

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

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?...

- 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
- To “*Fix broken windows*” - Pragmatic Programmers

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

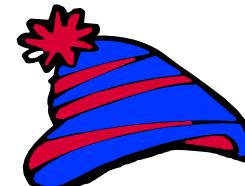
The Two Hats

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



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

Common Code Smells

- Inappropriate Naming
- Comments
- Dead Code
- Duplicated code
- Primitive Obsession
- Large Class
- Lazy Class
- Alternative Class with Different Interface
- Long Method
- Long Parameter List
- Switch Statements
- Speculative Generality
- Oddball Solution
- Feature Envy
- Refused Bequest
- Black Sheep

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)

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);
}
```

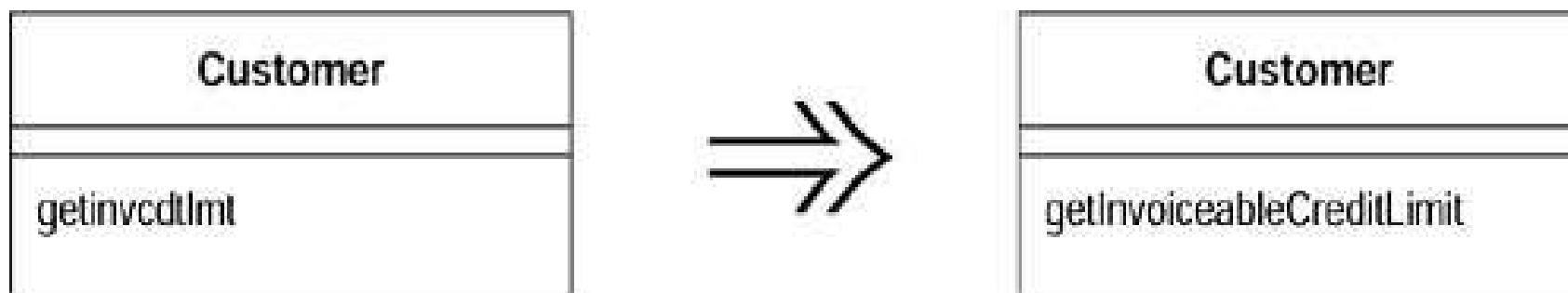
```
private bool ShouldGrow()
{
    return (m_size + 1) > GetCapacity();
}
```

```
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;
}

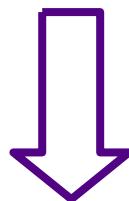
private void StoreElement(string element)
{
    m_elements[m_size++] = element;
}
```

Rename Method



Extract Method

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



```
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

```
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();  
}
```

Long Method

- ☒ A method is long when it is too hard to quickly comprehend.
- ☒ Long methods tend to hide behavior that ought to be shared, which leads to duplicated code in other methods or classes.
- ☒ Good OO code is easiest to understand and maintain with shorter methods with good names
- ☒ Remedies:
 - ☒ Extract Method
 - ☒ Replace Temp with Query
 - ☒ Introduce Parameter Object
 - ☒ Preserve Whole Object
 - ☒ Replace Method with Method Object.
 - ☒ Decompose Conditional



Long Method Makeover (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(">");
}

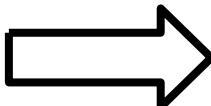
private void writeEndTagTo(StringBuffer result)
{
    result.append("</");
    result.append(name);
    result.append(">");
}
```

```
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);
    }
}
```

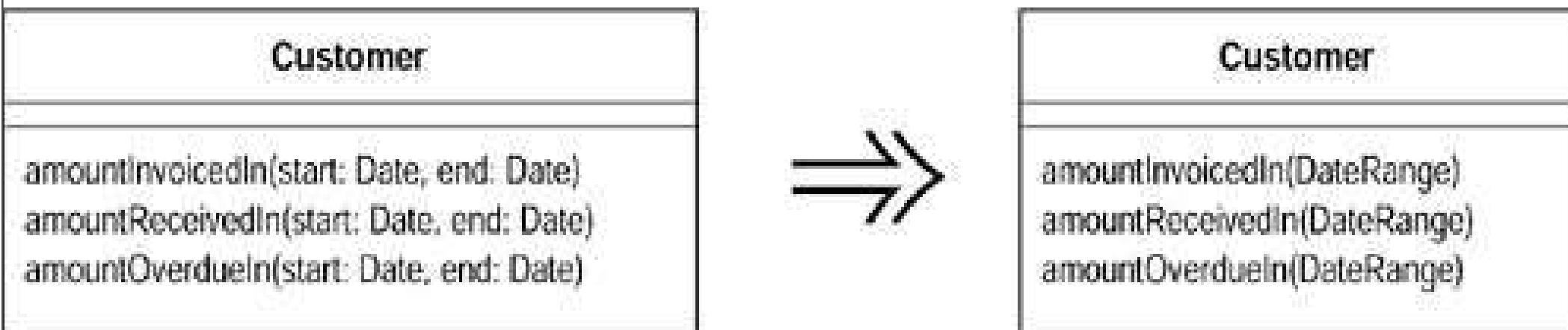
Replace Temp with Query

```
double basePrice = _quanity * _itemPrice;  
if(basePrice > 1000) {  
    return basePrice * 0.95;  
}  
else {  
    return basePrice * 0.98;  
}
```



```
if(getBasePrice() > 1000) {  
    return getBasePrice() * 0.95;  
}  
else {  
    return getBasePrice() * 0.98;  
}  
  
double getBasePrice() {  
    return _quaniyi * itemPrice;  
}
```

Introduce Parameter Object

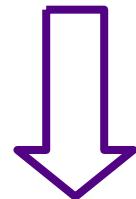


Preserve Whole Object

```
int low = daysTempRange().getLow();
```

```
int high = daysTempRange().getHigh();
```

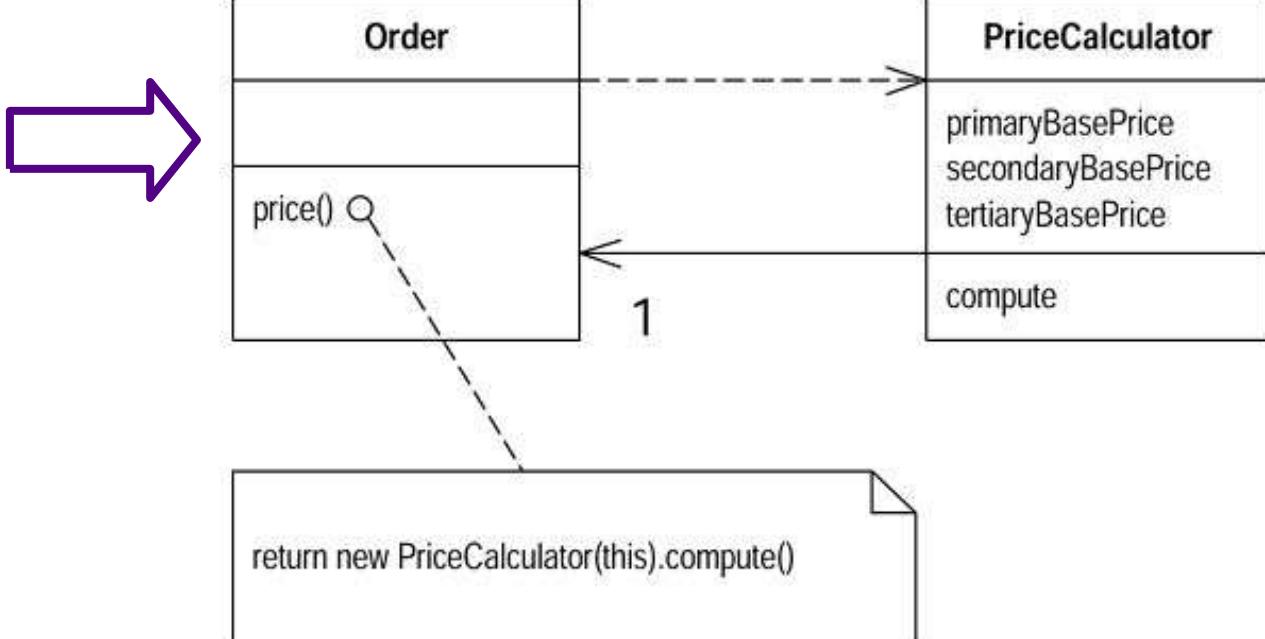
```
withinPlan = plan.withinRange(low, high);
```



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

Replace Method with Method Object

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

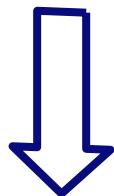


Decompose Conditional

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);
```

Example of Conditional Complexity

```
public bool ProvideCoffee(CoffeeType coffeeType)
{
    if(_change < _CUP_PRICE || !AreCupsSufficient || !IsHotWaterSufficient || !IsCoffeePowderSufficient)
    {
        return false;
    }
    if((coffeeType == CoffeeType.Cream || coffeeType == CoffeeType.CreamAndSugar) && !IsCreamPowderSufficient)
    {
        return false;
    }
    if((coffeeType == CoffeeType.Sugar || coffeeType == CoffeeType.CreamAndSugar) && !IsSugarSufficient)
    {
        return false;
    }

    _cups--;
    _hotWater -= _CUP_HOT_WATER;
    _coffeePowder -= _CUP_COFFEE_POWDER;
    if(coffeeType == CoffeeType.Cream || coffeeType == CoffeeType.CreamAndSugar)
    {
        _creamPowder -= _CUP_CREAM_POWDER;
    }
    if(coffeeType == CoffeeType.Sugar || coffeeType == CoffeeType.CreamAndSugar)
    {
        _sugar -= _CUP_SUGAR;
    }

    ReturnChange();
    return true;
}
```

Long Parameter List

- Methods that take too many parameters produce client code that is awkward and difficult to work with.
- Remedies:
 - Introduce Parameter Object
 - Replace Parameter with Method
 - Preserve Whole Object



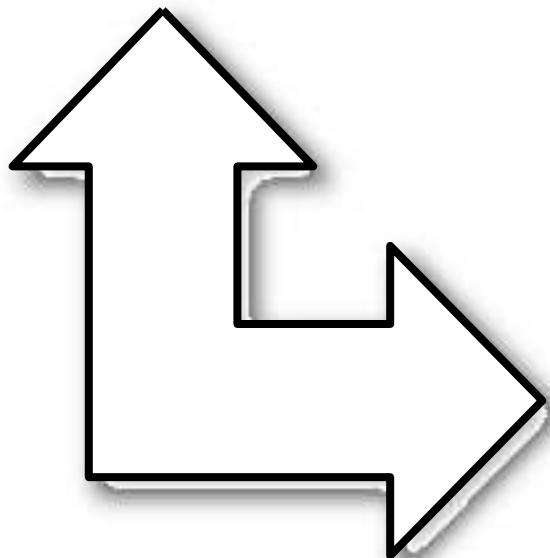
Example

```
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());  
}
```

Introduce Parameter Object

Customer

AmoutInvoicedIn(Date start, Date end)
AmoutReceivedIn(Date start, Date end)
AmoutOverdueIn(Date start, Date end)



Customer

AmoutInvoicedIn(DateRange range)
AmoutReceivedIn(DateRange range)
AmoutOverdueIn(DateRange range)

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

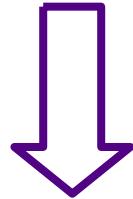
```
public double getPrice() {  
    int basePrice = _quantity * _itemPrice;  
    int discountLevel = getDiscountLevel();  
    double finalPrice = discountedPrice (basePrice);  
    return finalPrice;  
}  
  
private double discountedPrice (int basePrice) {  
    if (getDiscountLevel() == 2) return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

Preserve Whole Object

```
int low = daysTempRange().getLow();
```

```
int high = daysTempRange().getHigh();
```

```
withinPlan = plan.withinRange(low, high);
```



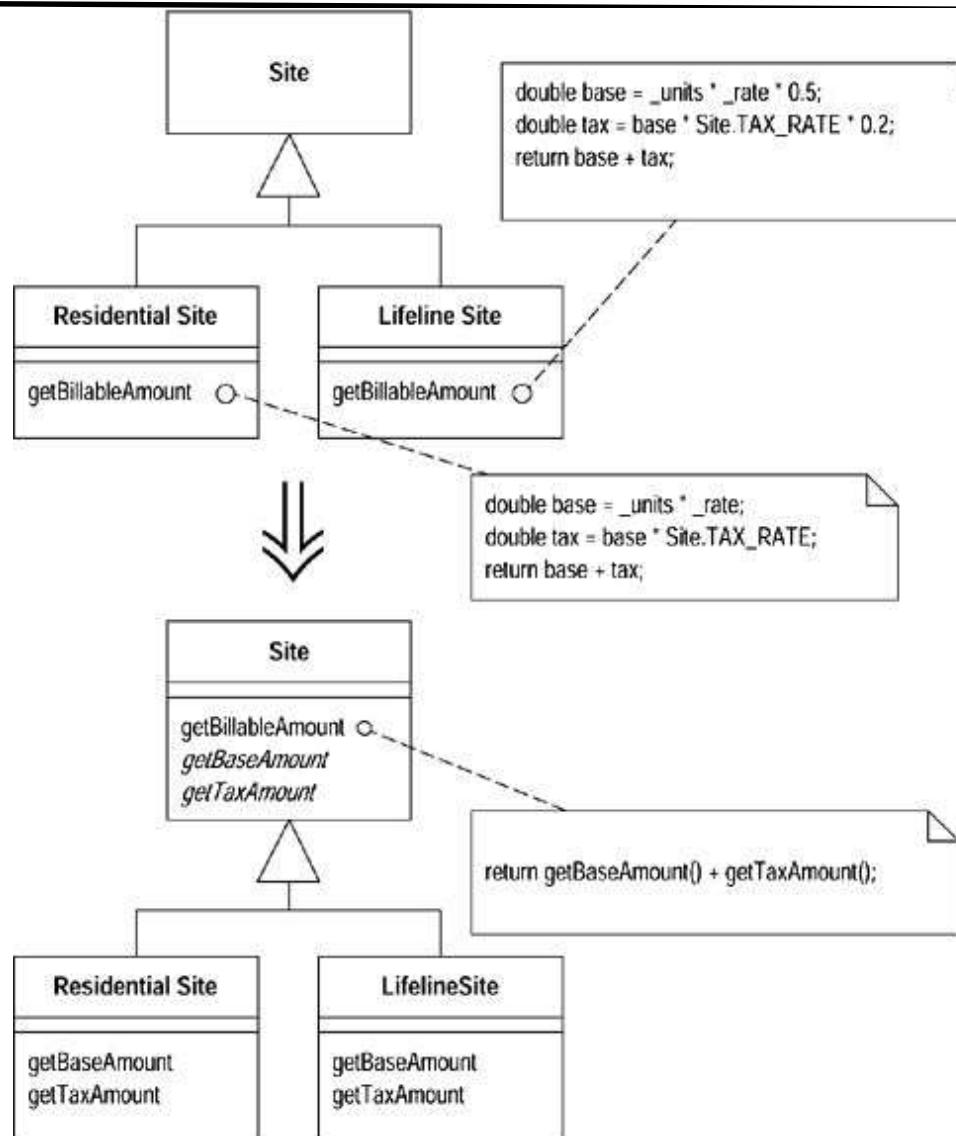
```
withinPlan = plan.withinRange(daysTempRange());
```

Feature Envy

- 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.
-
- Remedies:
 - Move Field
 - Move Method
 - Extract Method



Form Template Method



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 "";  
}
```

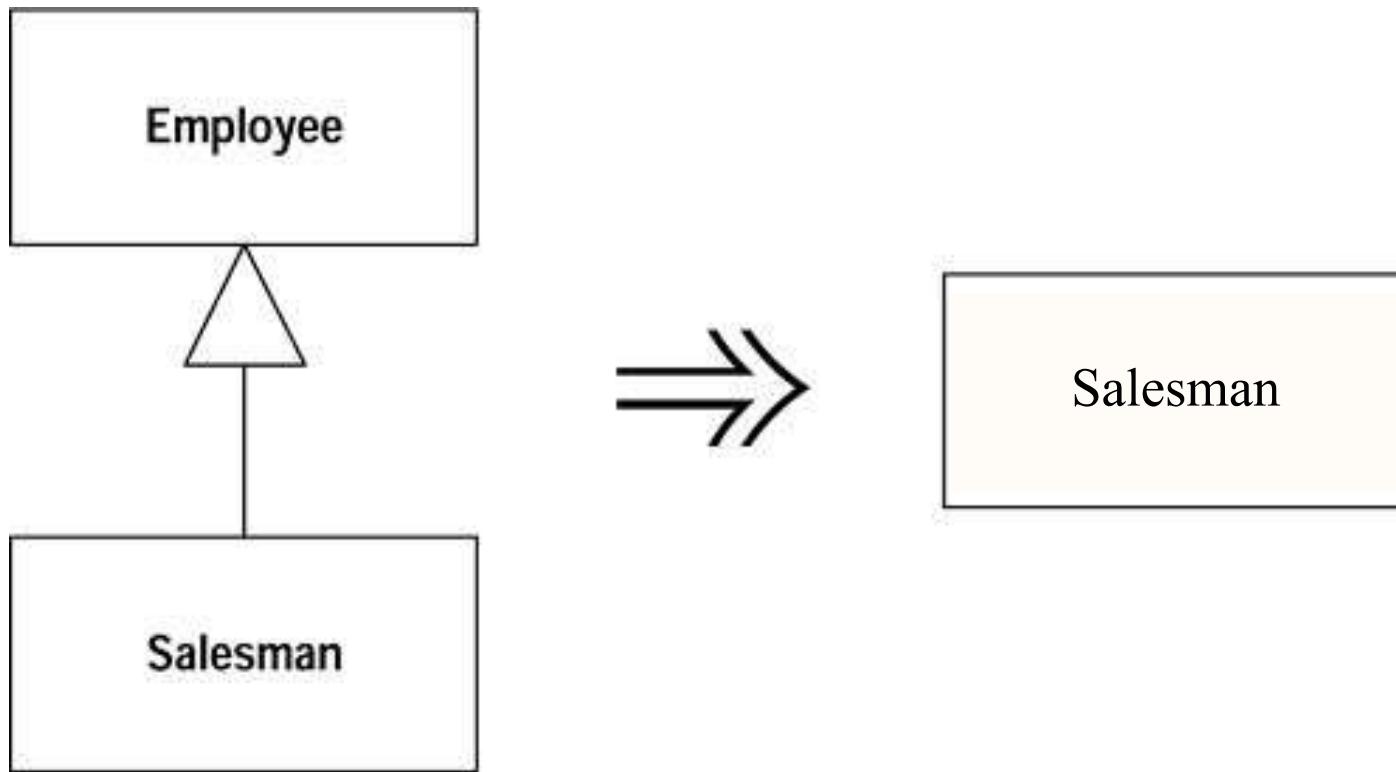


Speculative Generality

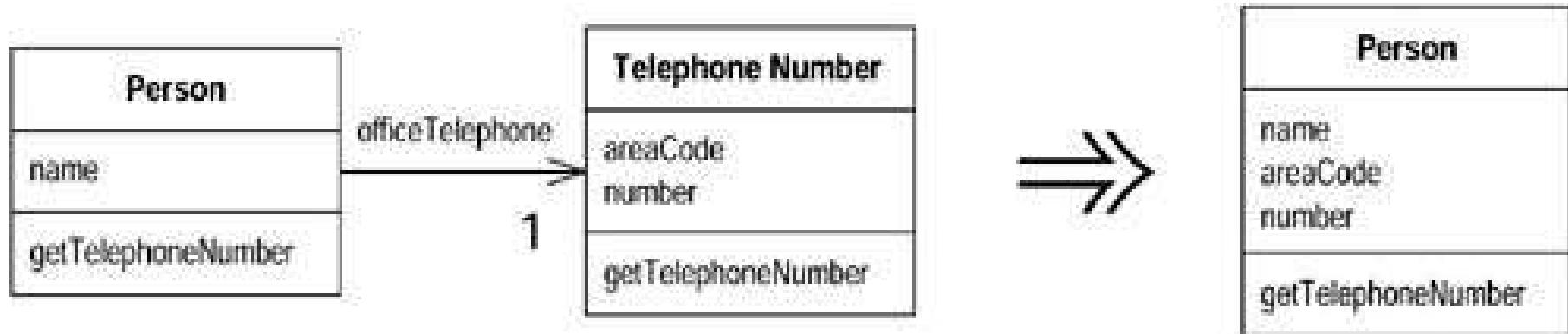
- You get this smell when people say "Oh, I think we will need the ability to do that someday" and thus want all sorts of hooks and special cases to handle things that aren't required.
- This odor exists when you have generic or abstract code that isn't actually needed today. Such code often exists to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



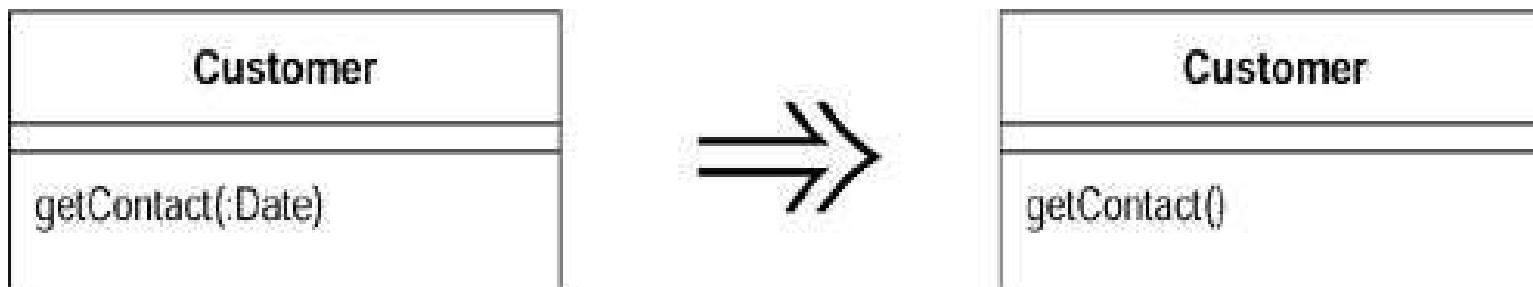
Collapse Hierarchy



Inline Class



Remove Parameter

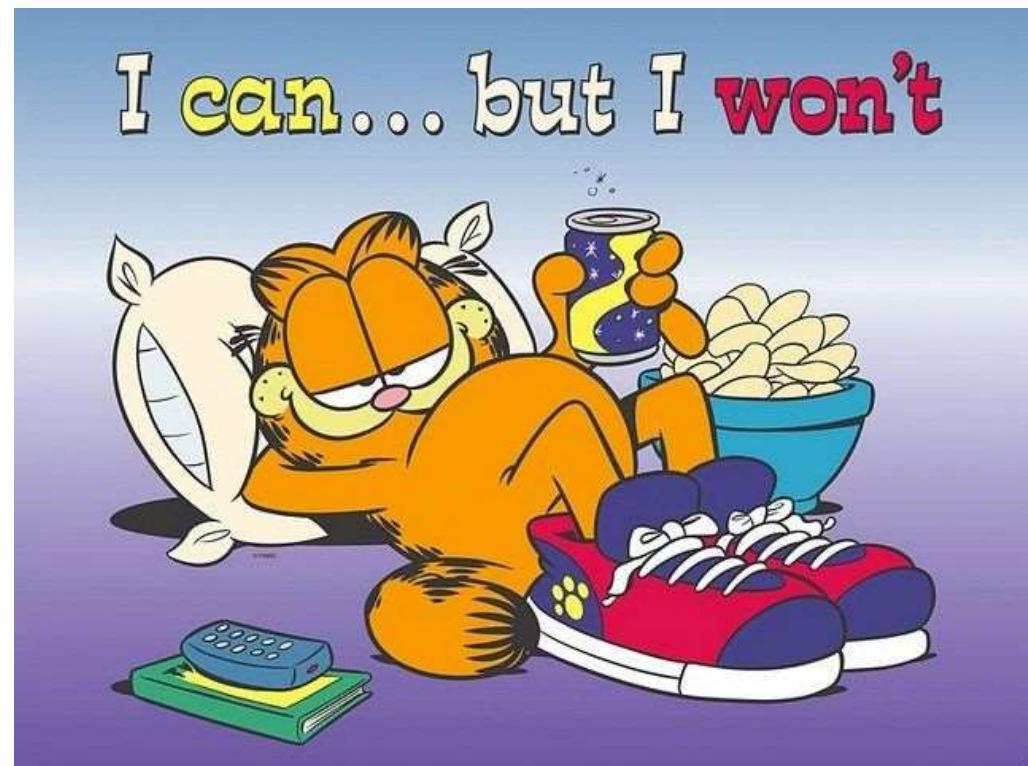


Lazy Class

- 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 Clazz 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

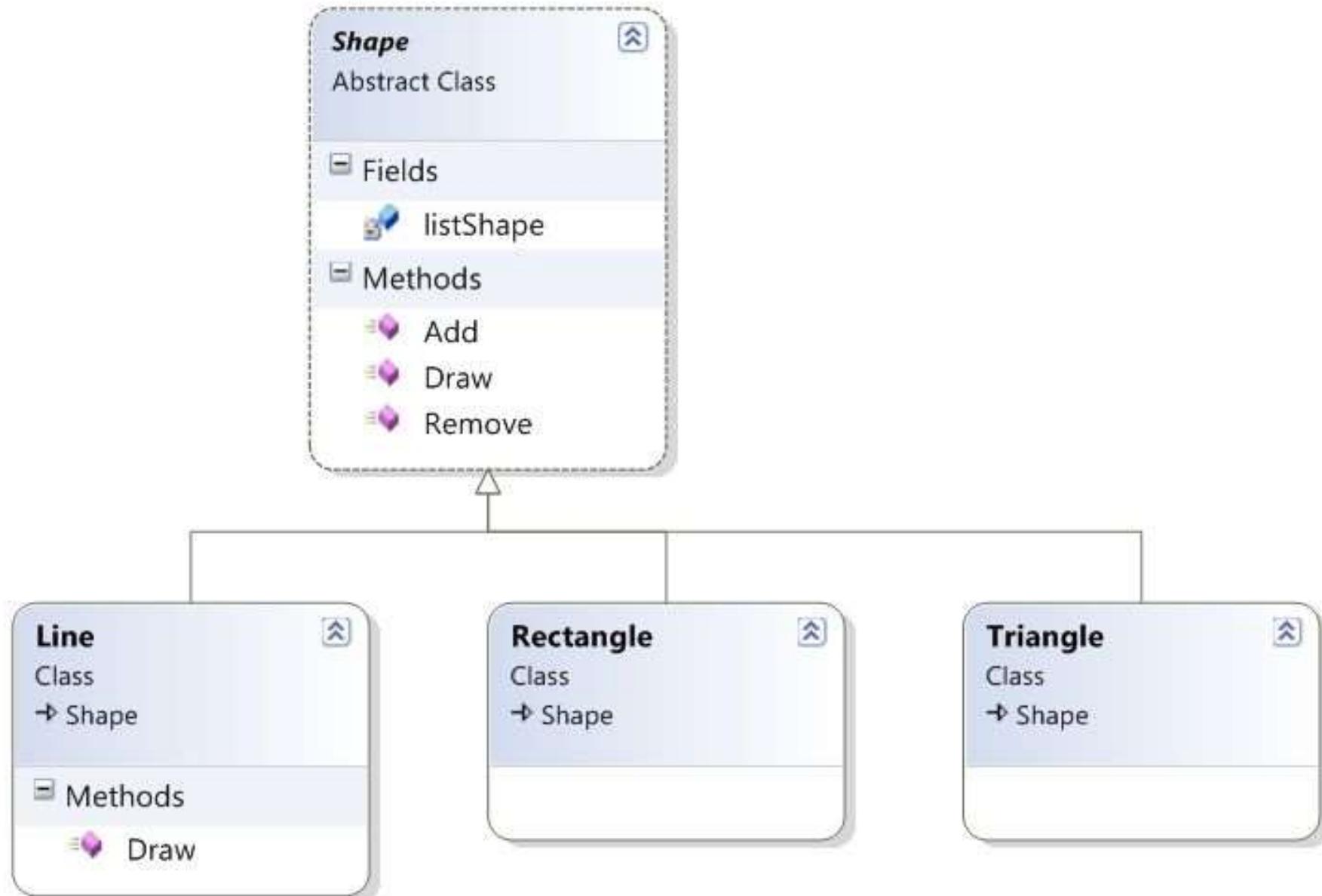
```
public class Letter {  
    private final String content;  
  
    public Letter(String content) {  
        this.content = content;  
    }  
  
    public String getContent() {  
        return content;  
    }  
}
```

Refused Bequest

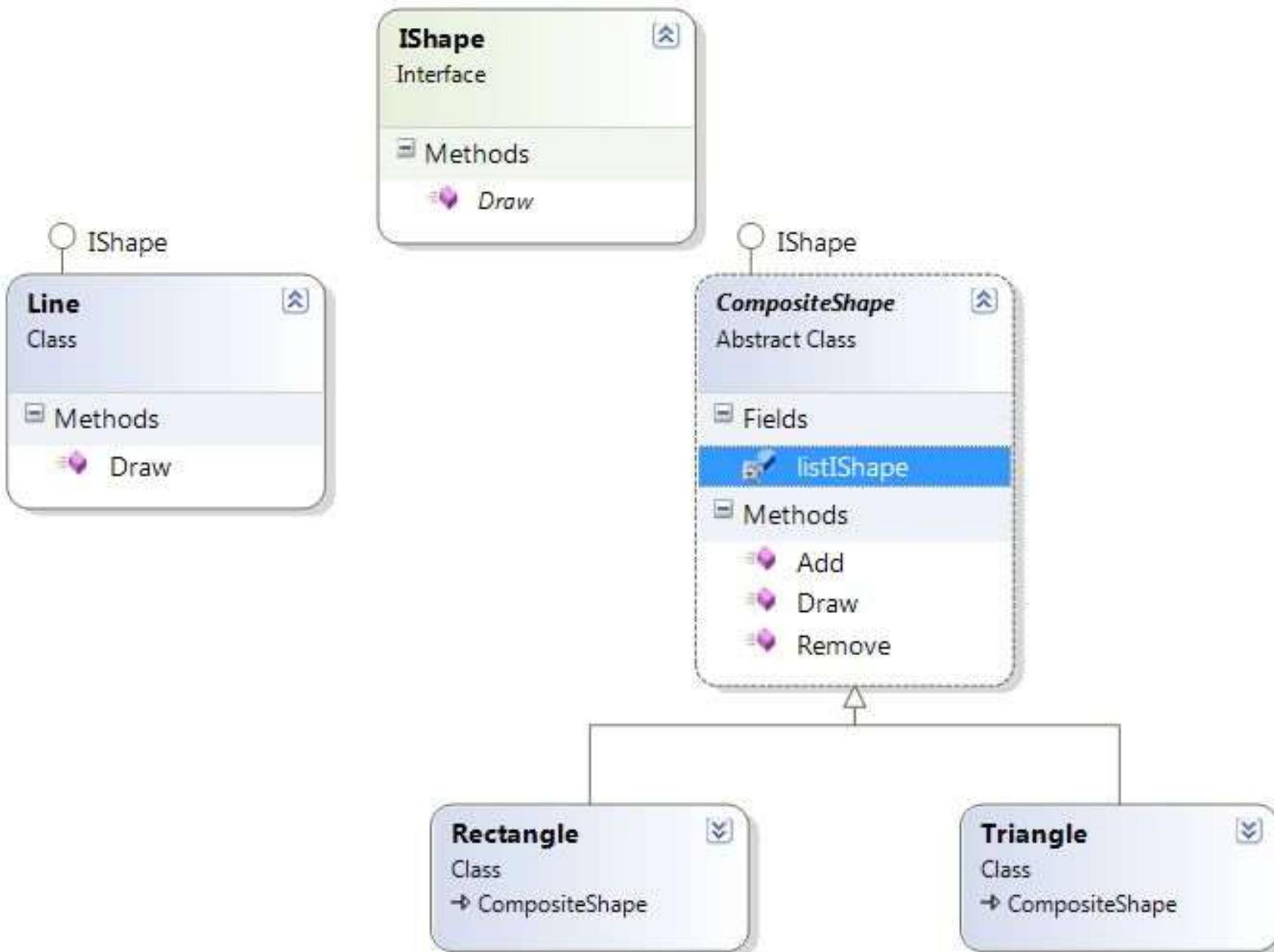
- This rather potent odor results when *subclasses inherit code that they don't want*.
In some cases, a subclass may “refuse the bequest” by providing a *do-nothing implementation* of an inherited method.
- Remedies
 - Push Down Field
 - Push Down Method



Example of Refused Bequest



Refused Bequest Make Over



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];  
    }  
}
```

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.
-
- Remedies
 - Extract Class
 - Replace Data Value with Object
 - Replace Type Code with Class
 - Introduce Parameter Object
 - Replace Array with Object



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);
    }

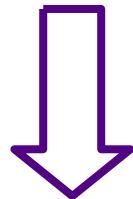
    public void Remove(IShape shape)
    {
        shapeList.Remove(shape);
    }
}
```

Replace Array with Object

```
String[] row = new String[3];  
row [0] = "Liverpool";  
row [1] = "15";
```

Replace Array with Object

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



```
Performance row = new Performance("Liverpool", "15");
```

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

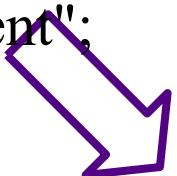
```
private void grow() {  
    Object[] newElements = new Object[elements.length + 10];  
    for (int i = 0; i < size; i++)  
        newElements[i] = elements[i];  
  
    elements = newElements;  
}  
  
private void anotherGrow() {  
    Object[] newElements = new Object[elements.length + INITIAL_CAPACITY];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

Large Class

- Like people, classes suffer when they take on too many responsibilities.
- GOD Objects
- 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

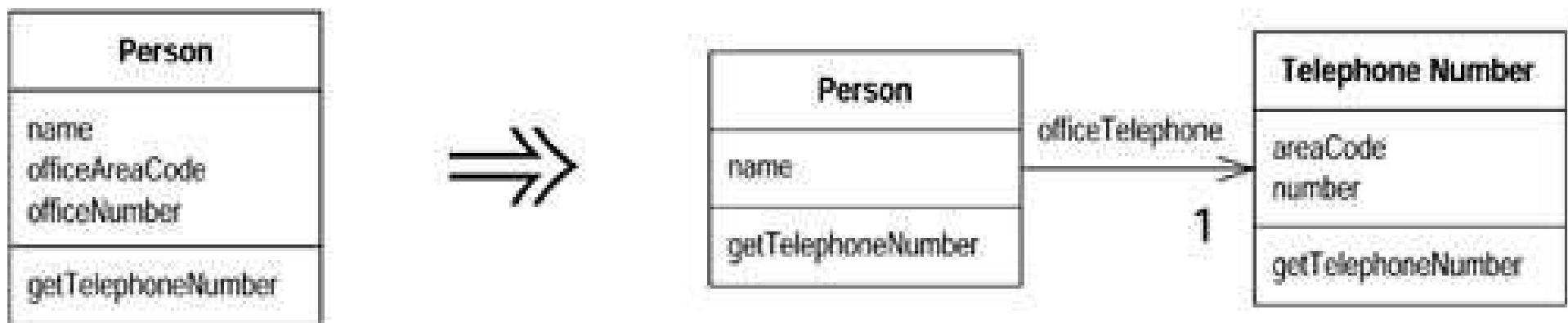


```

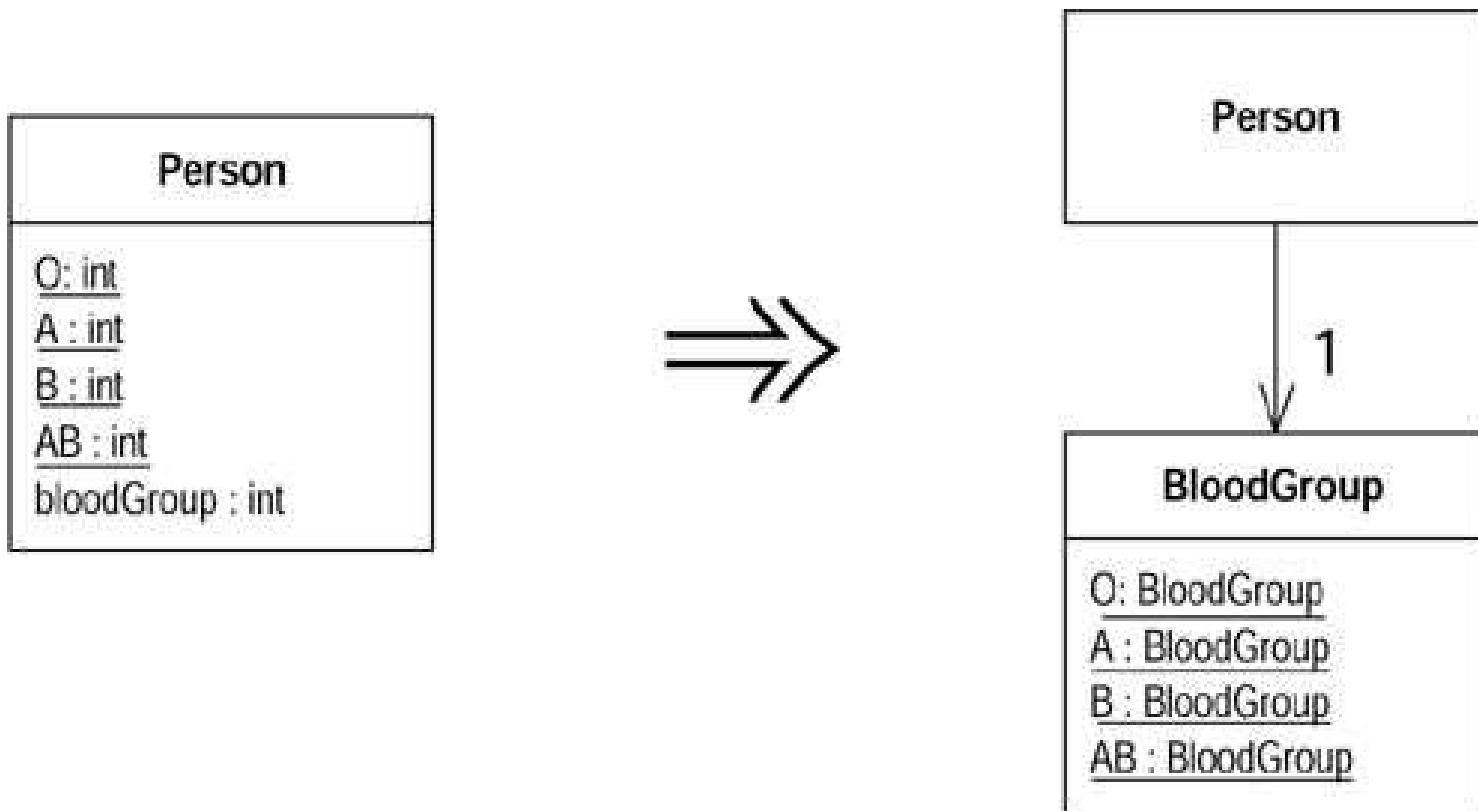
public Vector clearWorkspaceAfterPayrollGeneration(long payrollId, String dName,
    String username, String password, String debug) {
    long returnCode = 201;
    String errorCode = "Uncaught exception";
    PayrollDB db = null;
    Connection conn = null;
    Payroll payroll = null;
    PayrollProject 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);
}

```

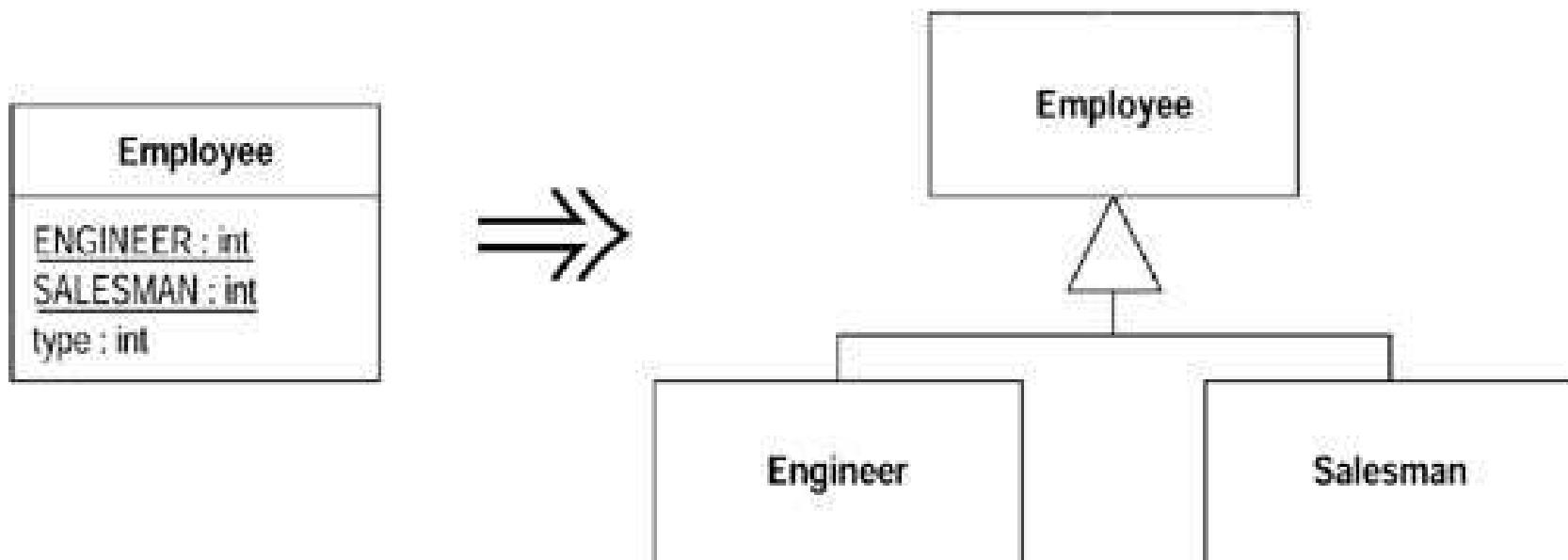
Extract Class



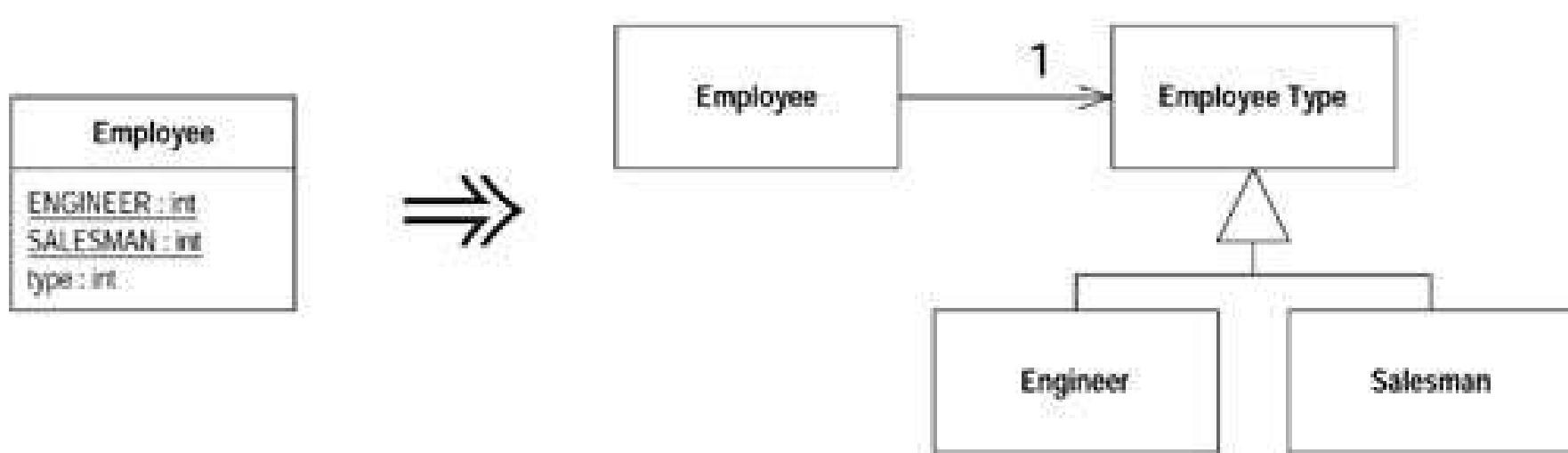
Replace Type Code with Class



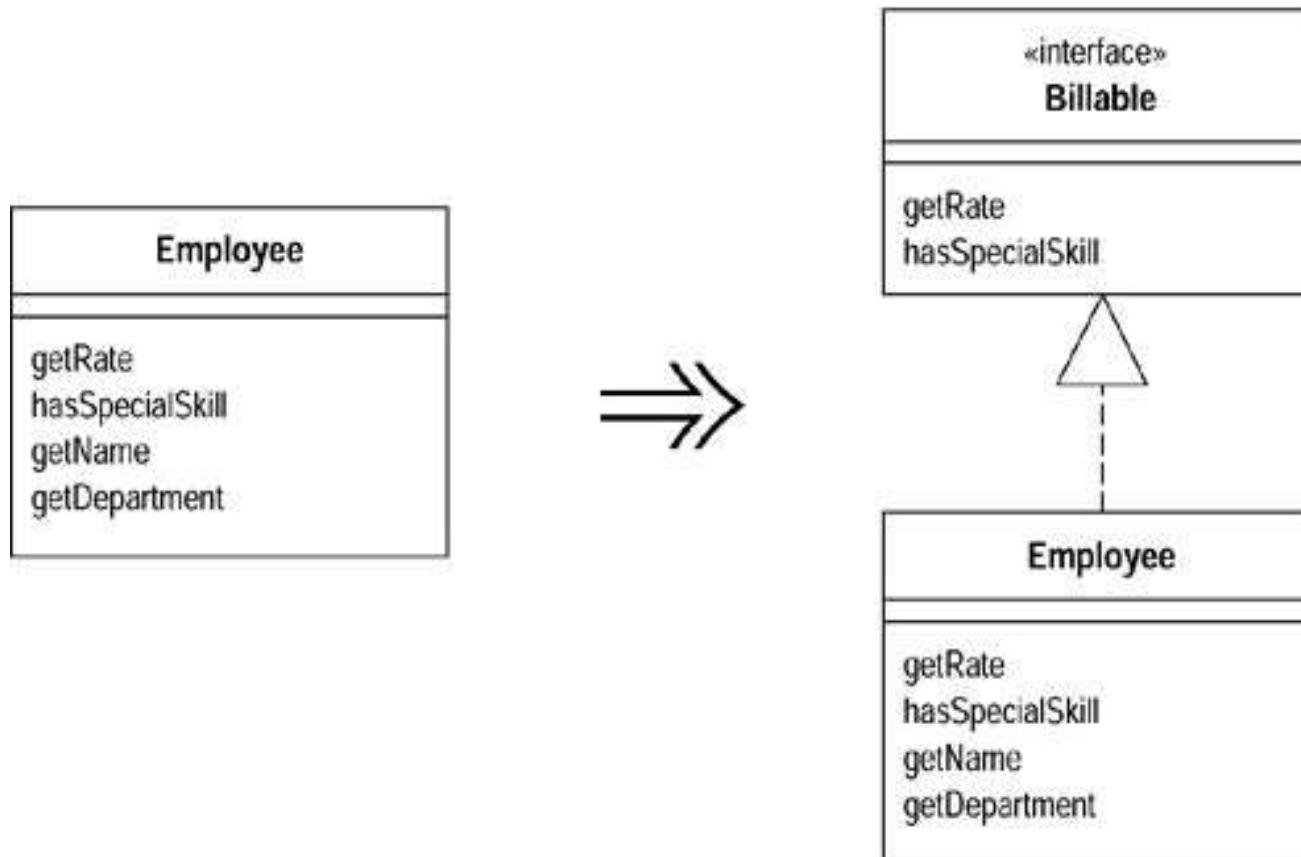
Replace Type Code with Subclasses



Replace Type Code with State/Strategy



Extract (Narrow) Interface



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 Polymorphism
 - Replace Type Code with State / Strategy
 - Replace Parameter with Explicit Methods
 - Introduce Null Object.



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

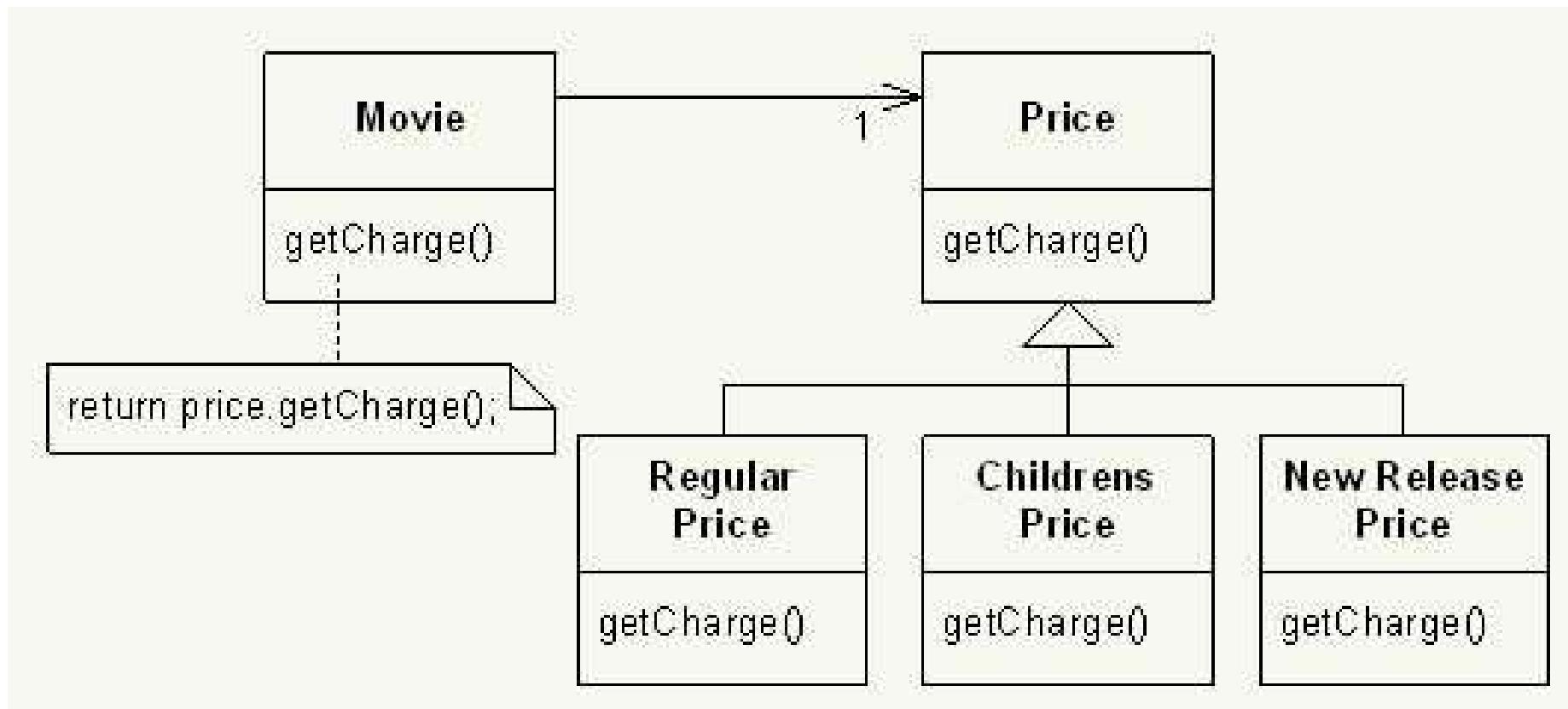
```
switch(strategy){  
    case 1:  
        if(!(name == null))  
            this.name.put("Name", name);  
        break;  
    case 2:  
        if(!(address == null))  
            this.address.put("Address", address);  
        break;  
    case 3:  
        if(!(city==null))  
            this.city.put("City", 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;
    }
}
```

Replace Type Code with Polymorphism

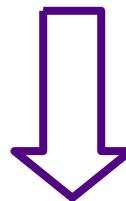


Replace Parameter with Method

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

Replace Parameter with Method

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



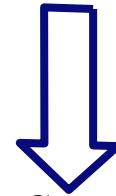
```
void setHeight(int h) {  
    this.height = h;  
}  
  
void setWidth (int w) {  
    this.width = w;  
}
```

Introduce Null Object

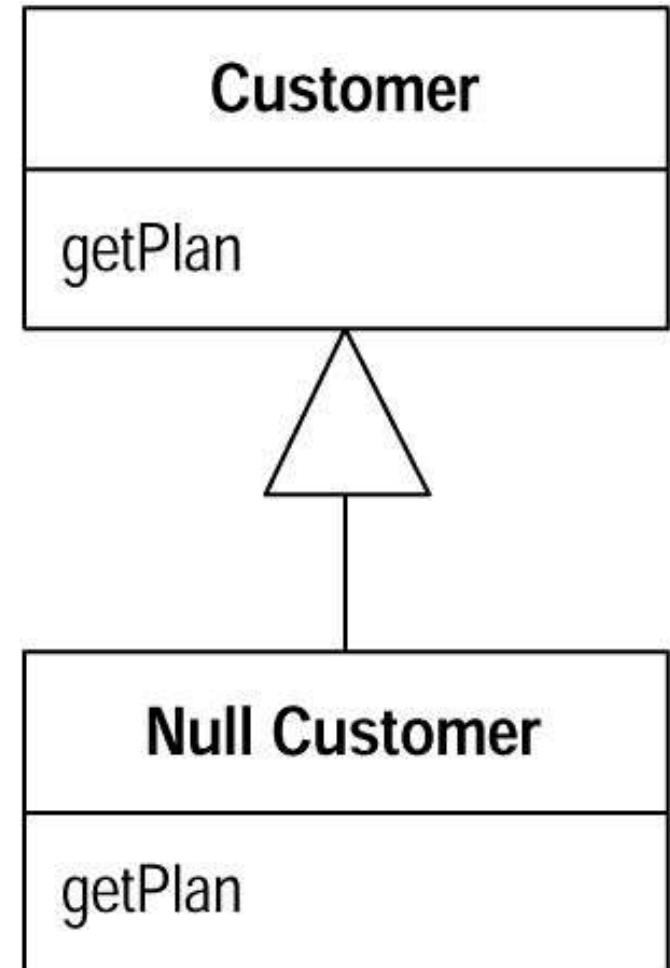
```
// In client class  
Customer customer = site.getCustomer();  
BillingPlan plan;  
if (customer == null)  
    plan = BillingPlan.basic();  
else  
    plan = customer.getPlan();
```

Introduce Null Object

```
// 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