Refactoring Code Smells

Table of Contents

[Refactoring 4](#_Toc135244270)

[Benefits of Refactoring 4](#_Toc135244271)

[Readability 5](#_Toc135244272)

[Refactoring Use Cases 6](#_Toc135244273)

[The Two Hats 6](#_Toc135244274)

[How to Refactor 7](#_Toc135244275)

[Refactoring Advice 7](#_Toc135244276)

[Code Smells 7](#_Toc135244277)

[Inappropriate Naming 8](#_Toc135244278)

[Comments 9](#_Toc135244279)

[Rename Method 10](#_Toc135244280)

[Extract Method 11](#_Toc135244281)

[Introduce Assertions 12](#_Toc135244282)

[Long Method 12](#_Toc135244283)

[Extract Method 13](#_Toc135244284)

[Replace Temp with Query 14](#_Toc135244285)

[Introduce Parameter Object 14](#_Toc135244286)

[Preserve Whole Object 15](#_Toc135244287)

[Replace Method with Method Object 16](#_Toc135244288)

[Decompose Conditional 16](#_Toc135244289)

[Long Parameter List 17](#_Toc135244290)

[Introduce Parameter Object 18](#_Toc135244291)

[Replace Parameter with Method 19](#_Toc135244292)

[Preserve Whole Object 19](#_Toc135244293)

[Feature Envy 20](#_Toc135244294)

[Dead Code 21](#_Toc135244295)

[Duplicated Code 21](#_Toc135244296)

[Pull Up Field 23](#_Toc135244297)

[Form Template Method 23](#_Toc135244298)

[Substitute Algorithm 25](#_Toc135244299)

[Speculative Generality 26](#_Toc135244300)

[Collapse Hierarchy 26](#_Toc135244301)

[Inline Class 27](#_Toc135244302)

[Remove Parameter 27](#_Toc135244303)

[Lazy Class 27](#_Toc135244304)

[Refused Bequest 28](#_Toc135244305)

[Black Sheep 29](#_Toc135244306)

[Primitive Obsession 29](#_Toc135244307)

[Replace Array with Object 30](#_Toc135244308)

[Oddball Solution 31](#_Toc135244309)

[Large Class 31](#_Toc135244310)

[Extract Class 31](#_Toc135244311)

[Replace Type Code with Class 32](#_Toc135244312)

[Replace Type Code with Subclasses 32](#_Toc135244313)

[Replace Type Code with State/Strategy 33](#_Toc135244314)

[Extract Narrow Interface 33](#_Toc135244315)

[Switch Statement 34](#_Toc135244316)

[Replace Type Code with Polymorphism 35](#_Toc135244317)

[Replace Parameter with Method 35](#_Toc135244318)

[Introduce Null Object 36](#_Toc135244319)

## Refactoring

**Refactoring** is a series of small steps, each of which changes the program’s internal structure without changing its external behaviour.

To verify that there are no changes to the external behaviour, we can take several steps:

* Testing
* Using the right tool (IDE)
* Formal code analysis using the chosen tool
* Being very, very careful

The following are some cases where refactoring is valid or invalid:

* ✘ Refactor the code to support logging (changed behaviour)
* ✘ Refactor the code so that it authenticates against LDAP (changed behaviour)
* ✔ Refactor the code to remove duplication
* ✔ Refactor the code to reduce class size
* ✘ Refactor the code to support caching (changed behaviour)

### Benefits of Refactoring

Some of the benefits of refactoring our code include:

* 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
* 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"

### Readability

As an example of improved readability due to refactoring, consider the two code blocks below. One of them has better readability as a result of refactoring.

if (date.before(Summer\_Start) || date.after(Summer\_End)) {  
 charge = quantity \* winterRate + winterServiceCharge;  
} else {  
 charge = quantity \* summerRate;  
}

JAVA

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

JAVA

### Refactoring Use Cases

We should refactor our code in three cases:

* Before adding **new functionality**:
  + Refactor the existing code until you understand it
  + Refactor the design to make it simple to add the new functionality
* To **find bugs**:
  + Refactor the code so that you understand it
* For **code reviews**:
  + There is an immediate effect on code reviews
  + Refactoring allows for higher-level suggestions

### The Two Hats

There are essentially two separate hats when programming.

The first hat is for **adding new functionality**. When wearing this hat, we add new capabilities, write tests and make sure the tests are working.

The second had is for **refactoring**. While wearing this hat, we do not add new functionality and we do not add new tests (but we might change some tests). All we do is restructure the code to remove redundancy.

### How to Refactor

To refactor our code, we have to look for **code smells**. These are things that are not quite right or will cause us pain if we do not fix them soon.

### Refactoring Advice

There are two important pieces of advice to follow when refactoring code:

1. Take baby steps.
2. The Hippocratic Oath – First, do no harm.

## Code Smells

**Code Smells** identify frequently occurring **design problems** in a way that is more specific and targeted than general design guidelines. A code smell is a design that duplicates, complicates, bloats or tightly couples code.

Common code smells include:

* 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
* Train Wreck

## Inappropriate Naming

Names given to **variables** and **methods** should be clear and meaningful.

A variable name should say exactly what it is.

* Bad: private String s
* Good: private String salary

A method name should describe what the method does.

* Bad: public double calc(double s)
* Good: public double calculateFederalTaxes(double salary)

## Comments

**Comments** are a failure to express an idea using code. We should try to make our code self-documenting and intention-revealing, so that the comment is unnecessary.

Consider the following block of code:

public class MyList  
{  
 int INITIAL\_CAPACITY = 10;  
 boolean 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;  
 }  
 }  
}

JAVA

This can be refactored to the following:

void AddToList(String element) {  
 if (m\_readOnly) return;  
 if (ShouldGrow()) Grow();  
 StoreElement(element);  
}  
  
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;  
}  
  
private boolean ShouldGrow() {  
 return (m\_size + 1) > GetCapacity();  
}

JAVA

There are three mechanisms that can help us to refactor the comments code smell:

* Extract Method
* Rename Method
* Introduce Assertion

### Rename Method

**Rename method** simply follows the spirit of fixing inappropriate naming. Method names should be renamed to be descriptive of their functionality.

### Extract Method

**Extract Method** involves shifting a part of one method into another method to make the original method less overwhelming. The existence of the new method also gives us the opportunity to use the name of the new method to convey what is happening inside that block instead of using a comment.

Original:

void PrintOwning(double amount){  
 PrintBanner();  
  
 // print details  
 System.*out*.println("name: "+ name);  
 System.*out*.println("amount: "+ amount);  
}

JAVA

Refactored:

void PrintOwning(double amount){  
 PrintBanner();  
 PrintDetails(amount);  
}  
  
void PrintDetails(double amount){  
 System.*out*.println("name: "+ name);  
 System.*out*.println("amount: "+ amount);  
}

JAVA

### Introduce Assertions

If we require specific conditions to be met, using comments to note these is a poor solution. Instead, we can use an assertion. This conveys the same message but also allows us to provide an explicit check for the conditions.

Original:

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

JAVA

Refactored:

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();  
}

JAVA

## Long Method

A **long method** is one which is difficult to quickly comprehend. Since you have no clue what is actually happening in the method, you end up duplicating the code in other places. We should make methods short and easy to understand.

### Extract Method

The same **extract method** process can used to solve the long method issue. Just extract parts of the code into separate methods so that it doesn’t become overwhelming or distract you from what the main purpose of the method is.

Original:

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();  
}

JAVA

Refactored:

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

JAVA

### Replace Temp with Query

Instead of creating a temporary variable that holds some calculated value, we can use a query to retrieve the value from another method. This hides the details of how the value is calculated, this reducing the amount of information presented to us at once.

Original:

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

JAVA

Refactored:

if(getBasePrice() > 1000) return getBasePrice() \* 0.95;  
else return getBasePrice() \* 0.98;

JAVA

### Introduce Parameter Object

Instead of having multiple parameters being passed to a method, it may be cleaner to combine the parameters into a single object and pass that instead.

Original:

public class Customer {  
 public void amountInvoicedIn(Date start, Date end);  
}

JAVA

Refactored:

public class Customer {  
 public void amountInvoicedIn(DateRange dateRange);  
}

JAVA

### Preserve Whole Object

We may face situations where we are inadvertently taking values from one object, storing them in temporary variables only to then go pass those variables as parameters to another method. Watch out for these and consider passing the object directly instead.

Original:

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

JAVA

Refactored:

withinPlan = plan.withinRange(daysTempRange());

JAVA

### Replace Method with Method Object

Particularly complicated methods may benefit from being shifted to a separate class dedicated to handling the complicacy.

Original:

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

JAVA

Refactored:

double price() {  
 return priceCalculator.compute(this);  
}

JAVA

### Decompose Conditional

Conditional Statements (if-then-else statements) can become overly complicated if we directly use all of the possible conditions they depend on. Instead, we should extract the condition checking to dedicated methods.

Original:

if (date.before (SUMMER\_START) || date.after(SUMMER\_END))  
 charge = quantity \* \_winterRate + \_winterServiceCharge;  
else charge = quantity \* \_summerRate;

JAVA

Refactored:

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

JAVA

## Long Parameter List

Having a large number of parameters in a single method makes the method awkward to work with.

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()  
 );  
}

JAVA

### Introduce Parameter Object

Instead of having multiple parameters being passed to a method, it may be cleaner to combine the parameters into a single object and pass that instead.

Original:

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

JAVA

Refactored:

AmoutInvoicedIn(DateRange range);  
AmoutRecivedIn(DateRange range);  
AmoutOverdueIn(DateRange range);

JAVA

### Replace Parameter with Method

Similar to replacing temporary variables with queries, we can replace parameters with methods that retrieve the value for the parameter.

Original:

int discountLevel;  
if (\_quantity > 100)  
 discountLevel = 2;  
else  
 discountLevel = 1;  
double finalPrice = discountedPrice (basePrice, discountLevel);

JAVA

Refactored:

int discountLevel = getDiscountLevel();  
double finalPrice = discountedPrice (basePrice, discountLevel);

JAVA

### Preserve Whole Object

This is the same as before.

Original:

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

JAVA

Refactored:

withinPlan = plan.withinRange(daysTempRange());

JAVA

## Feature Envy

Data and behavior that acts on the data belong together. When a method makes too many calls to other classes to get the data from other classes, it is called **feature envy**.

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;  
 }  
}

JAVA

There are three ways to resolve this: Move Field, Move Method and Extract Method.

Move Field and Move Methods involves simply moving a field (a variable) or a method to the envious class. We can do this if it is determined that the field or method is better attributed to the envious class.

Extract Method works the same as before. For example, we can simply place all of the computation occurring above into a new method in the Loan class.

## Dead Code

Over time, a codebase will start to accumulate dead code. This is code that isn’t being used anywhere in the system. Most likely, it was used at one point but changes made elsewhere resulted in it not being used anymore. Dead code in the codebase increases its complexity unnecessarily and should be removed.

## Duplicated Code

Code duplication is the most common code smell and can be of many forms. There could be obvious code duplication, which is literally copying and pasting code, but also more subtle duplications such as those caused by parallel inheritance hierarchies or similar algorithms.

There are several levels of duplication:

* **Literal Duplication** – The same code is used in multiple places, such as a for loop.
* **Semantic Duplication** – The same code, but written differently. For example, pushing elements to a stack one by one vs using a for loop to push the elements to a stack.

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

JAVA

VS

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

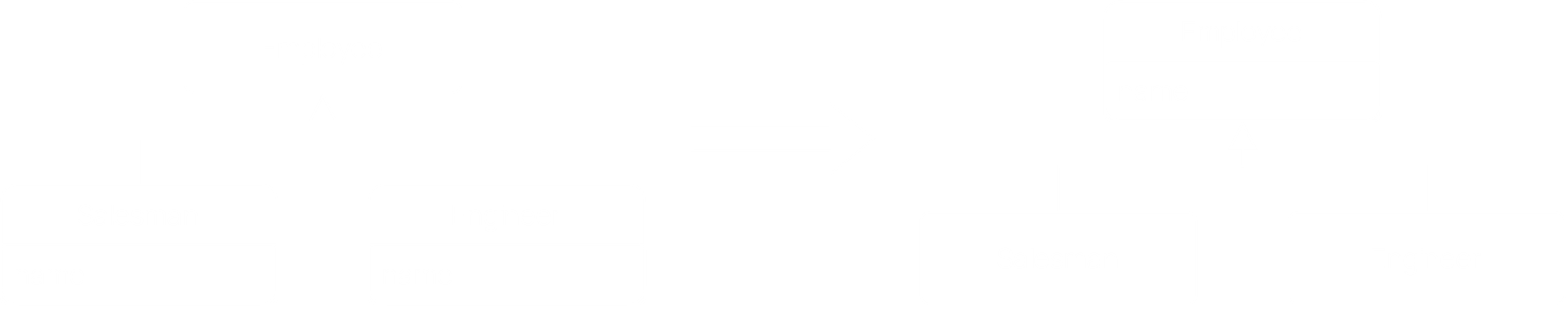
JAVA

* **Data Duplication** – The same data is declared in multiple places, such as different classes.
* **Conceptual Duplication** – Different methods are used to achieve the same outcome, such as using two different sorting algorithms in two places.
* **Logical Steps Duplication** – The same set of steps is repeated in different places, such as the same validation sets occurring repeatedly at different points in our code.

Remedies for code duplication include using **extract method** (discussed previously), **pull up field**, **form template method** and **substitute algorithm**.

### Pull Up Field

Common data can be ‘pulled up’ to a common ancestor to reduce duplication.



### Form Template Method

A similar method in two classes can be places in a common ancestor. If there are any differences in the data being used inside the method or if there are variations in the steps, we can make those parts variable. In the example below, the bill calculation mechanism is the same for two classes, but the base value and the tax value is calculated differently. As such, we can have two abstract methods in the parent class to retrieve these values and make the bill calculation method common to both classes.

Original:

public abstract class Site {  
 double \_units, \_rate, tax\_rate;  
}  
  
public class ResidentialSite extends Site {  
 public double getBillableAmount() {  
 double base = \_units \* \_rate;  
 double tax = base \* tax\_rate;  
 return base + tax;  
 }  
}  
  
public class LifelineSite extends Site {  
 public double getBillableAmount() {  
 double base = \_units \* \_rate \* 0.5;  
 double tax = base \* tax\_rate \* 0.2;  
 return base + tax;  
 }  
}

JAVA

Refactored:

public abstract class Site {  
 double \_units, \_rate, tax\_rate;  
 public abstract double getBaseAmount();  
 public abstract double getTaxAmount();  
 public double getBillableAmount() {  
 return getBaseAmount() + getTaxAmount();  
 }  
}  
  
public class ResidentialSite extends Site {  
 public double getBaseAmount() {  
 return \_units \* \_rate;  
 }  
  
 public double getTaxAmount() {  
 return getBaseAmount() \* tax\_rate;  
 }  
}

public class LifelineSite extends Site {  
 public double getBaseAmount() {  
 return \_units \* \_rate \* 0.5;  
 }  
  
 public double getTaxAmount() {  
 return getBaseAmount() \* tax\_rate \* 0.2;  
 }  
}

JAVA

### Substitute Algorithm

Repetitive steps can often be substituted with iterations. In the example below, instead of using a separate if block for each person, we can write a for loop that checks against a list of people.

Original:

public 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 "";  
}

JAVA

Refactored:

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

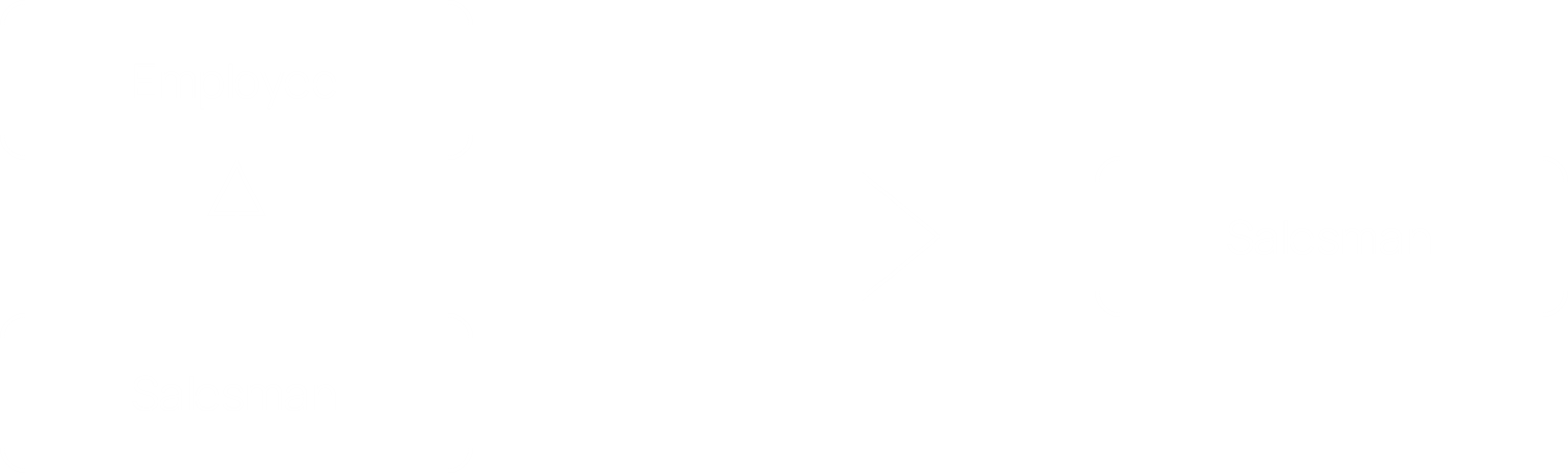
JAVA

## Speculative Generality

**Speculative Generality** is the abstraction of code under the assumption that the abstraction will be required in the future.

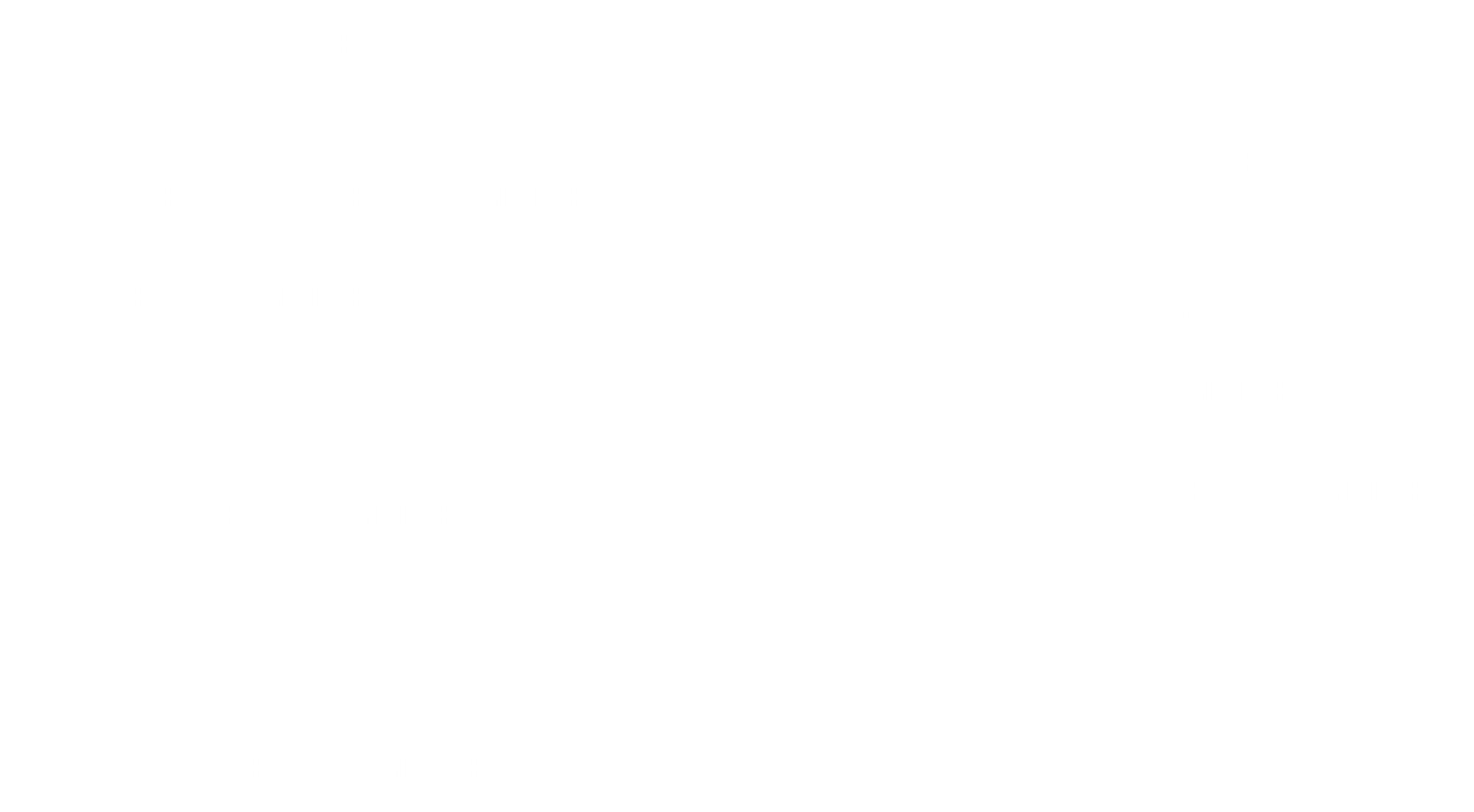
### Collapse Hierarchy

Remove unnecessary hierarchies, such as ones which only have a single child.



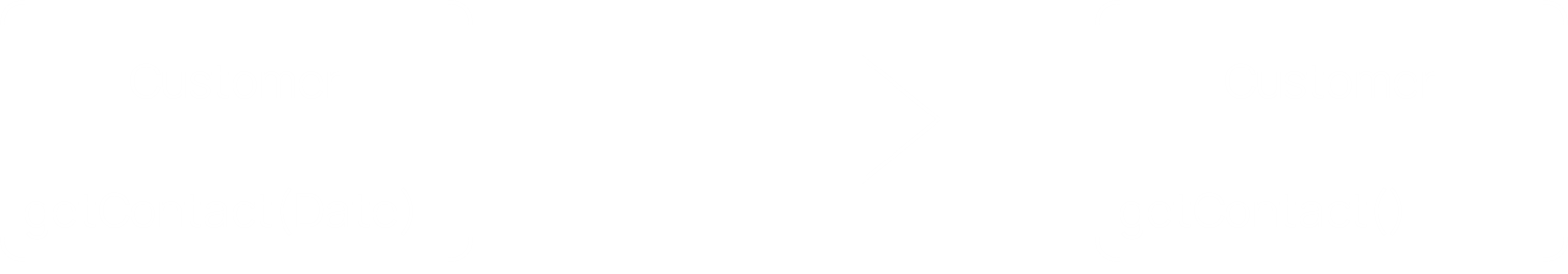
### Inline Class

If data or methods have been separated out into a different class, check that this separation is actually necessary. If the base class is simple enough, it may not be necessary at all.



### Remove Parameter

Remove unused parameters.



## Lazy Class

A **lazy class** is a class that is doing so little that the existence of a separate class just to do this amount of work cannot be justified.

Example:

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

JAVA

The issue can be resolved using **inline class** and **collapse hierarchy**, both of which have been previously discussed.

## Refused Bequest

Sometimes, subclasses inherit code that does not make sense for them. As a workaround, the implementations for the code are changed to ‘do nothing’. This suggests that the unwanted code has been placed too high up in the hierarchy and should instead be placed lower down, perhaps in one of the siblings of this subclass.

This issue can be resolved using **push down field** and **push down method**. Neither of these requires further discussion since they literally mean pushing down a field or a method to a child class.

## Black Sheep

Sometimes, a class or method just does not fit into the family. A subclass may be too different from its siblings or a method might be too different from the other methods in the class. In these cases, we should consider moving the subclass or method elsewhere.

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];  
 }**}

JAVA

## Primitive Obsession

**Primitive obsession** is when we use primitives (strings, integers, etc.) for high-level operations instead of using classes.

Original:

if (string.indexOf("substring") != -1)

JAVA

Refactored:

if (string.contains("substring"))

JAVA

Remedies for this code smell include **extract class**, **replace data value with object**, **replace type code with class**, **introduce parameter object** and **replace array with object**, only the last of which is being discussed further here.

### Replace Array with Object

Different related properties should be stored in an object. However, this is sometimes achieved by storing them as an array.

Original:

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

JAVA

Refactored:

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

JAVA

## Oddball Solution

An **oddball solution** is when a problem is solved in one way everywhere in the codebase but in a different way in one or two places. The different solution usually indicates subtly duplicated code.

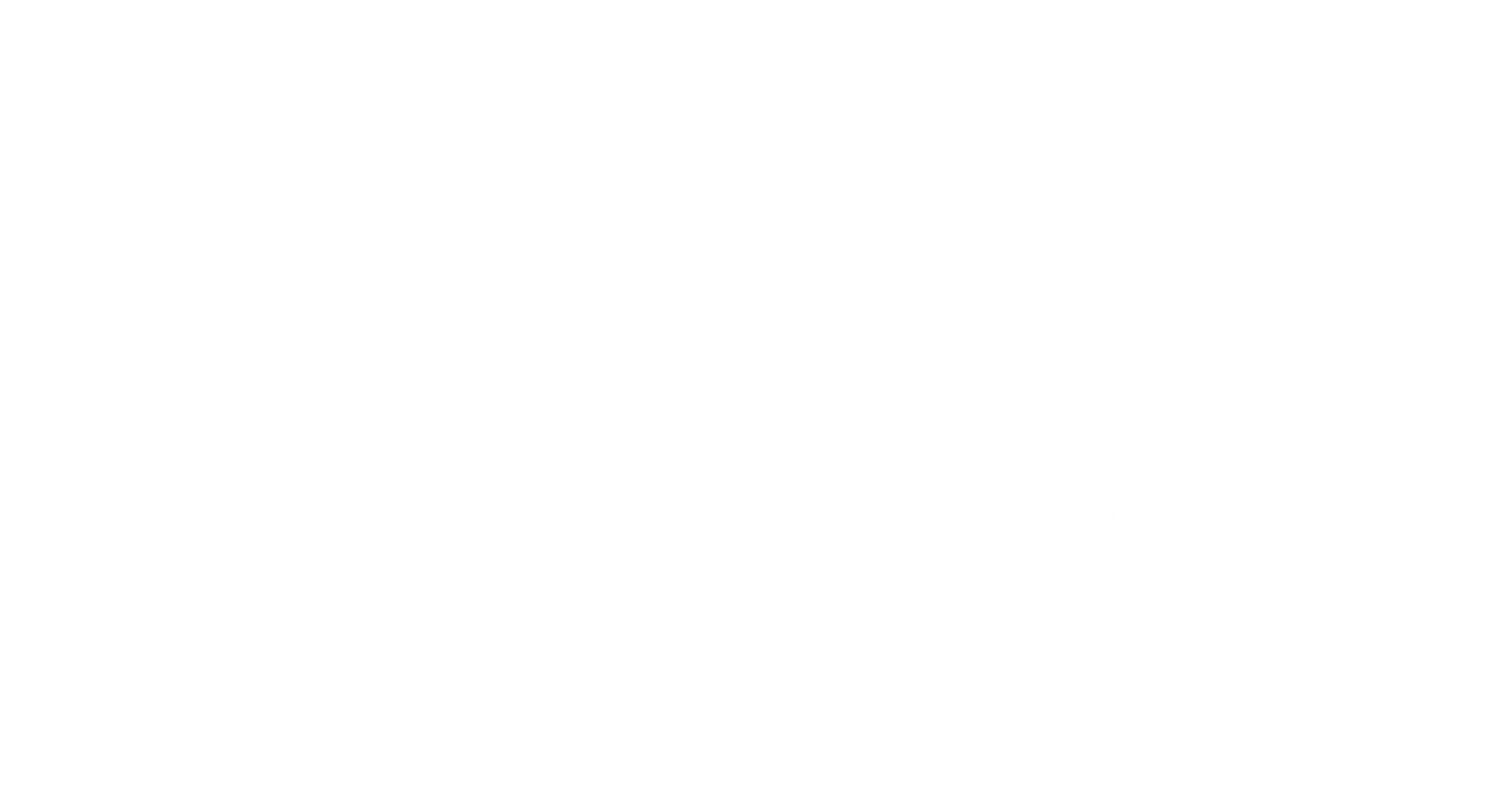
The remedy to this smell is to use **substitute algorithm**.

## Large Class

A large class is one which does too much. The presence of too many instance variables is usually indicative of the class doing too much.

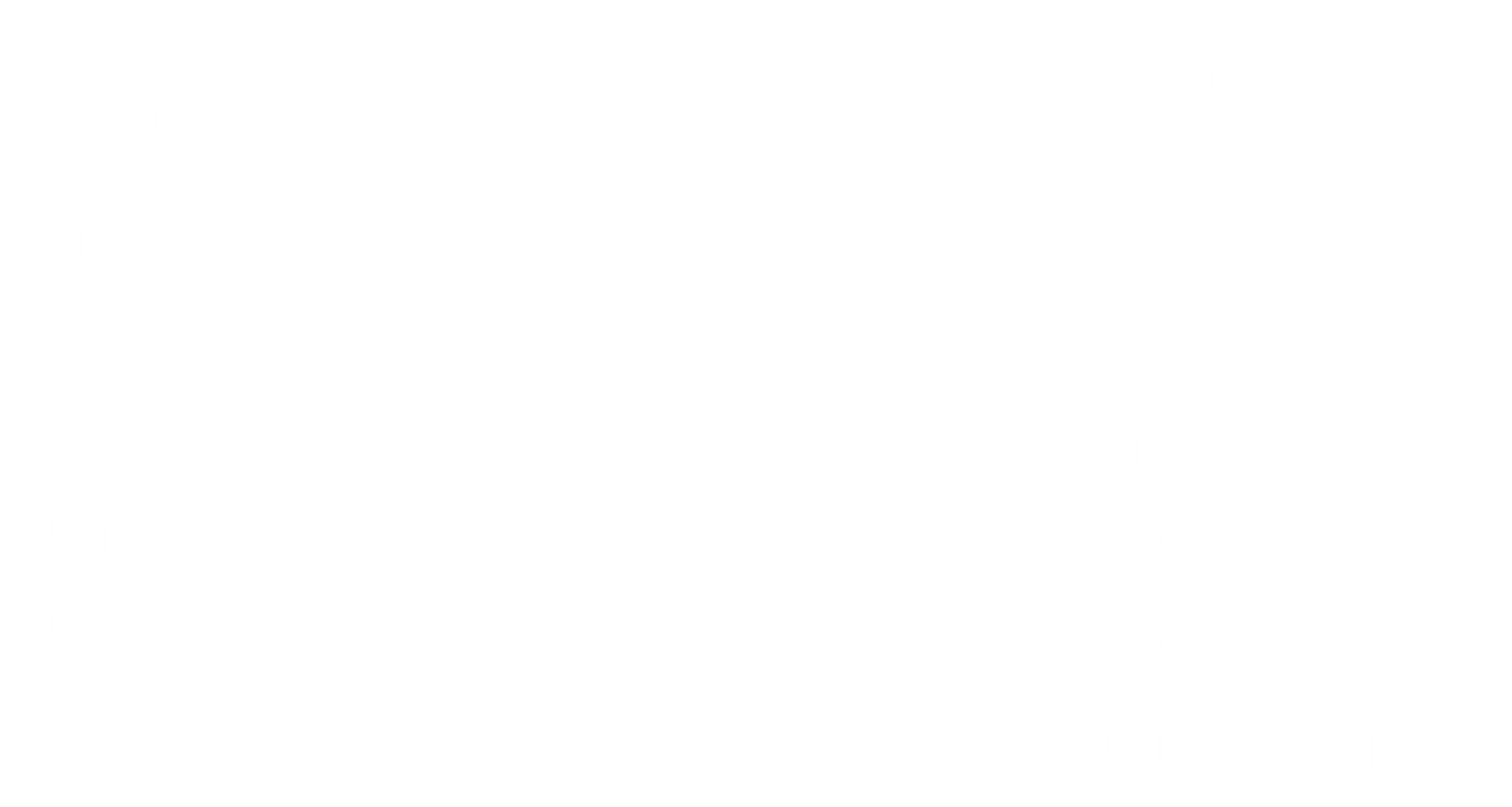
### Extract Class

If part of the class can be identified to have its own separate responsibilities, it should be extracted to its own class.



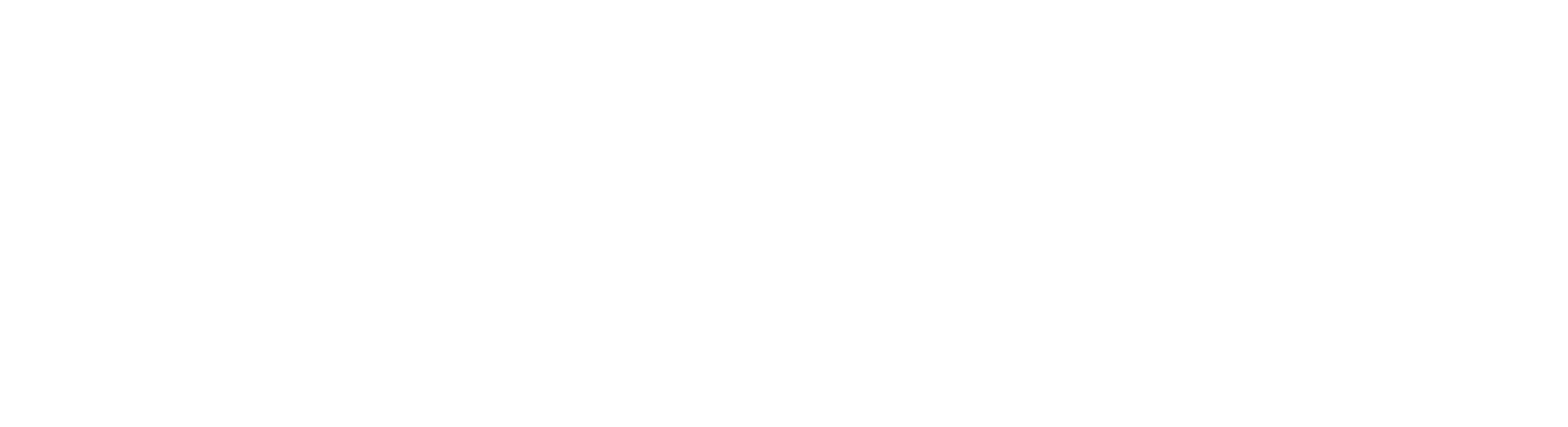
### Replace Type Code with Class

If there are multiple variables representing different ‘types’ of the same thing, we should place the types in their own class.



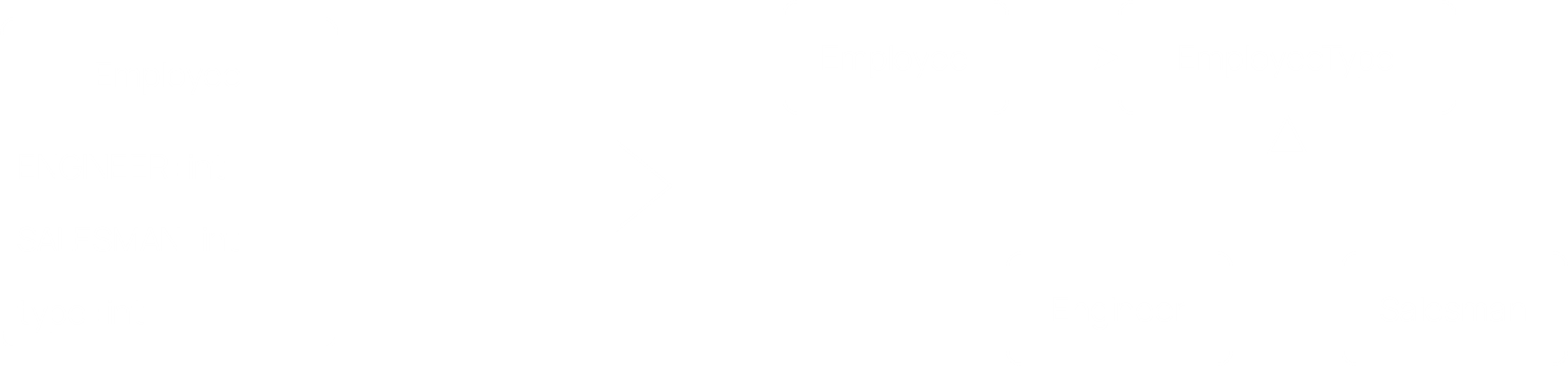
### Replace Type Code with Subclasses

Depending on the situation, it may make sense to replace type code with subclasses instead of extracting them to a separate class.



### Replace Type Code with State/Strategy

We could also replace type code by using a state variable for the different types. This is somewhat similar to replacing the type code with a class.



### Extract Narrow Interface

Extracting methods of the class to an interface can help if we need to identify which parts of the code are usable by clients.



## Switch Statement

This smell exists when a switch statement or a group of if-else statements is duplicated across the codebase or excessively used. This indicates that proper object-oriented principles have not been followed.

Example:

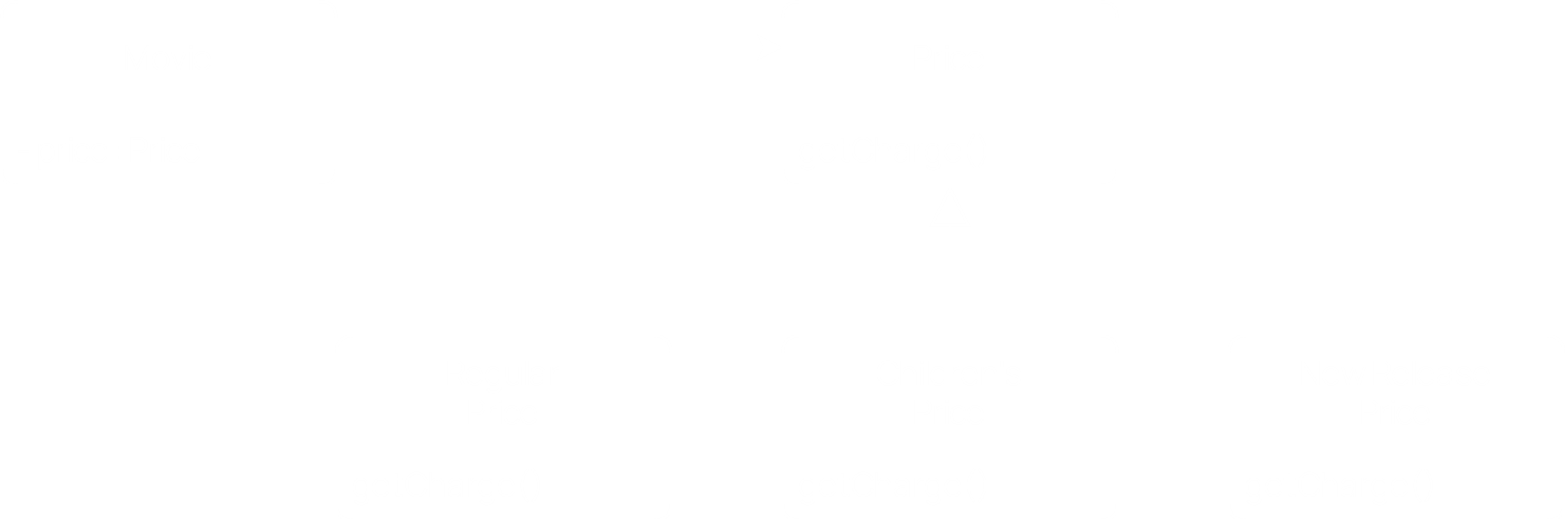
switch(strategy) {  
 case 1:  
 if (!(name == null))  
 this.name.put("Name", name);  
 break;  
 case 2:  
 if (!(address == null))  
 this.address.put("Address", address);  
 break;  
 case 3:  
 if (!(city == null))  
 this.city.put("City", city);  
 break;  
 case 4:  
 if (!(age == 0))  
 this.age.put("Age", 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;  
}

JAVA

Remedies to this issue include **replace type code with polymorphism**, **replace type code with state/strategy**, **replace parameter with explicit methods** and **introduce null object**.

### Replace Type Code with Polymorphism

The different types represented by the different cases of the switch statement could be turned into their own hierarchy which will allow us to retrieve the required results using a parameter indicated the class to be used.



### Replace Parameter with Method

The use of a switch statement may also indicate that the method is handling the work of multiple methods. In this case, we should split the work into their own methods and use the one required.

Original:

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

JAVA

Refactored:

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

JAVA

### Introduce Null Object

A switch case could also be used to handle the case where the value of an object is null. To make our code cleaner, we could actually have an explicit object that is a **null object**.

Original:

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

JAVA

Refactored:

// in client class  
Customer customer = site.getCustomer();  
BillingPlan plan = customer.getPlan();  
  
// in NullCustomer

public BillingPlan getPlan() {  
 return BillingPlan.*basic*();  
}

JAVA