



CSC40232: SOFTWARE ENGINEERING

Guest Lecturer: Jin Guo
SOLID Principles
sarec.nd.edu/courses/SE2017



Department of
Computer Science and
Engineering



SOLID
Software Development is not a Jenga game

<http://www.kirkk.com/modularity/2009/12/solid-principles-of-class-design/>
<http://blog.gauffin.org/2012/05/11/solid-principles-with-real-world-examples/>
<http://code.tutsplus.com/series/the-solid-principles--cms-634>

The SOLID Principles

- Single Responsibility Principle
- Open Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

3



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

4

Single Responsibility

A class should have only one reason to change.

```
public class Book {
    int currentPage = 0;

    public String getTitle(){
        return "A Great book";
    }

    public String getAuthor(){
        return "John Doe";
    }

    public int turnPage(){
        return currentPage++;
    }

    public void printCurrentPage(){
        // Print page
    }
}
```

What about this class?



5

Single Responsibility



Separate out the responsibilities...

```
public class Book2 {
    int currentPage = 0;
    Printer printer = new PlainTextPrinter();

    public String getTitle(){
        return "A Great book";
    }

    public String getAuthor(){
        return "John Doe";
    }

    public int turnPage(){
        return currentPage++;
    }

    public String getCurrentPage(){
        // Print page
        String currentPageText = null;
        return currentPageText;
    }

    public void save(){
        printer.printPage(getCurrentPage());
    }
}

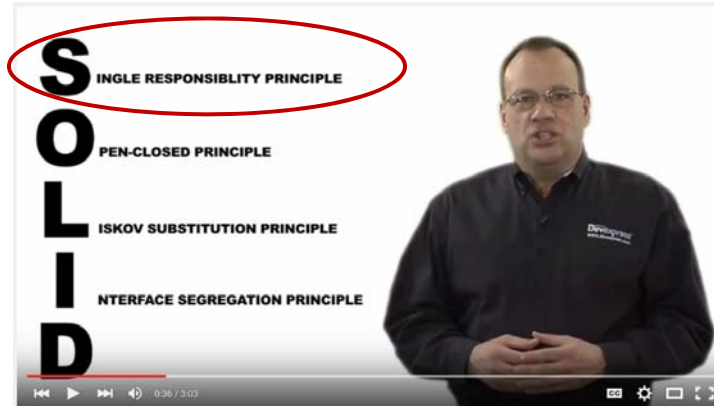
public interface Printer {
    public void printPage(String str);
}

public class PlainTextPrinter implements Printer{
    @Override
    public void printPage(String str) {
        // Print page
    }
}

public class HtmlPrinter implements Printer {
    @Override
    public void printPage(String str) {
        // Print page
    }
}
```

6

Single Responsibility



<https://www.youtube.com/watch?v=nyxaChZ1row&list=PL4CE9F710017EA77A&index=1>

7

Open Closed Principle



If you modify a class you may break the API/Contract such that classes that depend on it may fail.

It is better to reuse the class to add new features through inheritance or aggregation.

This way the base class is untouched.


8

Open Closed Principle

```

public void Parse() throws IOException
{
    BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
    StringBuilder scope = new StringBuilder();
    String line = reader.readLine();
    while (line != null)
    {
        char[] chars = line.toCharArray();
        char ch = chars[0];
        switch (ch)
        {
            case '$':
                // Process the entire "line" and add to a collection of KeyValuePair.
                AddToVariables(line);
                break;
            case '!':
                // Call various functions according to the ! qualifier
                if (line == "!execute")
                    ExecuteScope(scope);
                else if (line == "!custom_command")
                    RunCustomCommand(line, scope);
                else if (line == "!single_line_directive")
                    ProcessDirective(line);

                scope = new StringBuilder();
                break;
            default:
                // No processing directive, i.e. add the "line"
                // to the current scope.
                scope.append(line);
                break;
        }
        line = reader.readLine();
    }
}
    
```



9

Open Closed Principle

```

public interface IMyHandler {
    void Process(IProcessContext context, String line);
}

public class Parser{
    private Map<Character, IMyHandler> _handlers = new HashMap<Character, IMyHandler>();
    private IMyHandler _defaultHandler;


    public void Add(Character controlCharacter, IMyHandler handler){
        _handlers.put(controlCharacter, handler);
    }

    private void Parse(BufferedReader buffer) throws IOException
    {
        StringBuilder scope = new StringBuilder();
        IPProcessContext context = null; // create your context here.
        String line = buffer.readLine();
        while (line != null) {
            IMyHandler handler = null;
            char contextSelector = line.toCharArray()[0];
            if(!_handlers.containsKey(contextSelector))
                handler = _defaultHandler;
            else
                handler = _handlers.get(contextSelector);

            handler.Process(context, line);
            line = buffer.readLine();
        }
    }
}
    
```

Step 1. Create an interface

Step 2. Get the correct concrete handler and delegate the processing to it.

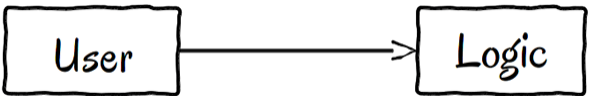


10

Open Closed Principle

Open Closed Principle

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



```
graph LR; User[User] --> Logic[Logic]
```

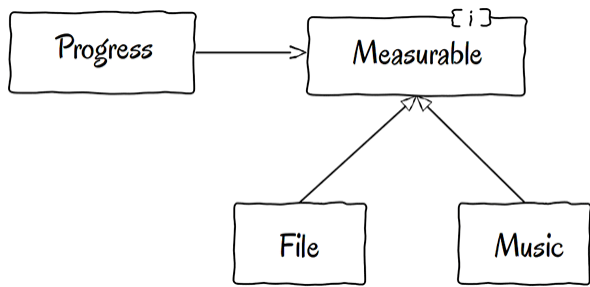
Violates OCP because user is tied directly to Logic class.

11

Open Closed Principle

Open Closed Principle

The most common solution..

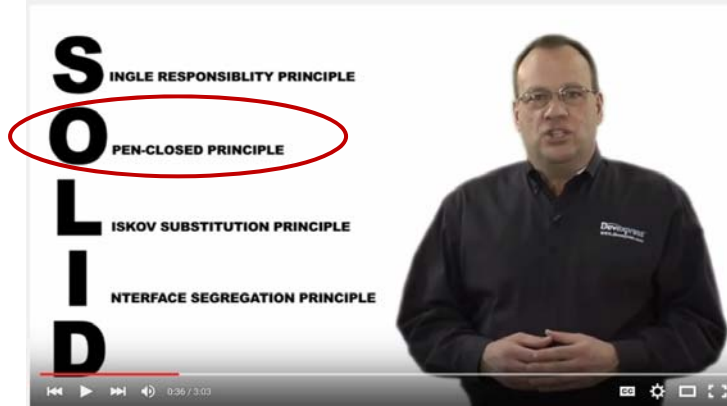


```
graph TD; Progress[Progress] --> Measurable[Measurable]; File[File] --> Measurable; Music[Music] --> Measurable
```

Correct

12

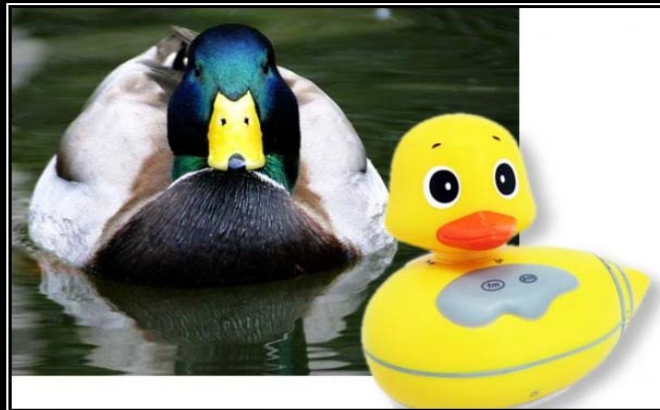
Single Responsibility



<https://www.youtube.com/watch?v=EpvfSQEJq68&list=PL4CE9F710017EA77A&index=2>

13

Liskov Substitution Principle



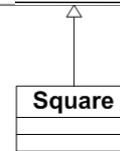
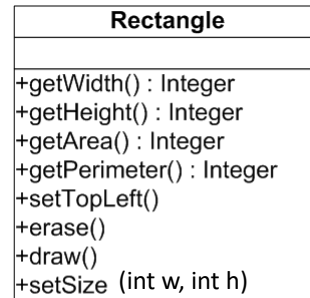
LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

14

Liskov Substitution Principle

- Should inheritance be used between the square and rectangle classes?
- Every square 'is-a' rectangle.
- Good opportunities for re-use.



15

```

package example1;
import java.awt.Graphics;

public class Rectangle {
    private int x, y, width, height;
    public Rectangle(int x, int y, int w, int h){
        this.x = x; this.y=y; this.width=w; this.height=h;
    }
    public int getWidth() { return width;}
    public int getHeight() { return height;}
    public int getArea() { return width * height;}
    public int getPerimeter() { return 2 * (width + height);}
    public void setTopLeft(int newX, int newY) { x = newX; y = newY;}
    public void erase(Graphics g) {}
    public void draw(Graphics g){}
    public void setSize(int w, int h) { width = w; height = h;}
}

class Square extends Rectangle {
    public Square (int x, int y, int side ){
        super(x,y,side,side);
    }
}

```



However there are serious problems in the design because the square inherits unwanted methods such as `setSize(int w, int h)`.

Patching the problem

- We can easily override the unwanted behavior:

```
Public void setSize(int w, int h) {  
    width = h;  
    height = h;  
}
```

- Unfortunately the code now behaves in an unexpected way:

```
public void stretch(Rectangle r, int dx, Graphics g){  
    r.erase(g);  
    r.setSize(r.getWidth()+dx, r.getHeight());  
    r.draw(g);  
}
```

which is NOT an elegant solution. Code should behave in ways expected by the programmer.

17

Design principle of LEAST ASTONISHMENT.

- A designer of a class or interface **MUST** specify the semantics of each method.
- All subclasses must conform to this expected behavior.

18

Liskov Substitution Principle

- Inheritance should ensure that any property proved about super-type objects also holds for subtype objects.
- Let $q(x)$ be a property provable about objects x of type T .
- Then $q(y)$ should be true for objects y of type S where S is a subtype of T .

19

Liskov Substitution Principle

- Preconditions cannot be strengthened in a subclass - you cannot have a subclass that has stronger preconditions than its superclass.
- Postconditions cannot be weakened in a subclass - you cannot have a subclass that has weaker postconditions than its superclass.
- No new exceptions should be thrown by methods of the subclass, except where those exceptions are themselves subtypes of exceptions thrown by the methods of the superclass.

20

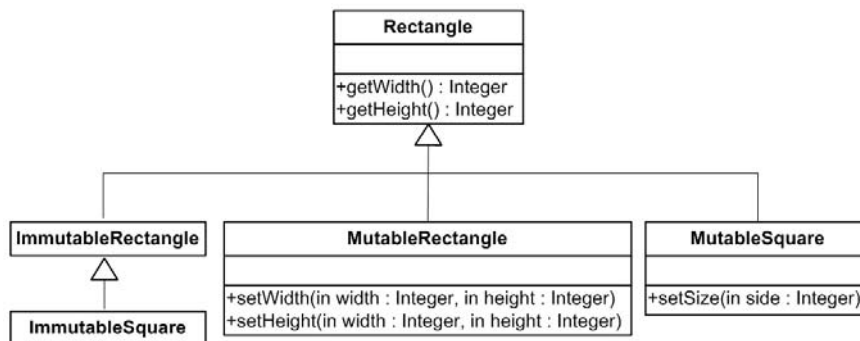
Liskov Substitution Principle

- Subtype methods should ‘look like’ corresponding supertype method
- Subtype methods should extend behavior of supertype method in a consistent manner
- Subtype methods should not change or eliminate supertype method properties

Adhering to this principle yields well-behaved subclasses that support parametric polymorphism

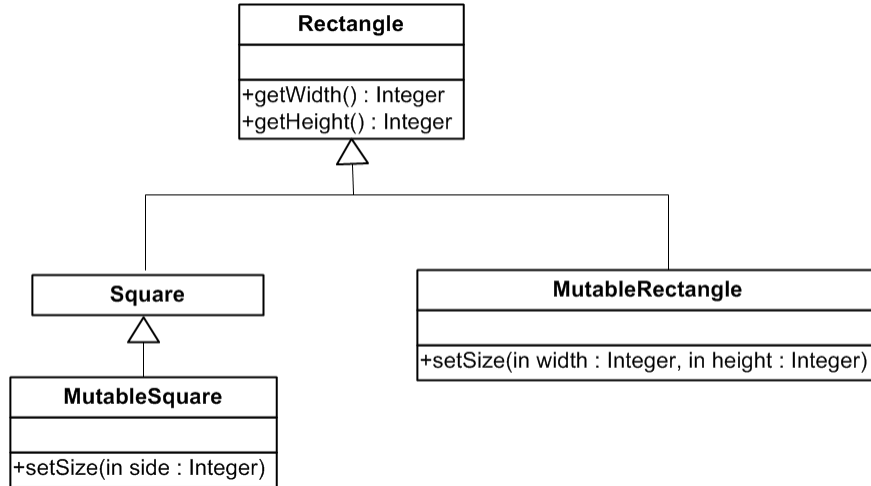
21

A possible solution

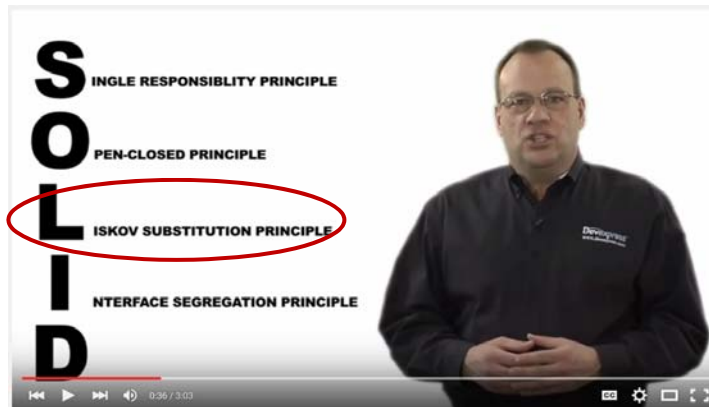


- Gets around the problem of square and rectangle’s behavior not matching (i.e. public methods of Rectangle were not all appropriate for square), by creating a more sophisticated hierarchy.

Another possible solution

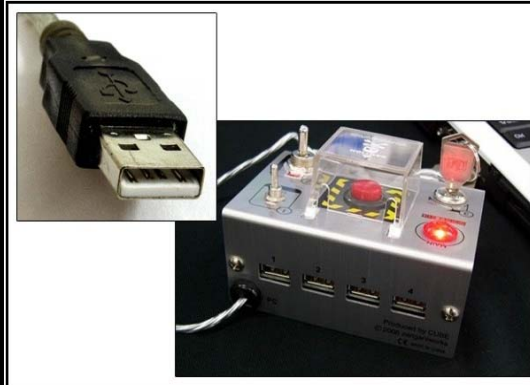


Single Responsibility



<https://www.youtube.com/watch?v=LkqWvVXNDKw&index=3&list=PL4CE9F710017EA77A>

Interface Segregation Principle



INTERFACE SEGREGATION PRINCIPLE
You Want Me To Plug This In, Where?

There is nothing that says that there should be a one-to-one mapping between classes and interfaces. It's in fact much better if you can create several smaller interfaces instead

25

Interface Segregation Principle

- Any interface we define should be cohesive.
- There must be some kind of interface which a client can rely on. Its purpose is to communicate to the client code how the module should be used.
- So what should go into the interface? In this example we expose all functionalities that we'd like to offer.

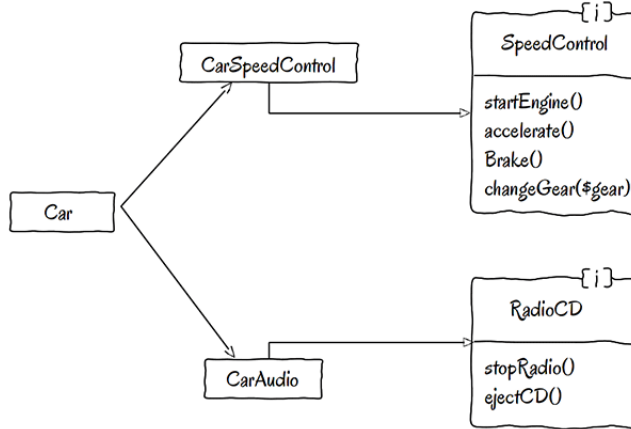
```

interface VEHICLE {
    startEngine()
    accelerate()
    brake()
    lightsOn()
    signalLeft()
    signalRight()
    changeGear($gear)
    stopRadio()
    ejectCD()
}
    
```



26

Interface Segregation Principle

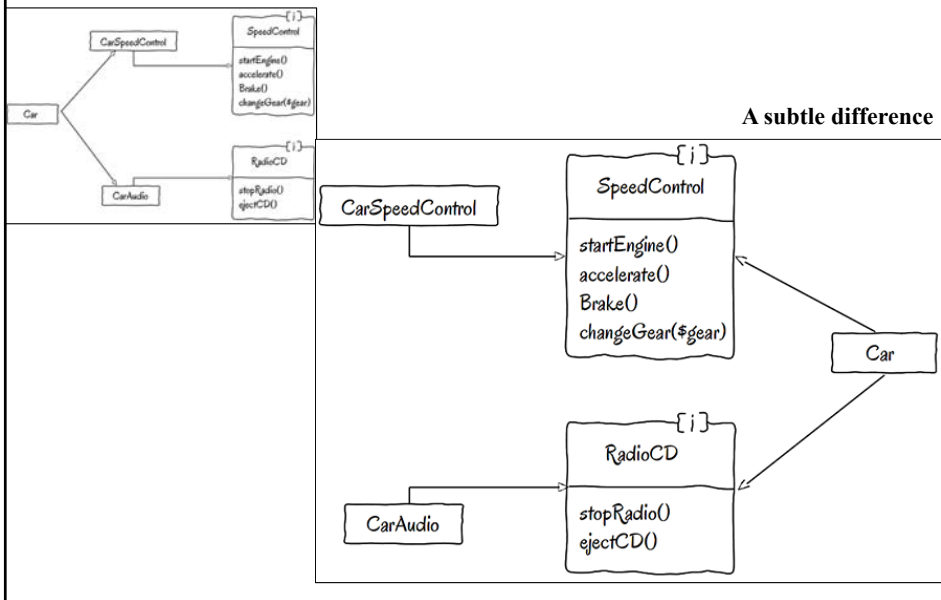


We could break the interface into pieces specialized to each implementation.

Various cars can incorporate (aggregate) different objects.

The car uses the implementations but depends on the interfaces.

Interface Segregation Principle



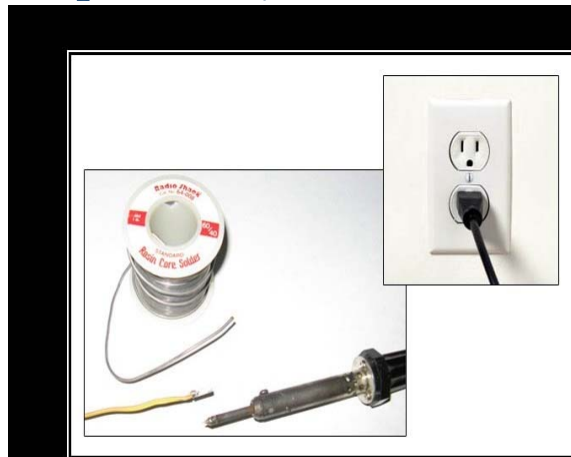
Single Responsibility



<https://www.youtube.com/watch?v=dmKvJyihAQ&index=4&list=PL4CE9F710017EA77A>

29

Dependency Inversion Principle



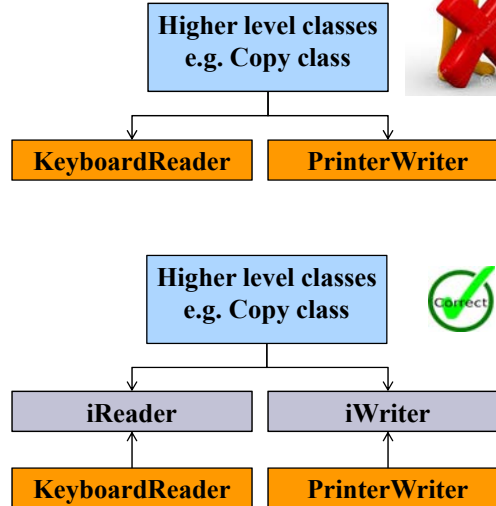
DEPENDENCY INVERSION PRINCIPLE
Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Let the caller create the dependencies instead of letting the class itself create the dependencies. Hence inverting the dependency control (from letting the class control them to letting the caller control them).

30

Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.



31

What about this?

```

class Worker {
    public void work() {
        // ....working
    }
}

class Manager {
    Worker worker;
    public void setWorker(Worker w) {
        worker = w;
    }
    public void manage() {
        worker.work();
    }
}

class SuperWorker {
    public void work() {
        //.... working much more
    }
}
    
```



32

Fixed to comply with DIP

```
// Dependency Inversion Principle - Good example
interface IWorker {
    public void work();
}

class Worker implements IWorker{
    public void work() {
        // ....working
    }
}

class SuperWorker implements IWorker{
    public void work() {
        //.... working much more
    }
}

class Manager {
    IWorker worker;

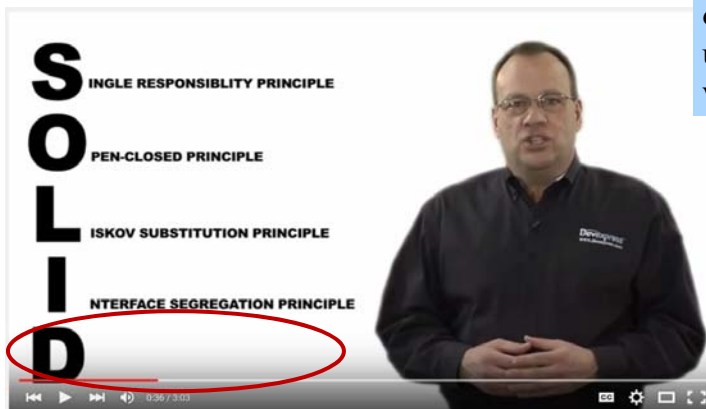
    public void setWorker(IWorker w) {
        worker = w;
    }

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



33

Single Responsibility



Deferring this discussion until next week.

<https://www.youtube.com/watch?v=d4uBBvnreXw&list=PL4CE9F710017EA77A&index=5>

34

