
Design Pattern

SE 3109 (2 Credit)



About Course

- ❑ The course contains 3 part
 - I. Design Principles
 - II. Design Patterns
 - III. Code Refactoring

A bit knowledge of Design Patterns

- A general reusable solution for the common problems occurs in software design
- Represents an idea
- Typically show relationships and interactions between classes or objects
- Idea is to speed up the development process by providing well tested, proven development/design paradigm
- Programming language independent strategies for solving a common problem

A bit knowledge of Design Patterns

- Design patterns define a common language that you and your teammates can use to communicate more efficiently. You can say, “Oh, just use a Singleton for that,” and everyone will understand the idea behind your suggestion. No need to explain what a singleton is if you know the pattern and its name.

Design Principles

- A set of guidelines that helps developers to avoid having a bad design.
- 3 important characteristics of a bad design that should be avoided
 - ✓ Rigidity - It is hard to change because every change affects too many other parts of the system.
 - ✓ Fragility - When you make a change, unexpected parts of the system break.
 - ✓ Immobility - It is hard to reuse in another application because it cannot be disentangled from the current application.

Software design principles: SOLID

- Five basic principles
- Help to create good software architecture
 - ✓ S stands for SRP (Single responsibility principle)
 - ✓ O stands for OCP (Open closed principle)
 - ✓ L stands for LSP (Liskov substitution principle)
 - ✓ I stand for ISP (Interface segregation principle)
 - ✓ D stands for DIP (Dependency inversion principle)

Single responsibility principle

- Each responsibility should be a separate class, because each responsibility is an axis of change.
- A class or module should have one, and only one, reason to be changed.

Single responsibility principle

```
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.
    }
}

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

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


Single responsibility principle: Benefit

- A class with one responsibility will have far fewer test cases
- Less functionality in a single class will have fewer dependencies
- Smaller, well-organized classes are easier to search than monolithic ones

Open close principle

- Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.
- In other words, (in an ideal world...) you should never need to change existing code or classes: All new functionality can be added by adding new subclasses or methods, or by reusing existing code through delegation.
- This **prevents** you from introducing **new bugs** in existing code. If you never change it, you can't break it.

Open close principle

- A class should be written in such a manner that it performs its job flawlessly without the assumption that people in the future will simply come and change it
- The class should remain closed for modification, but it should have the option to get extended

Open close principle

- Ways of extending the class include:
 - Inheriting from the class
 - Overwriting the required behaviors from the class
 - Extending certain behaviors of the class
- Ex: installing extensions in chrome browser

OCP Example

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

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

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

```
public class AreaCalculator
{
    public double calculateRectangleArea(Rectangle rectangle)
    {
        return rectangle.length * rectangle.width;
    }
    public double calculateCircleArea(Circle circle)
    {
        return (22/7)*circle.radius*circle.radius;
    }
}
```

New shape?
Pentagon?

OCP Example

```
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 (22/7)*radius*radius;
    }
}
```

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

Liskov substitution principle

- Derived types must be completely substitutable for their base types.
- **if class *A* is a subtype of class *B*, then we should be able to replace *B* with *A* without disrupting the behavior of our program.**
- This principle is very closely related to Open Closed Principle (OCP), violation of LSP in turn violates the OCP

LSP Example

```
class TransportationDevice
{
    String name;
    String getName() { ... }
    void setName(String n) { ... }

    double speed;
    double getSpeed() { ... }
    void setSpeed(double d) { ... }

    Engine engine;
    Engine getEngine() { ... }
    void setEngine(Engine e) { ... }

    void startEngine() { ... }
}
```

```
class Car extends TransportationDevice
{
    @Override
    void startEngine() { ... }
}
```

```
class Bicycle extends TransportationDevice
{
    @Override
    void startEngine() /*problem!*/
}
```


LSP Example

```
class TransportationDevice
{
    String name;
    String getName() { ... }
    void setName(String n) { ... }

    double speed;
    double getSpeed() { ... }
    void setSpeed(double d) { ... }
}
```

```
class Car extends DevicesWithEngines
{
    @Override
    void startEngine() { ... }
}
```

```
class DevicesWithoutEngines extends TransportationDevice
{
    void startMoving() { ... }
}
```

```
class DevicesWithEngines extends TransportationDevice
{
    Engine engine;
    Engine getEngine() { ... }
    void setEngine(Engine e) { ... }

    void startEngine() { ... }
}
```

```
class Bicycle extends DevicesWithoutEngines
{
    @Override
    void startMoving() { ... }
}
```

Interface segregation principle

- Client should not be forced to depend on methods that they do not use.
- The ISP says that once an interface has become too 'fat' it needs to be split into smaller and more specific interfaces so that any clients of the interface will only know about the methods that pertain to them

ISP Example

- There is a Restaurant interface which contains methods for accepting orders from
 - online customers,
 - dial-in or telephone customers and
 - walk-in customers.
- It also contains methods for
 - handling online payments and
 - in-person payments (for walk-in customers as well as telephone customers when their order is delivered at home).

ISP Example

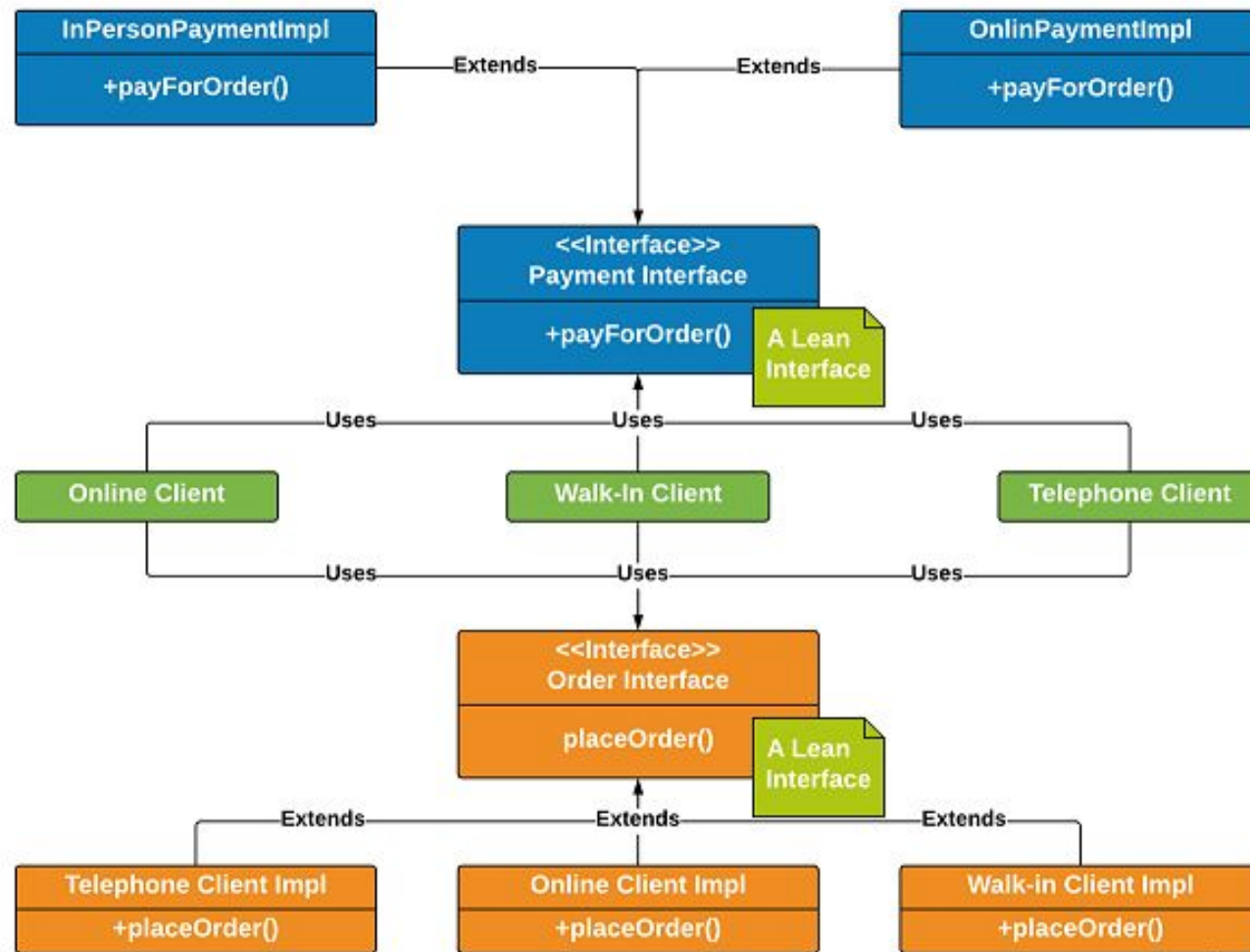
```
public interface RestaurantInterface
{
    public void acceptOnlineOrder();
    public void takeTelephoneOrder();
    public void payOnline();
    public void walkInCustomerOrder();
    public void payInPerson();
}

public class OnlineClientImpl implements RestaurantInterface
{
    public void acceptOnlineOrder()
    {
        //logic for placing online order
    }
    public void takeTelephoneOrder()
    {
        //Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }
    public void payOnline()
    {
        //logic for paying online
    }
    public void walkInCustomerOrder()
    {
        //Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }
    public void payInPerson() {
        //Not Applicable for Online Order
        throw new UnsupportedOperationException();
    }
}
```

ISP Example

- To overcome the above mentioned problems , we apply Interface Segregation Principle to refactor the above design.
 - Separate out payment and order placement functionalities into two separate lean interfaces, `PaymentInterface.java` and `OrderInterface.java`.
 - Each of the clients use one implementation each of `PaymentInterface` and `OrderInterface`. For example – `OnlineClient.java` uses `OnlinePaymentImpl` and `OnlineOrderImpl` and so on.
 - Single Responsibility Principle is now attached as Payment interface(`PaymentInterface.java`) and Ordering interface(`OrderInterface`).
 - Change in any one of the order or payment interfaces does not affect the other. They are independent now. There will be no need to do any dummy implementation or throw an `UnsupportedOperationException` as each interface has only methods it will always use.

ISP Example



Dependency inversion principle

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

DIP Example

- Card example: The type of credit card or debit card that you have for paying does not even matter; they will simply swipe it.
- allows a programmer to remove hardcoded dependencies so that the application becomes loosely coupled and extendable


```

public class BackEndDeveloper {
    public void writeJava() {
    }
}
public class FrontEndDeveloper {
    public void writeJavascript() {
    }
}
public class Project {
    private BackEndDeveloper backEndDeveloper;
    private FrontEndDeveloper frontEndDeveloper;
    public void implement () {
        backEndDeveloper.writeJava();
        frontEndDeveloper.writeJavascript();
    }
}

```

```

public interface Developer {
    void develop();
}
public class BackEndDeveloper implements Developer {
    @Override
    public void develop() {
        writeJava();
    }
    private void writeJava() {
    }
}
public class FrontEndDeveloper implements Developer {
    @Override
    public void develop() {
        writeJavascript();
    }
    public void writeJavascript() {
    }
}
public class Project {
    private List<Developer> developers;
    public Project(List<Developer> developers) {
        this.developers = developers;
    }
    public void implement() {
        developers.forEach(d->d.develop());
    }
}

```



Refactoring Code Smells

If it Stinks, change it!



What is Refactoring?

What is Refactoring?

- ☑ A series of **small** steps, each of which changes the program's **internal structure** without changing its **external behavior** - Martin Fowler

What is Refactoring?

- ☑ A series of **small** steps, each of which changes the program's **internal structure** without changing its **external behavior** - Martin Fowler

- ☑
 - ☑ Verify no change in external behavior by
 - ☑ Testing
 - ☑ Using the right tool - IDE
 - ☑ Formal code analysis by tool
 - Being very, very careful



What if you hear...



What if you hear...

- ☒ We'll just refactor the code to support logging

What if you hear...

- 
- ☒ We'll just refactor the code to support logging

What if you hear...



- ☒ We'll just refactor the code to support logging
- ☒ Can you refactor the code so that it authenticates against LDAP instead of Database?




What if you hear...

- ☒ We'll just refactor the code to support logging
- ☒ Can you refactor the code so that it authenticates against LDAP instead of Database?




What if you hear...

- ☒ We'll just refactor the code to support logging
- ☒ Can you refactor the code so that it authenticates against LDAP instead of Database?
- ☒ We have too much duplicate code, we need to refactor the code to eliminate duplication





What if you hear...

- ☒ We'll just refactor the code to support logging 
- ☒ Can you refactor the code so that it authenticates against LDAP instead of Database? 
- ☒ We have too much duplicate code, we need to refactor the code to eliminate duplication 





What if you hear...

- ☒ We'll just refactor the code to support logging 
- ☒ Can you refactor the code so that it authenticates against LDAP instead of Database? 
- ☒ We have too much duplicate code, we need to refactor the code to eliminate duplication 
- ☒ This class is too big, we need to refactor it






What if you hear...

- ☒ We'll just refactor the code to support logging 
- ☒ Can you refactor the code so that it authenticates against LDAP instead of Database? 
- ☒ We have too much duplicate code, we need to refactor the code to eliminate duplication 
- ☒ This class is too big, we need to refactor it 

What if you hear...

- ☒ We'll just refactor the code to support logging 
- ☒ Can you refactor the code so that it authenticates against LDAP instead of Database? 
- ☒ We have too much duplicate code, we need to refactor the code to eliminate duplication 
- ☒ This class is too big, we need to refactor it 
- ☒ Caching?

What if you hear...

- ☒ We'll just refactor the code to support logging 
- ☒ Can you refactor the code so that it authenticates against LDAP instead of Database? 
- ☒ We have too much duplicate code, we need to refactor the code to eliminate duplication 
- ☒ This class is too big, we need to refactor it 
- ☒ Caching? 



Why do we Refactor?



Why do we Refactor?

- ☑ Helps us deliver **more business value faster**

Why do we Refactor?

- ☑ Helps us deliver **more business value faster**
- ☑ Improves the **design** of our software
 - ☑ Easier to maintain and understand
 - ☑ Easier to facilitate change
 - ☑ More flexibility
 - ☑ Increased re-usability



Why do we Refactor?...



Why do we Refactor?...

- ☑ Minimizes *technical debt*



Why do we Refactor?...

- ☑ Minimizes *technical debt*
- ☑ Keep **development** at *speed*

Why do we Refactor?...

- ☑ Minimizes *technical debt*
- ☑ Keep **development** at *speed*
- ☑ To make the software easier to **understand**
 - ☑ Write for people, not the compiler
 - ☑ Understand unfamiliar code

Why do we Refactor?...

- ☑ Minimizes *technical debt*
- ☑ Keep **development** at *speed*
- ☑ To make the software easier to **understand**
 - ☑ Write for people, not the compiler
 - ☑ Understand unfamiliar code
- ☑ To help find **bugs**
 - ☑ refactor while debugging to clarify the code

Why do we Refactor?...

- ☑ Minimizes *technical debt*
- ☑ Keep **development** at *speed*
- ☑ To make the software easier to **understand**
 - ☑ Write for people, not the compiler
 - ☑ Understand unfamiliar code
- ☑ To help find **bugs**
 - ☑ refactor while debugging to clarify the code
- ☑

Readability

Which code segment is easier to read?

Sample 1

```
if (date.Before(Summer_Start) || date.After(Summer_End)){  
    charge = quantity * winterRate + winterServiceCharge;  
else  
    charge = quantity * summerRate;  
}
```

Sample 2

```
if (IsSummer(date)) {  
    charge = SummerCharge(quantity);  
else  
    charge = WinterCharge(quantity);  
}
```



When should you refactor?



When should you refactor?

- ☑ To add **new functionality**
 - ☑ refactor existing code until you understand it
 - ☑ refactor the design to make it simple to add

When should you refactor?

- ☑ To add **new functionality**
 - ☑ refactor existing code until you understand it
 - ☑ refactor the design to make it simple to add
- ☑ To find **bugs**
 - ☑ refactor to understand the code

When should you refactor?

- ☑ To add **new functionality**
 - ☑ refactor existing code until you understand it
 - ☑ refactor the design to make it simple to add
- ☑ To find **bugs**
 - ☑ refactor to understand the code
- ☑ For **code reviews**
 - ☑ immediate effect of code review
 - ☑ allows for higher level suggestions

When should you refactor?

- ☑ To add **new functionality**

- ☑ refactor existing code until you understand it
- ☑ refactor the design to make it simple to add

- ☑ To find **bugs**

- ☑ refactor to understand the code

- ☑ For **code reviews**

- ☑ immediate effect of code review
- ☑ allows for higher level suggestions

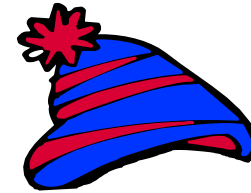
The Two Hats

Adding Function



- ☒ Add new capabilities to the system
- ☒ Adds new tests
- ☒ Get the test working

Refactoring



- ☒ Does not add any new features
- ☒ Does not add tests (but may change some)
- ☒ Restructure the code to remove redundancy

How do we Refactor?

- ☑ We look for Code-Smells
- ☑ Things that we suspect are not quite right or will cause us severe pain if we do not fix



2 Piece of Advice before Refactoring

2 Piece of Advice before Refactoring

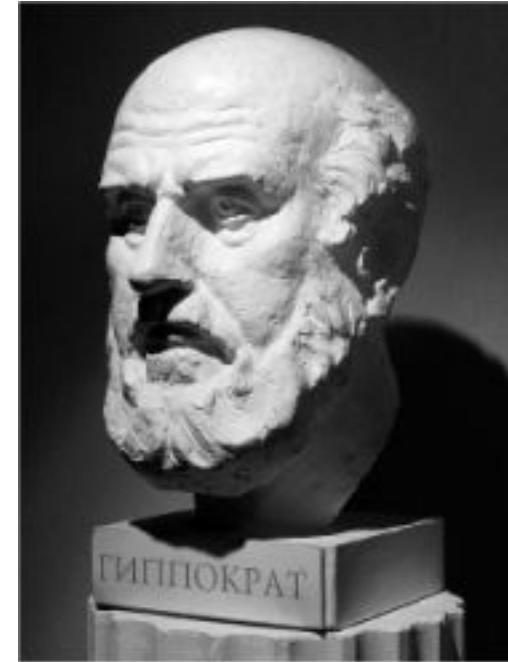


Baby Steps

2 Piece of Advice before Refactoring



Baby Steps



The Hippocratic Oath

First Do No Harm!

Code Smells?

Code Smells identify *frequently* occurring **design problems** in a way that is more *specific or targeted* than general design guidelines (like “loosely coupled code” or “duplication-free code”). - Joshua K

A code smell is a design that duplicates, complicates, **bloats or tightly couples code**

A short history of Code Smells

- ☑ If it stinks, change it!
- ☑ Kent Beck coined the term code smell to signify something in code that needed to be changed.



Code Smells

- Bloaters
 - code, methods and classes that have increased to such gargantuan proportions that they are hard to work with
- Object-Orientation Abusers
 - incomplete or incorrect application of object-oriented programming principles.
- Change Preventers
 - if you need to change something in one place in your code, you have to make many changes in other places too
- Dispensables
 - something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand
- Couplers
 - excessive coupling between classes



Inappropriate Naming

Inappropriate Naming

- ☑ Names given to variables (fields) and methods should be clear and meaningful.

Inappropriate Naming

- ☑ Names given to variables (fields) and methods should be clear and meaningful.
- ☑ A variable name should say exactly what it is.
 - ☑ Which is better?
 - ☑ `private String s;` OR `private String salary;`

Inappropriate Naming

- ☑ Names given to variables (fields) and methods should be clear and meaningful.
- ☑ A variable name should say exactly what it is.
 - ☑ Which is better?
 - ☑ `private String s;` OR `private String salary;`
- ☑ A method should say exactly what it does.
 - ☑ Which is better?
 - ☑ `public double calc(double s)`
 - ☑ `public double calculateFederalTaxes(double salary)`

Bloaters

- Long Method
- Long Parameter List
- Large Class
- Primitive Obsession

Long Method

- ☑ A method is long when it is too hard to quickly comprehend.



Long Method

- ☑ A method is long when it is too hard to quickly comprehend.
- ☑ A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.



Long Method

- ☑ A method is long when it is too hard to quickly comprehend.
- ☑ A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.
- ☑ The more lines found in a method, the harder it is to figure out what the method does. This is the main reason for this refactoring.



Long Method

- ☑ A method is long when it is too hard to quickly comprehend.
- ☑ A method contains too many lines of code. Generally, any method longer than ten lines should make you start asking questions.
- ☑ The more lines found in a method, the harder it is to figure out what the method does. This is the main reason for this refactoring.
- ☑ Remedies:
 - ☑ Extract Method
 - ☑ Replace Temp with Query
 - ☑ Introduce Parameter Object
 - ☑ Preserve Whole Object
 - ☑ Replace Method with Method Object.
 - ☑ Decompose Conditional



Long Method Example

```
private String toStringHelper(StringBuffer result)
{
    result.append("<");
    result.append(name);
    result.append(attributes.toString());
    result.append(">");
    if (!value.equals(""))
        result.append(value);
    Iterator it = children().iterator();
    while (it.hasNext())
    {
        TagNode node = (TagNode)it.next();
        node.toStringHelper(result);
    }
    result.append("</");
    result.append(name);
    result.append(">");
    return result.toString();
}
```

Long Method

- To reduce the length of a method body, use **Extract Method**.

To reduce the length of method body: **Extract**

Method

```
private String toStringHelper(StringBuffer result)
{
    writeOpenTagTo(result);
    writeValueTo(result);
    writeChildrenTo(result);
    writeEndTagTo(result);
    return result.toString();
}
```

```
private void writeOpenTagTo(StringBuffer result)
{
    result.append("<");
    result.append(name);
    result.append(attributes.toString());
    result.append(">");
}
```

```
private void writeEndTagTo(StringBuffer result)
{
    result.append("</");
    result.append(name);
    result.append(">");
}
```

```
private void writeValueTo(StringBuffer result)
{
    if (!value.equals(""))
        result.append(value);
}
```

```
private void writeChildrenTo(StringBuffer result)
{
    Iterator it = children().iterator();
    while (it.hasNext())
    {
        TagNode node = (TagNode)it.next();
        node.toStringHelper(result);
    }
}
```

Long Method

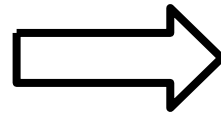
- If local variables and parameters interfere with extracting a method, use
 - **Replace Temp with Query,**
 - **Introduce Parameter Object** or
 - **Preserve Whole Object.**

Replace Temp with Query

```
double calculateTotal() {  
    double basePrice = quantity * itemPrice;  
    if (basePrice > 1000) {  
        return basePrice * 0.95;  
    }  
    else {  
        return basePrice * 0.98;  
    }  
}
```

Replace Temp with Query

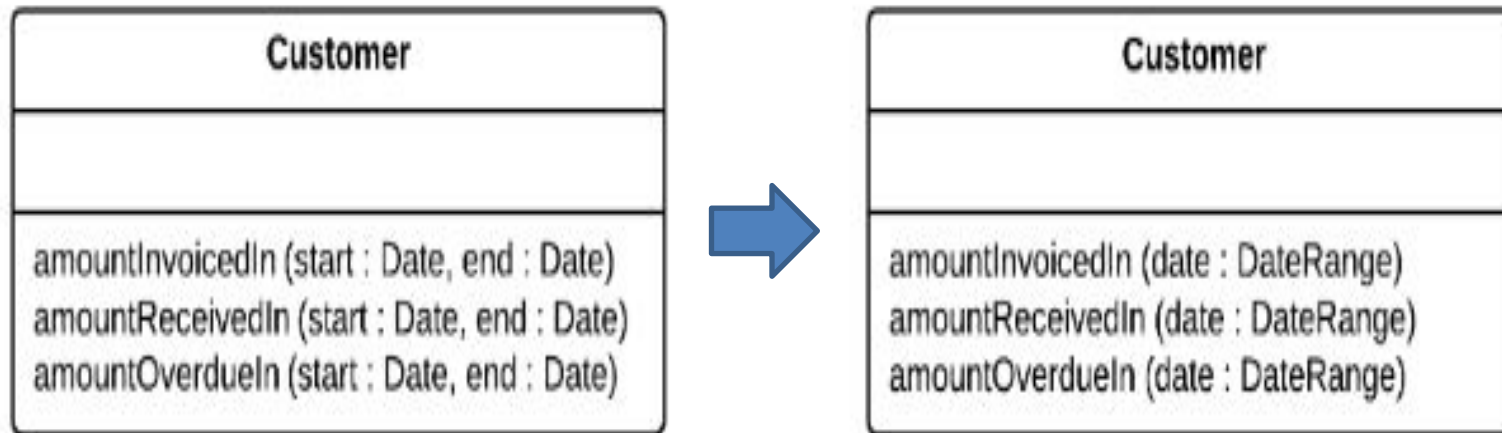
```
double calculateTotal() {  
    double basePrice = quantity * itemPrice;  
    if (basePrice > 1000) {  
        return basePrice * 0.95;  
    }  
    else {  
        return basePrice * 0.98;  
    }  
}
```



```
double calculateTotal() {  
    if (basePrice() > 1000) {  
        return basePrice() * 0.95;  
    }  
    else {  
        return basePrice() * 0.98;  
    }  
}  
  
double basePrice() {  
    return quantity * itemPrice;  
}
```

The same expression may sometimes be found in other methods as well, which is one reason to consider creating a common method.

Introduce Parameter Object



Introduce Parameter Object

- Create a new class that will represent your group of parameters. Make the class immutable.
- In the method that you want to refactor, use Add Parameter, which is where your parameter object will be passed. In all method calls, pass the object created from old method parameters to this parameter.
- Now start deleting old parameters from the method one by one, replacing them in the code with fields of the parameter object. Test the program after each parameter replacement.

Immutable class

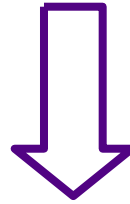
- Immutable class means that once an object is created, we cannot change its content.
- Following are the requirements:
 - The class must be declared as final (So that child classes can't be created)
 - Data members in the class must be declared as final (So that we can't change the value of it after object creation)
 - A parameterized constructor
 - Getter method for all the variables in it
 - No setters (To not have the option to change the value of the instance variable)

Preserve Whole Object

```
int low = daysTempRange.getLow();  
int high = daysTempRange.getHigh();  
withinPlan = plan.withinRange(low, high);
```

Preserve Whole Object

```
int low = daysTempRange.getLow();  
int high = daysTempRange.getHigh();  
withinPlan = plan.withinRange(low, high);
```



```
withinPlan = plan.withinRange(daysTempRange);
```

Long Method

- If none of the previous conditions help, try moving the entire method to a separate object via **Replace Method with Method Object**.

Replace Method with Method Object

```
//class Order...
```

```
double price() {
```

```
    double primaryBasePrice;
```

```
    double secondaryBasePrice;
```

```
    double tertiaryBasePrice;
```

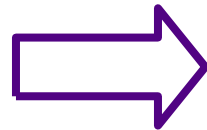
```
    // long computation;
```

```
    ...
```

```
}
```

Replace Method with Method Object

```
class Order {  
    double price() {  
  
        double primaryBasePrice;  
  
        double secondaryBasePrice;  
  
        double tertiaryBasePrice;  
  
        // long computation;  
  
        ...  
    }  
}
```



```
class Order {  
    public double price() {  
        return new PriceCalculator(this).compute();  
    }  
}  
  
class PriceCalculator {  
    private double primaryBasePrice;  
    private double secondaryBasePrice;  
    private double tertiaryBasePrice;  
  
    public PriceCalculator(Order order) {  
        // Copy relevant information from the  
        // order object.  
    }  
  
    public double compute() {  
        // Perform long computation.  
    }  
}
```

Long Method

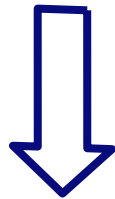
- Conditional operators and loops are a good clue that code can be moved to a separate method. For conditionals, use **Decompose Conditional**. If loops are in the way, try **Extract Method**.

Decompose Conditional

You have a complicated conditional (if-then-else) statement.

Extract methods from the condition, then part, and else parts.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))  
    charge = quantity * _winterRate + _winterServiceCharge;  
else charge = quantity * _summerRate;
```



```
if (notSummer(date))  
    charge = winterCharge(quantity);  
else charge = summerCharge (quantity);
```


Long Parameter List

- Methods that take too many parameters produce client code that is awkward and difficult to work with.
- More than three or four parameters for a method.
- A long list of parameters might happen after several types of algorithms are merged in a single method.
- Remedies
 - Introduce Parameter Object
 - Replace Parameter with Method
 - Preserve Whole Object



Example

```
private void createUserInGroup() {  
    GroupManager groupManager = new GroupManager();  
    Group group = groupManager.create(TEST_GROUP, false,  
        GroupProfile.UNLIMITED_LICENSES, "",  
        GroupProfile.ONE_YEAR, null);  
    user = userManager.create(USER_NAME, group, USER_NAME, "jack",  
        USER_NAME, LANGUAGE, false, false, new Date(),  
        "blah", new Date());  
}
```

Long Parameter List

- Check what values are passed to parameters. If some of the arguments are just results of method calls of another object, use **Replace Parameter with Method**

Replace Parameter with Method

```
public double getPrice() {  
    int basePrice = _quantity * _itemPrice;  
    int discountLevel;  
    if (_quantity > 100)  
        discountLevel = 2;  
    else  
        discountLevel = 1;  
    double finalPrice = discountedPrice (basePrice, discountLevel);  
    return finalPrice;  
}  
  
private double discountedPrice (int basePrice, int discountLevel) {  
    if (discountLevel == 2)  
        return basePrice * 0.1;  
    else  
        return basePrice * 0.05; }  
}
```

Replace Parameter with Method

```
public double getPrice() {  
    int basePrice = _quantity * _itemPrice;  
    int discountLevel = getDiscountLevel();  
    double finalPrice = discountedPrice (basePrice, discountLevel);  
    return finalPrice;  
}  
  
private int getDiscountLevel() {  
    if (_quantity > 100) return 2;  
    else return 1;  
}  
  
private double discountedPrice (int basePrice, int discountLevel) {  
    if (getDiscountLevel() == 2) return basePrice * 0.1;  
    else return basePrice * 0.05;  
}
```

Replace Parameter with Method

```
public double getPrice() {  
    int basePrice = _quantity * _itemPrice;  
    int discountLevel = getDiscountLevel();  
    double finalPrice = discountedPrice (basePrice);  
    return finalPrice;  
}  
  
private int getDiscountLevel() {  
    if (_quantity > 100) return 2;  
    else return 1;  
}  
  
private double discountedPrice (int basePrice) {  
    if (getDiscountLevel() == 2) return basePrice * 0.1;  
    else return basePrice * 0.05;  
}
```

Long Parameter List

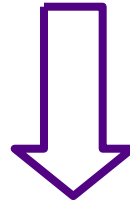
- Instead of passing a group of data received from another object as parameters, pass the object itself to the method, by using **Preserve Whole Object**

Preserve Whole Object

```
int low = daysTempRange.getLow();  
int high = daysTempRange.getHigh();  
withinPlan = plan.withinRange(low, high);
```


Preserve Whole Object

```
int low = daysTempRange.getLow();  
int high = daysTempRange.getHigh();  
withinPlan = plan.withinRange(low, high);
```



```
withinPlan = plan.withinRange(daysTempRange);
```

Long Parameter List

- If there are several unrelated data elements, sometimes you can merge them into a single parameter object via **Introduce Parameter Object**.

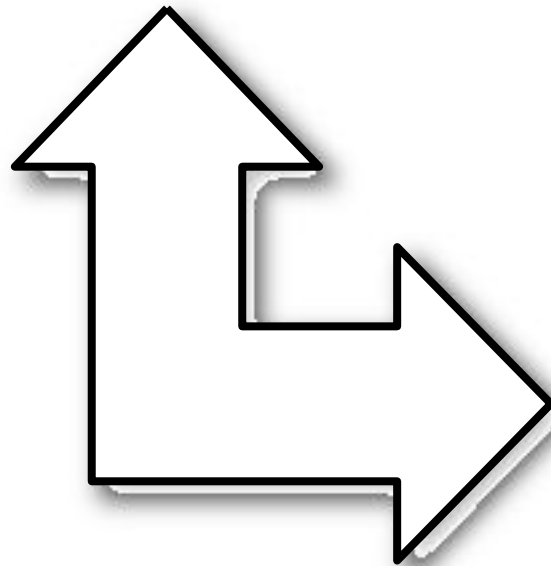
Introduce Parameter Object

Introduce Parameter Object

Customer
AmoutInvoicedIn(Date start, Date end) AmoutRecivedIn(Date start, Date end) AmoutOverdueIn(Date start, Date end)

Introduce Parameter Object

Customer
AmoutInvoicedIn(Date start, Date end)
AmoutRecivedIn(Date start, Date end)
AmoutOverdueIn(Date start, Date end)



Customer
AmoutInvoicedIn(DateRange range)
AmoutRecivedIn(DateRange range)
AmoutOverdueIn(DateRange range)

Long Parameter List

- When to Ignore
 - Do not get rid of parameters if doing so would cause unwanted dependency between classes (coupling).

Large Class

- ☑ Like people, classes suffer when they take on too many responsibilities.
- ☑ Classes usually start small. But over time, they get bloated as the program grows.
- ☑ Fowler and Beck note that the presence of too many instance variables usually indicates that a class is trying to do too much. In general, large classes typically contain too many responsibilities.



Remedies

- ☑ Extract Class
- ☑ Replace Type Code with Class/Subclass
- ☑ Replace Type Code with State/Strategy
- ☑ Replace Conditional with Polymorphism



```

public Vector clearWorkspaceAfterPayrollGeneration(long payrollId, String dName,
String username, String password, String debug) {
    long returnCode = 201;
    String errorCode = "Uncaught exception";
    PayrollDB db = null;
    Connection conn = null;
    Payroll payroll = null;
    PayrollProject payrollProject = null;
    WorkspaceUtil wsu = new WorkspaceUtil();
    Workspace ws = null;
    try {
        for (int i = 0; i < 11; i++) {
            ws = wsu.getWorkspace();
            if (ws == null) {
                returnCode = 101;
                errorCode = "unable to connect to workspace";
                ws.logMessage(errorCode + "\n", true);
                break;
            }
            java.util.Date curTime = new java.util.Date(); // get current
            // time
            ws.logMessage("\n-----" + curTime
                + "-----\n", true);
            ws.logMessage("clearWorkspaceAfterPayrollGeneration(" + payrollId + "," + dName
                + "," + username + ",*****)\n", true);
            errorCode = "Creating PayrollDb object";
            db = new PayrollDB(username, password, dName);
            errorCode = "Connecting to database";
            conn = db.getConnection();
            errorCode = "Converting payrollId to Integer";
            Integer iPayrollId = new Integer((int) payrollId);
            errorCode = "Creating Payroll object";
            payroll = new Payroll(db);
            errorCode = "Calling payroll.selectRowById";
            if (!(payroll.selectRowById(iPayrollId))) {
                returnCode = 102;
                errorCode = "selectPayroll(" + iPayrollId + ") failed";
                ws.logMessage(errorCode + "\n", true);
                break;
            }
            errorCode = "PayrollName (" + payroll.getPayrollName() + ") PayrollType ("
                + payroll.getPayrollType() + ")";
            if (debug.equals("Y"))
                ws.logMessage(errorCode + "\n", true);
            errorCode = "Creating payrollProject object";
            payrollProject = new PayrollProject(db);
            errorCode = "Calling payrollProject.selectRowsById";
            if (!(payrollProject.selectRowsById(iPayrollId))) {
                returnCode = 103;
                errorCode = "selPayrollProject(" + iPayrollId + ") failed";
                ws.logMessage(errorCode + "\n", true);
                break;
            }
            errorCode = "Unloading (" + payrollProject.getRowCnt() + " projects found)";
            boolean bRowFound = payrollProject.firstRow();
            String projectName = null;
            String projectVersion = null;
            Boolean payrollProjectFlag = null;
            long lUnloadCnt = 0;
            OrganizationProjects paygej = null;

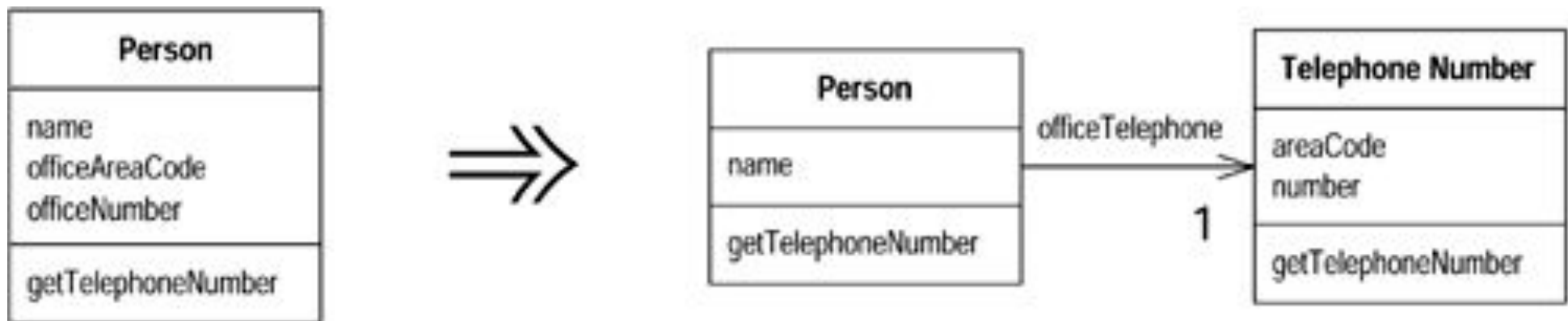
```

```

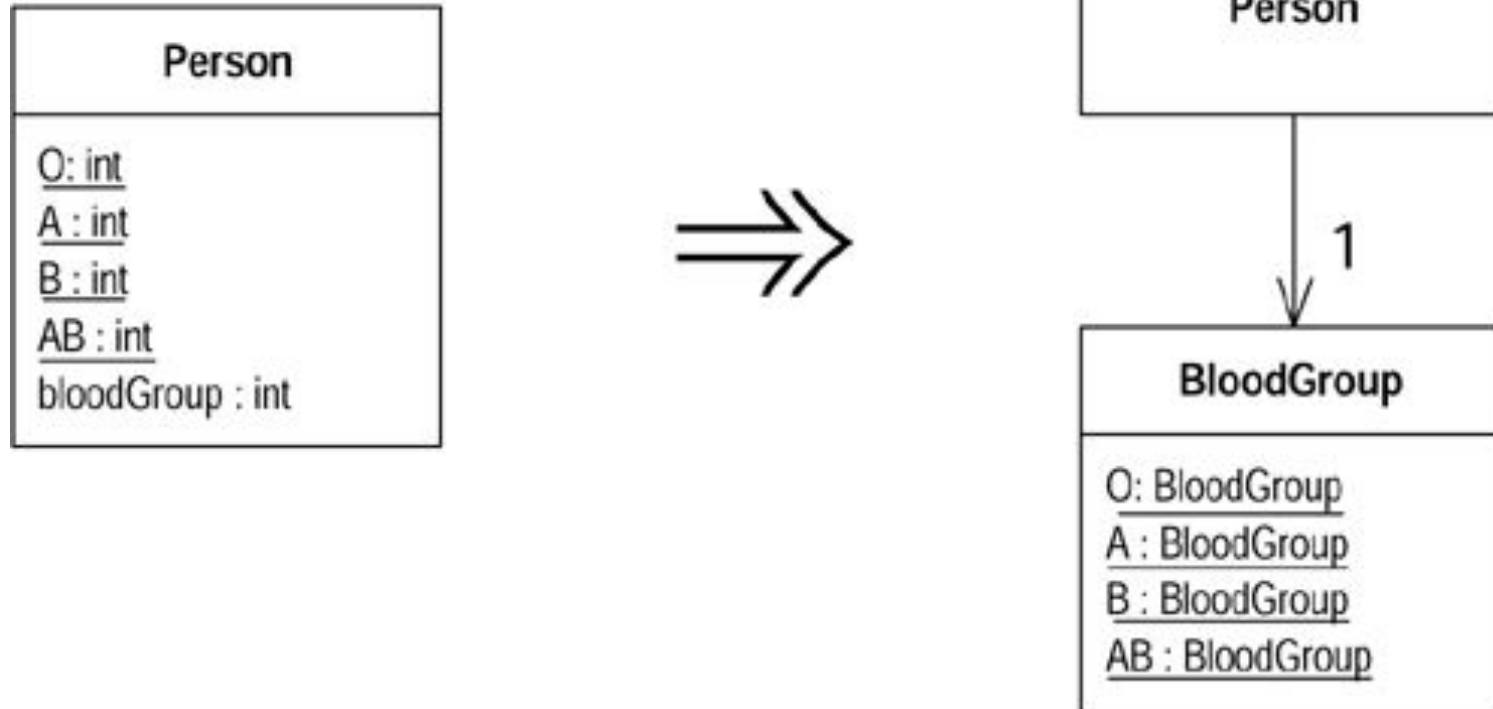
while (bRowFound) {
    projectName = payrollProject.getProjectName();
    projectVersion = payrollProject.getProjectVersion();
    payrollProjectFlag = payrollProject.getPayrollProjectFlag();
    errorCode = "-UnloadingProject (" + projectName + ") Ver (" + projectVersion
        + ") PayrollProject (" + payrollProjectFlag + ")";
    try {
        paygej = new OrganizationProjects(projectName);
    } catch (Exception e) {
        paygej = null;
        errorCode = errorCode + " (non-standard project ignored)";
    }
    if (paygej != null) {
        if (paygej.isOrganizationWideProject() && paygej.isNonPayrollProject()) {
            errorCode = errorCode + " (bypassed common project)";
        } else {
            if (paygej.isATPPProject()) {
                errorCode = errorCode + " unloadATPPProject";
                lUnloadCnt = wsu.unloadATPPProject(paygej, null);
            } else {
                errorCode = errorCode + " unloadProject";
                if (wsu.unloadProject(projectName))
                    lUnloadCnt = 1;
                else
                    lUnloadCnt = 0;
            }
            errorCode = errorCode + " (unloaded " + lUnloadCnt + " projects)";
        }
    }
    if (debug.equals("Y"))
        ws.logMessage(errorCode + "\n", true);
    bRowFound = payrollProject.nextRow();
}
returnCode = 0;
errorCode = "Success";
ws.logMessage(errorCode + "\n", true);
}
} catch (Exception e) {
    errorCode = errorCode + ": excp(" + e + ")";
    if (ws != null) {
        try {
            ws.logMessage(errorCode + "\n", true);
        } catch (Exception eee) {
            errorCode = errorCode + " LOG FAILED: excp(" + eee + ")";
        }
    }
} finally {
    if (db != null) {
        try {
            db.closeConnection();
        } catch (Exception ee) {
            // ignore
        }
    }
}
return commandUtil.commandVector(returnCode, errorCode);

```

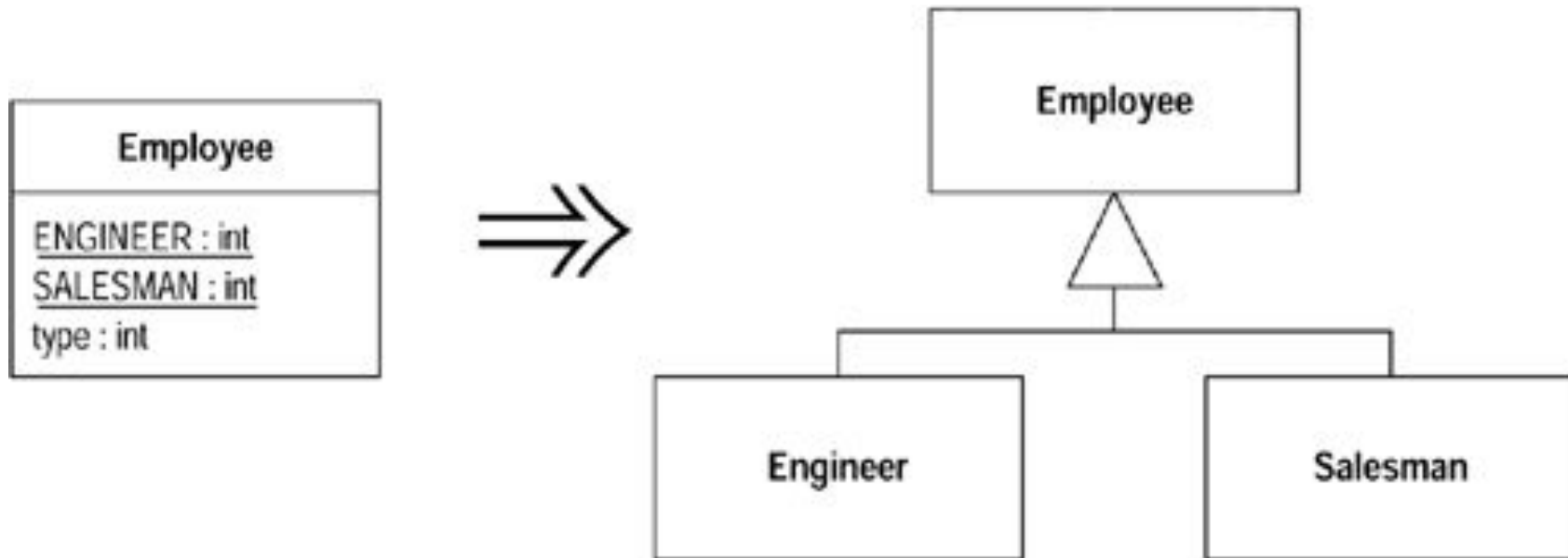

Extract Class



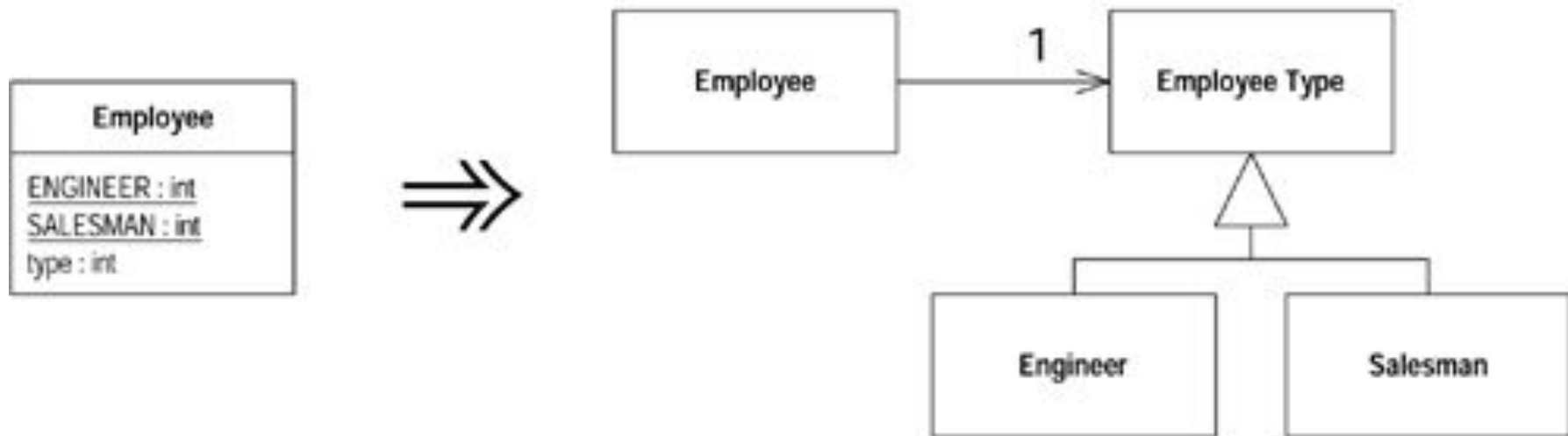
Replace Type Code with Class



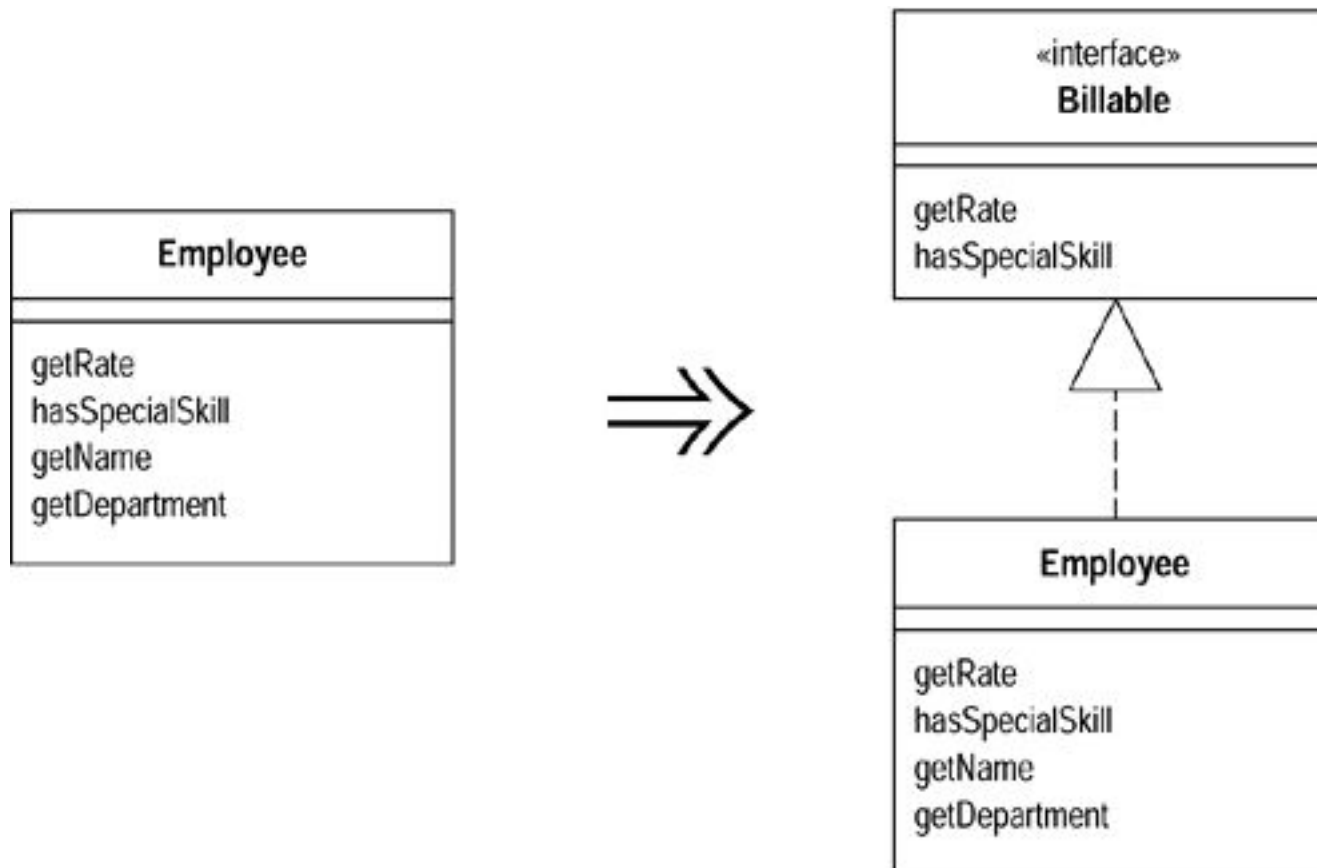
Replace Type Code with Subclasses



Replace Type Code with State/Strategy



Extract (Narrow) Interface



Primitive Obsession

- ☑ This smell exists when primitives, such as strings, doubles, arrays or low-level language components, are used for high-level operations instead of using classes.
- ☑ This typically occurs when you haven't yet seen how a higher-level abstraction can clarify or simplify your code.

- ☑
- ☑
- ☑
- ☑
- ☑
- ☑



Primitive Obsession Example

```
if (someString.indexOf("substring") != -1)
```



```
if(someString.contains("substring"))
```

Primitive Obsession Example

```
private void Grow() {  
    Object[] newElements = new Object[elements.length + 10];  
    for (int i = 0; i < size; i++)  
        newElements[i] = elements[i];  
  
    elements = newElements;  
}
```



```
private void Grow() {  
    Object[] newElements = new Object[elements.length +  
    INITIAL_CAPACITY];  
    System.arraycopy(elements, 0, newElements, 0, size);  
    elements = newElements;  
}
```


Primitive Obsession Example

```
public class CompositeShape
{
    IShape [] arr = new IShape[100];
    int count = 0;

    public void Add(IShape shape){
        arr[count++] = shape;
    }

    public void Remove(IShape shape)
    {
        for (int i = 0; i < 100; i++)
        {
            if (shape == arr[i])
            {
                //code to remove
            }
        }
    }
}
```

Primitive Obsessed Code - Make Over

```
public class CompositeShape
{
    List<IShape> shapeList = new List<IShape>();

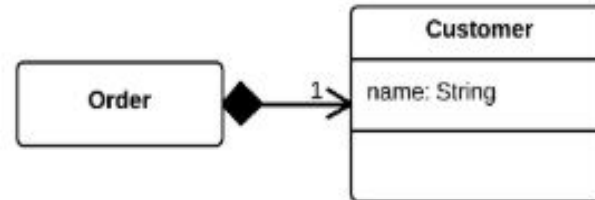
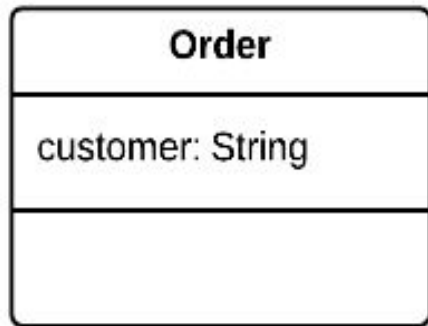
    public void Add(IShape shape)
    {
        shapeList.Add(shape);
    }

    public void Remove(IShape shape)
    {
        shapeList.Remove(shape);
    }
}
```

Primitive Obsession

- If you have a large variety of primitive fields, it may be possible to logically group some of them into their own class. Even better, move the behavior associated with this data into the class too. For this task, try **Replace Data Value with Object**.

Replace Data Value with Object

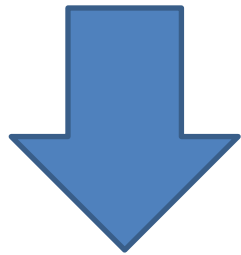


Primitive Obsession

- If the values of primitive fields are used in method parameters, go with **Introduce Parameter Object** or **Preserve Whole Object**.
- When complicated data is coded in variables, use **Replace Type Code with Class**, **Replace Type Code with Subclasses** or **Replace Type Code with State/Strategy**.
- If there are arrays among the variables, use **Replace Array with Object**.

Replace Array with Object.

```
String[] row = new String[2];  
row[0] = "Liverpool";  
row[1] = "15";
```



```
Performance row = new Performance();  
row.setName("Liverpool");  
row.setWins("15");
```

Dispensables

- Comments
- Duplicate Code
- Lazy Class
- Dead Code
- Speculative Generality

Comments

- ☑ Comments are often used as deodorant
- ☑ Comments represent a *failure to express an idea in the code*. Try to make your code self-documenting or intention-revealing
- ☑ When you feel like writing a comment, first try "to refactor so that the comment becomes superfluous."
- ☑ Remedies:
 - ☑ Extract Method
 - ☑ Rename Method
 - ☑ Introduce Assertion



Comment: “Grow the Array” smells

```
public class MyList
{
    int INITIAL_CAPACITY = 10;
    bool m_readOnly;
    int m_size = 0;
    int m_capacity;
    string[] m_elements;

    public MyList()
    {
        m_elements = new string[INITIAL_CAPACITY];
        m_capacity = INITIAL_CAPACITY;
    }

    int GetCapacity() {
        return m_capacity;
    }
}
```

```
void AddToList(string element)
{
    if (!m_readOnly)
    {
        int newSize = m_size + 1;
        if (newSize > GetCapacity())
        {
            // grow the array
            m_capacity += INITIAL_CAPACITY;
            string[] elements2 = new string[m_capacity];
            for (int i = 0; i < m_size; i++)
                elements2[i] = m_elements[i];

            m_elements = elements2;
        }
        m_elements[m_size++] = element;
    }
}
```

Comment Smells Make-over

```
void AddToList(string element)
{
    if (m_readOnly)
        return;
    if (ShouldGrow())
    {
        Grow();
    }
    StoreElement(element);
}
```

```
private bool ShouldGrow()
{
    return (m_size + 1) > GetCapacity();
}
```

```
private void Grow()
{
    m_capacity += INITIAL_CAPACITY;
    string[] elements2 = new string[m_capacity];
    for (int i = 0; i < m_size; i++)
        elements2[i] = m_elements[i];

    m_elements = elements2;
}

private void StoreElement(string element)
{
    m_elements[m_size++] = element;
}
```

Rename Method



Extract Method

```
void PrintOwning(double amount){  
    PrintBanner();  
  
    // print details  
    System.Console.Out.WriteLine("name: "+ name);  
    System.Console.Out.WriteLine("amount: "+ amount);  
}
```

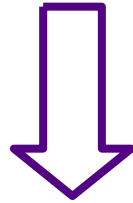
Extract Method

```
void PrintOwning(double amount){  
    PrintBanner();
```

```
    // print details
```

```
    System.Console.Out.WriteLine("name: "+ name);  
    System.Console.Out.WriteLine("amount: "+ amount);
```

```
}
```



```
void PrintOwning(double amount){  
    PrintBanner();  
    PrintDetails(amount);  
}
```

```
void PrintDetails(double amount){  
    System.Console.Out.WriteLine("name: "+ name);  
    System.Console.Out.WriteLine("amount: "+ amount);  
}
```

Introduce Assertion

Introduce Assertion

```
double getExpenseLimit() {  
    // should have either expense limit or a primary project  
    return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :  
        _primaryProject.GetMemberExpenseLimit();  
}
```

Introduce Assertion

```
double getExpenseLimit() {  
    // should have either expense limit or a primary project  
    return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :  
    _primaryProject.GetMemberExpenseLimit();  
}
```



```
double getExpenseLimit() {  
    assert(_expenseLimit != NULL_EXPENSE || _primaryProject != null,  
    "Both Expense Limit and Primary Project must not be null");  
  
    return (_expenseLimit != NULL_EXPENSE) ? _expenseLimit :  
    _primaryProject.GetMemberExpenseLimit();  
}
```


Introduce Assertion

- A portion of code assumes something about, for example, the current condition of an object or value of a parameter or local variable. Usually this assumption will always hold true except in the event of an error
- Make these assumptions obvious by adding corresponding assertions.
- Check for comments that describe the conditions under which a particular method will work
- When you see that a condition is assumed, add an assertion for this condition in order to make sure.

Dead Code

☑ Code that is no longer used in a system or related system is Dead Code.

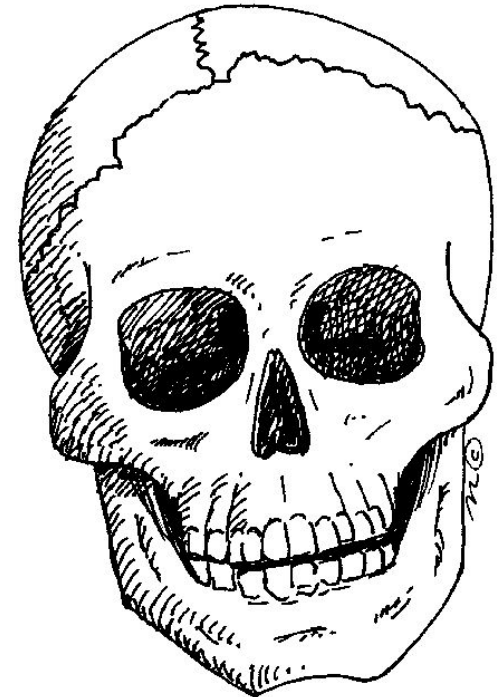
☑ Increased Complexity.

☑ Accidental Changes.

☑ More Dead Code

☑ Remedies

☑

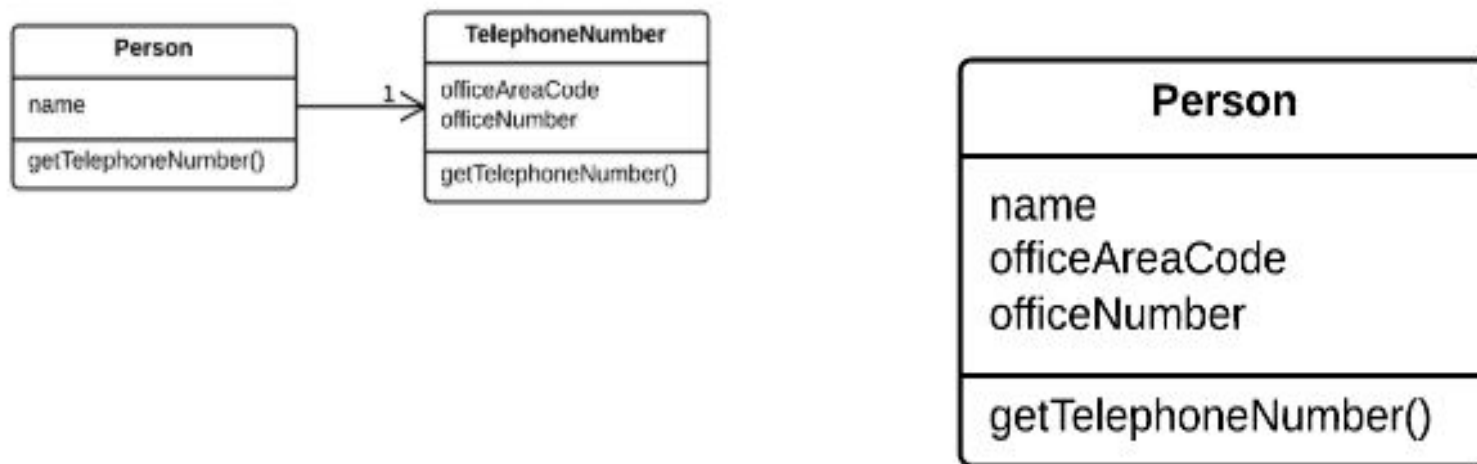


Dead Code

- Delete unused code and unneeded files.
- In the case of an unnecessary class, **Inline Class** or **Collapse Hierarchy** can be applied if a subclass or superclass is used.

Inline Class

- A class does almost nothing and is not responsible for anything, and no additional responsibilities are planned for it.
- Move all features from the class to another one.



Collapse Hierarchy

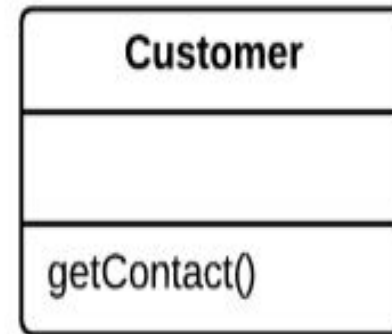
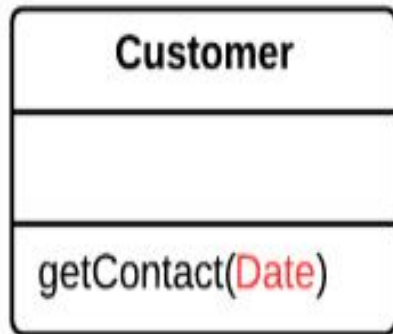
- You have a class hierarchy in which a subclass is practically the same as its superclass.
- Merge the subclass and superclass.



Dead Code

- To remove unneeded parameters, use **Remove Parameter**.

Remove Parameter

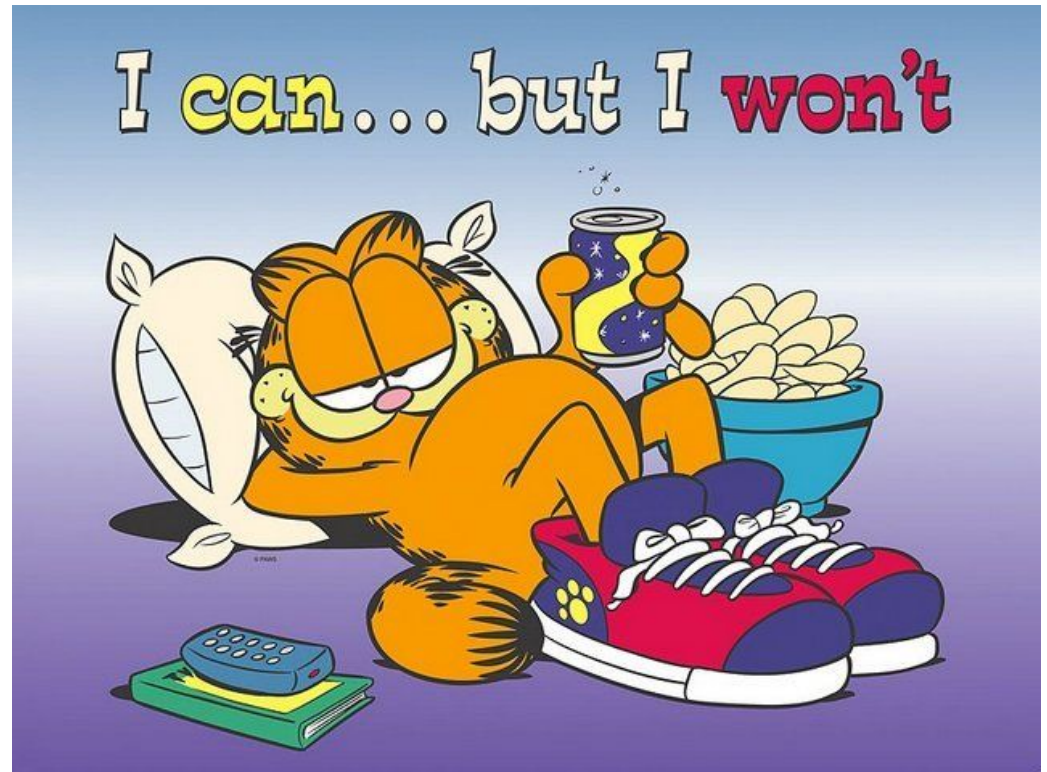


Lazy Class

- ☑ A class that isn't doing enough to carry its weight
- ☑ We let the class die with dignity
- ☑ Often this might be a class that used to pay its way but has been downsized with refactoring. Or it might be a class that was added because of changes that were planned but not made.

Remedies

- ☑ Inline Class
- ☑ Collapse Hierarchy



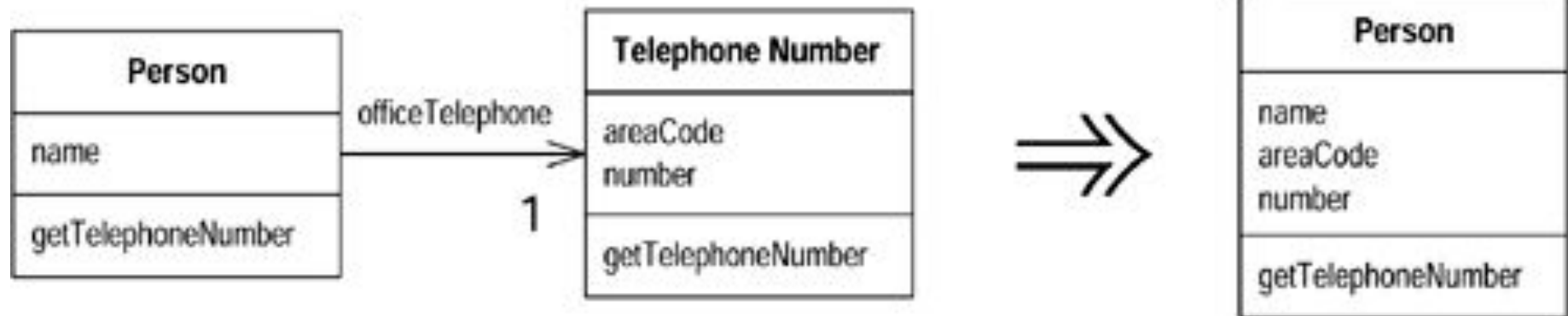
Lazy Class Example

```
public interface SomeInterface {  
    void methodOne();  
    void defaultMethod();  
}  
public abstract class LazyClazz implements SomeInterface {  
    public abstract void methodOne();  
    public void defaultMethod() {  
        //do nothing  
    }  
}  
public class WorkerClazz extends LazyClazz {  
    public void methodOne() {  
        // some actual code here  
    }  
    public void defaultMethod() {  
        //some more actual code  
    }  
}
```

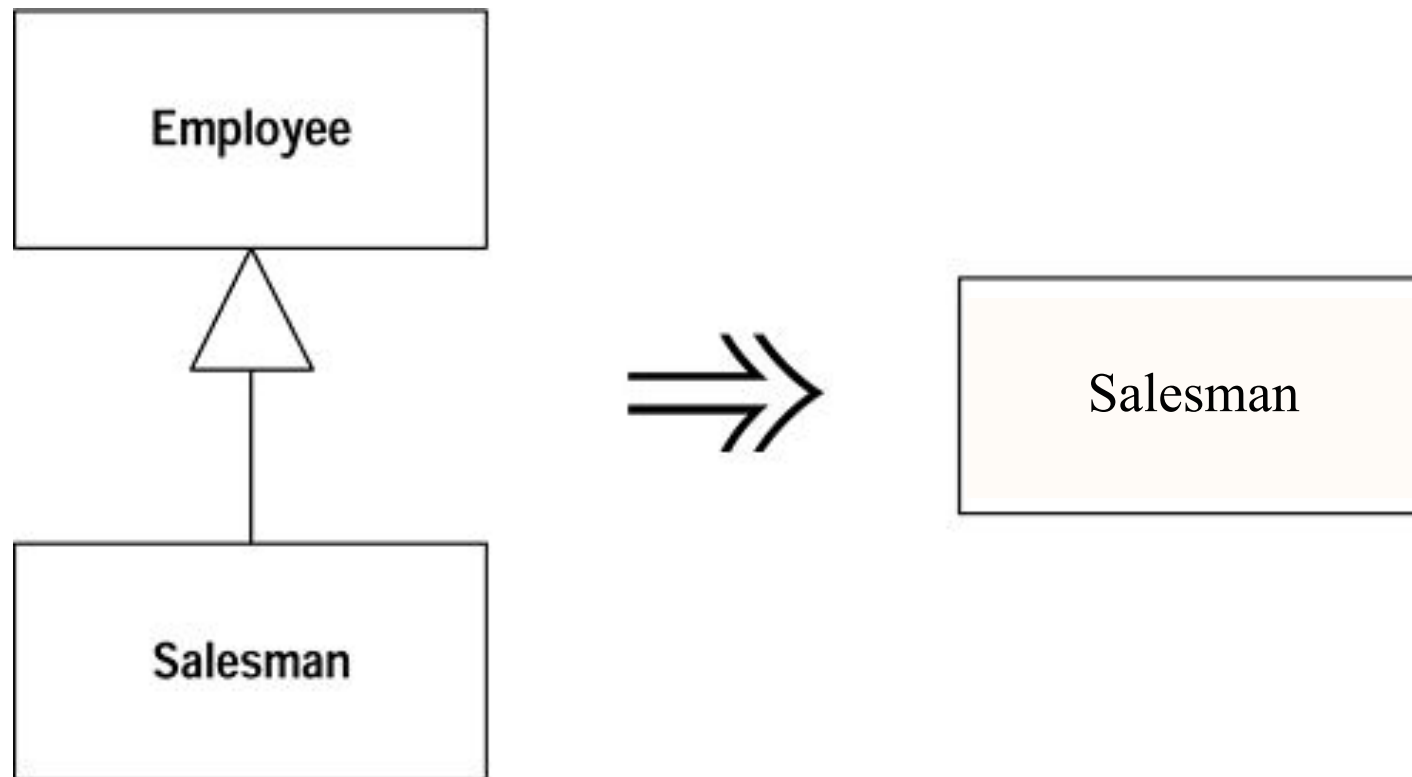
Another Lazy Class

```
public class Letter {  
    private final String content;  
  
    public Letter(String content) {  
        this.content = content;  
    }  
  
    public String getContent() {  
        return content;  
    }  
}
```

Inline Class



Collapse Hierarchy



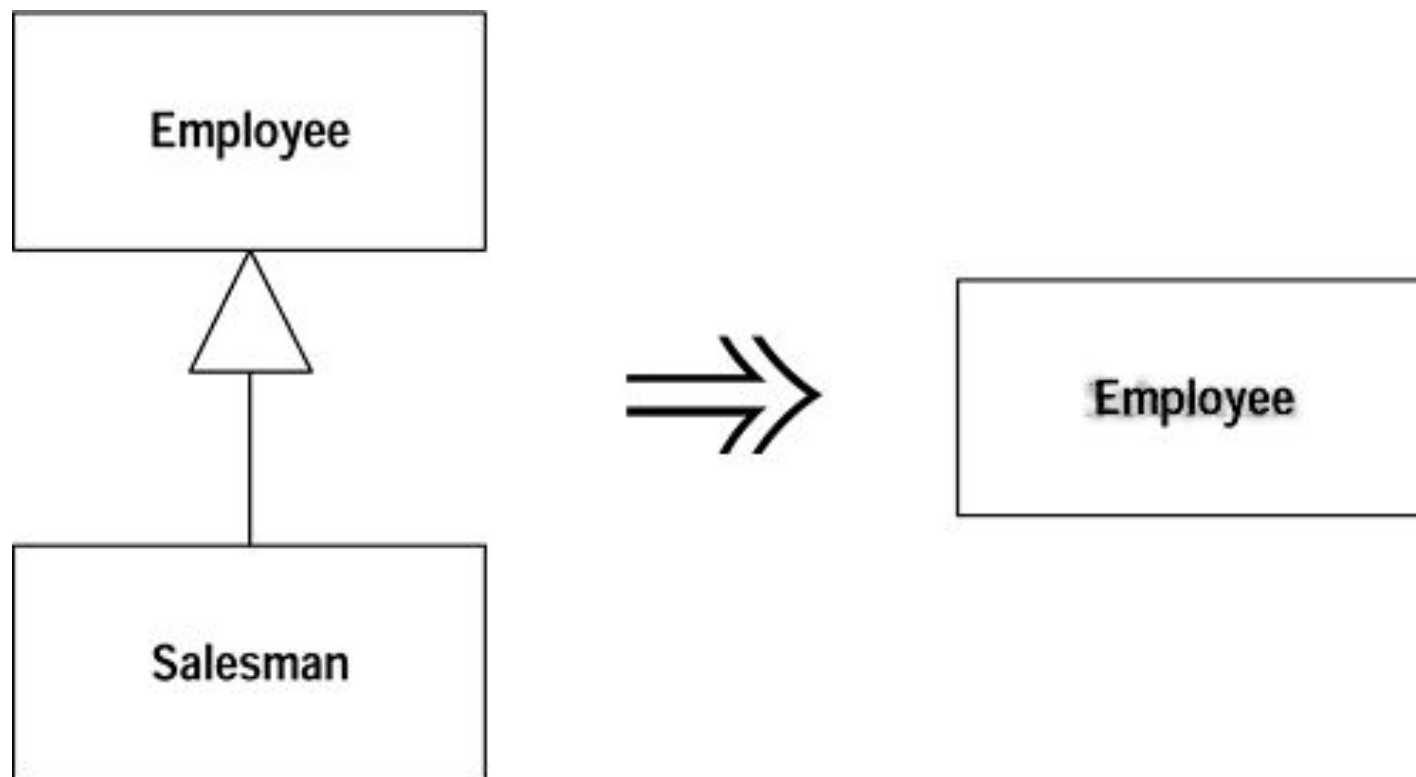
Speculative Generality

- ☑ You get this smell when people say "Oh, I think we will need the ability to do that someday" and thus want all sorts of hooks and special cases to handle things that aren't required.
- ☑ This odor exists when you have generic or abstract code that isn't actually needed today. Such code often exists to support future behavior, which may or may not be necessary in the future.
- ☑ Remedies
 - ☑ Collapse Hierarchy
 - ☑ Inline Class
 - ☑ Remove Parameter



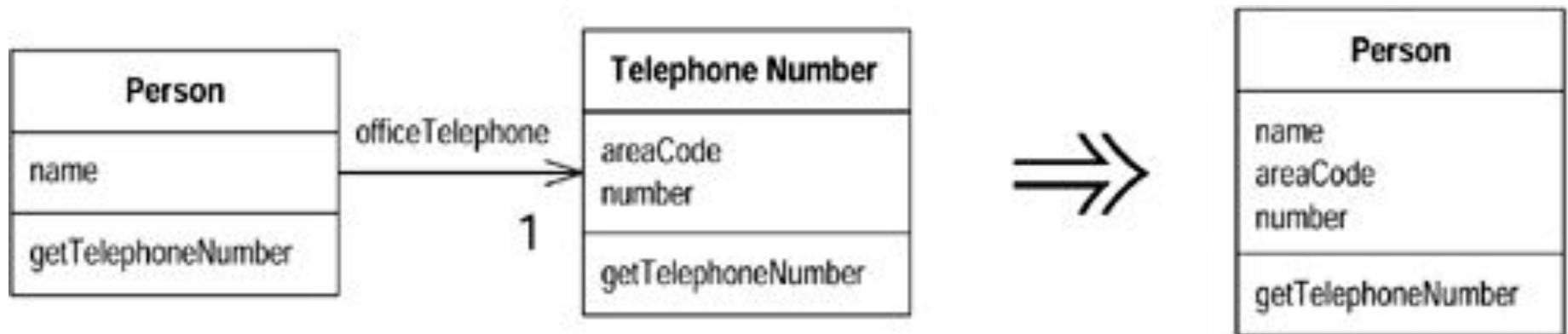
Collapse Hierarchy

- For removing unused abstract classes, try **Collapse Hierarchy**



Inline Class

- Unnecessary delegation of functionality to another class can be eliminated via **Inline Class**.



Remove Parameter

- Methods with unused parameters should be given a look with the help of **Remove Parameter**.



Duplicated Code

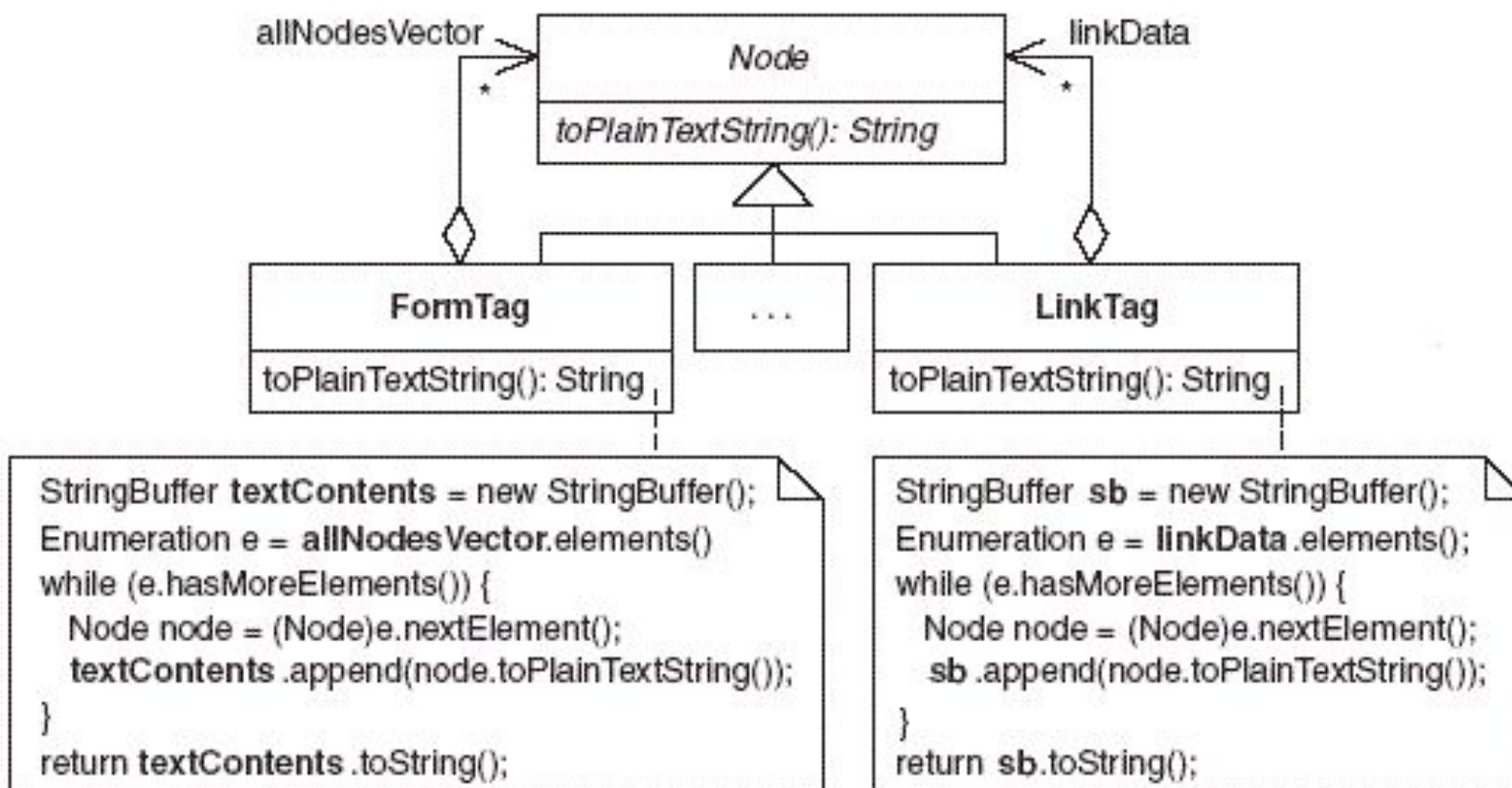
- usually occurs when multiple programmers are working on different parts of the same program at the same time
- when specific parts of code look different but actually perform the same job. This kind of duplication can be hard to find and fix.
- copying and pasting the relevant code
- There are subtle or non-obvious duplications
 - Such as parallel inheritance hierarchies.
 - Similar algorithms

Ctl+C Ctl+V Pattern

```
public static MailTemplate getStaticTemplate(Languages language) {
    MailTemplate mailTemplate = null;
    if(language.equals(Languages.English)) {
        mailTemplate = new EnglishLanguageTemplate();
    } else if(language.equals(Languages.French)) {
        mailTemplate = new FrenchLanguageTemplate();
    } else if(language.equals(Languages.Chinese)) {
        mailTemplate = new ChineseLanguageTemplate();
    } else {
        throw new IllegalArgumentException("Invalid language type specified");
    }
    return mailTemplate;
}

public static MailTemplate getDynamicTemplate(Languages language, String content) {
    MailTemplate mailTemplate = null;
    if(language.equals(Languages.English)) {
        mailTemplate = new EnglishLanguageTemplate(content);
    } else if(language.equals(Languages.French)) {
        mailTemplate = new FrenchLanguageTemplate(content);
    } else if(language.equals(Languages.Chinese)) {
        mailTemplate = new ChineseLanguageTemplate(content);
    } else {
        throw new IllegalArgumentException("Invalid language type specified");
    }
    return mailTemplate;
}
```

Example Of Obvious Duplication



```

private void AddOrderMaterials(int iOrderId)
{

    if (iOrderType == 1)
    {
        OrderMaterial oOrderMaterialCoffee = new OrderMaterial();
        oOrderMaterialCoffee.MaterialId = 1;
        oOrderMaterialCoffee.OrderId = iOrderId;
        oOrderMaterialCoffee.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCoffee);

        oDataContext.SubmitChanges();
    }
    else if (iOrderType == 2)
    {
        OrderMaterial oOrderMaterialCoffee = new OrderMaterial();
        oOrderMaterialCoffee.MaterialId = 1;
        oOrderMaterialCoffee.OrderId = iOrderId;
        oOrderMaterialCoffee.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCoffee);

        OrderMaterial oOrderMaterialCream = new OrderMaterial();
        oOrderMaterialCream.MaterialId = 2;
        oOrderMaterialCream.OrderId = iOrderId;
        oOrderMaterialCream.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCream);

        oDataContext.SubmitChanges();
    }
    else if (iOrderType == 3)
    {
        OrderMaterial oOrderMaterialCoffee = new OrderMaterial();
        oOrderMaterialCoffee.MaterialId = 1;
        oOrderMaterialCoffee.OrderId = iOrderId;
        oOrderMaterialCoffee.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialCoffee);

        OrderMaterial oOrderMaterialSugar = new OrderMaterial();
        oOrderMaterialSugar.MaterialId = 3;
        oOrderMaterialSugar.OrderId = iOrderId;
        oOrderMaterialSugar.Quantity = 2;
        oDataContext.OrderMaterials.Inserton<x>submit(oOrderMaterialSugar);

        oDataContext.SubmitChanges();
    }
    else if (iOrderType == 4)

```



Levels of Duplication



Literal Duplication

Same for loop in 2 places

Semantic Duplication

1stLevel - For and For Each Loop

```
stack.push(1); stack.push(3);  
stack.push(5); stack.push(10);  
stack.push(15);
```

v/s

```
for(int i : asList(1,3,5,10,15))  
stack.push(i);
```

2ndLevel - Loop v/s Lines repeated




Data Duplication

Some constant declared in 2 classes (test and production)



Conceptual Duplication

2 Algorithm to Sort elements (Bubble sort and Quick sort)



Logical Steps - Duplication

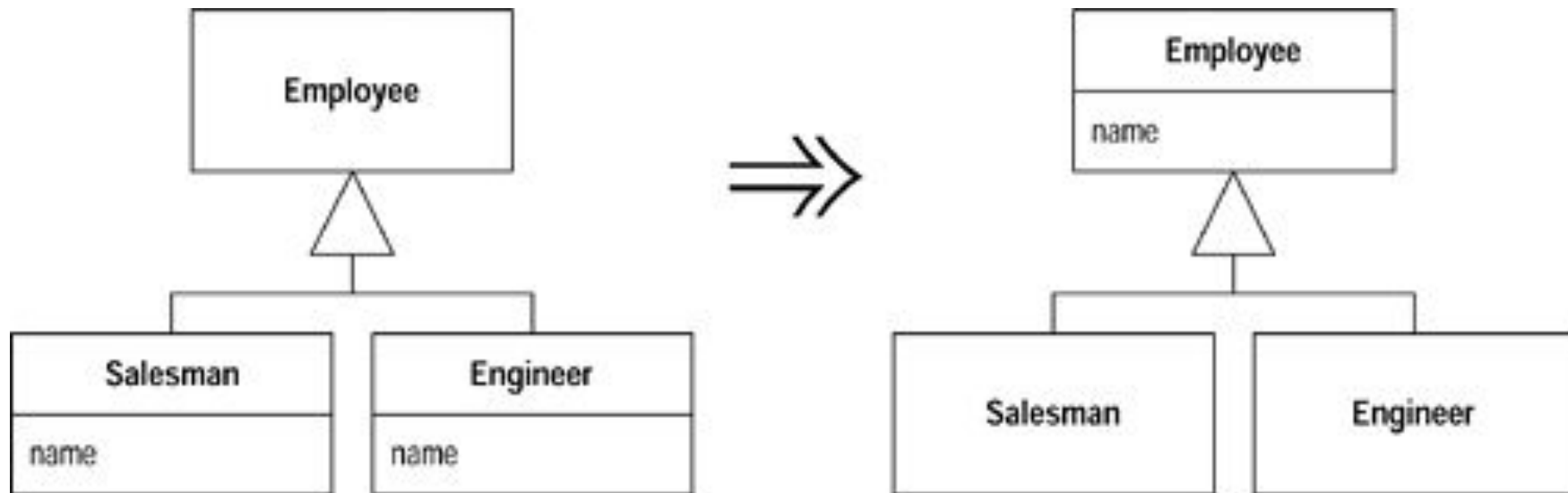
Same set of steps repeat in different scenarios.

Ex: Same set of validations in various points in your applications

Duplicated Code

- If the same code is found in two or more methods in the same class: use **Extract Method** and place calls for the new method in both places.
- If the same code is found in two subclasses of the same level:
 - Use **Extract Method** for both classes, followed by **Pull Up Field** for the fields used in the method that you are pulling up.

Pull Up Field



If the same code is found in two subclasses of the same level

- If the duplicate code is inside a constructor, use **Pull Up Constructor Body**.

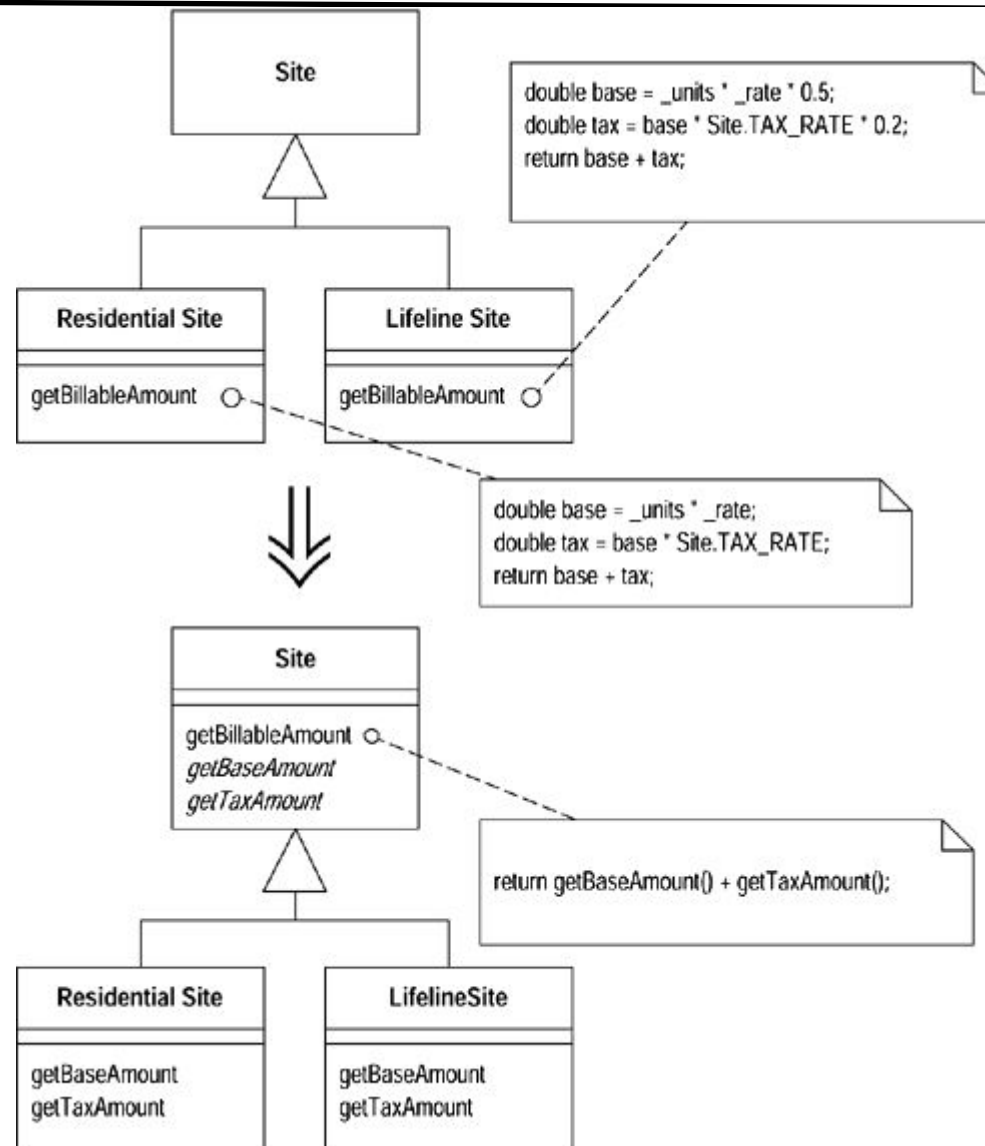
```
class Manager extends Employee {  
    public Manager(String name, String id,  
int grade) {  
        this.name = name;  
        this.id = id;  
        this.grade = grade;  
    }  
    // ...  
}
```

```
class Manager extends Employee {  
    public Manager(String name, String id, int grade)  
    {  
        super(name, id);  
        this.grade = grade;  
    }  
    // ...  
}
```

If the same code is found in two subclasses
of the same level

- If the duplicate code is similar but not completely identical, use **Form Template Method**.

Form Template Method



If the same code is found in two subclasses
of the same level

- If two methods do the same thing but use different algorithms, select the best algorithm and apply **Substitute Algorithm**.

Substitute Algorithm

```
String foundPerson(String[] people){  
  for (int i = 0; i < people.length; i++) {  
    if (people[i].equals ("Don")){  
      return "Don";  
    }  
    if (people[i].equals ("John")){  
      return "John";  
    }  
    if (people[i].equals ("Kent")){  
      return "Kent";  
    }  
  }  
  return ""; }  
}
```

Substitute Algorithm

```
String foundPerson(String[] people){
```

```
    for (int i = 0; i < people.length; i++) {
```

```
        if (people[i].equals ("Don")){
```

```
            return "Don";
```

```
        }
```

```
        if (people[i].equals ("John")){
```

```
            return "John";
```

```
        }
```

```
        if (people[i].equals ("Kent")){
```

```
            return "Kent";
```

```
        }
```

```
    }
```

```
    return "";
```

```
}
```

```
String foundPerson(String[] people){
```

```
    List candidates = Arrays.asList(new String[] {"Don",  
"John", "Kent"});
```

```
    for (String person : people)
```

```
        if (candidates.contains(person))
```

```
            return person;
```

```
    return "";
```

```
}
```

If duplicate code is found in two different classes

- If the classes are not part of a hierarchy, use **Extract Superclass** in order to create a single superclass for these classes that maintains all the previous functionality.
- If it is difficult or impossible to create a superclass, use **Extract Class** in one class and use the new component in the other.

- If a large number of conditional expressions are present and perform the same code (differing only in their conditions), merge these operators into a single condition using **Consolidate Conditional Expression** and use **Extract Method** to place the condition in a separate method with an easy-to-understand name.

Consolidate Conditional Expression

- You have multiple conditionals that lead to the same result or action.
- The main purpose of consolidation is to extract the conditional to a separate method for greater clarity.

```
double disabilityAmount() {  
    if (seniority < 2) {  
        return 0;  
    }  
    if (monthsDisabled > 12) {  
        return 0;  
    }  
    if (isPartTime) {  
        return 0;  
    }  
    // Compute the disability amount.  
    // ...  
}
```

```
double disabilityAmount() {  
    if (isNotEligibleForDisability()) {  
        return 0;  
    }  
    // Compute the disability amount.  
    // ...  
}
```

- Consolidate the conditionals in a single expression by using **and** and **or**.

- If the same code is performed in all branches of a conditional expression: place the identical code outside of the condition tree by using **Consolidate Duplicate Conditional Fragments**.

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
    send();  
}  
else {  
    total = price * 0.98;  
    send();  
}
```

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
}  
else {  
    total = price * 0.98;  
}  
send();
```

Object-Orientation Abusers

- Switch Statements
- Refused Bequest
- Alternative Classes with Different Interfaces

Switch Statement

- ☑ This smell exists when the same switch statement (or “if...else if...else if” statement) is duplicated across a system.
- ☑ Such duplicated code reveals a lack of object-orientation and a missed opportunity to rely on the elegance of polymorphism.
- ☑ Remedies:
 - ☑ Replace Type Code with Subclasses
 - ☑ Replace Type Code with State / Strategy
 - ☑ Replace Parameter with Explicit Methods
 - ☑ Introduce Null Object Pattern.



Switch Smell Examples

```
if( "=".equalsIgnoreCase(operator.trim())){
    for(int i=0;i<ruleCriteria.getCriteriaSize();i++){
        if(ruleCriteria.getCriteria(i).equalsIgnoreCase("Name"))
            criteriaCompareStrategy[i] = new Integer(nameFlag);
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("City"))
            criteriaCompareStrategy[i]=new Integer(cityFlag);
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("Address"))
            criteriaCompareStrategy[i]=new Integer(addressFlag);
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("Age"))
            criteriaCompareStrategy[i]=new Integer(ageFlag);
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("Income"))
            criteriaCompareStrategy[i]=new Integer(incomeFlag);
        else if(ruleCriteria.getCriteria(i).equalsIgnoreCase("TotalPurchase"))
            criteriaCompareStrategy[i]=new Integer(spendingFlag);
    }
}
```

More Switch Smell Examples

```
switch(strategy){
    case 1:
        if(!(name == null))
            this.name.put("Name", name);
        break;
    case 2:
        if(!(address == null))
            this.address.put("Address", address);
        break;
    case 3:
        if(!(city==null))
            this.city.put("City", city);
        break;
    case 4:
        if(!(age == 0))
            this.age.put("Age", new Integer(age));
        break;
    case 5:
        if(!(sal==0))
            this.income.put("Income", new Double(sal));
        break;
    case 6:
        if(!(spending==0))
            this.totalPurchase.put("TotalPurchase", new Double(spending));
        break;
}
```

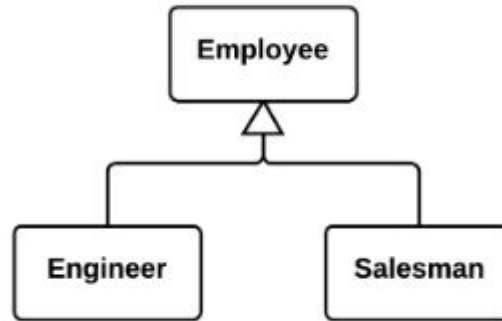
Evil Switch Example

```
while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental)rentals.nextElement();

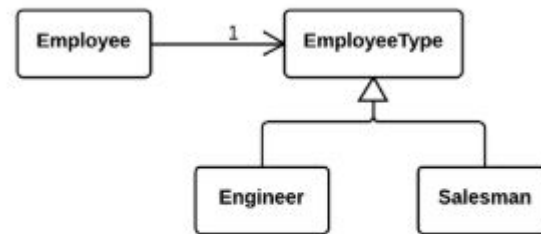
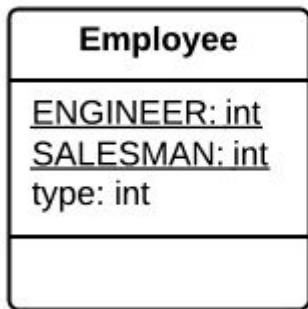
    //determine amounts for each line
    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
    }
}
```

Reason & Remedies

- Relatively rare use of switch and case operators is one of the hallmarks of object-oriented code
- As a rule of thumb, when you see switch you should think of polymorphism.
- If a `switch` is based on type code, such as when the program's runtime mode is switched, use **Replace Type Code with Subclasses** or **Replace Type Code with State/Strategy**.
- Type code occurs when, instead of a separate data type, you have a set of numbers or strings that form a list of allowable values for some entity.



Replace Type Code with Subclasses



Replace Type Code with State/Strategy

After specifying the inheritance structure, use **Replace Conditional with Polymorphism**

Replace Conditional with Polymorphism

- You have a conditional that performs various actions depending on object type or properties then follow the following the steps
 - Create subclasses matching the branches of the conditional.
 - In them, create a shared method and move code from the corresponding branch of the conditional to it.
 - Then replace the conditional with the relevant method call.
 - The result is that the proper implementation will be attained via polymorphism depending on the object class.

Replace Conditional with Polymorphism

```
class Bird {  
    // ...  
    double getSpeed() {  
        switch (type) {  
            case EUROPEAN:  
                return getBaseSpeed();  
            case AFRICAN:  
                return getBaseSpeed() - getLoadFactor() *  
numberOfCoconuts;  
            case NORWEGIAN_BLUE:  
                return (isNailed) ? 0 :  
getBaseSpeed(voltage);  
        }  
        throw new RuntimeException("Should be  
unreachable");  
    }  
}
```

```
abstract class Bird {  
    // ...  
    abstract double getSpeed(); }
```

```
class European extends Bird {  
    double getSpeed() {  
        return getBaseSpeed(); } }
```

```
class African extends Bird {  
    double getSpeed() {  
        return getBaseSpeed() - getLoadFactor() *  
numberOfCoconuts; } }
```

```
class NorwegianBlue extends Bird {  
    double getSpeed() {  
        return (isNailed) ? 0 :  
getBaseSpeed(voltage); } }
```

```
// Somewhere in client code  
speed = bird.getSpeed();
```

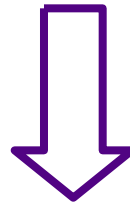

- If there are not too many conditions in the operator and they all call same method with different parameters, polymorphism will be superfluous. If this case, you can break that method into multiple smaller methods with **Replace Parameter with Explicit Methods** and change the `switch` accordingly.

Replace Parameter with Explicit Method

```
void setValue (String name, int value) {  
    if (name.equals("height"))  
        this.height = value;  
    else if (name.equals("width"))  
        this.width = value;  
}
```

Replace Parameter with Explicit Method

```
void setValue (String name, int value) {  
    if (name.equals("height"))  
        this.height = value;  
    else if (name.equals("width"))  
        this.width = value;  
}
```



```
void setHeight(int h) {  
    this.height = h;  
}  
  
void setWidth (int w) {  
    this.width = w;  
}
```

Introduce Null Object Pattern

```
// In client class
Customer customer = site.getCustomer();
BillingPlan plan;
if (customer == null)
    plan = BillingPlan.basic();
else
    plan = customer.getPlan();
```

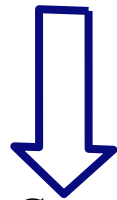
Introduce Null Object Pattern

```
// In client class
Customer customer = site.getCustomer();

BillingPlan plan;

if (customer == null) plan = BillingPlan.basic();

else plan = customer.getPlan();
```



```
// In client class
Customer customer = site.getCustomer();

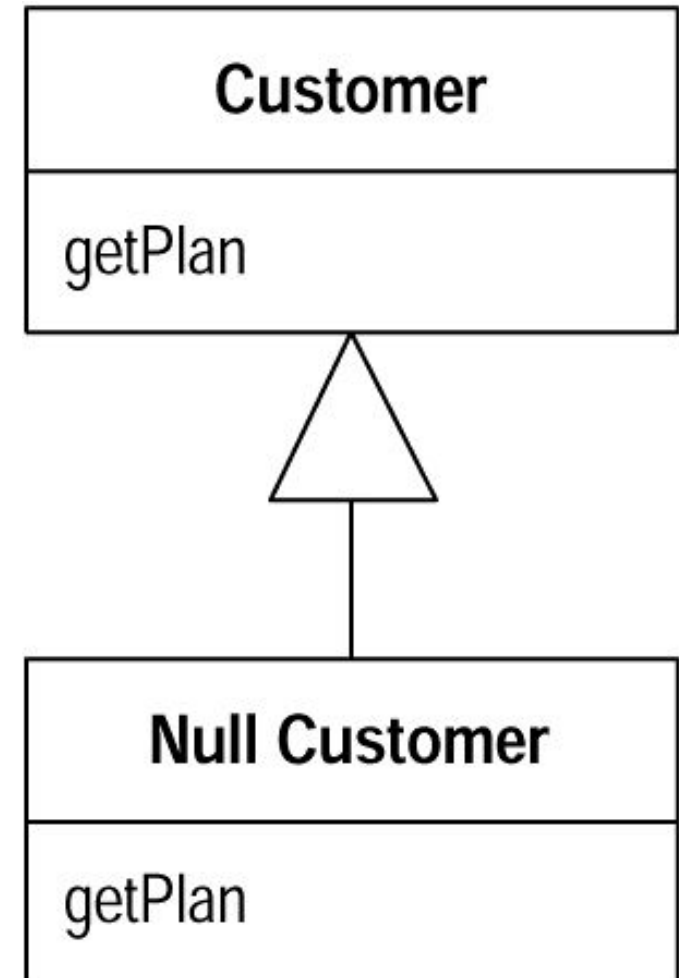
BillingPlan plan = customer.getPlan();

// In Null Customer

public BillingPlan getPlan(){

    return BillingPlan.basic();

}
```



Refactoring & Patterns

- ☑ There is a natural relation between patterns and refactorings. Patterns are where you want to be; refactorings are ways to get there from somewhere else. - Martin Fowler

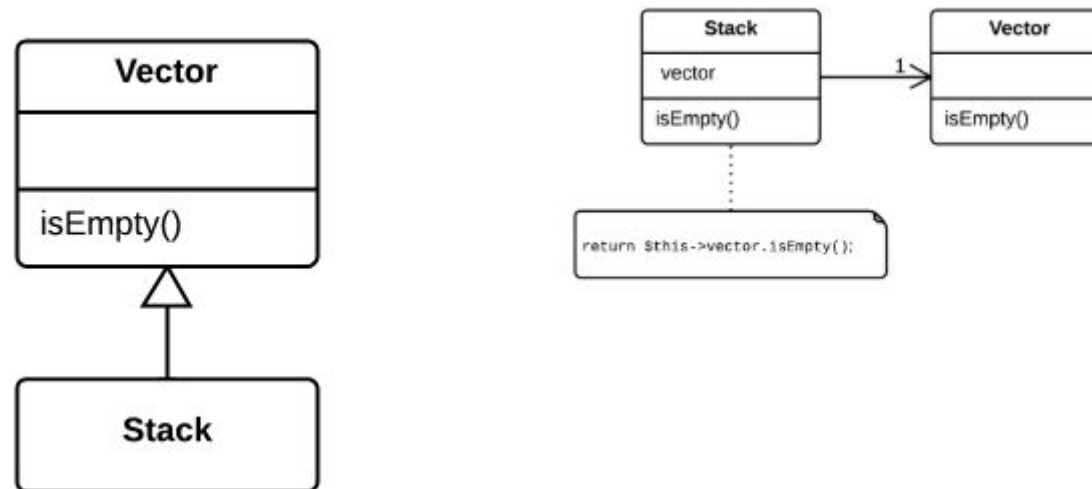
Refused Bequest

- ☑ This rather potent odor results when *subclasses inherit code that they don't want*. In some cases, a subclass may “refuse the bequest” by providing a *do-nothing implementation* of an inherited method.
- ☑ Remedies
 - ☑ Replace Inheritance with Delegation
 - ☑ Extract Superclass



Replace Inheritance with Delegation

- Follow Liskov substitution principle



- If inheritance is appropriate, get rid of unneeded fields and methods in the subclass. Extract all fields and methods needed by the subclass from the parent class, put them in a new subclass, and set both classes to inherit from it by using **Extract Superclass (Liskov substitution principle)**.

Alternative Classes with Different Interfaces

- Two classes perform identical functions but have different method names.
- Try to put the interface of classes in terms of a common denominator
 - **Rename Methods** to make them identical in all alternative classes
 - **Move Method, Add Parameter** and **Parameterize Method** to make the signature and implementation of methods the same
 - If only part of the functionality of the classes is duplicated, try using **Extract Superclass**. In this case, the existing classes will become subclasses.
 - After you have determined which treatment method to use and implemented it, you may be able to delete one of the classes

Change Preventers

- Divergent Change
- Shotgun Surgery

Divergent Change

- You find yourself having to change many unrelated methods when you make changes to a class
- due to poor program structure or "copypasta programming".
- Remedies
 - Split up the behavior of the class via **Extract Class**.
 - If different classes have the same behavior, you may want to combine the classes through inheritance (**Extract Superclass** and **Extract Subclass**).

Shotgun Surgery

- Making any modifications requires that you make many small changes to many different classes.
- Shotgun Surgery resembles Divergent Change but is actually the opposite smell.
 - Divergent Change is when many changes are made to a single class.
 - Shotgun Surgery refers to when a single change is made to multiple classes simultaneously.
- A single responsibility has been split up among a large number of classes. This can happen after overzealous application of **Divergent Change**.

Shotgun Surgery : Remedies

- Use **Move Method** and **Move Field** to move existing class behaviors into a single class. If there is no class appropriate for this, create a new one.
- If moving code to the same class leaves the original classes almost empty, try to get rid of these now-redundant classes via **Inline Class**.



Couplers

- Feature Envy
- Inappropriate Intimacy

Feature Envy (method)

- ☑ A method that seems more interested in some other class than the one it is in.
- ☑ Data and behavior that acts on that data belong together. When a method makes too many calls to other classes to obtain data or functionality, Feature Envy is in the air. This smell may occur after fields are moved to a data class.
- ☑ Remedies:
 - ☑ Move Field
 - ☑ Move Method
 - ☑ Extract Method



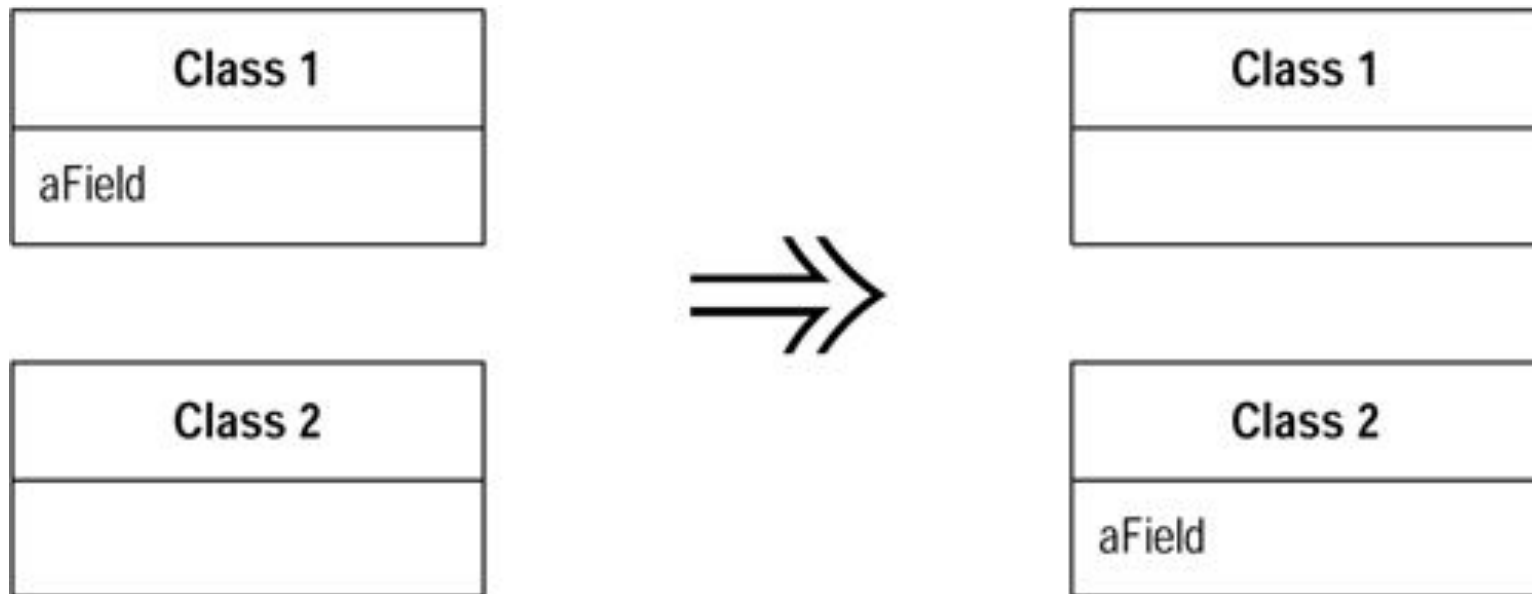
Example

```
Public class CapitalStrategy{
    double capital(Loan loan)
    {
        if (loan.getExpiry() == NO_DATE && loan.getMaturity() != NO_DATE)
            return loan.getCommitmentAmount() * loan.duration() * loan.riskFactor();

        if (loan.getExpiry() != NO_DATE && loan.getMaturity() == NO_DATE)
        {
            if (loan.getUnusedPercentage() != 1.0)
                return loan.getCommitmentAmount() * loan.getUnusedPercentage() *
loan.duration() * loan.riskFactor();
            else
                return (loan.outstandingRiskAmount() * loan.duration() * loan.riskFactor()) +
                    (loan.unusedRiskAmount() * loan.duration() * loan.unusedRiskFactor());
        }

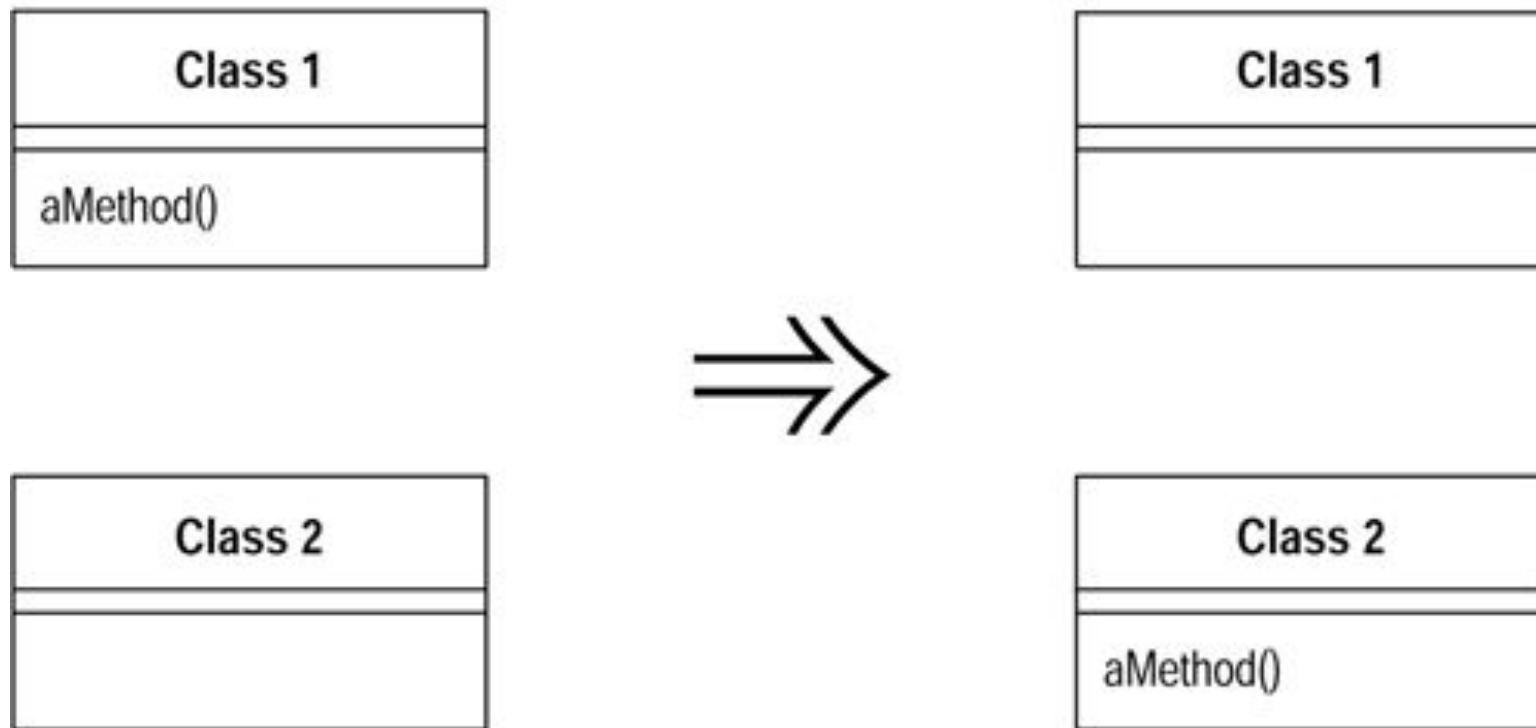
        return 0.0;
    }
}
```


Move Field



Move Method:

If a method clearly should be moved to another place



- If only part of a method accesses the data of another object, use **Extract Method**

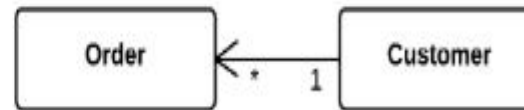
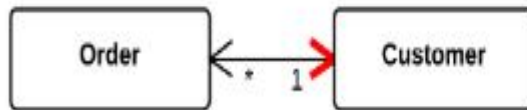
Inappropriate Intimacy (class)

- One class uses the internal fields and methods of another class.
- Remedies
 - **Move Method** and **Move Field** to move parts of one class to the class in which those parts are used. But this works only if the first class truly does not need these parts.
 - Extract Class



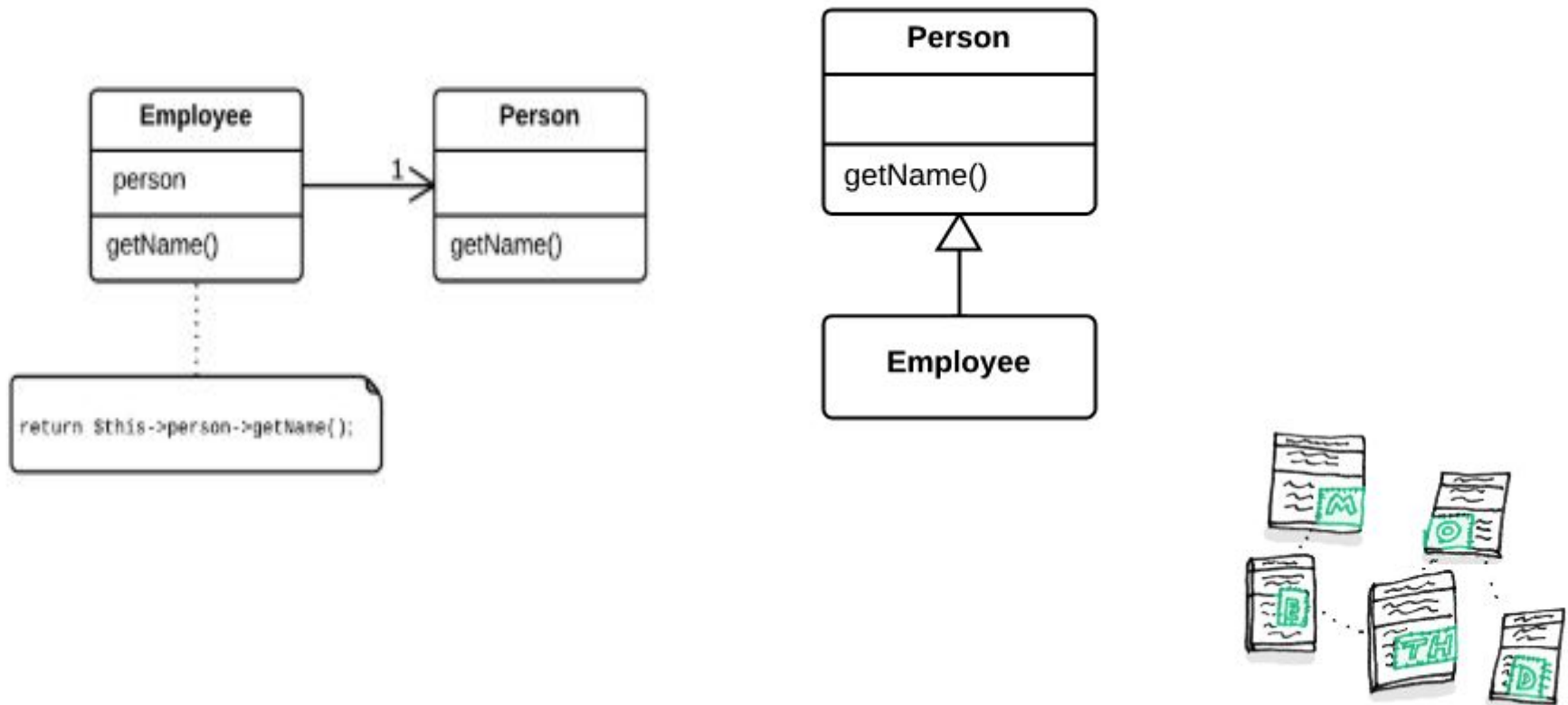
Inappropriate Intimacy (class)

- If the classes are mutually interdependent, you should use **Change Bidirectional Association to Unidirectional**.



Inappropriate Intimacy (class)

- If this “intimacy” is between a subclass and the superclass, consider **Replace Delegation with Inheritance**.



Other Smells

Black Sheep

- ☑ Sometimes a subclass or method *doesn't fit* in so well with its *family*.
- ☑ A subclass that is substantially different in nature than other subclasses in the hierarchy.
- ☑ A method in a class that is noticeably different from other methods in the class.



Example

```
public class StringUtil {  
    public static String pascalCase(String string) {  
        return string.substring(0,1).toUpperCase() + string.substring(1);  
    }  
  
    public static String camelCase(String string) {  
        return string.substring(0,1).toLowerCase() + string.substring(1);  
    }  
  
    public static String numberAndNoun(int number, String noun) {  
        return number + " " + noun + (number != 1 ? "s" : "");  
    }  
  
    public static String extractCommandNameFrom(Map parameterMap) {  
        return ((String[]) parameterMap.get("command"))[0];  
    }  
}
```

Oddball Solution

- ☑ When a problem is solved one way throughout a system and the same problem is solved another way in the same system, one of the solutions is the oddball or inconsistent solution. The presence of this smell usually indicates subtly duplicated code.



Oddball Solution Example

```
string LoadUserProfileAction::process()
{
    //some code here
    return process("ViewAction");
}
string UploadAction::process() {
    //some code here
    return process("ViewAction");
}
string ShowLoginAction::process() {
    //some other code here
    Action* viewAction = actionProcessor().get("ViewAction");
    return viewAction->process();
}
```

Oddball Solution Example

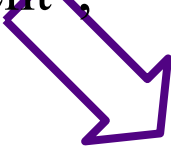
```
private void grow() {  
    Object[] newElements = new Object[elements.length + 10];  
    for (int i = 0; i < size; i++)  
        newElements[i] = elements[i];  
  
    elements = newElements;  
}  
  
private void anotherGrow() {  
    Object[] newElements = new Object[elements.length + INITIAL_CAPACITY];  
    System.arraycopy(elements, 0, newElements, 0, size);  
    elements = newElements;  
}
```

Substitute Algorithm

```
String foundPerson(String[] people){  
  for (int i = 0; i < people.length; i++) {  
    if (people[i].equals ("Don")){  
      return "Don";  
    }  
    if ("John".equals (people[i])){  
      return "John";  
    }  
    if (people[i].equals ("Kent")){  
      return "Kent";  
    }  
  }  
  return ""; }  
}
```

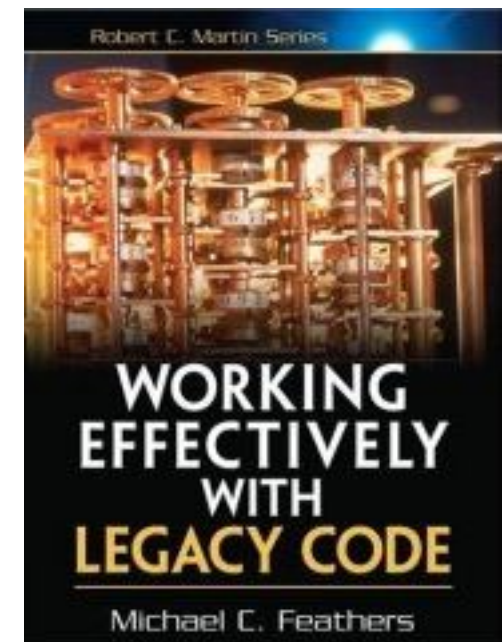
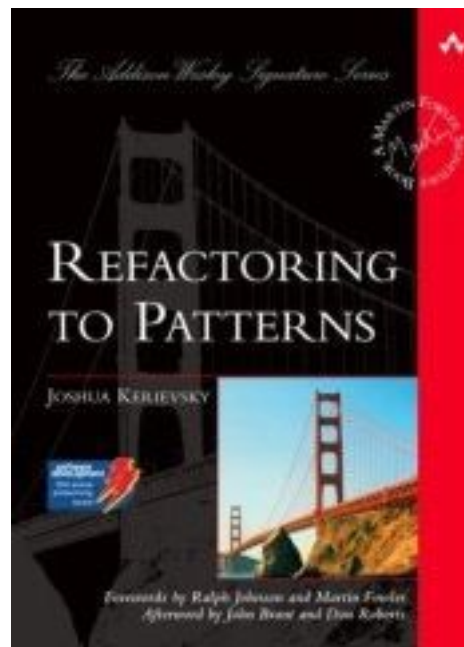
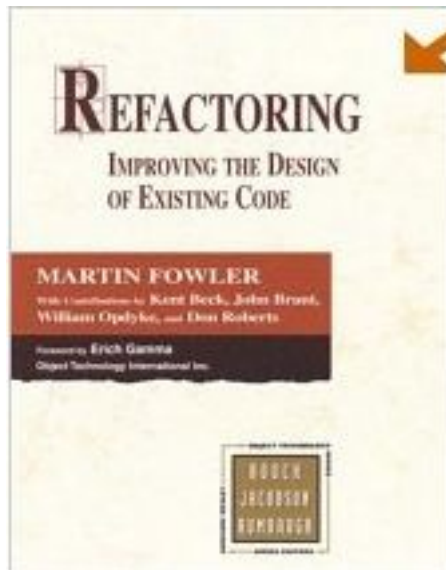
Substitute Algorithm

```
String foundPerson(String[] people){  
for (int i = 0; i < people.length; i++) {  
    if (people[i].equals ("Don")){  
        return "Don";  
    }  
    if ("John".equals (people[i])){  
        return "John";  
    }  
    if (people[i].equals ("Kent")){  
        return "Kent";  
    }  
}  
return "";
```



```
String foundPerson(String[] people){    List candidates =  
Arrays.asList(new String[] {"Don", "John", "Kent"});  
for (String person : people)  
    if (candidates.contains(person))  
        return person;  
return "";
```

Reference Reading



Further Information On Code Smells and Refactoring

- ☑ Wiki Discussion About Code Smells: <http://c2.com/cgi/wiki?CodeSmell>
- ☑ Mika's Smell Taxonomy: <http://www.soberit.hut.fi/mmantyla/BadCodeSmellsTaxonomy.htm>
- ☑ Bill Wake's book, "Refactoring Workbook"
- ☑ Refactoring Catalog Online: <http://www.refactoring.com/catalog/index.html>
- ☑ Refactoring to Patterns Catalog Online: <http://industriallogic.com/xp/refactoring/catalog.html>