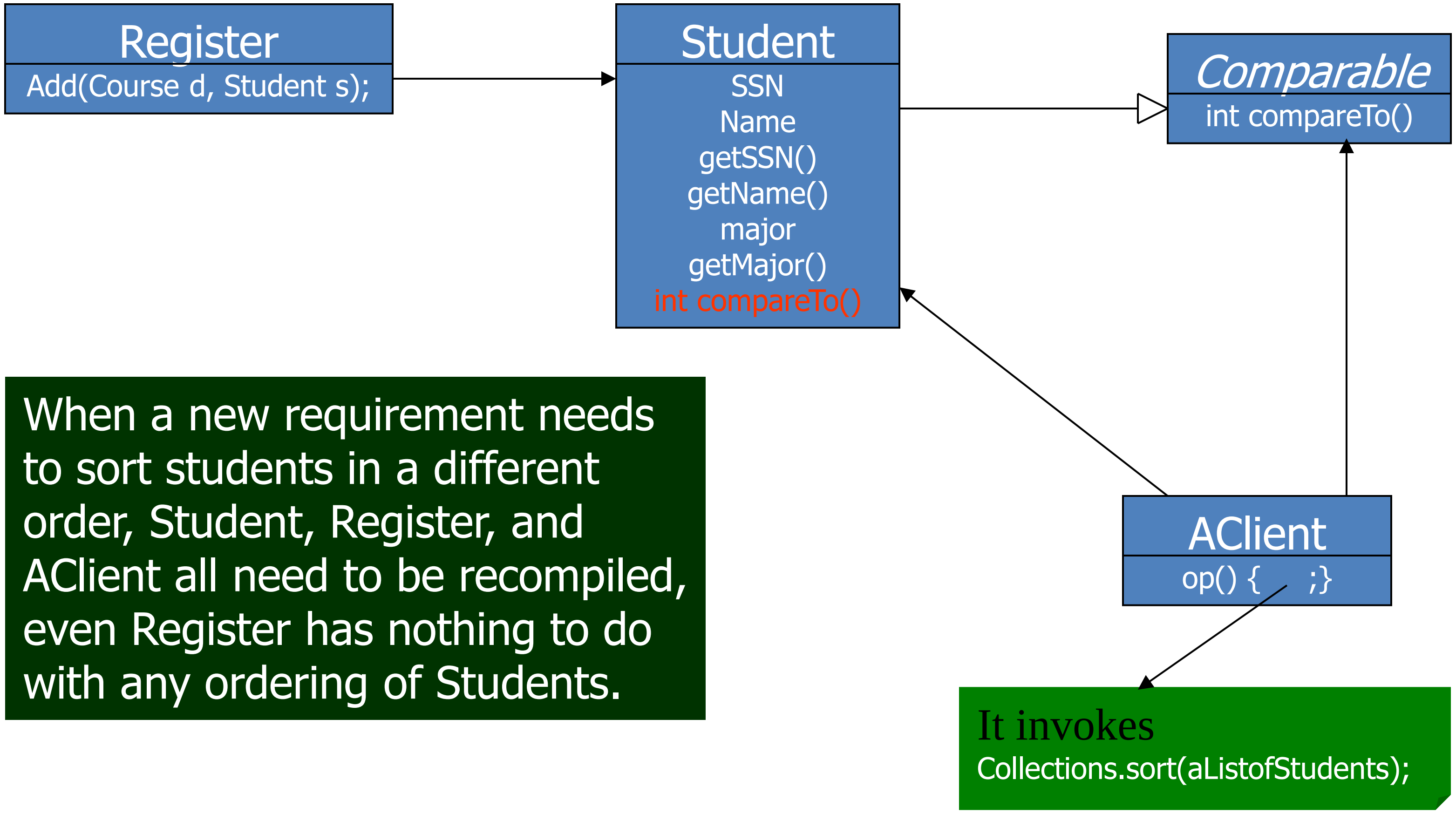
```
class Bird{
 public void eat(){}
}

class FlightBird extends Bird{
  public void fly()()
}
class NonFlight extends Bird{}
```

The single-responsibility principle

- “A class should have only one reason to change.”
- A responsibility = a reason to change
- Separate coupled responsibilities into separate classes

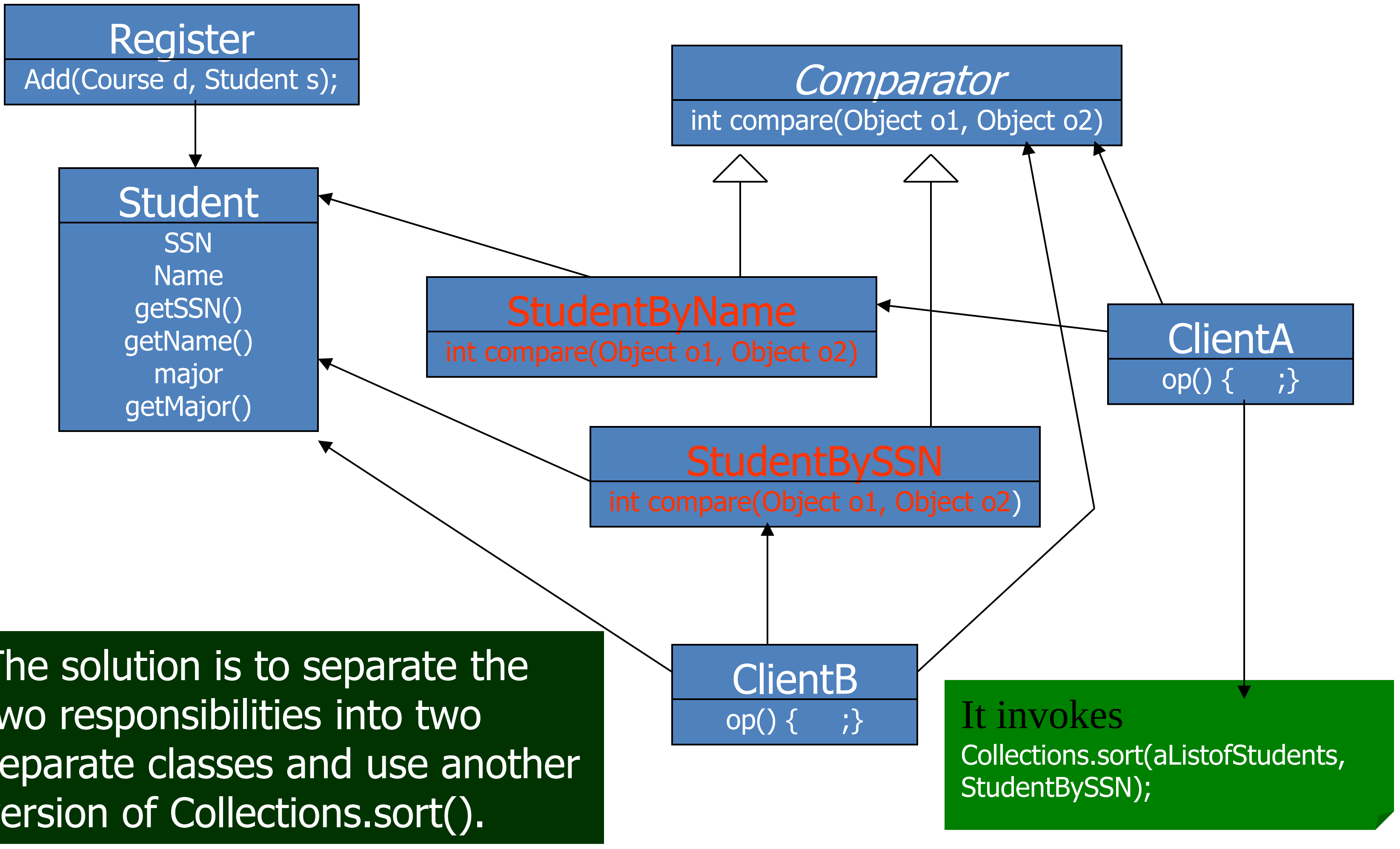
The single-responsibility principle



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

It invokes Collections.sort(aListofStudents);

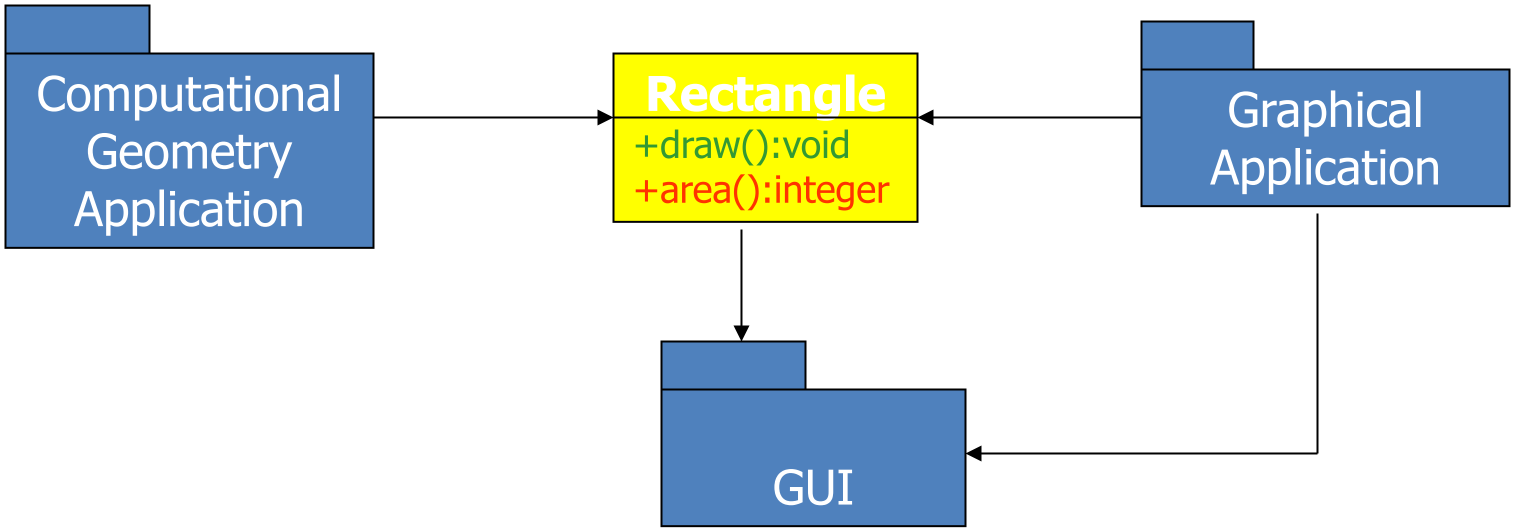
The single-responsibility principle


Register
Add(Course d, Student s);
Student
SSN
Name
getSSN()
getName()
major
getMajor()
Comparator
int compare(Object o1, Object o2)
StudentByName
int compare(Object o1, Object o2)
StudentBySSN
int compare(Object o1, Object o2)
ClientA
op() { ;}
ClientB
op() { ;}

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

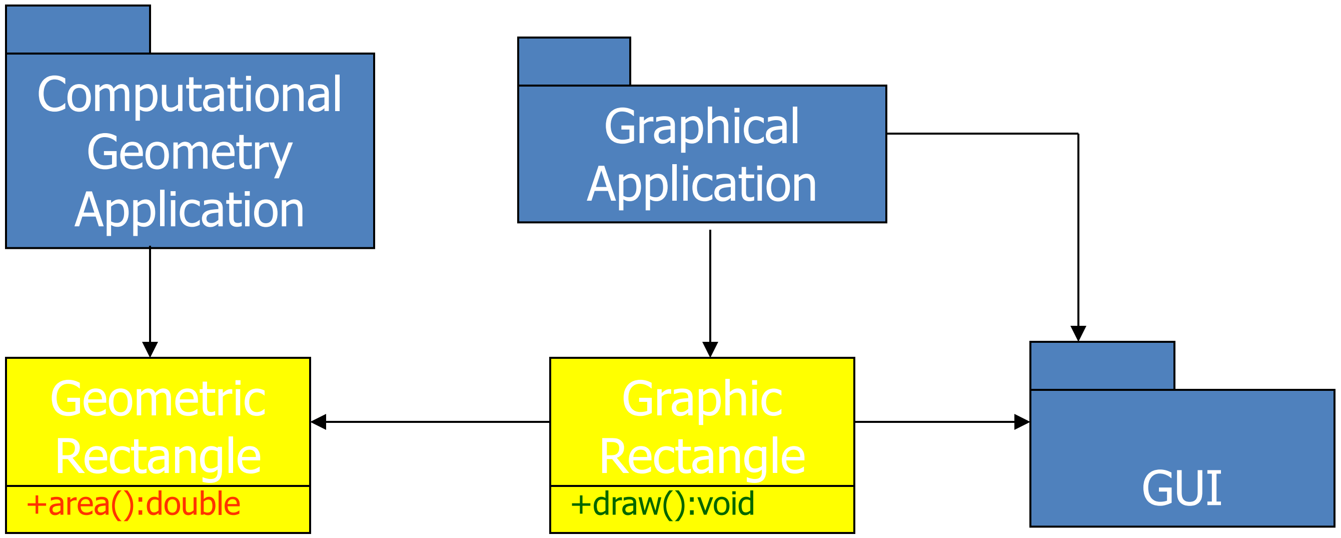
It invokes
Collections.sort(aListofStudents, StudentBySSN);

The single-responsibility principle



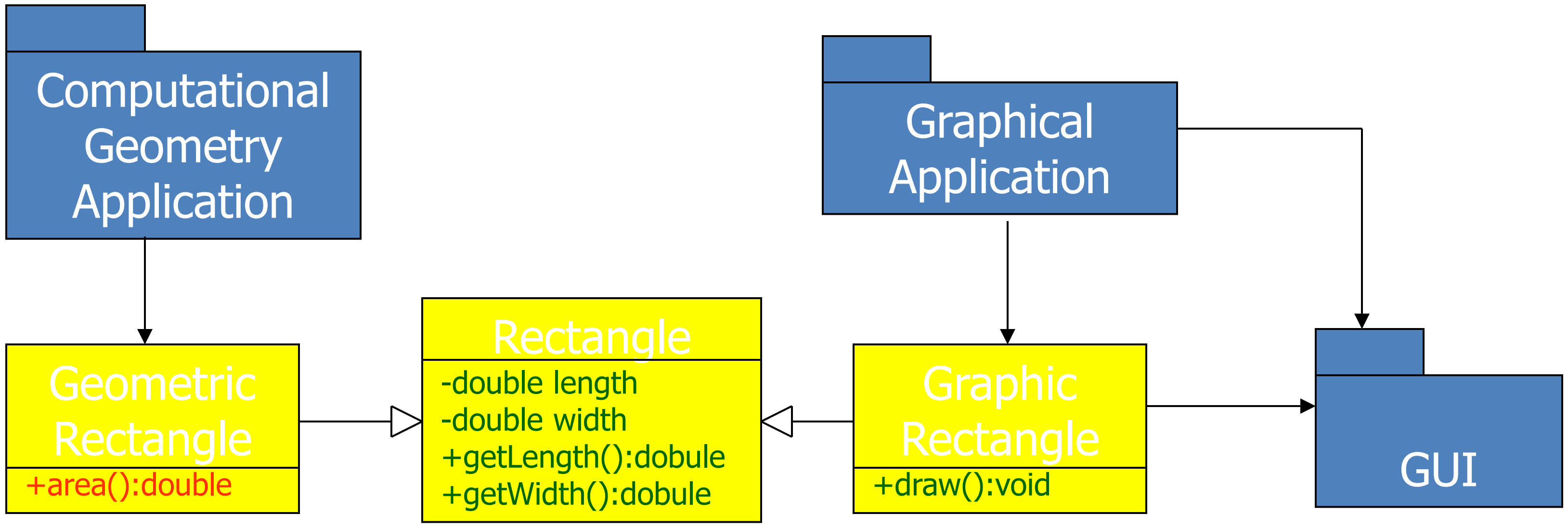
Class Rectangle may be forced to make changes from two different unrelated sources. One is from the Computational Geometry Application (CGA). E.g., adding area function for length and width of type double. The other is from Graphical Application (GA). E.g., add draw() in Windows XP to the existing draw in X Windows. A change from either of the two source would still cause the other application to recompile.

The single-responsibility principle



- 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.

The single-responsibility principle



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.

NOTE: this does not violate LSP, since Rectangle does not have any client.

Open-Closed Principle

- **In object-oriented programming, the open/closed principle states "software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"**

- That is, once a class is written it should not be required to be rewritten unless its specification changes!!!!!

Open-Close

- The Open Close Principle states that the design and writing of the code should be done in a way that new functionality should be added with minimum changes in the existing code. The design should be done in a way to allow the adding of new functionality as new classes, keeping as much as possible existing code unchanged.

?????

- If you write code that requires class B to be updated if class A changes….

- YOU GOT IT WORNG – YOUR SOLUTION NEEDS SCRAPPED!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

- IT IS COMPLETELY UNECONOMICAL!!!!!!!!!!

Bad Code

```
class GraphicEditor {

public void drawShape(Shape s) {
if (s.m_type==1)
drawRectangle(s);
else if (s.m_type==2)
drawCircle(s);
}
public void drawCircle(Circle r)
{....}
public void
drawRectangle(Rectangle r) {....}
}
```

```
class Shape {
int m_type;
}

class Rectangle extends Shape {
Rectangle() {
super.m_type=1;
}
}

class Circle extends Shape {
Circle() {
super.m_type=2;
}
}
```

Good Code

```
class GraphicEditor {
public void drawShape(Shape
s) {
            s.draw();
}
}
```

```
abstract class Shape {
            abstract void draw();
 }

 class Rectangle extends Shape
{
            public void draw() {....
            }
 }
```

Dependency Inversion Principle

- Low level classes, the classes which implement basic and primary operations
- high level classes, the classes which encapsulate complex logic
- A natural way of implementing such structures would be to write low level classes and once we have them to write the complex high level classes.
- What happens if we need to replace a low level class?

Example

- Let's take the classical example of a copy module which reads characters from the keyboard and writes them to the printer device.
- The high level class containing the logic is the Copy class.
- The low level classes are KeyboardReader and PrinterWriter.

Going Wrong

- In a bad design the high level class uses directly and depends heavily on the low level classes.
- In such a case if we want to change the design to direct the output to a new FileWriter class
- we have to make changes in the Copy class.

Back on path

- In order to avoid such problems we can introduce an abstraction layer between high level classes and low level classes.
- Since the high level modules contain the complex logic they should not depend on the low level modules
- so the new abstraction layer should not be created based on low level modules.

Low level modules are to be created based on the abstraction layer.

from high level modules to the low level modules:

High Level Classes -->

Abstraction Layer -->

Low Level Classes

Use it!

Can we have a real example

- We have the manager class which is a high level class, and the low level class called Worker. We need to add a new module to our application to model the changes in the company structure determined by the employment of new specialized workers. We created a new class SuperWorker for this.
- Let's assume the Manager class is quite complex, containing very complex logic. And now we have to change it in order to introduce the new SuperWorker. Let's see the disadvantages:
- we have to change the Manager class (remember it is a complex one and this will involve time and effort to make the changes).

And …

- some of the current functionality from the manager class might be affected.
- the unit testing should be redone.
- All those problems could take a lot of time to be solved and they might induce new errors in the old functionality.
- The situation would be different if the application had been designed following the Dependency Inversion Principle.
- It means we design the manager class, an IWorker interface and the Worker class implementing the IWorker interface.
- When we need to add the SuperWorker class all we have to do is implement the IWorker interface for it. No additional changes in the existing classes.

Bad Code

```
class Manager {
Worker worker;

public void setWorker(Worker
w) {
worker = w;
}

public void manage() {
worker.work();
}
}
```

```
class Worker {
public void work() {
 ....
}
}


class SuperWorker {
public void work() {
....
}
}
```

Good Code

```
class Manager {
IWorker worker;

public void setWorker(IWorker
w) {
worker = w;
}

public void manage() {
worker.work();
}
}
```

```
abstract class IWorker {
abstract void work();
}


class Worker extends IWorker{
public void work() {....
}}

class SuperWorker extends
IWorker{
public void work() {....
}}
```