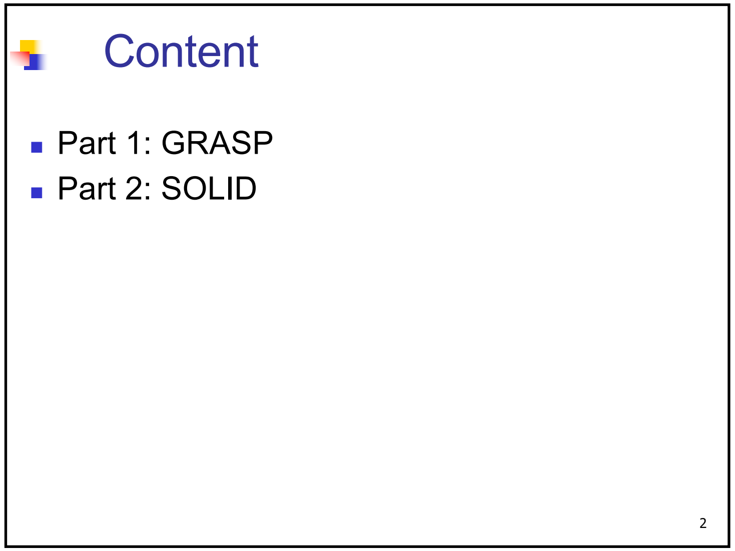
13. Design principles: GRASP & SOLID

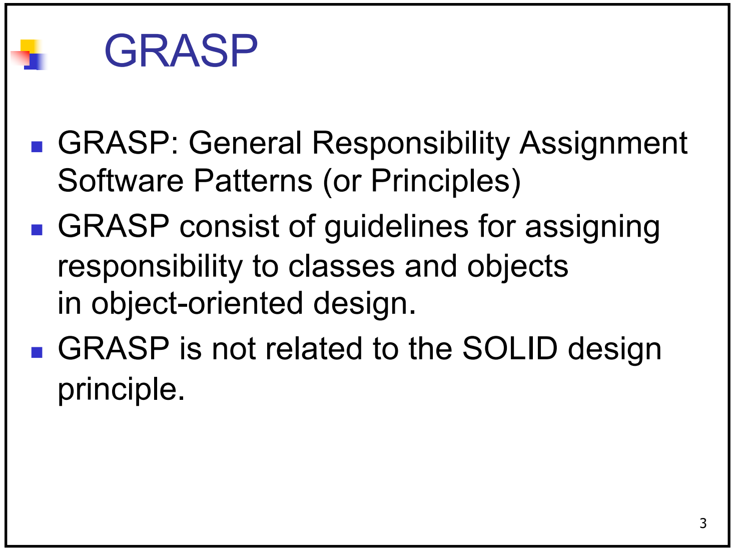
1



Content

- Part 1: GRASP
- Part 2: SOLID

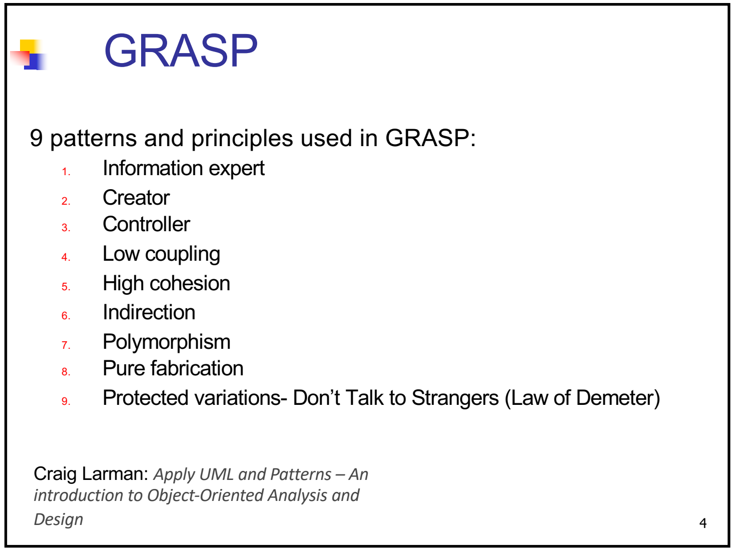
2



GRASP

- GRASP: General Responsibility Assignment Software Patterns (or Principles)
- GRASP consist of guidelines for assigning responsibility to classes and objects in object-oriented design.
- GRASP is not related to the SOLID design principle.

3



GRASP

9 patterns and principles used in GRASP:

1. Information expert
2. Creator
3. Controller
4. Low coupling
5. High cohesion
6. Indirection
7. Polymorphism
8. Pure fabrication
9. Protected variations- Don’t Talk to Strangers (Law of Demeter)

Craig Larman: *Apply UML and Patterns – An introduction to Object-Oriented Analysis and Design*

4

Information Expert

- Given an object o, which responsibilities can be assigned to o?
- Expert principle says – assign those responsibilities to o for which o has the information to fulfill that responsibility.
- They have all the information needed to perform operations, or in some cases they collaborate with others to fulfill their responsibilities.

5

5

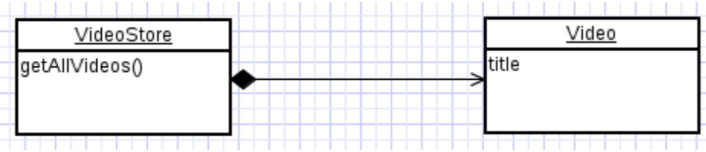
Example for Expert

- Assume we need to get all the videos of a VideoStore.
- Since VideoStore knows about all the videos, we can assign this responsibility of giving all the videos can be assigned to VideoStore class.
- VideoStore is the information expert.

6

6

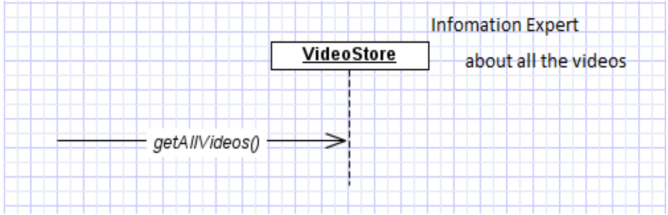
Example for Expert



7

7

Example for Expert



8

8

Creator

- Who creates an Object? Or who should create a new instance of some class?
- Container" object creates "contained" objects.
- Decide who can be creator based on the objects association and their interaction: Assign class B the responsibility to create object A if one of these is true (more is better)
  - B contains or compositely aggregates A
  - B records A
  - B closely uses A
  - B has the initializing data for A

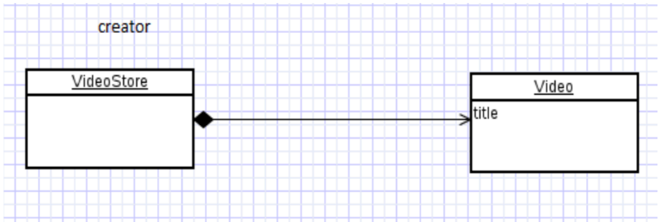
9

Example for Creator

- Consider VideoStore and Video in that store.
- VideoStore has an aggregation association with Video. I.e, VideoStore is the container and the Video is the contained object.
- So, we can instantiate video object in VideoStore class

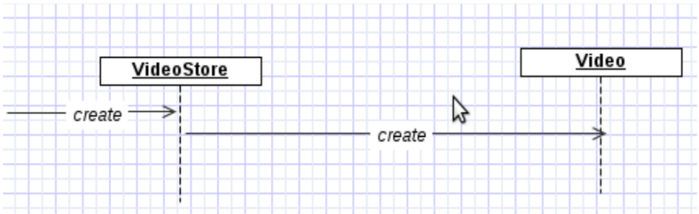
10

Example diagram



11

Example for creator



12

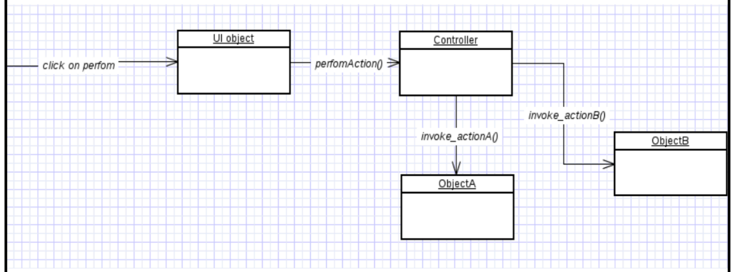
Controller

- Deals with how to delegate the request from the UI layer objects to domain layer objects.
- when a request comes from UI layer object, Controller pattern helps us in determining what is that first object that receive the message from the UI layer objects.
- This object is called controller object which receives request from UI layer object and then controls/coordinates with other object of the domain layer to fulfill the request.
- It delegates the work to other class and coordinates the overall activity

Controller

- We can make an object as Controller, if
  - Object represents the overall system (facade controller)
  - Object represent a use case, handling a sequence of operations (use case or session controller).
- Benefits
  - can reuse this controller class.
  - Can use to maintain the state of the use case.
  - Can control the sequence of the activities

Example for Controller



Bloated Controllers

- Controller class is called bloated, if
  - The class is overloaded with too many responsibilities.
    - Solution – Add more controllers
  - Controller class also performing many tasks instead of delegating to other class.
    - Solution – controller class has to delegate things to others.

Low Coupling

- How strongly the objects are connected to each other?
- Coupling – object depending on other object.
- When depended upon element changes, it affects the dependant also.
- Low Coupling – How can we reduce the impact of change in depended upon elements on dependant elements.
- Prefer low coupling – assign responsibilities so that coupling remain low.
- Minimizes the dependency hence making system maintainable, efficient and code reusable

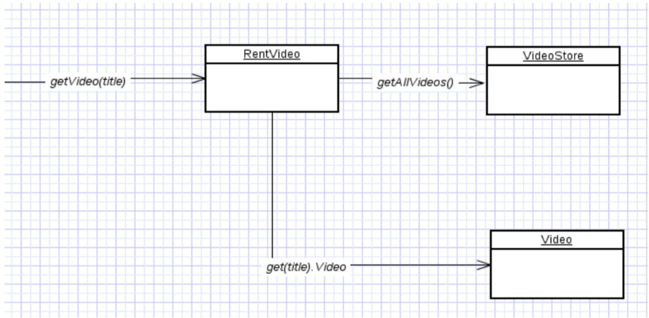
Low coupling

Two elements are coupled, if

- One element has aggregation/composition association with another element.
- One element implements/extends other element.

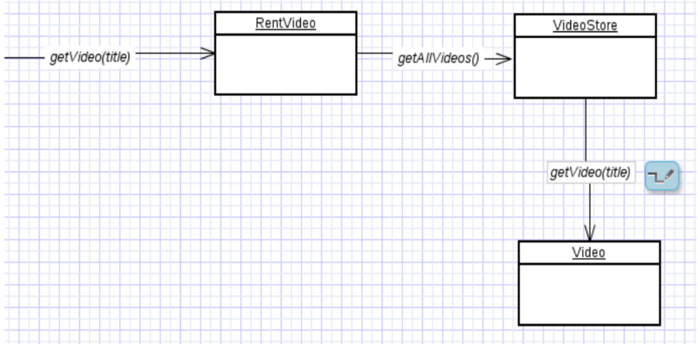
Example for poor coupling

- Here class Rent knows about both VideoStore and Video objects. Rent is depending on both the classes.



Example for low coupling

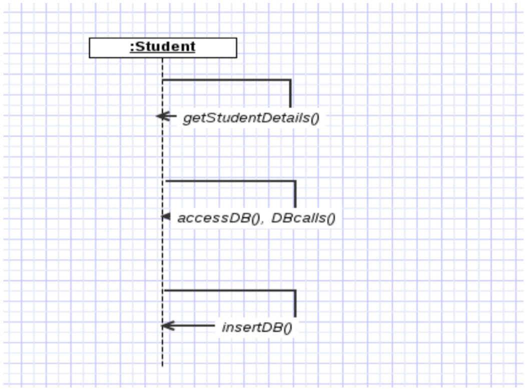
- VideoStore and Video class are coupled, and Rent is coupled with VideoStore. Thus providing low coupling



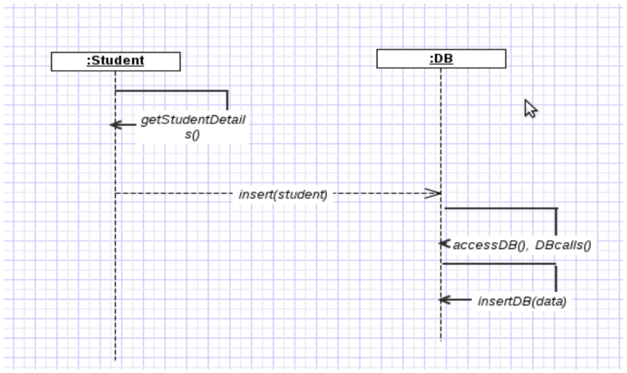
High Cohesion

- How are the operations of any element are functionally related?
- Related responsibilities in to one manageable unit.
- Prefer high cohesion
- Clearly defines the purpose of the element
- Benefits
  - Easily understandable and maintainable.
  - Code reuse
  - Low coupling

Example for low cohesion



Example for High Cohesion

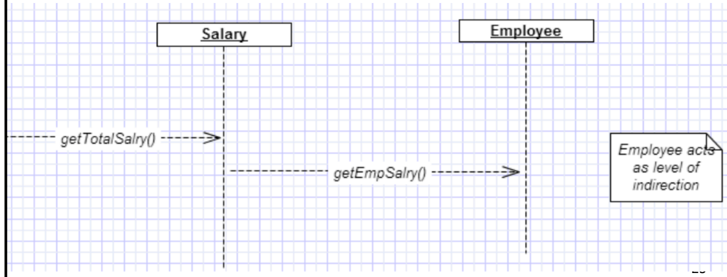


Indirection

- How can we avoid a direct coupling between two or more elements.
- Indirection introduces an intermediate unit to communicate between the other units, so that the other units are not directly coupled.
- Benefits: low coupling
- Example: Adapter, Facade, Obserever

Example for Indirection

- Here polymorphism illustrates indirection
- Class Employee provides a level of indirection to other units of the system

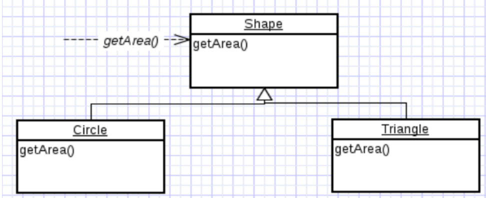


Polymorphism

- How to handle related but varying elements based on element type?
- Polymorphism guides us in deciding which object is responsible for handling those varying elements.
- Benefits: handling new variations will become easy.

Example for Polymorphism

- the getArea() varies by the type of shape, so we assign that responsibility to the subclasses.



- By sending message to the Shape object, a call will be made to the corresponding sub class object – Circle or Triangle

Pure Fabrication

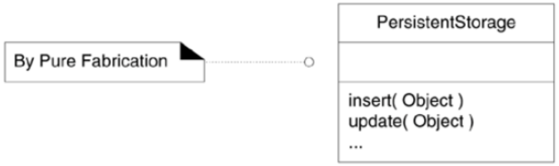
- Problem: Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling
- Solution: Assign a highly cohesive set of responsibilities to an artificial or convenience "behavior” class that does not represent a problem domain concept — something made up, in order to support high cohesion, low coupling, and reuse.

Example 1

- We need to save Sale instances in a relational database. By Information Expert, we assign this responsibility to the Sale class itself, because the sale has the data that needs to be saved. But
  - the Sale class becomes incohesive.
  - The Sale class has to be coupled to the relational database interface (such as JDBC in Java technologies), so its coupling goes up.
  - Saving objects in a relational database is a very general task. Placing these responsibilities in the Sale class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.

Example 1

- Solution: create a new class (PersistentStorage) that is solely responsible for saving objects
- Advantages:
  - The Sale remains well-designed, with high cohesion and low coupling.
  - The PersistentStorage class is relatively cohesive, having the sole purpose of storing or inserting objects in a persistent storage medium.
  - The PersistentStorage class is a very generic and reusable object.

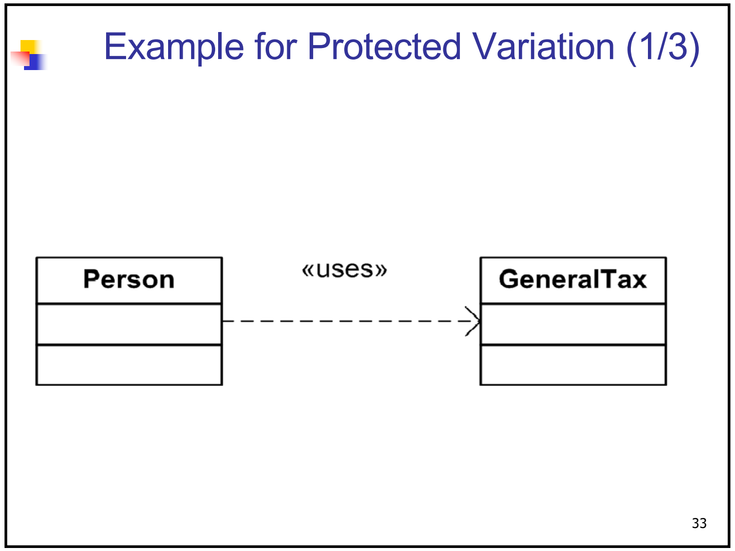


Example 2

```
interface IForeignExchange {
   List<ConversionRate> getConversionRates();
}
class ConversionRate{
   private String from;
   private String to;
   private double rate;
   public ConversionRate(String from, String to, double rate) {
      this.from = from;
      this.to = to;
      this.rate = rate;
   }
}
public class ForeignExchange implements IForeignExchange {
   public List<ConversionRate> getConversionRates() {
      List<ConversionRate> rates = ForeignExchange.getConversionRatesFromExternalApi();
      return rates;
   }
   private static List<ConversionRate> getConversionRatesFromExternalApi() {
      //  Communication with external API. Here is only mock.
      List<ConversionRate> conversionRates = new ArrayList<ConversionRate>();
      conversionRates.add(new ConversionRate("USD", "EUR", 0.88));
      conversionRates.add(new ConversionRate("EUR", "USD", 1.13));
      return conversionRates;
   }
}
```

Protected Variation

- Problem: How to design objects, subsystems and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
- Solution: Identify points of predicted variation or instability, assign responsibilities to create a stable interface around them.



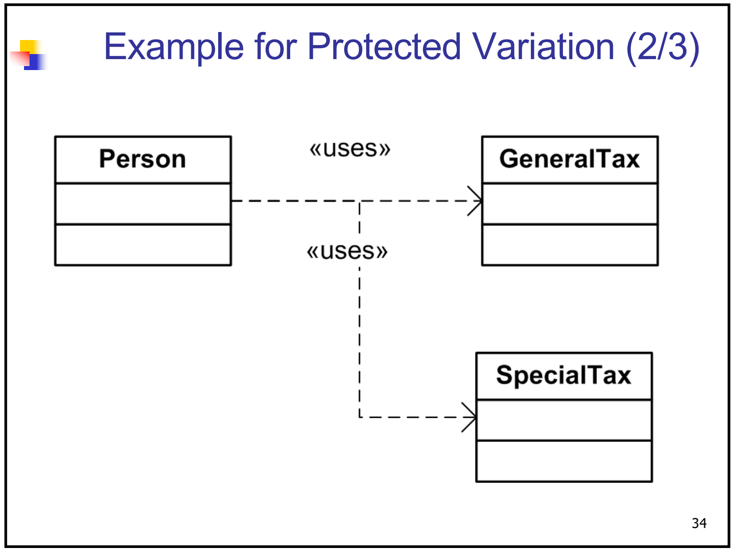
Example for Protected Variation (1/3)

Person

«uses»

GeneralTax

33



Example for Protected Variation (2/3)

Person

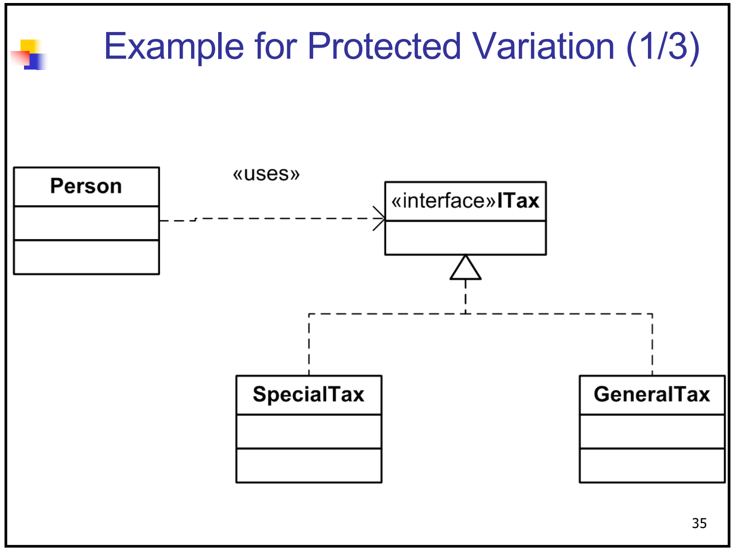
«uses»

GeneralTax

«uses»

SpecialTax

34



Example for Protected Variation (1/3)

Person

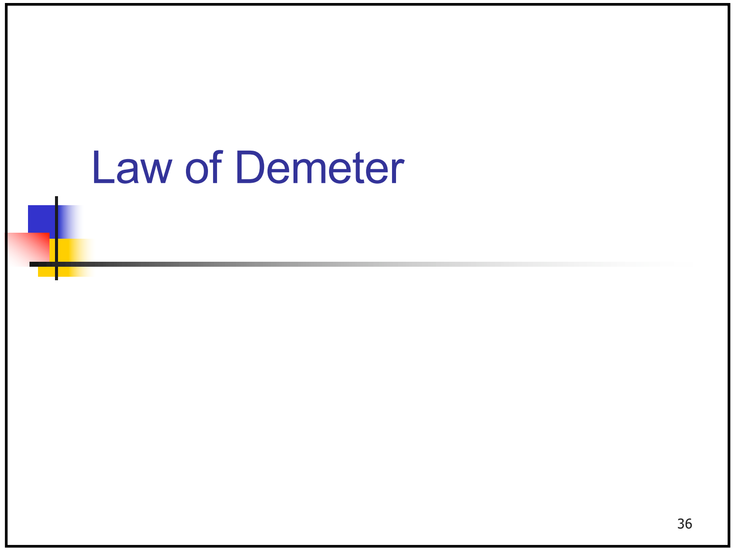
«uses»

«interface»**ITax**

SpecialTax

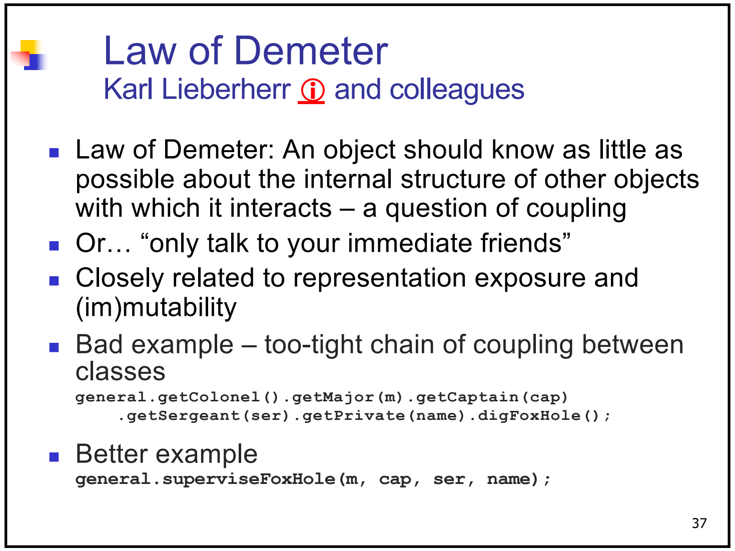
GeneralTax

35



Law of Demeter

36



Law of Demeter
Karl Lieberherr ⓘ and colleagues

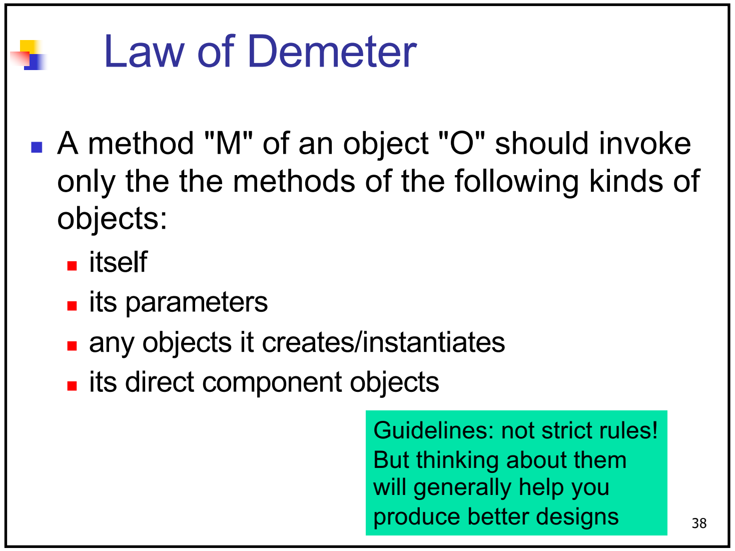
- Law of Demeter: An object should know as little as possible about the internal structure of other objects with which it interacts – a question of coupling
- Or… "only talk to your immediate friends"
- Closely related to representation exposure and (im)mutability
- Bad example – too-tight chain of coupling between classes

```
general.getColonel().getMajor(m).getCaptain(cap)
    .getSergeant(ser).getPrivate(name).digFoxHole();
```

- Better example

```
general.superviseFoxHole(m, cap, ser, name);
```

37

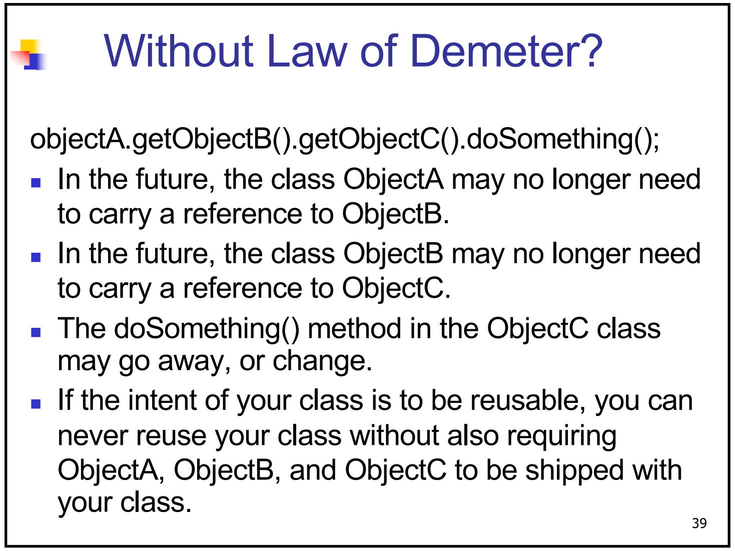


Law of Demeter

- A method "M" of an object "O" should invoke only the the methods of the following kinds of objects:
  - itself
  - its parameters
  - any objects it creates/instantiates
  - its direct component objects

Guidelines: not strict rules! But thinking about them will generally help you produce better designs

38

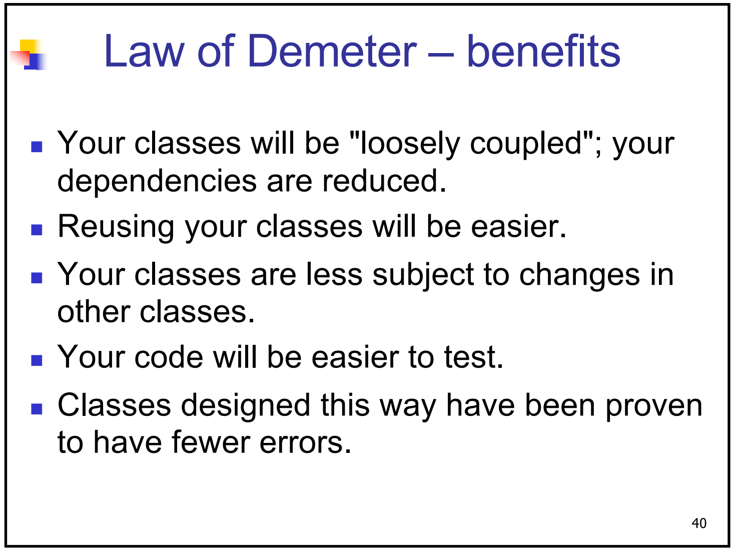


Without Law of Demeter?

objectA.getObjectB().getObjectC().doSomething();

- In the future, the class ObjectA may no longer need to carry a reference to ObjectB.
- In the future, the class ObjectB may no longer need to carry a reference to ObjectC.
- The doSomething() method in the ObjectC class may go away, or change.
- If the intent of your class is to be reusable, you can never reuse your class without also requiring ObjectA, ObjectB, and ObjectC to be shipped with your class.

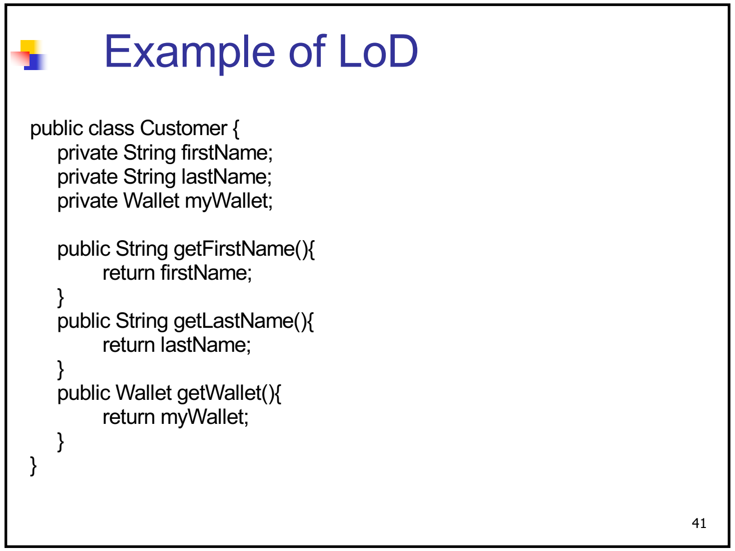
39



Law of Demeter – benefits

- Your classes will be "loosely coupled"; your dependencies are reduced.
- Reusing your classes will be easier.
- Your classes are less subject to changes in other classes.
- Your code will be easier to test.
- Classes designed this way have been proven to have fewer errors.

40

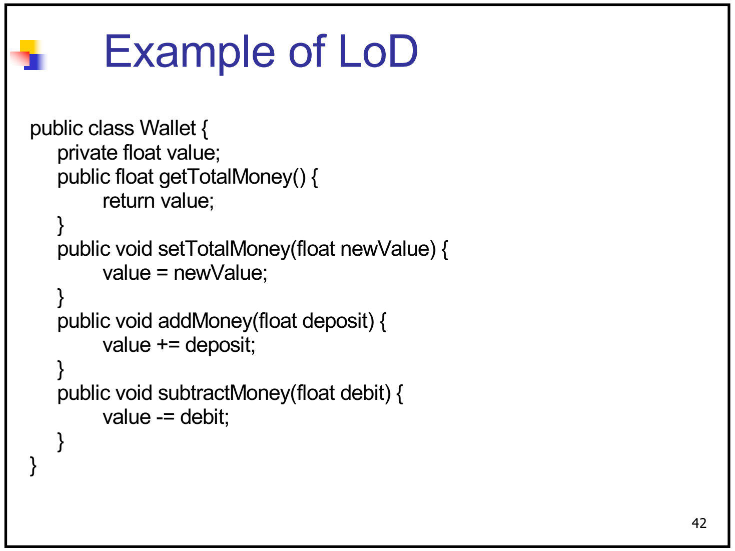


Example of LoD

```
public class Customer {
    private String firstName;
    private String lastName;
    private Wallet myWallet;

    public String getFirstName(){
          return firstName;
    }
    public String getLastName(){
          return lastName;
    }
    public Wallet getWallet(){
          return myWallet;
    }
}
```

41

41

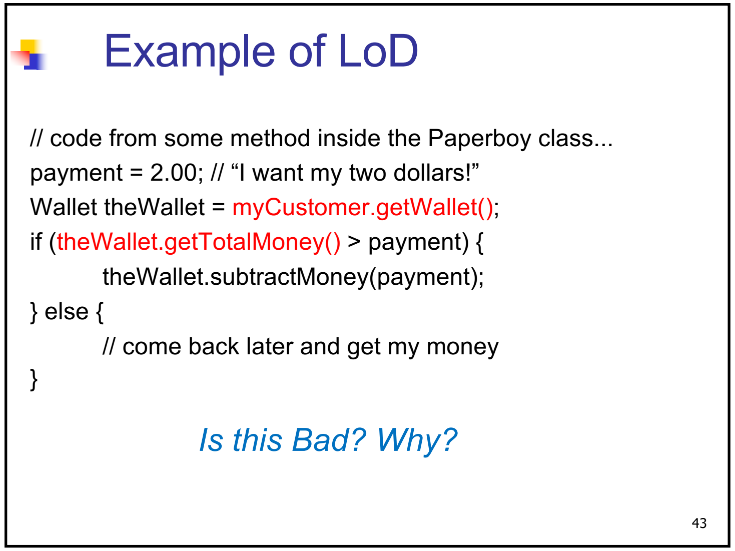


Example of LoD

```
public class Wallet {
    private float value;
    public float getTotalMoney() {
          return value;
    }
    public void setTotalMoney(float newValue) {
          value = newValue;
    }
    public void addMoney(float deposit) {
          value += deposit;
    }
    public void subtractMoney(float debit) {
          value -= debit;
    }
}
```

42

42



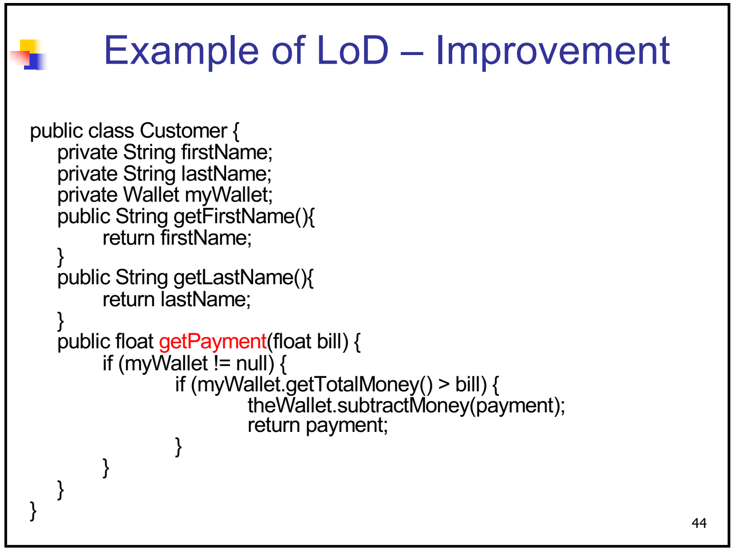
Example of LoD

```
// code from some method inside the Paperboy class...
payment = 2.00; // "I want my two dollars!"
Wallet theWallet = myCustomer.getWallet();
if (theWallet.getTotalMoney() > payment) {
          theWallet.subtractMoney(payment);
} else {
          // come back later and get my money
}
```

Is this Bad? Why?

43

43

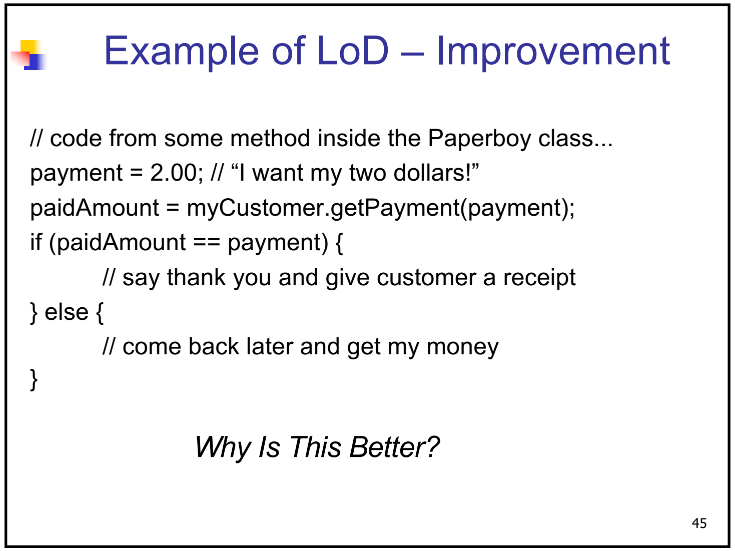


Example of LoD – Improvement

```
public class Customer {
    private String firstName;
    private String lastName;
    private Wallet myWallet;
    public String getFirstName(){
          return firstName;
    }
    public String getLastName(){
          return lastName;
    }
    public float getPayment(float bill) {
          if (myWallet != null) {
                if (myWallet.getTotalMoney() > bill) {
                      theWallet.subtractMoney(payment);
                      return payment;
                }
          }
    }
}
```

44

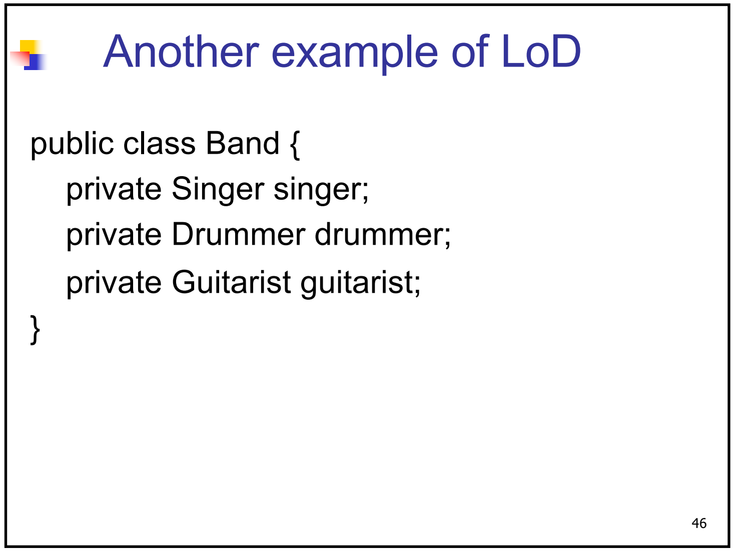
44



Example of LoD – Improvement

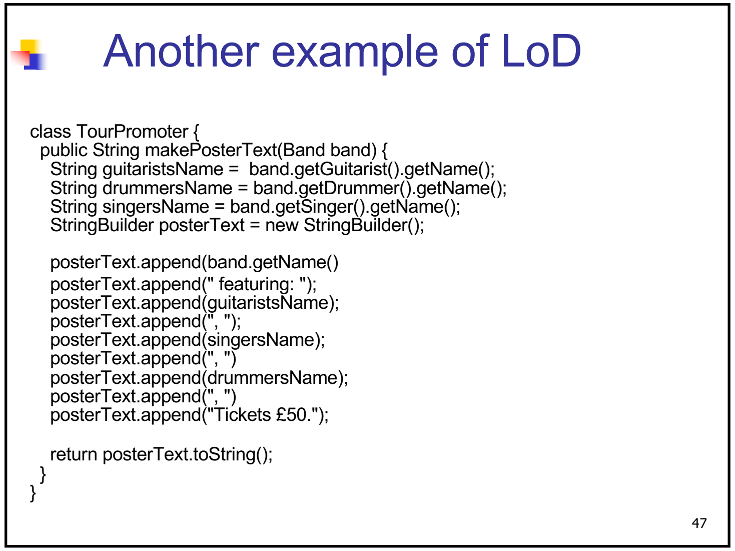
```
// code from some method inside the Paperboy class...
payment = 2.00; // “I want my two dollars!”
paidAmount = myCustomer.getPayment(payment);
if (paidAmount == payment) {
        // say thank you and give customer a receipt
} else {
        // come back later and get my money
}
```

Why Is This Better?



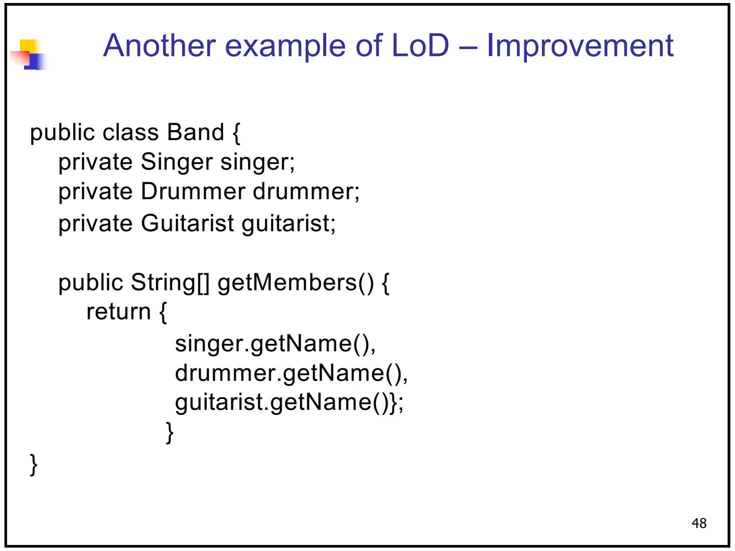
Another example of LoD

```
public class Band {
   private Singer singer;
   private Drummer drummer;
   private Guitarist guitarist;
}
```



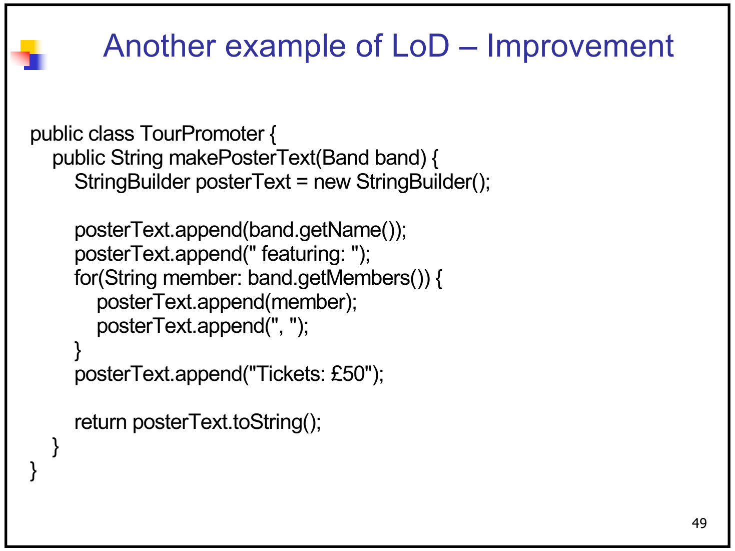
Another example of LoD

```
class TourPromoter {
  public String makePosterText(Band band) {
    String guitaristsName =  band.getGuitarist().getName();
    String drummersName = band.getDrummer().getName();
    String singersName = band.getSinger().getName();
    StringBuilder posterText = new StringBuilder();

    posterText.append(band.getName()
    posterText.append(" featuring: ");
    posterText.append(guitaristsName);
    posterText.append(", ");
    posterText.append(singersName);
    posterText.append(", ")
    posterText.append(drummersName);
    posterText.append(", ")
    posterText.append("Tickets £50.");

    return posterText.toString();
  }
}
```



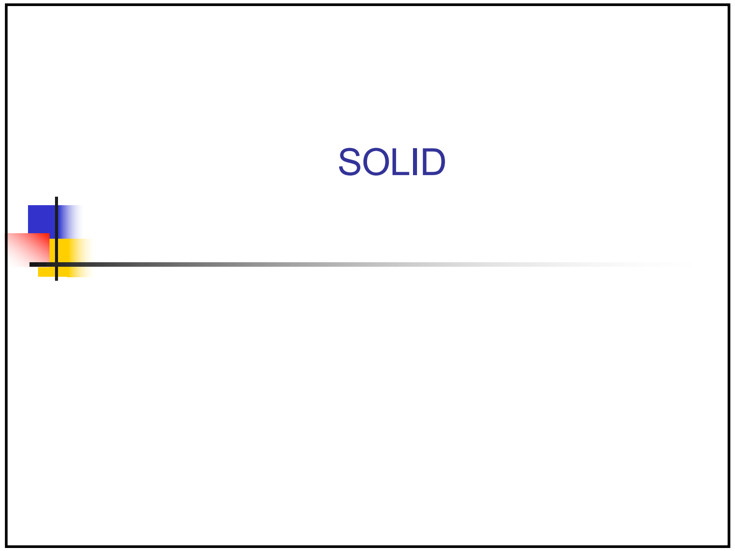
Another example of LoD – Improvement

```
public class Band {
   private Singer singer;
   private Drummer drummer;
   private Guitarist guitarist;

   public String[] getMembers() {
      return {
               singer.getName(),
               drummer.getName(),
               guitarist.getName()};
              }
}
```

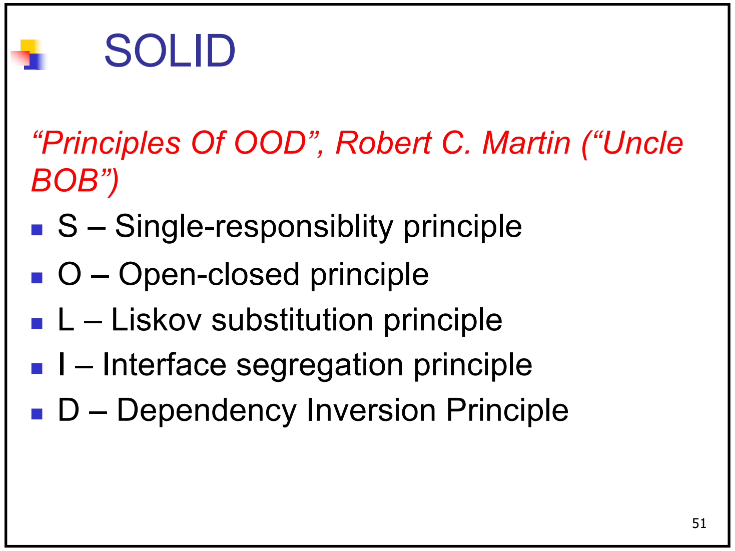


Another example of LoD – Improvement

```
public class TourPromoter {
   public String makePosterText(Band band) {
      StringBuilder posterText = new StringBuilder();

      posterText.append(band.getName());
      posterText.append(" featuring: ");
      for(String member: band.getMembers()) {
         posterText.append(member);
         posterText.append(", ");
      }
      posterText.append("Tickets: £50");

      return posterText.toString();
   }
}
```



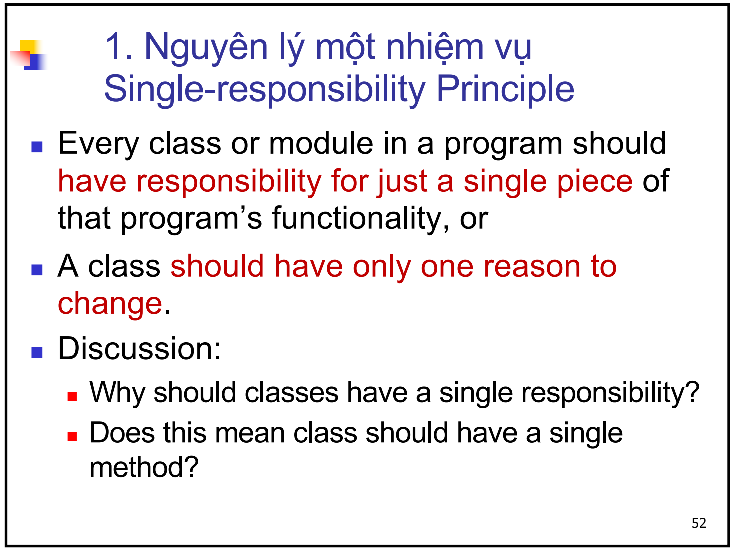
SOLID



SOLID

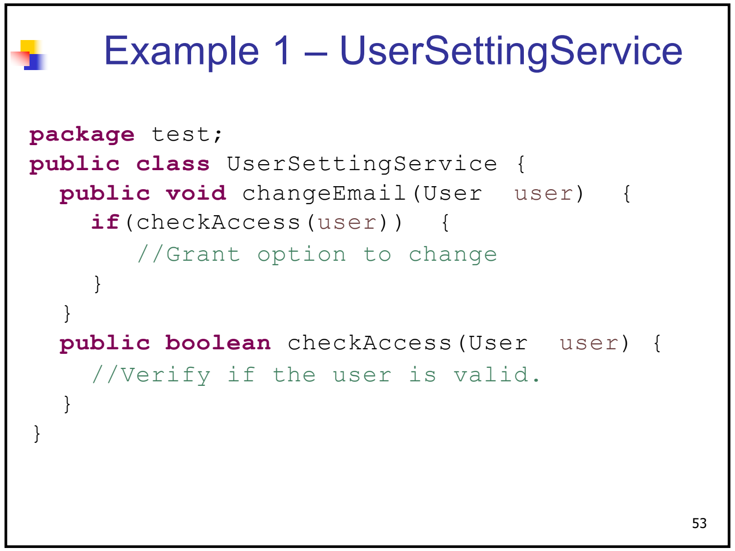
“Principles Of OOD”, Robert C. Martin (“Uncle BOB”)

- S – Single-responsiblity principle
- O – Open-closed principle
- L – Liskov substitution principle
- I – Interface segregation principle
- D – Dependency Inversion Principle



1. Nguyên lý một nhiệm vụ Single-responsibility Principle

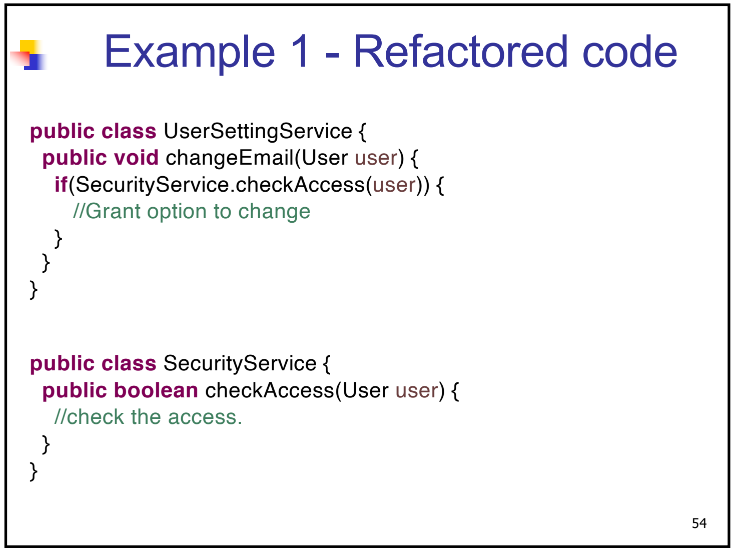
- Every class or module in a program should have responsibility for just a single piece of that program’s functionality, or
- A class should have only one reason to change.
- Discussion:
  - Why should classes have a single responsibility?
  - Does this mean class should have a single method?



Example 1 – UserSettingService

```
package test;
public class UserSettingService {
  public void changeEmail(User  user)   {
    if(checkAccess(user))   {
       //Grant option to change
    }
  }
  public boolean checkAccess(User  user) {
    //Verify if the user is valid.
  }
}
```

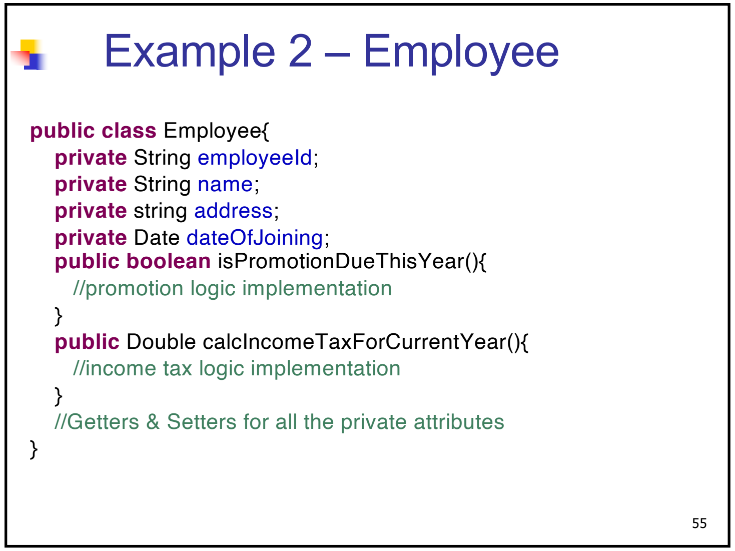
53



Example 1 - Refactored code

```
public class UserSettingService {
  public void changeEmail(User user) {
    if(SecurityService.checkAccess(user)) {
       //Grant option to change
    }
  }
}


public class SecurityService {
  public boolean checkAccess(User user) {
    //check the access.
  }
}
```

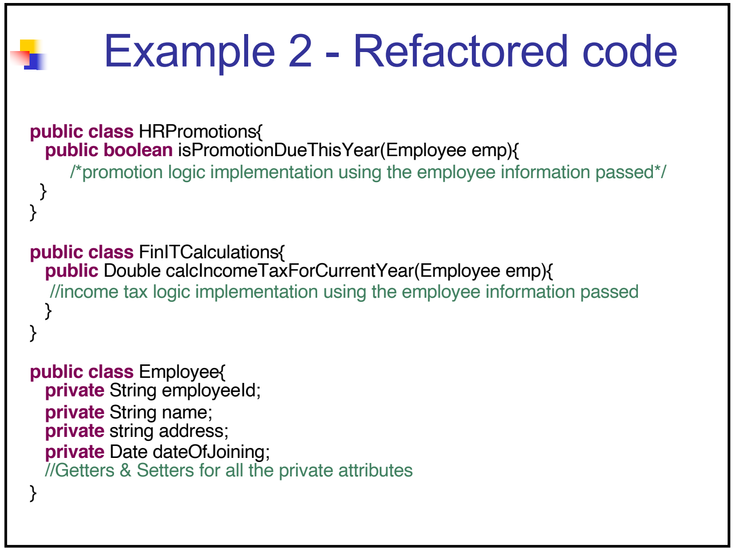
54



Example 2 – Employee

```
public class Employee{
   private String employeeId;
   private String name;
   private string address;
   private Date dateOfJoining;
   public boolean isPromotionDueThisYear(){
      //promotion logic implementation
   }
   public Double calcIncomeTaxForCurrentYear(){
      //income tax logic implementation
   }
   //Getters & Setters for all the private attributes
}
```

55



Example 2 - Refactored code

```
public class HRPromotions{
   public boolean isPromotionDueThisYear(Employee emp){
        /*promotion logic implementation using the employee information passed*/
  }
}

public class FinITCalculations{
   public Double calcIncomeTaxForCurrentYear(Employee emp){
    //income tax logic implementation using the employee information passed
   }
}

public class Employee{
   private String employeeId;
   private String name;
   private string address;
   private Date dateOfJoining;
   //Getters & Setters for all the private attributes
}
```

56

2. Nguyên lý đóng mở
Open-closed Principle

- Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification
- We should strive to write code that doesn't have to be changed every time the requirements change
- "*Open for extension*": A module (class) should provide extension point to change its behaviors
- *Closed for modification*": we should not have to change old code in order to implement new behavior (old code is untouched and therefore avoiding cascading breakage)

Example 1 - HealthInsuranceSurveyor

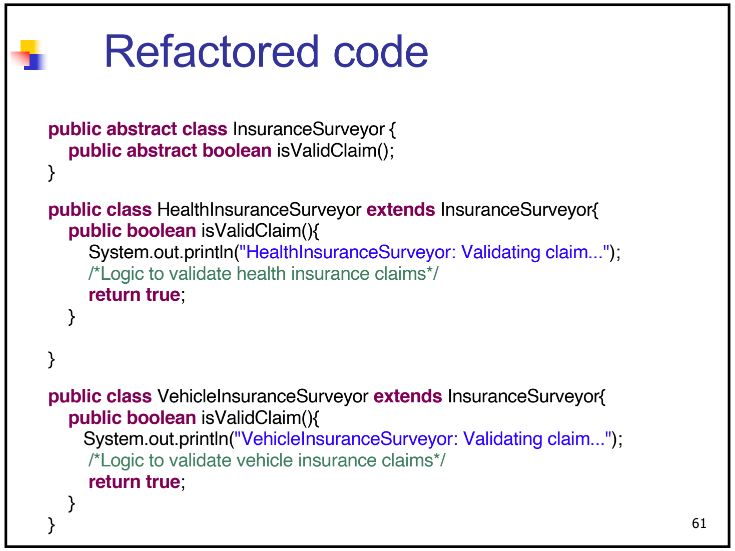
```
public class HealthInsuranceSurveyor{
   public boolean isValidClaim(){
      System.out.println("Validating ...");
      /*Logic to validate health insurance claims*/
      return true;
   }
}
```

ClaimApprovalManager

```
public class ClaimApprovalManager {
   public void processHealthClaim (HealthInsuranceSurveyor surveyor) {
      if(surveyor.isValidClaim()){
        System.out.println("Valid claim. Processing claim for approval....");
      }
   }
}
```

ClaimApprovalManager

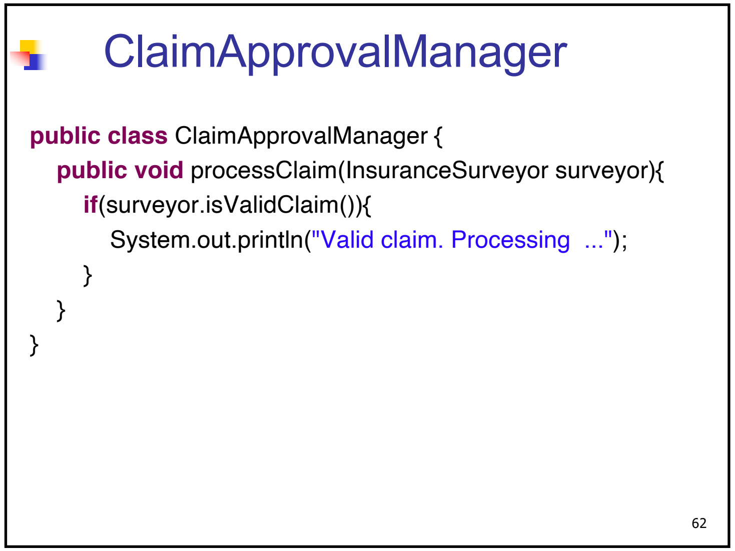
```
public class ClaimApprovalManager {
   public void processHealthClaim (HealthInsuranceSurveyor surveyor)
   {
      if(surveyor.isValidClaim()){
         System.out.println("Valid claim. Processing ...");
      }
   }
   public void processVehicleClaim (VehicleInsuranceSurveyor surveyor)
   {
      if(surveyor.isValidClaim()){
         System.out.println("Valid claim. Processing ...");
      }
   }
}
```



Refactored code

```
public abstract class InsuranceSurveyor {
   public abstract boolean isValidClaim();
}

public class HealthInsuranceSurveyor extends InsuranceSurveyor{
   public boolean isValidClaim(){
      System.out.println("HealthInsuranceSurveyor: Validating claim...");
      /*Logic to validate health insurance claims*/
      return true;
   }

}

public class VehicleInsuranceSurveyor extends InsuranceSurveyor{
   public boolean isValidClaim(){
     System.out.println("VehicleInsuranceSurveyor: Validating claim...");
      /*Logic to validate vehicle insurance claims*/
      return true;
   }
}
```

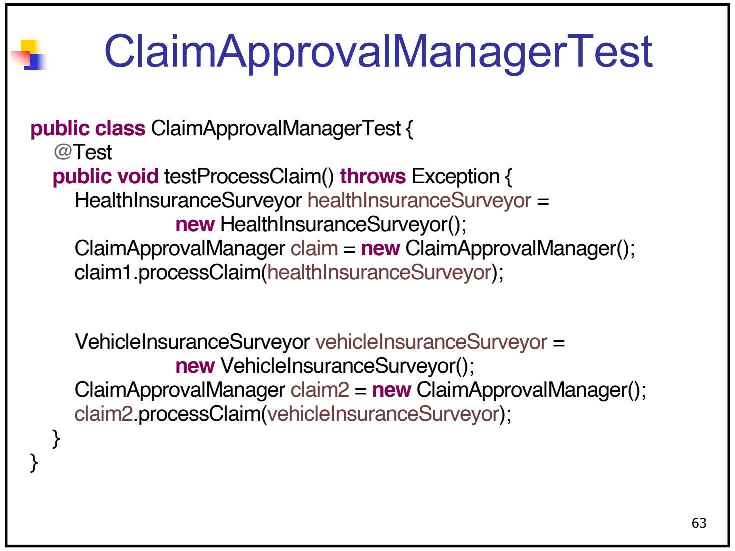
61



ClaimApprovalManager

```
public class ClaimApprovalManager {
   public void processClaim(InsuranceSurveyor surveyor){
      if(surveyor.isValidClaim()){
         System.out.println("Valid claim. Processing  ...");
      }
   }
}
```

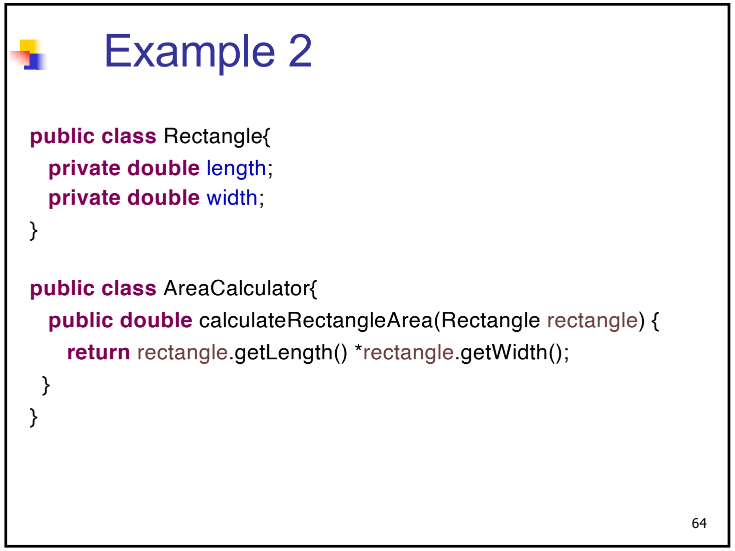
62



ClaimApprovalManagerTest

```
public class ClaimApprovalManagerTest {
   @Test
   public void testProcessClaim() throws Exception {
      HealthInsuranceSurveyor healthInsuranceSurveyor =
                    new HealthInsuranceSurveyor();
      ClaimApprovalManager claim = new ClaimApprovalManager();
      claim1.processClaim(healthInsuranceSurveyor);


      VehicleInsuranceSurveyor vehicleInsuranceSurveyor =
                    new VehicleInsuranceSurveyor();
      ClaimApprovalManager claim2 = new ClaimApprovalManager();
      claim2.processClaim(vehicleInsuranceSurveyor);
   }
}
```

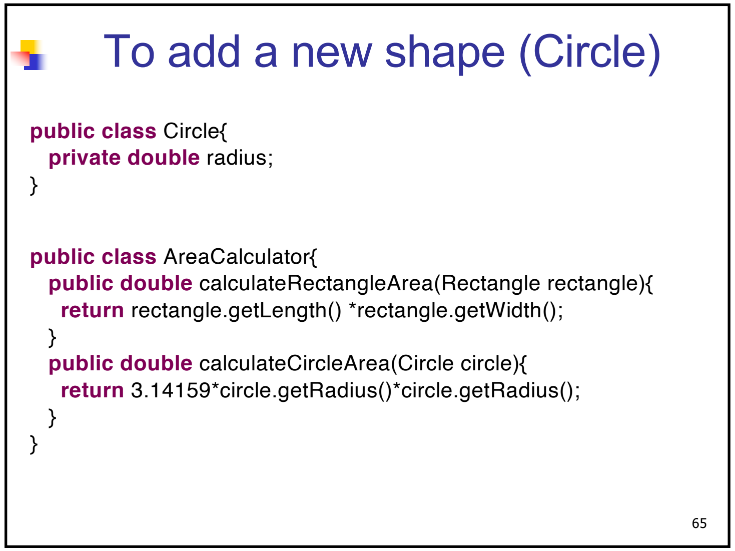
63



Example 2

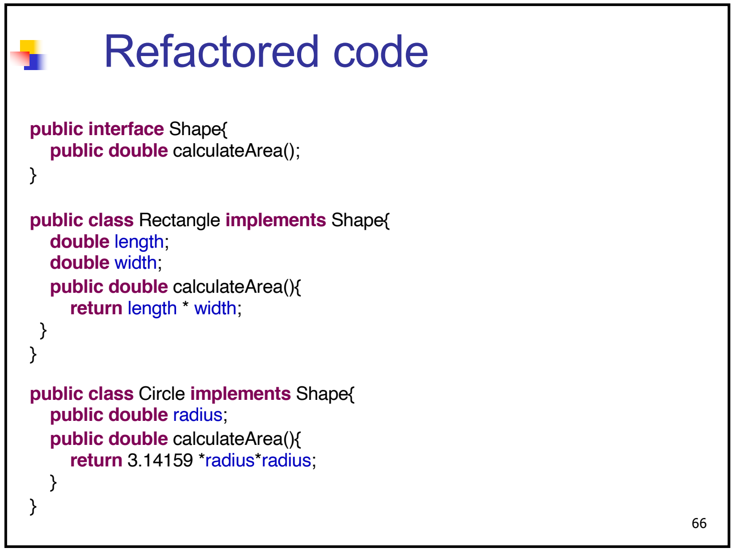
```
public class Rectangle{
  private double length;
  private double width;
}

public class AreaCalculator{
  public double calculateRectangleArea(Rectangle rectangle) {
    return rectangle.getLength() *rectangle.getWidth();
  }
}
```

64



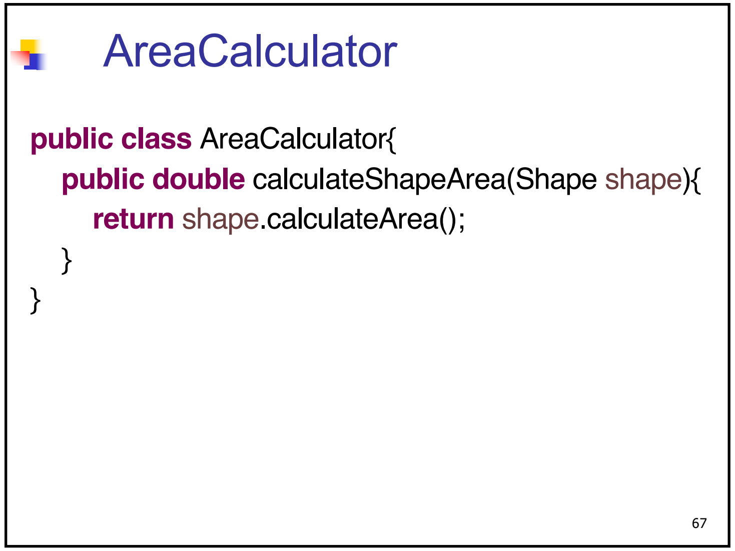
To add a new shape (Circle)

```
public class Circle{
   private double radius;
}


public class AreaCalculator{
   public double calculateRectangleArea(Rectangle rectangle){
     return rectangle.getLength() *rectangle.getWidth();
   }
   public double calculateCircleArea(Circle circle){
     return 3.14159*circle.getRadius()*circle.getRadius();
   }
}
```



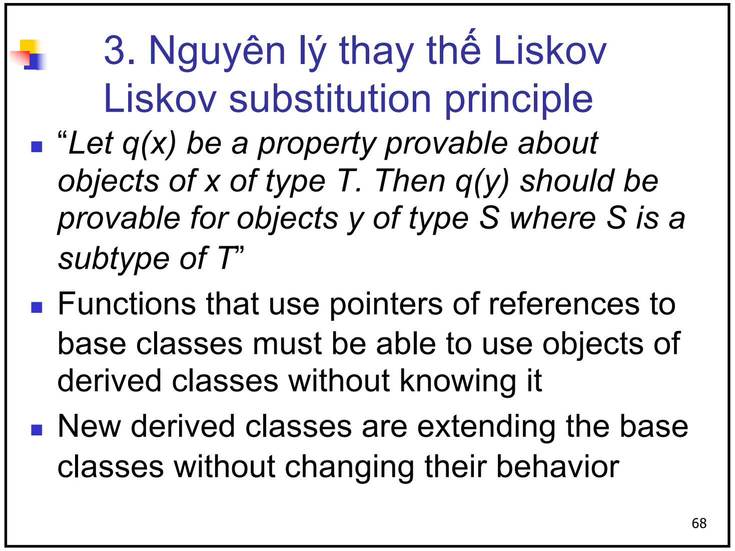
Refactored code

```
public interface Shape{
   public double calculateArea();
}

public class Rectangle implements Shape{
   double length;
   double width;
   public double calculateArea(){
      return length * width;
 }
}

public class Circle implements Shape{
   public double radius;
   public double calculateArea(){
      return 3.14159 *radius*radius;
   }
}
```



AreaCalculator

```
public class AreaCalculator{
   public double calculateShapeArea(Shape shape){
      return shape.calculateArea();
   }
}
```

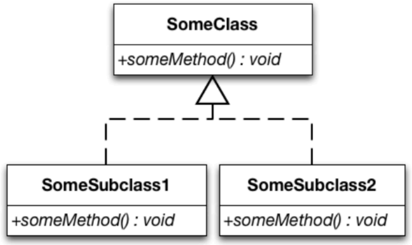


3. Nguyên lý thay thế Liskov Liskov substitution principle

- "*Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T*"
- Functions that use pointers of references to base classes must be able to use objects of derived classes without knowing it
- New derived classes are extending the base classes without changing their behavior

Liskov substitution principle

```
void clientMethod(SomeClass sc) {
   …
   sc.someMethod();
   …
}
```



Example – Rectangle

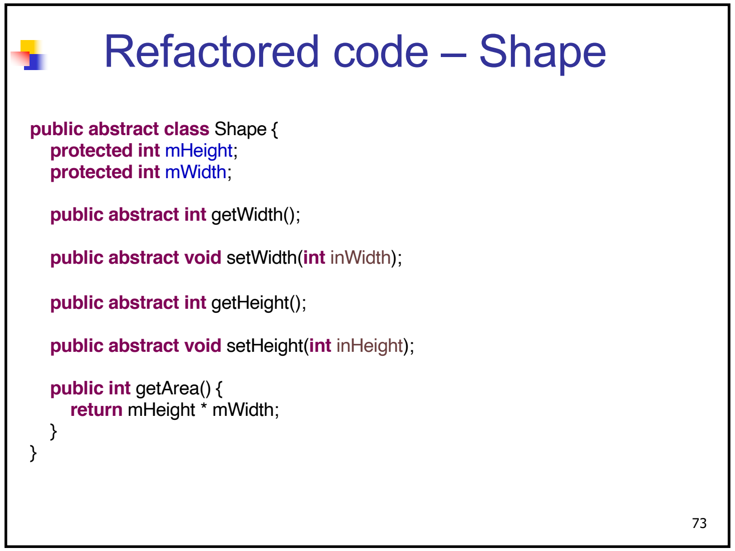
```
class Rectangle {
   protected int m_width;
   protected int m_height;

   public void setWidth(int width){
      m_width = width;
   }
   public void setHeight(int height){
      m_height = height;
   }
   public int getWidth(){
      return m_width;
   }
   public int getHeight(){
      return m_height;
   }
   public int getArea(){
      return m_width * m_height;
   }
}
```

Square

```
class Square extends Rectangle {
   @override
   public void setWidth(int width){
      m_width = width;
      m_height = width;
   }
   @override
   public void setHeight(int height){
      m_width = height;
      m_height = height;
   }
}
```

Test

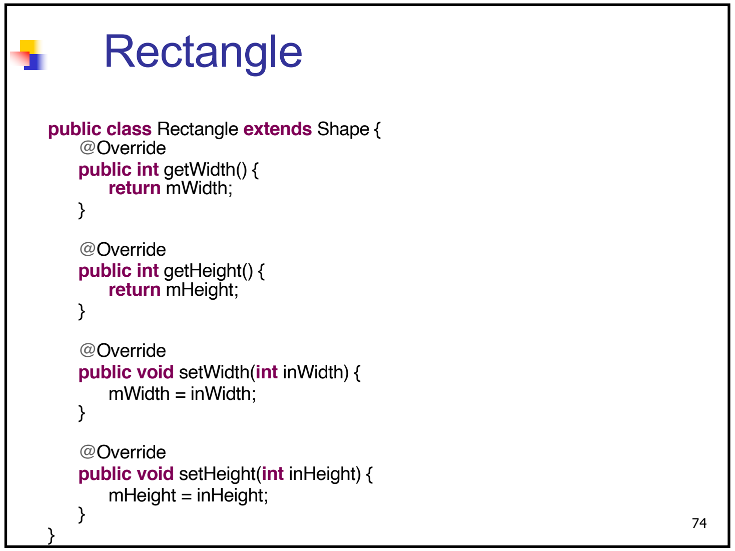
```
public class RectangleFactory {
   public static Rectangle generate(){
      return new Square(); // if we return new Rectangle(), everything is fine
   }
}

class LspTest {
   public static void main (String args[]) {
      Rectangle r = RectangleFactory.generate();

      r.setWidth(5);
      r.setHeight(10);
      // user knows that r it's a rectangle.
      // It assumes that he's able to set the width and height as for the base class

      System.out.println(r.getArea());
      // now he's surprised to see that the area is 100 instead of 50.
   }
}
```



Refactored code – Shape

```
public abstract class Shape {
   protected int mHeight;
   protected int mWidth;

   public abstract int getWidth();

   public abstract void setWidth(int inWidth);

   public abstract int getHeight();

   public abstract void setHeight(int inHeight);

   public int getArea() {
      return mHeight * mWidth;
   }
}
```

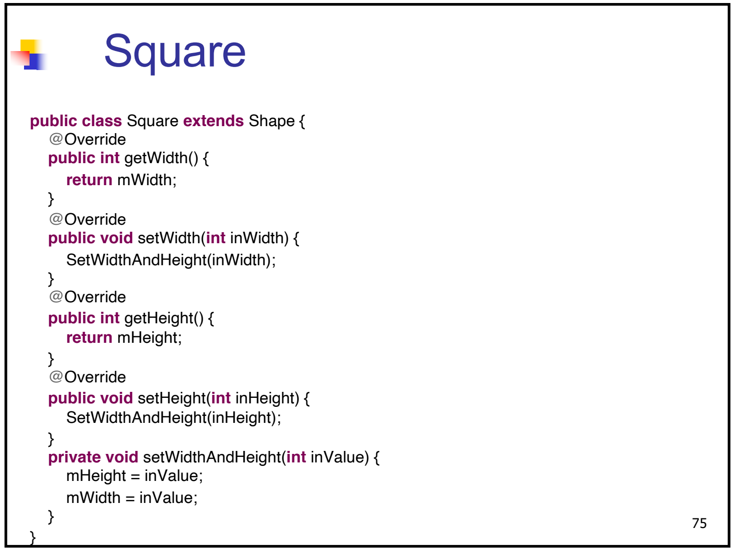
73



Rectangle

```
public class Rectangle extends Shape {
     @Override
     public int getWidth() {
          return mWidth;
     }

     @Override
     public int getHeight() {
          return mHeight;
     }

     @Override
     public void setWidth(int inWidth) {
          mWidth = inWidth;
     }

     @Override
     public void setHeight(int inHeight) {
          mHeight = inHeight;
     }
}
```

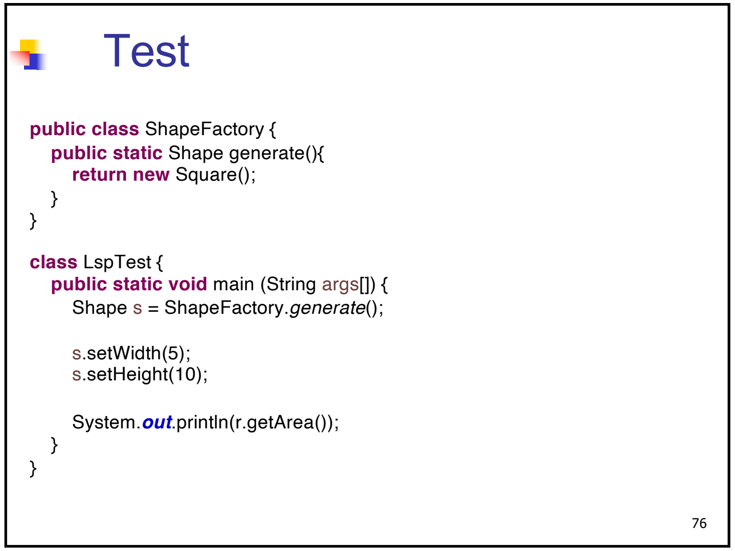
74



Square

```
public class Square extends Shape {
   @Override
   public int getWidth() {
      return mWidth;
   }
   @Override
   public void setWidth(int inWidth) {
      SetWidthAndHeight(inWidth);
   }
   @Override
   public int getHeight() {
      return mHeight;
   }
   @Override
   public void setHeight(int inHeight) {
      SetWidthAndHeight(inHeight);
   }
   private void setWidthAndHeight(int inValue) {
      mHeight = inValue;
      mWidth = inValue;
   }
}
```

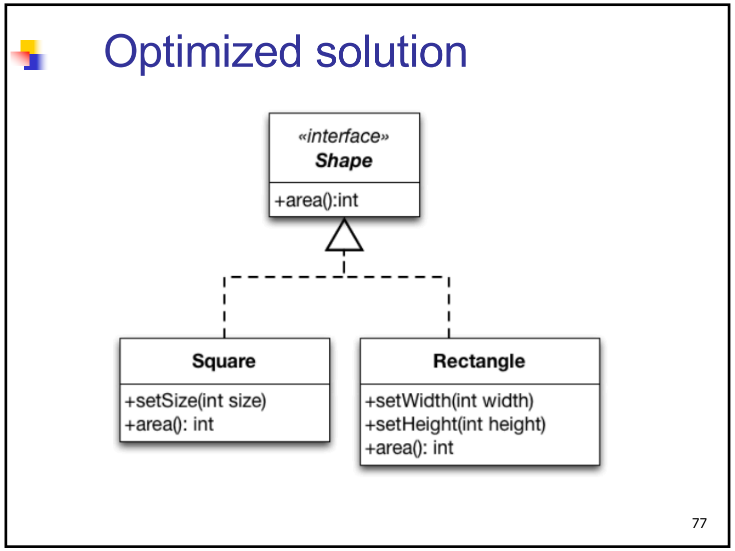
75



Test

```
public class ShapeFactory {
   public static Shape generate(){
      return new Square();
   }
}

class LspTest {
   public static void main (String args[]) {
      Shape s = ShapeFactory.generate();

      s.setWidth(5);
      s.setHeight(10);

      System.out.println(r.getArea());
   }
}
```

76

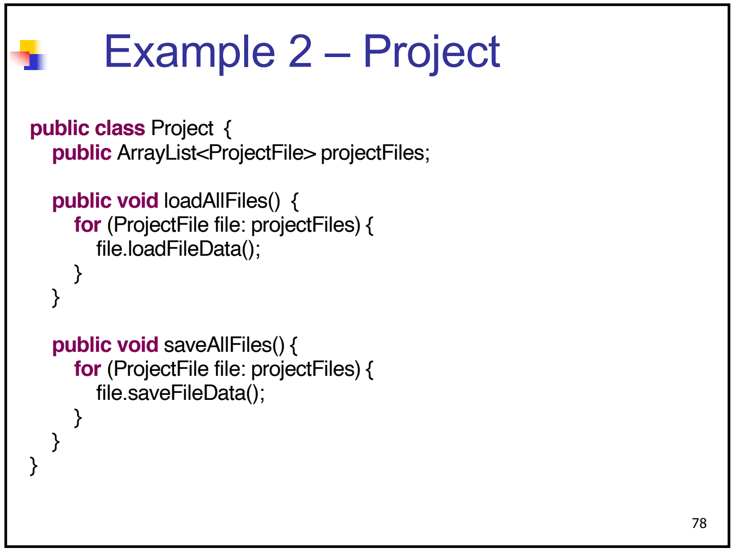


Optimized solution

«interface»
Shape
+area():int

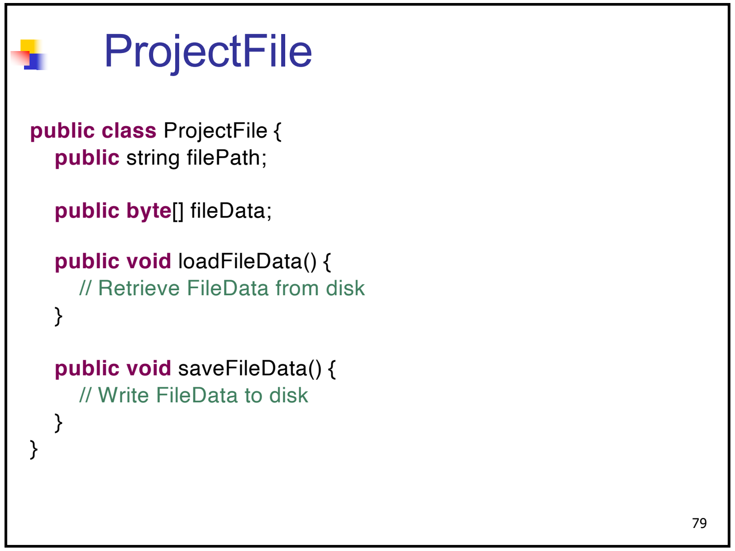
Square
+setSize(int size)
+area(): int

Rectangle
+setWidth(int width)
+setHeight(int height)
+area(): int



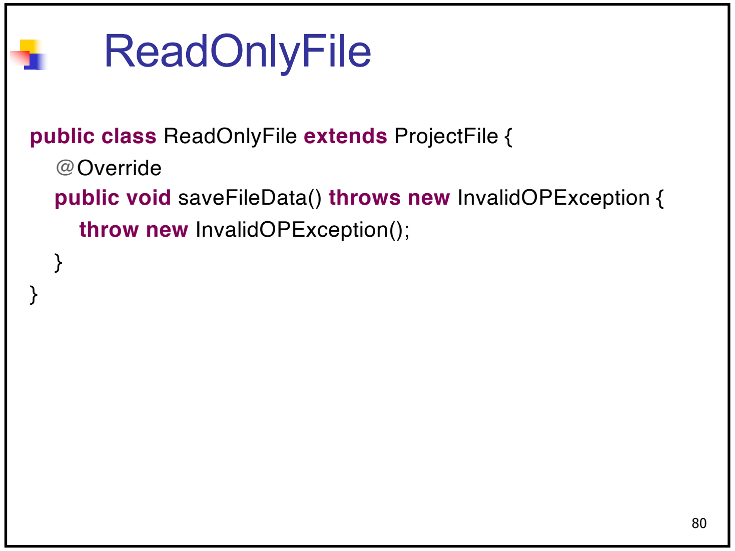
Example 2 – Project

```
public class Project  {
   public ArrayList<ProjectFile> projectFiles;

   public void loadAllFiles()  {
      for (ProjectFile file: projectFiles) {
         file.loadFileData();
      }
   }

   public void saveAllFiles() {
      for (ProjectFile file: projectFiles) {
         file.saveFileData();
      }
   }
}
```



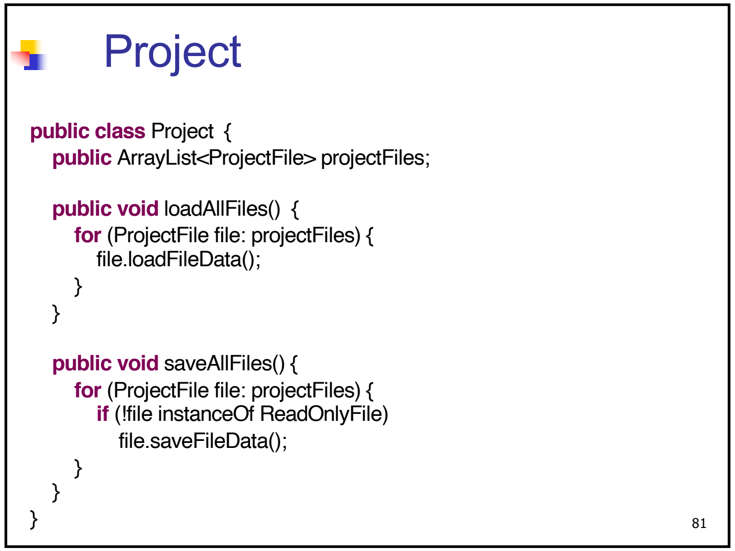
ProjectFile

```
public class ProjectFile {
   public string filePath;

   public byte[] fileData;

   public void loadFileData() {
      // Retrieve FileData from disk
   }

   public void saveFileData() {
      // Write FileData to disk
   }
}
```



ReadOnlyFile

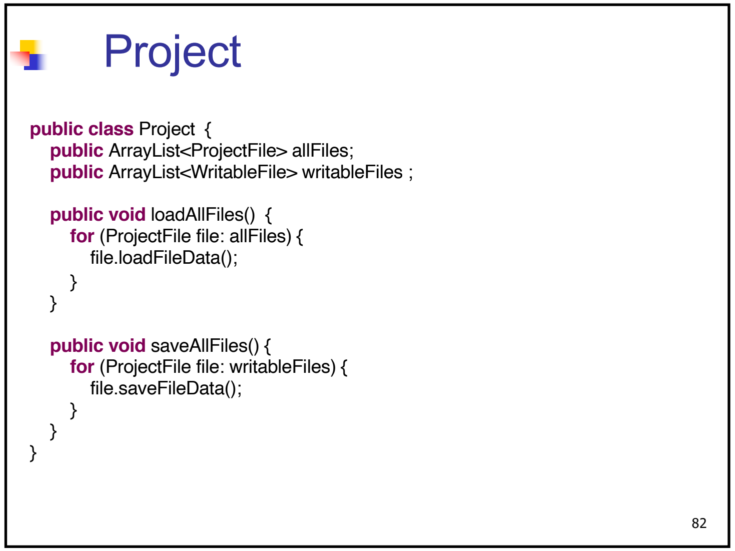
```
public class ReadOnlyFile extends ProjectFile {
   @Override
   public void saveFileData() throws new InvalidOPException {
      throw new InvalidOPException();
   }
}
```



Project

```
public class Project  {
   public ArrayList<ProjectFile> projectFiles;

   public void loadAllFiles()  {
      for (ProjectFile file: projectFiles) {
         file.loadFileData();
      }
   }

   public void saveAllFiles() {
      for (ProjectFile file: projectFiles) {
         if (!file instanceOf ReadOnlyFile)
            file.saveFileData();
      }
   }
}
```

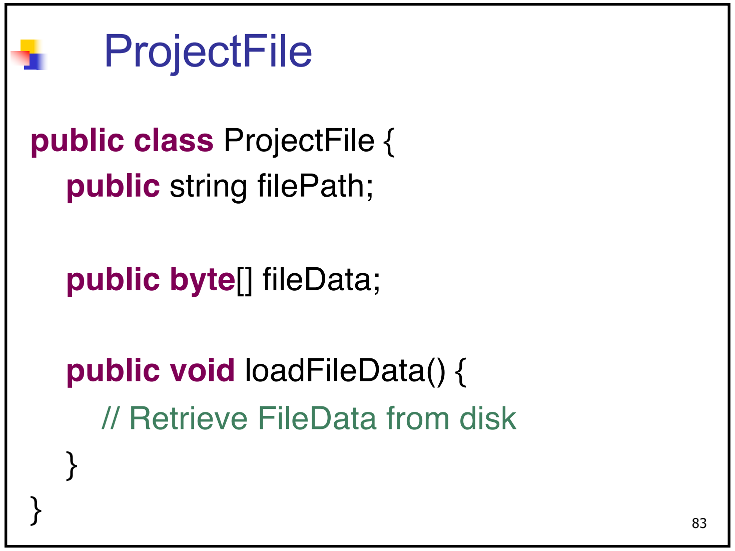
81



Project

```
public class Project  {
   public ArrayList<ProjectFile> allFiles;
   public ArrayList<WritableFile> writableFiles ;

   public void loadAllFiles()  {
      for (ProjectFile file: allFiles) {
         file.loadFileData();
      }
   }

   public void saveAllFiles() {
      for (ProjectFile file: writableFiles) {
         file.saveFileData();
      }
   }
}
```

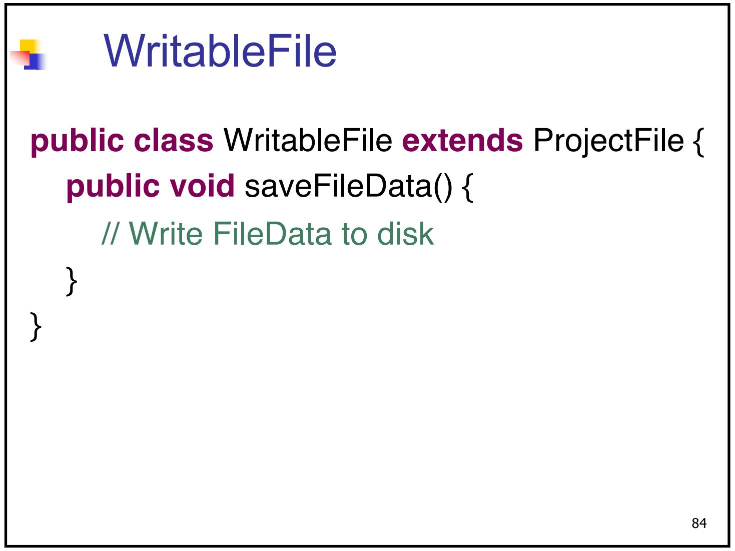
82



ProjectFile

```
public class ProjectFile {
   public string filePath;

   public byte[] fileData;

   public void loadFileData() {
      // Retrieve FileData from disk
   }
}
```

83



WritableFile

```
public class WritableFile extends ProjectFile {
   public void saveFileData() {
      // Write FileData to disk
   }
}
```

84

Discussion

- Does method overriding break the Liskov substitution principle?

Example

```
public class Report{
   private Foo foo;
   public String toString(){
      return "";
   }
}
```

3 subclasses of Report

1. HTMLReport
2. XMLReport
3. TextReport

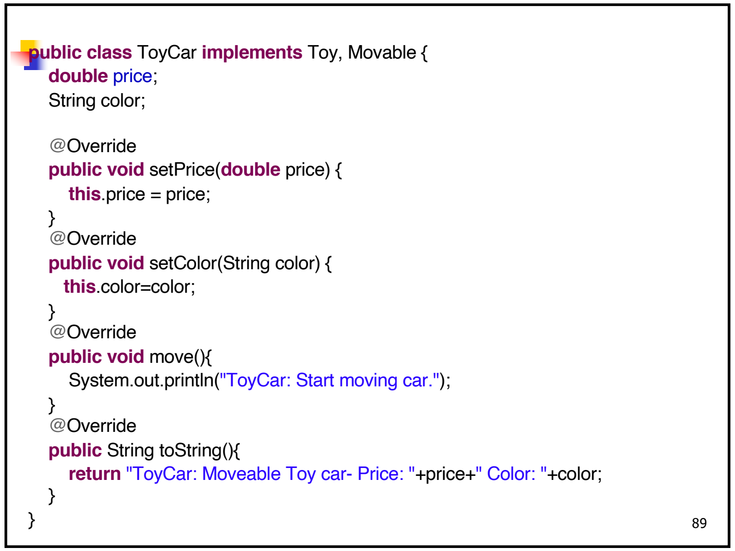
- Contract:
  - toString() returns a string, which is the representation of Foo in some format (HTML, XML, Text) (and return the empty string if the format is undefined).
  - toString() shall not mutate the Report object
  - toString() shall never throw an Exception

4. Nguyên lý phân tách giao diện Interface Segregation Principle

- An interface should NOT have methods that its implementing classes do not need (Client should NOT be forced to depend on methods it does not use)
- “Fat interface” should be split into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them
- Similar to the Single Responsibility Principle, the goal of the Interface Segregation Principle is to reduce the side effects and frequency of required changes by splitting the software into multiple, independent parts

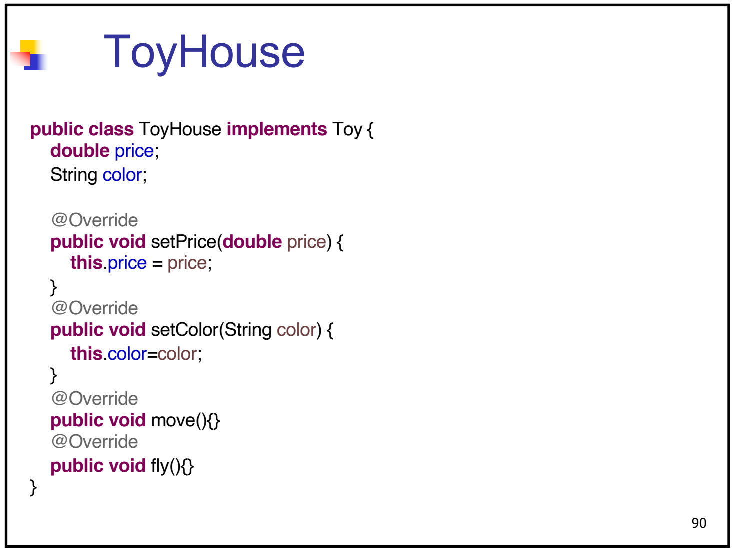
Example 1 – Toy

```
public interface Toy {
   void setPrice(double price);
   void setColor(String color);
   void move();
   void fly();
}
```



```
public class ToyCar implements Toy, Movable {
    double price;
    String color;

    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){
        System.out.println("ToyCar: Start moving car.");
    }
    @Override
    public String toString(){
        return "ToyCar: Moveable Toy car- Price: "+price+" Color: "+color;
    }
}
```

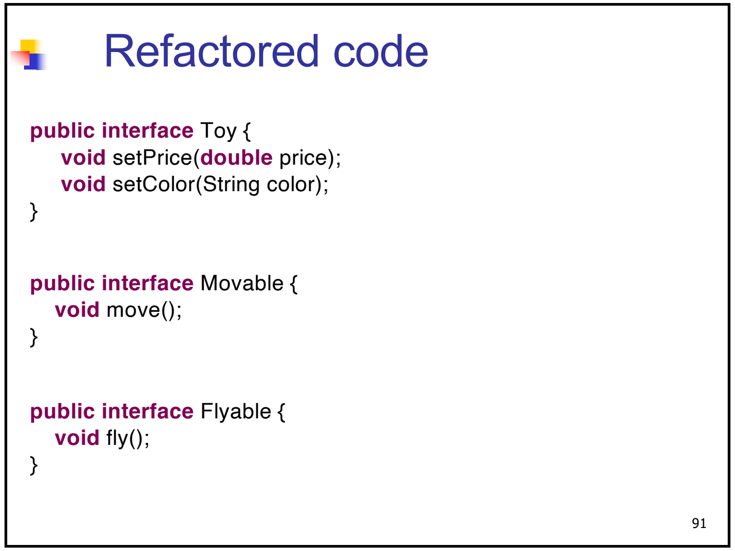
89



ToyHouse

```
public class ToyHouse implements Toy {
    double price;
    String color;

    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public void move(){}
    @Override
    public void fly(){}
}
```

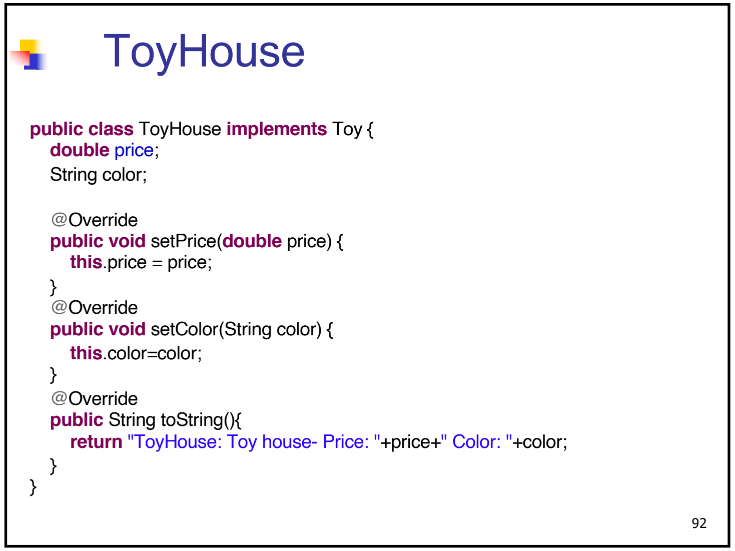
90



Refactored code

```
public interface Toy {
    void setPrice(double price);
    void setColor(String color);
}


public interface Movable {
    void move();
}


public interface Flyable {
    void fly();
}
```

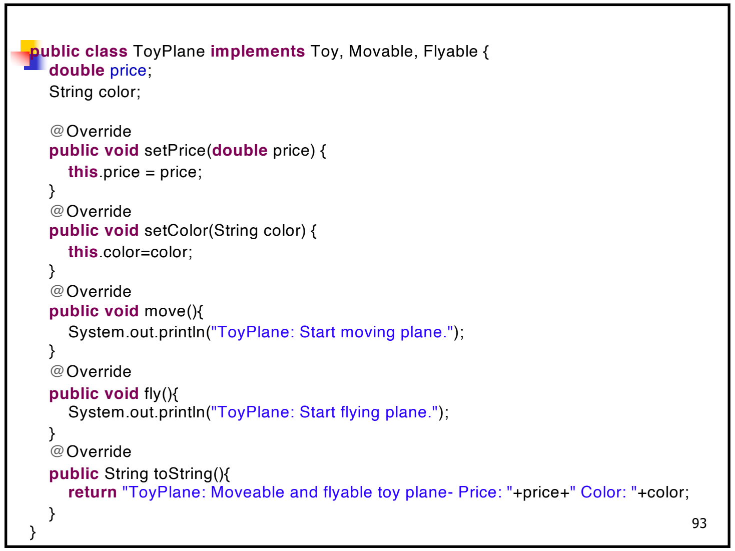
91



ToyHouse

```
public class ToyHouse implements Toy {
    double price;
    String color;

    @Override
    public void setPrice(double price) {
        this.price = price;
    }
    @Override
    public void setColor(String color) {
        this.color=color;
    }
    @Override
    public String toString(){
        return "ToyHouse: Toy house- Price: "+price+" Color: "+color;
    }
}
```

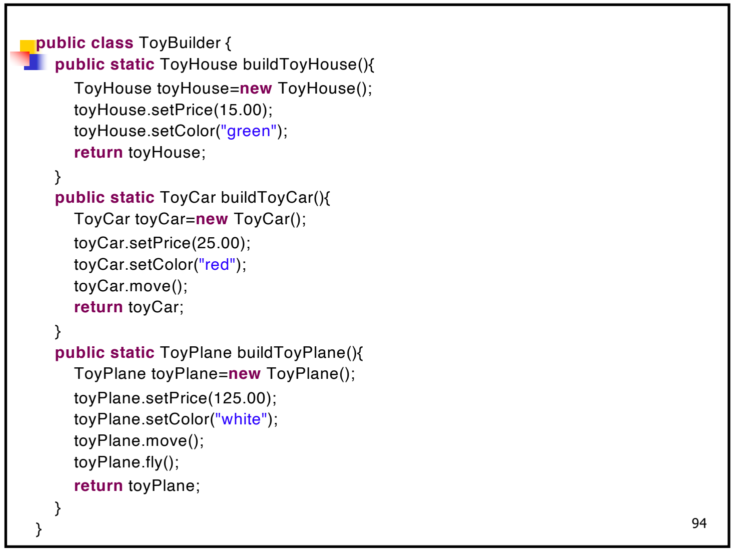
92



```
public class ToyPlane implements Toy, Movable, Flyable {
  double price;
  String color;

  @Override
  public void setPrice(double price) {
    this.price = price;
  }
  @Override
  public void setColor(String color) {
    this.color=color;
  }
  @Override
  public void move(){
    System.out.println("ToyPlane: Start moving plane.");
  }
  @Override
  public void fly(){
    System.out.println("ToyPlane: Start flying plane.");
  }
  @Override
  public String toString(){
    return "ToyPlane: Moveable and flyable toy plane- Price: "+price+" Color: "+color;
  }
}
```

93

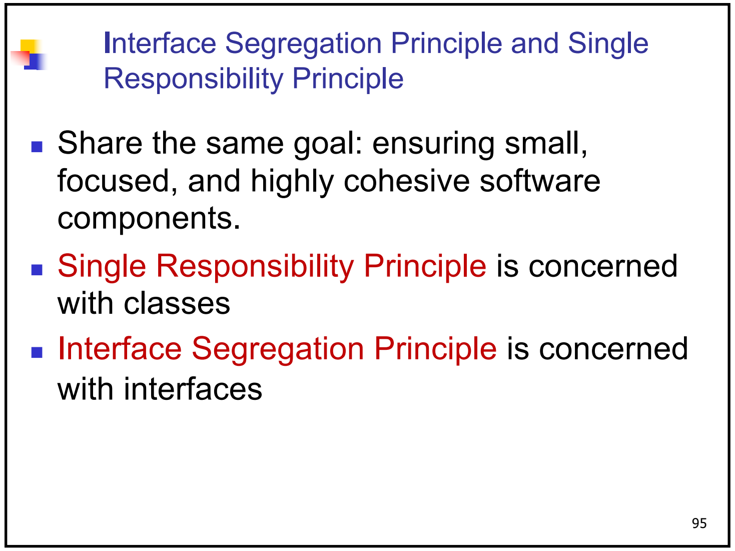
93



```
public class ToyBuilder {
  public static ToyHouse buildToyHouse(){
    ToyHouse toyHouse=new ToyHouse();
    toyHouse.setPrice(15.00);
    toyHouse.setColor("green");
    return toyHouse;
  }
  public static ToyCar buildToyCar(){
    ToyCar toyCar=new ToyCar();
    toyCar.setPrice(25.00);
    toyCar.setColor("red");
    toyCar.move();
    return toyCar;
  }
  public static ToyPlane buildToyPlane(){
    ToyPlane toyPlane=new ToyPlane();
    toyPlane.setPrice(125.00);
    toyPlane.setColor("white");
    toyPlane.move();
    toyPlane.fly();
    return toyPlane;
  }
}
```

94

94

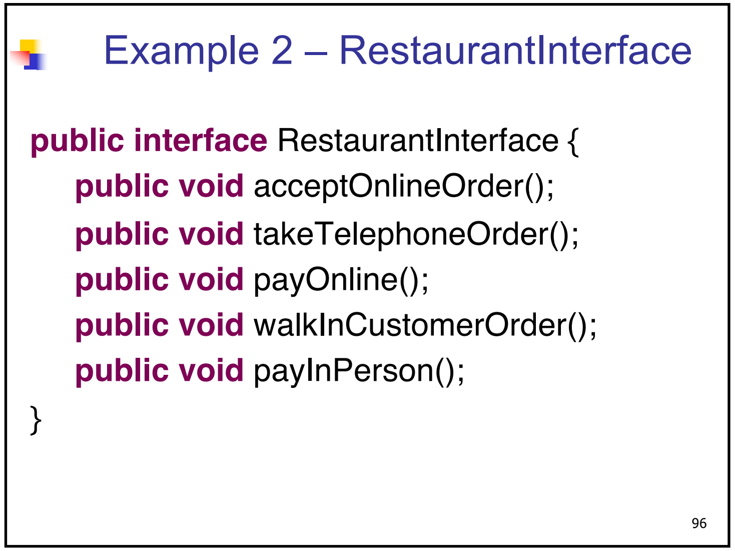


Interface Segregation Principle and Single Responsibility Principle

- Share the same goal: ensuring small, focused, and highly cohesive software components.
- Single Responsibility Principle is concerned with classes
- Interface Segregation Principle is concerned with interfaces

95

95

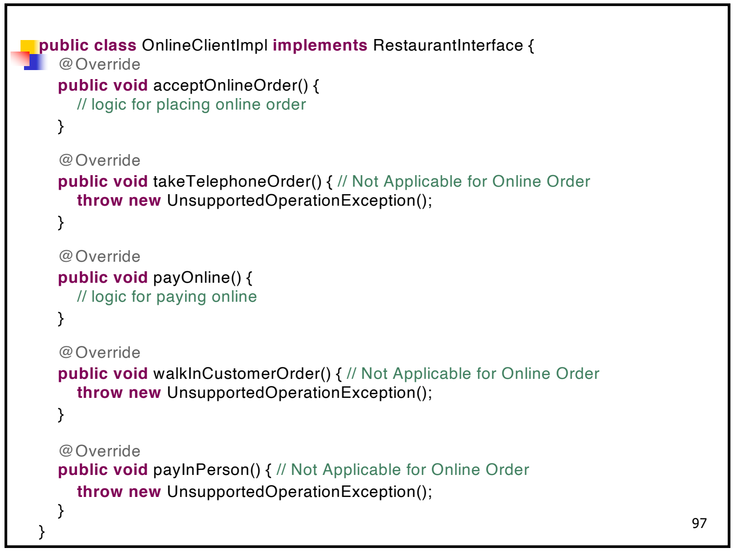


Example 2 – RestaurantInterface

```
public interface RestaurantInterface {
    public void acceptOnlineOrder();
    public void takeTelephoneOrder();
    public void payOnline();
    public void walkInCustomerOrder();
    public void payInPerson();
}
```

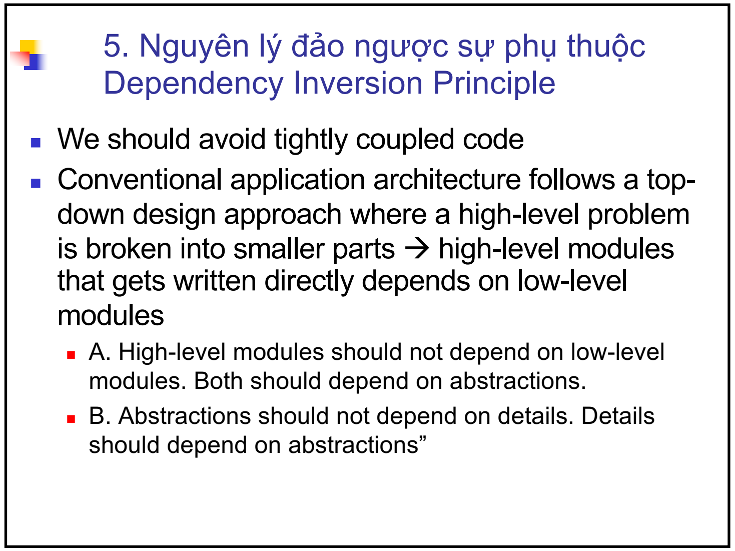
96

96



```
public class OnlineClientImpl implements RestaurantInterface {
    @Override
    public void acceptOnlineOrder() {
        // logic for placing online order
    }

    @Override
    public void takeTelephoneOrder() { // Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }

    @Override
    public void payOnline() {
        // logic for paying online
    }

    @Override
    public void walkInCustomerOrder() { // Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }

    @Override
    public void payInPerson() { // Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }
}
```

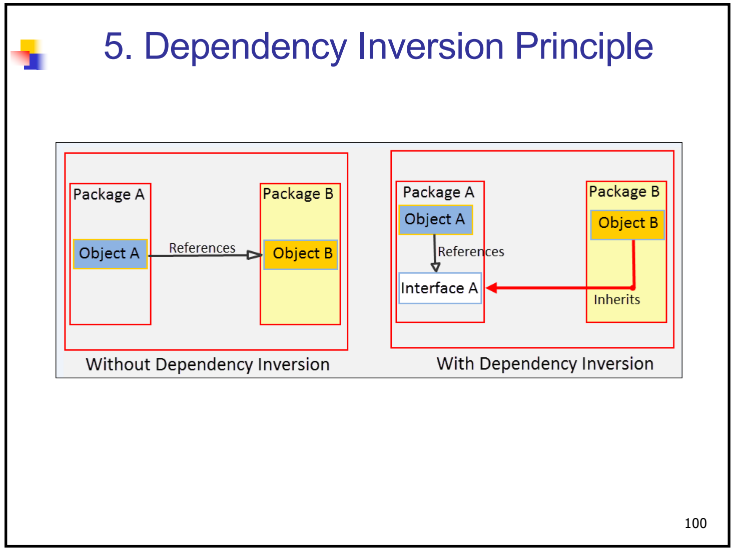
97



5. Nguyên lý đảo ngược sự phụ thuộc Dependency Inversion Principle

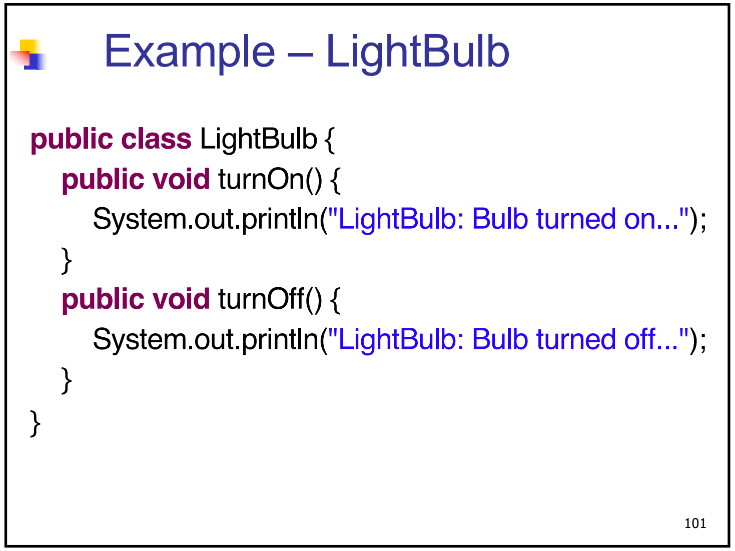
- We should avoid tightly coupled code
- Conventional application architecture follows a top-down design approach where a high-level problem is broken into smaller parts → high-level modules that gets written directly depends on low-level modules
  - A. High-level modules should not depend on low-level modules. Both should depend on abstractions.
  - B. Abstractions should not depend on details. Details should depend on abstractions”

99



5. Dependency Inversion Principle

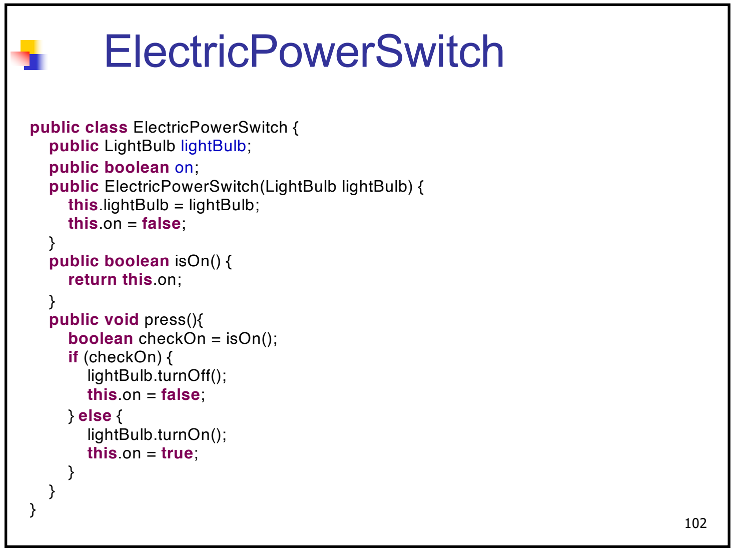
100



Example – LightBulb

```
public class LightBulb {
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```

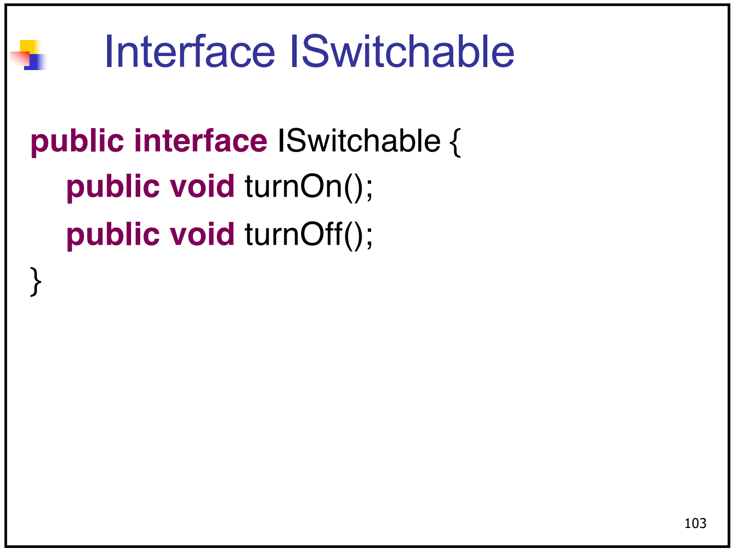
101



ElectricPowerSwitch

```
public class ElectricPowerSwitch {
    public LightBulb lightBulb;
    public boolean on;
    public ElectricPowerSwitch(LightBulb lightBulb) {
        this.lightBulb = lightBulb;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
    public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            lightBulb.turnOff();
            this.on = false;
        } else {
            lightBulb.turnOn();
            this.on = true;
        }
    }
}
```

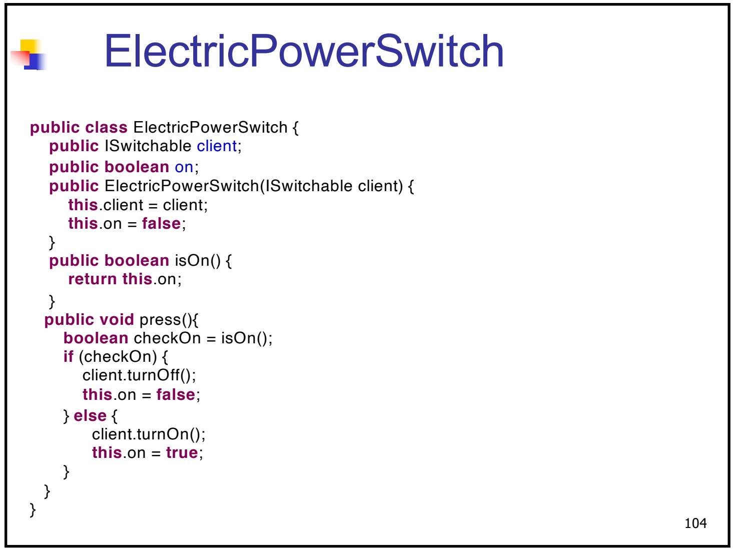
102



Interface ISwitchable

```
public interface ISwitchable {
    public void turnOn();
    public void turnOff();
}
```

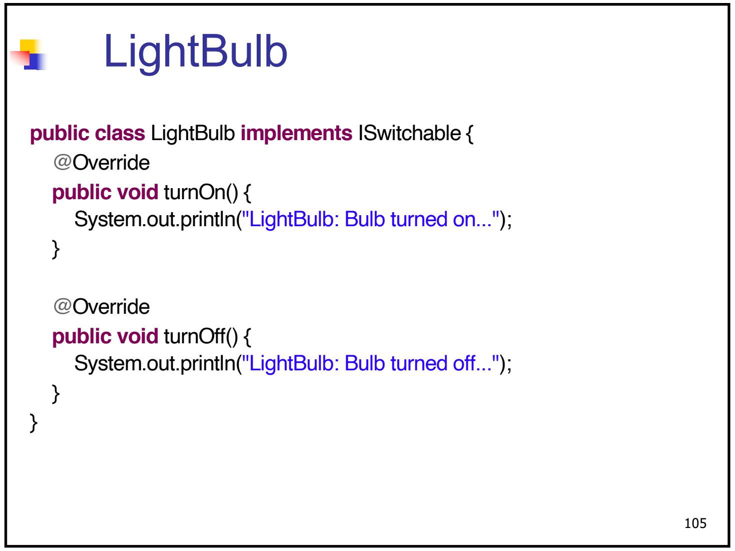
103



ElectricPowerSwitch

```
public class ElectricPowerSwitch {
    public ISwitchable client;
    public boolean on;
    public ElectricPowerSwitch(ISwitchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
    public void press(){
        boolean checkOn = isOn();
        if (checkOn) {
            client.turnOff();
            this.on = false;
        } else {
            client.turnOn();
            this.on = true;
        }
    }
}
```

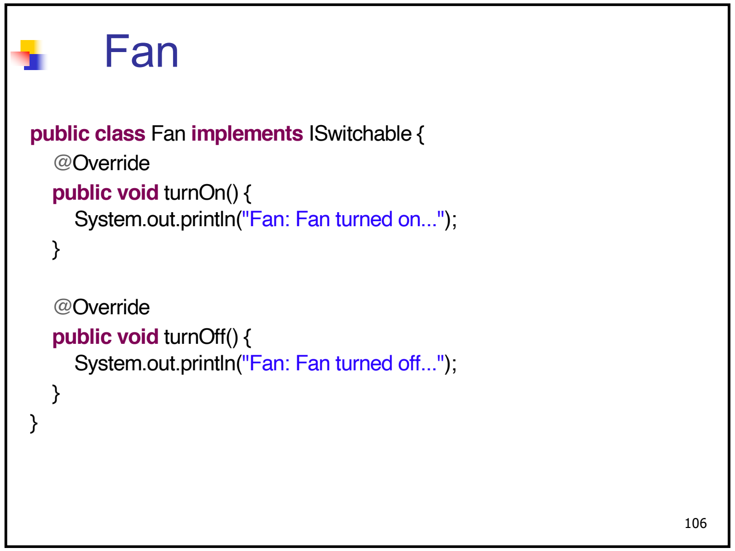
104



LightBulb

```
public class LightBulb implements ISwitchable {
    @Override
    public void turnOn() {
        System.out.println("LightBulb: Bulb turned on...");
    }

    @Override
    public void turnOff() {
        System.out.println("LightBulb: Bulb turned off...");
    }
}
```

105

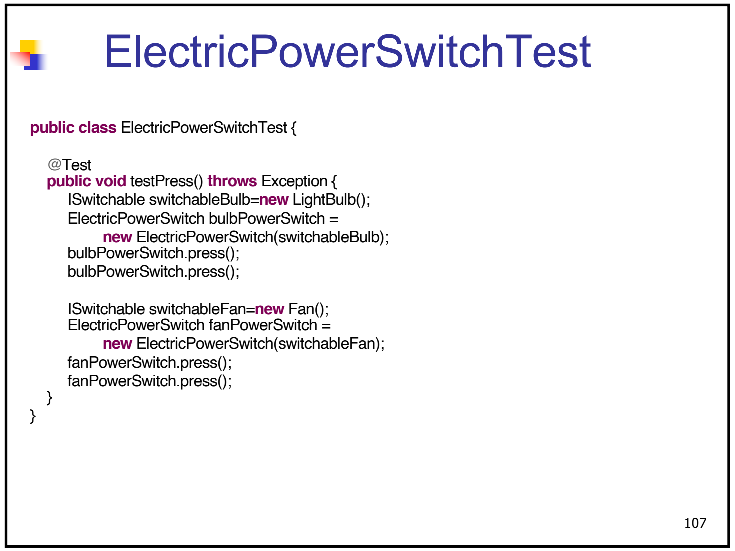


Fan

```
public class Fan implements ISwitchable {
   @Override
   public void turnOn() {
      System.out.println("Fan: Fan turned on...");
   }

   @Override
   public void turnOff() {
      System.out.println("Fan: Fan turned off...");
   }
}
```

106

106

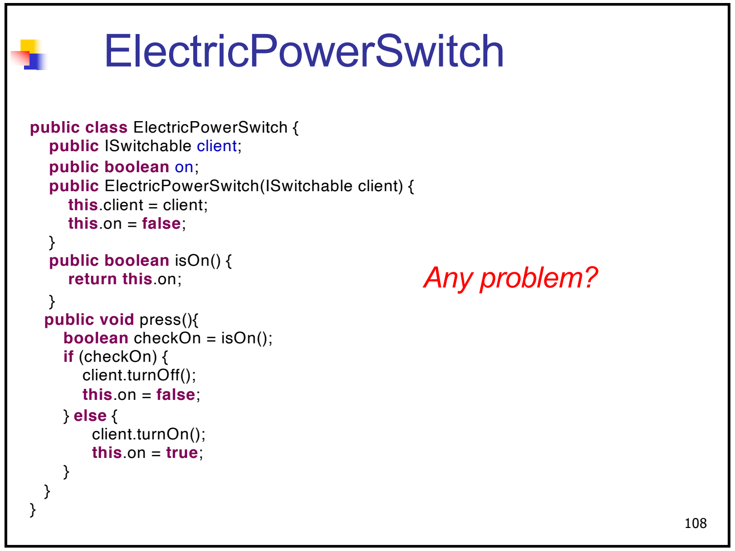


ElectricPowerSwitchTest

```
public class ElectricPowerSwitchTest {

   @Test
   public void testPress() throws Exception {
      ISwitchable switchableBulb=new LightBulb();
      ElectricPowerSwitch bulbPowerSwitch =
            new ElectricPowerSwitch(switchableBulb);
      bulbPowerSwitch.press();
      bulbPowerSwitch.press();

      ISwitchable switchableFan=new Fan();
      ElectricPowerSwitch fanPowerSwitch =
            new ElectricPowerSwitch(switchableFan);
      fanPowerSwitch.press();
      fanPowerSwitch.press();
   }
}
```

107

107



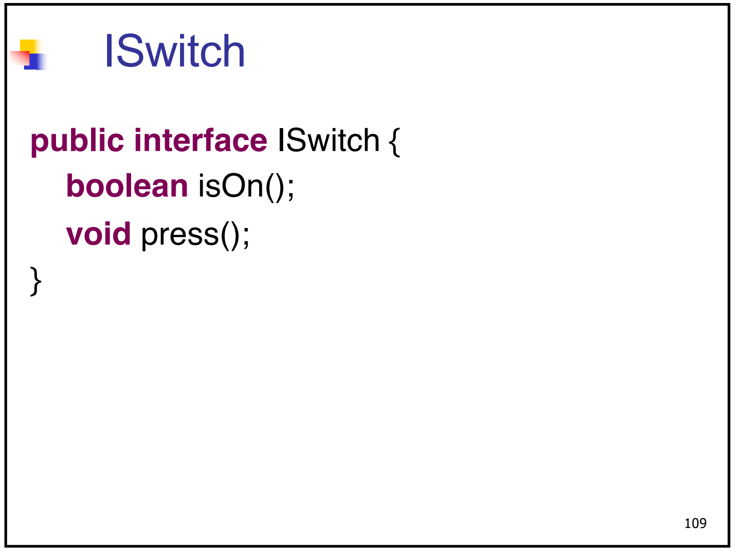
ElectricPowerSwitch

```
public class ElectricPowerSwitch {
   public ISwitchable client;
   public boolean on;
   public ElectricPowerSwitch(ISwitchable client) {
      this.client = client;
      this.on = false;
   }
   public boolean isOn() {
      return this.on;
   }
  public void press(){
     boolean checkOn = isOn();
     if (checkOn) {
        client.turnOff();
        this.on = false;
     } else {
          client.turnOn();
          this.on = true;
     }
  }
}
```

Any problem?

108

108

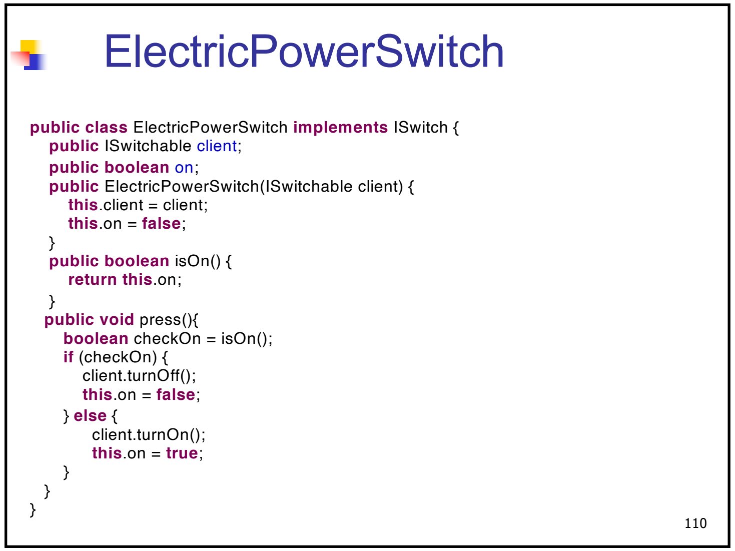


ISwitch

```
public interface ISwitch {
   boolean isOn();
   void press();
}
```

109

109



ElectricPowerSwitch

```
public class ElectricPowerSwitch implements ISwitch {
    public ISwitchable client;
    public boolean on;
    public ElectricPowerSwitch(ISwitchable client) {
        this.client = client;
        this.on = false;
    }
    public boolean isOn() {
        return this.on;
    }
   public void press(){
       boolean checkOn = isOn();
       if (checkOn) {
           client.turnOff();
           this.on = false;
       } else {
             client.turnOn();
             this.on = true;
       }
   }
}
```

110

110