CSE 470
Software Engineering
Refactoring and Code Smells

Imran Zahid
Lecturer
Computer Science and Engineering, BRAC University

Recap

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

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
- We follow the guiding principles
  - First do no harm
  - Baby steps

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

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
- Train Wreck

Code Smell - 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

Code Smell - Long Method

- Remedies:
  - Extract Method
  - Replace Temp with Query
  - Introduce Parameter Object
  - Preserve Whole 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();
}
```

Example Html tag:
<name> Jannet Jhonson </name>

Long Method: Extract Method

- Extracting methods is a solution to the long method code smell when parts of a method perform distinct, self-contained tasks.
- Breaking the method into smaller, well-named methods makes the code more readable, maintainable, testable, and reusable.

Long Method: 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: Replace Temp with Query

- When a temporary variable is being used to store the result of a computation that could instead be encapsulated in a dedicated method.
- This approach simplifies the method, enhances readability, and avoids clutter caused by excessive temporary variables.

Long Method: Replace Temp with Query

```
Method1(){
      double basePrice = _quanity * _itemPrice;
      if(basePrice > 1000)
            return basePrice * 0.95;
      else
         return basePrice*0.98;
}

Method2(){
      double basePrice = _quanity * _itemPrice;
      return basePrice + 100;
}
```

What if the basePrice calculation equation changes?

-> We would need to change two lines in the code

Long Method: Replace Temp with Query

```
Method1(){
     double basePrice = _quanity * _itemPrice;
     if(basePrice > 1000)
          return basePrice * 0.95;
     else
          return basePrice*0.98;
}

Method2(){
     double basePrice = _quanity * _itemPrice;
     return basePrice + 100;
}
```

⇨

```
Method1(){
     if(getBasePrice() > 1000)
          return getBasePrice() * 0.95;
     else
          return getBasePrice()*0.98;
}

Method2(){
     return getBasePrice() + 100;
}


double getBasePrice()  {
 return _quanitiy * itemPrice;
}
```

Long Method: Introduce Parameter Object

- When a method takes many parameters, particularly when these parameters are logically related.
- This refactoring simplifies the method signature, improves readability, and encapsulates the parameters into a cohesive object.

Long Method: Introduce Parameter Object

```
int MethodTooManyParameter (Date start, Date end, int value, string month, string yearStart, string
yearEnd) {

      // method body

}
```

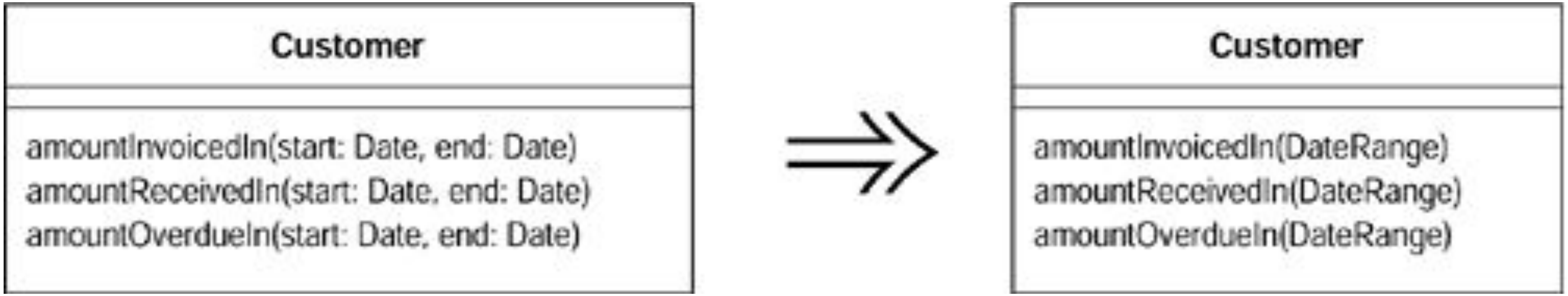
Long Method: Introduce Parameter Object

```
int MethodTooManyParameter (Date start, Date end, int value, string month, string yearStart, string
yearEnd) {

      // method body

}
```

⇩

```
public int MethodTooManyParameter(TimePeriod timePeriod, int value) {

     // method body

}
```

Long Method: Introduce Parameter Object



Customer

amountInvoicedIn(start: Date, end: Date)
amountReceivedIn(start: Date, end: Date)
amountOverdueIn(start: Date, end: Date)

Customer

amountInvoicedIn(DateRange)
amountReceivedIn(DateRange)
amountOverdueIn(DateRange)

Long Method: Preserve Whole Object

- When a method takes multiple parameters that originate from the same object, and passing the entire object improves clarity and maintainability.

Long Method: Preserve Whole Object

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

Long Method: Preserve Whole Object

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

⇩

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

Long 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;
```

Long Method: Decompose Conditional

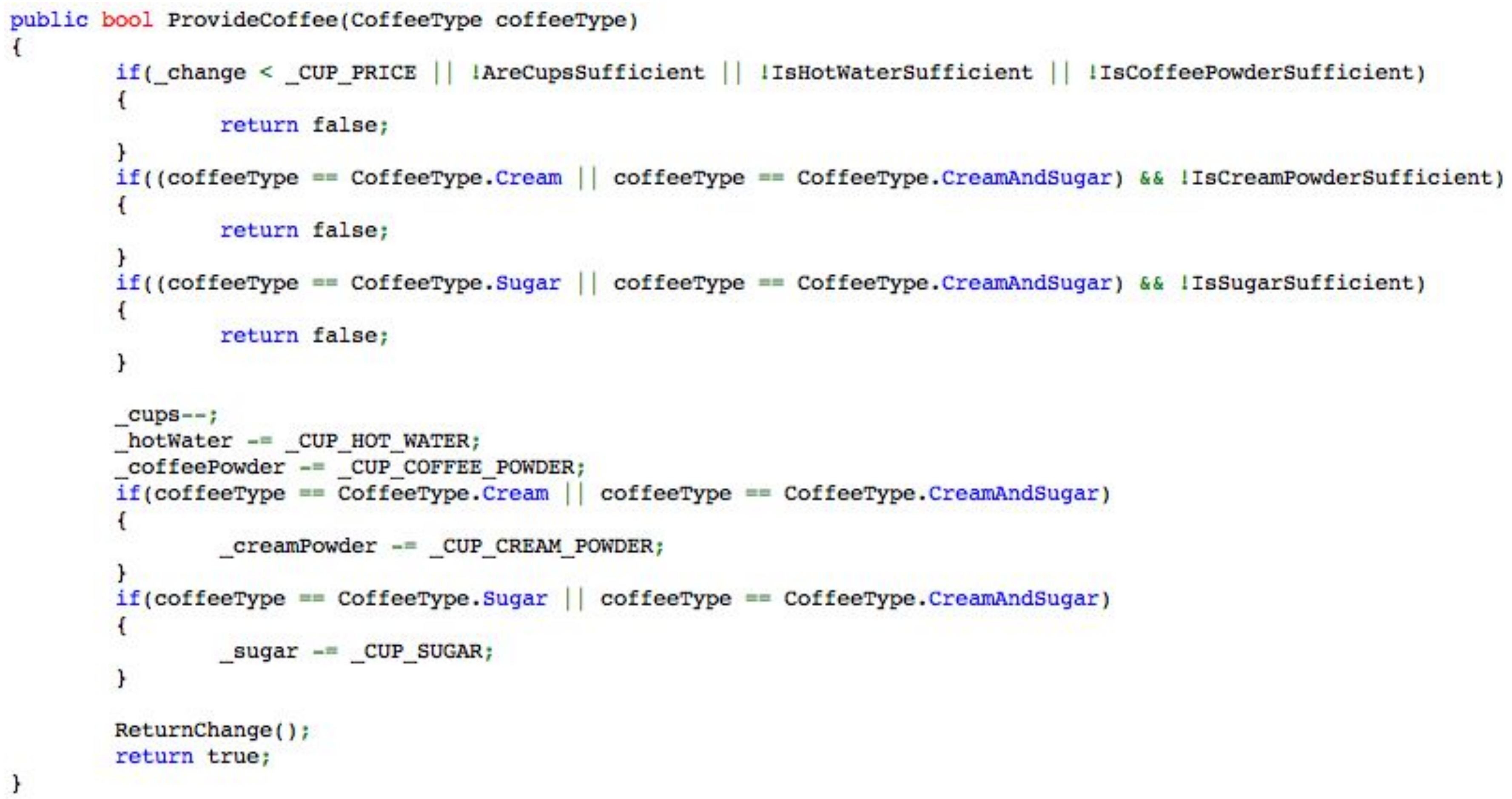
- You have a complicated conditional (if-then-else) statement.
- Extract methods from the condition, then part, and else parts.

```
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);
```

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;
}
```

Code Smell - 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

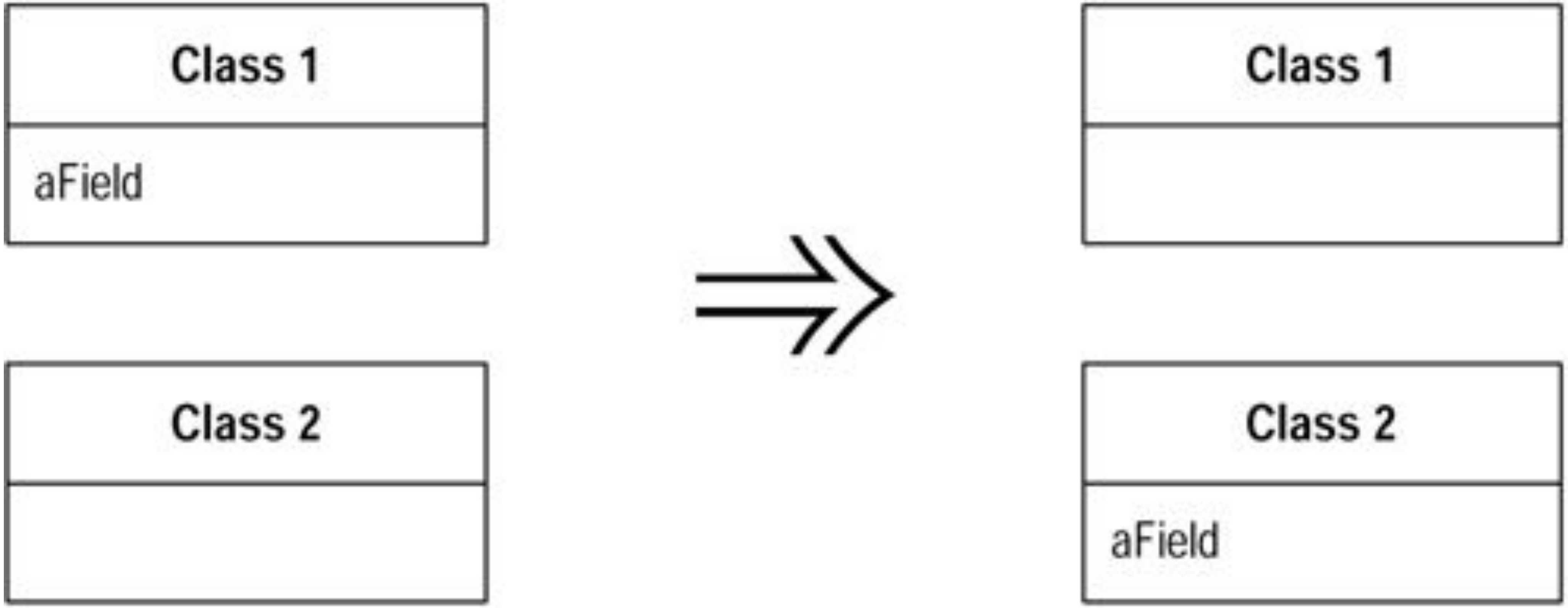
Feature Envy - Example

```
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;
    }
}
```

Feature Envy - Move Field

- We use "Move Field" to address the "Feature Envy" code smell when a field in one class is more frequently used by another class, indicating it logically belongs there.

Feature Envy - Move Field



Feature Envy - Move Field

```
class Person {
     String address;
     public String getAddress() {
     return address;
     }
     public void setAddress(String address) {
     this.address = address;
     }
}
class Order {
     Person customer;
     public void printShippingLabel() {
     System.out.println("Shipping to: " + customer.getAddress());
     }
}
```

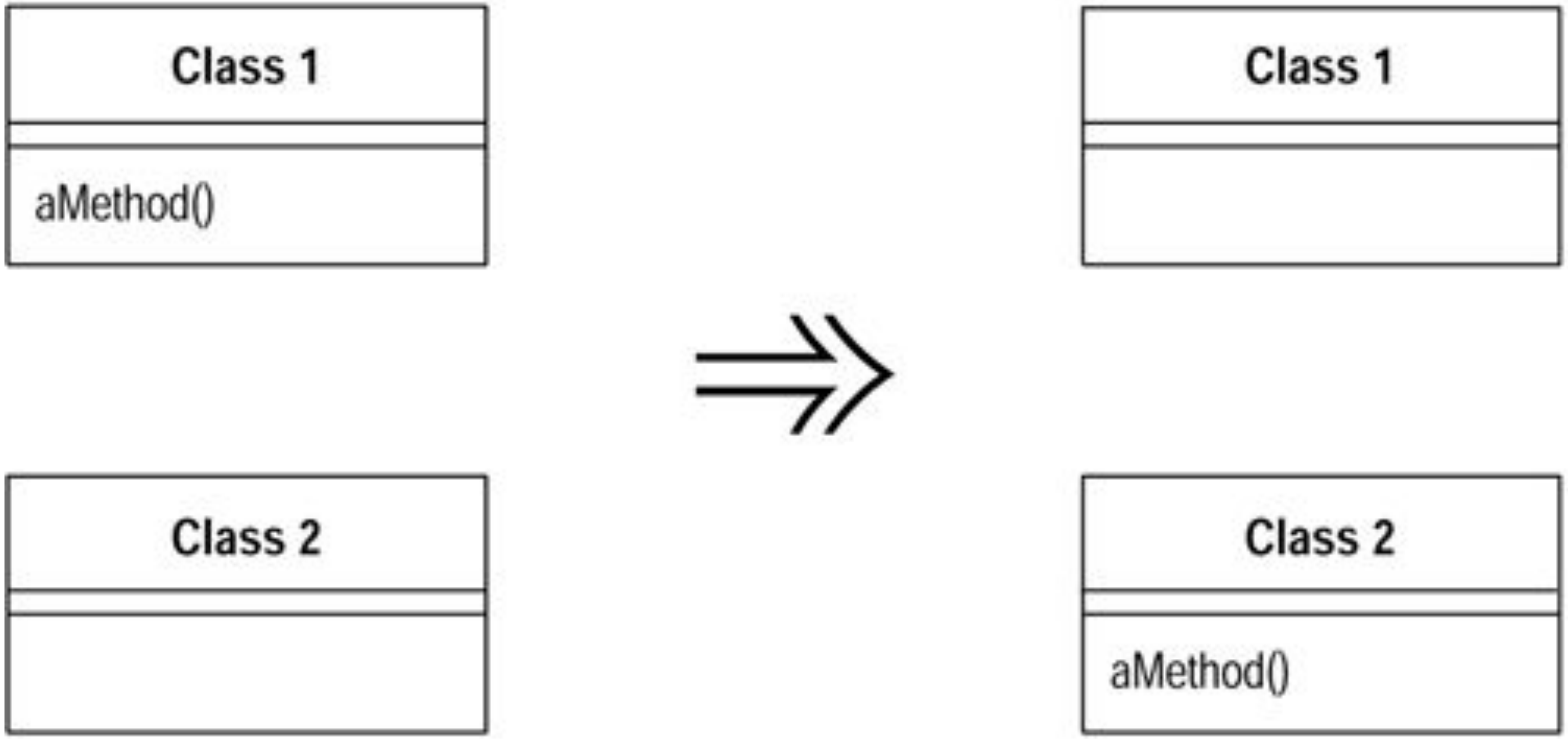
⇨

```
class Person {
}

class Order {
     String address;
     Person customer;

     public void printShippingLabel() {
     System.out.println("Shipping to: " + address);
     }
}
```

Feature Envy - Move Method

- We use "Move Method" to resolve the "Feature Envy" code smell when a method in one class excessively interacts with the data or methods of another class, suggesting it would function better in that class.

Feature Envy - Move Method



Feature Envy - Move Method

```
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;
    }
}
```

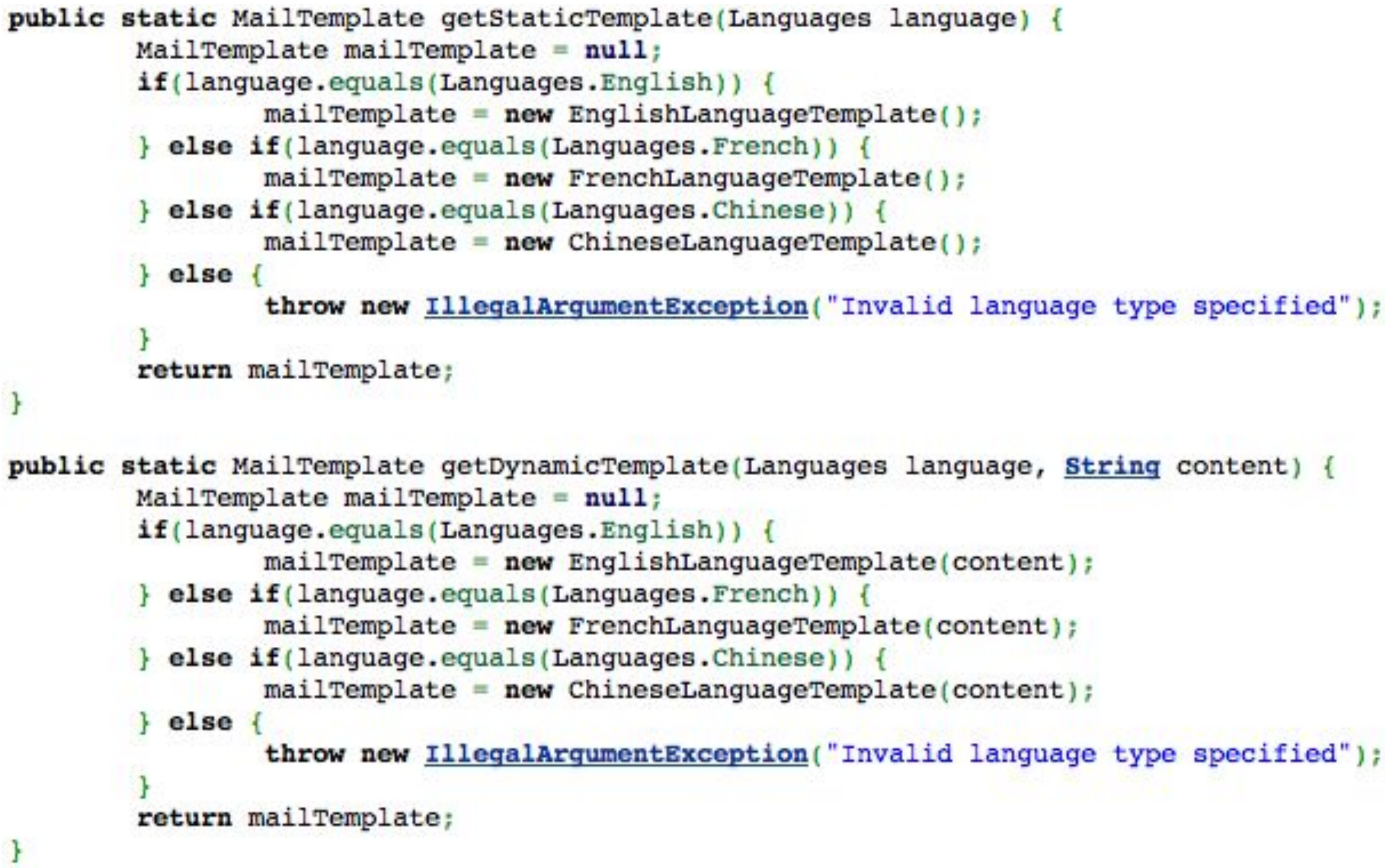
Feature Envy - Move Method

```
public class Loan {

    double capital(Loan loan) {
    if (getExpiry() == NO_DATE && getMaturity() != NO_DATE) {
        return getCommitmentAmount() * duration() * riskFactor();
    }

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

    return 0.0;
    }
}
```

Code Smell - Duplicated Code

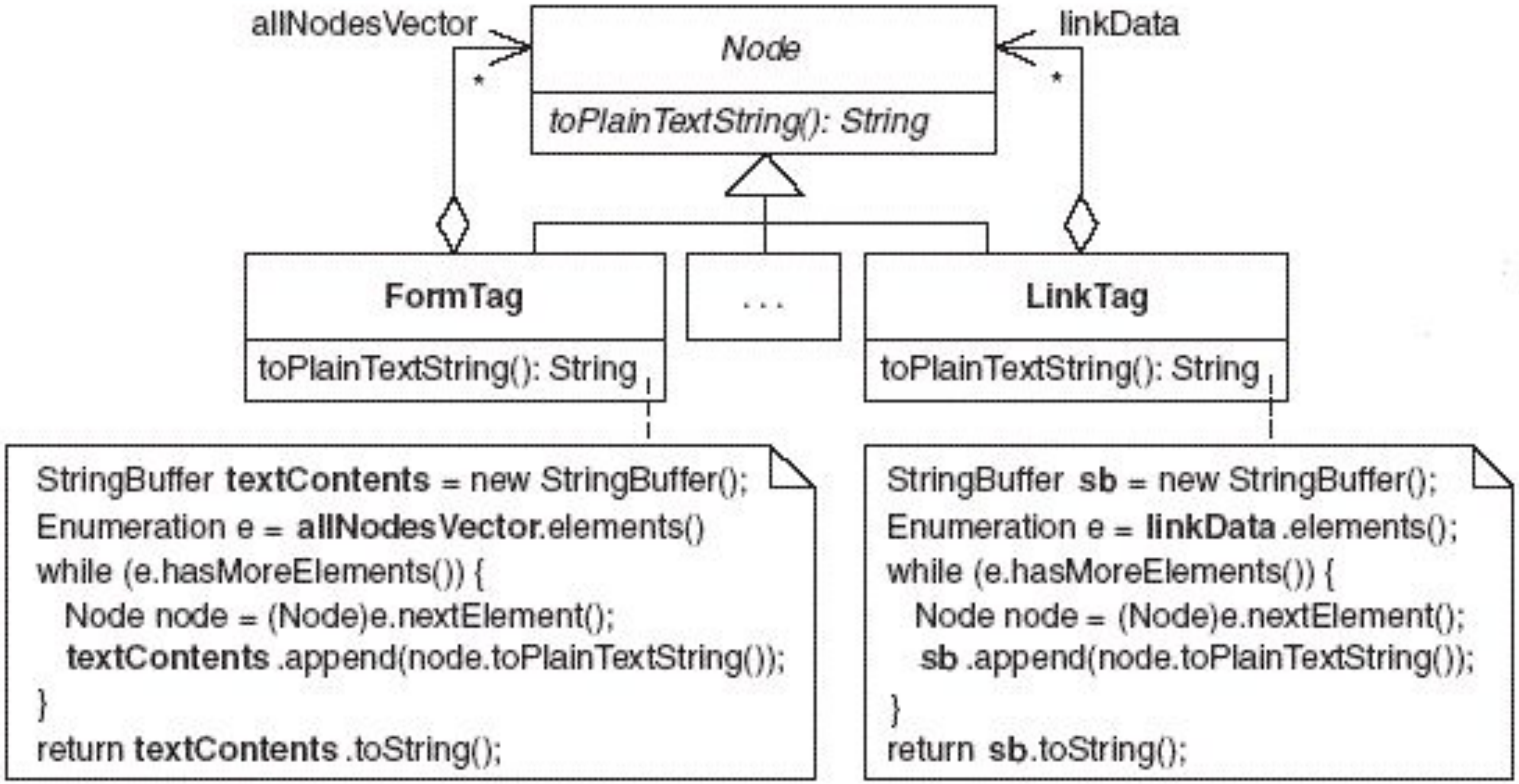
- The most pervasive and pungent smell in software
- There is obvious or blatant duplication
  - Such as copy and paste
- There are subtle or non-obvious duplications
  - Similar algorithms
- Remedies
  - Extract Method
  - Pull Up Field
  - Form Template Method
  - Substitute Algorithm

Example Of Obvious Duplication



```
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


allNodesVector
*
Node
toPlainTextString(): String
*
linkData
FormTag
toPlainTextString(): String
. . .
LinkTag
toPlainTextString(): String

```
StringBuffer textContents = new StringBuffer();
Enumeration e = allNodesVector.elements()
while (e.hasMoreElements()) {
  Node node = (Node)e.nextElement();
  textContents .append(node.toPlainTextString());
}
return textContents .toString();
```

```
StringBuffer sb = new StringBuffer();
Enumeration e = linkData .elements();
while (e.hasMoreElements()) {
  Node node = (Node)e.nextElement();
  sb .append(node.toPlainTextString());
}
return sb.toString();
```

Levels of Duplication

- Semantic Duplication - Runs the same, with semantic differences
  - Same for loop in 2 places
- Data Duplication - Variables/constants that represent the same data
  - Some constant declared in 2 classes (test and production)
- Conceptual Duplication - Runs differently, but does the same task
  - 2 Algorithm to Sort elements (Bubble sort and Quicksort)
- Logical Steps Duplication - Same set of steps repeat in different scenarios.
  - Same set of validations in various points in your applications

Semantic Duplication

- For and For Each Loop
- Loop v/s Lines repeated

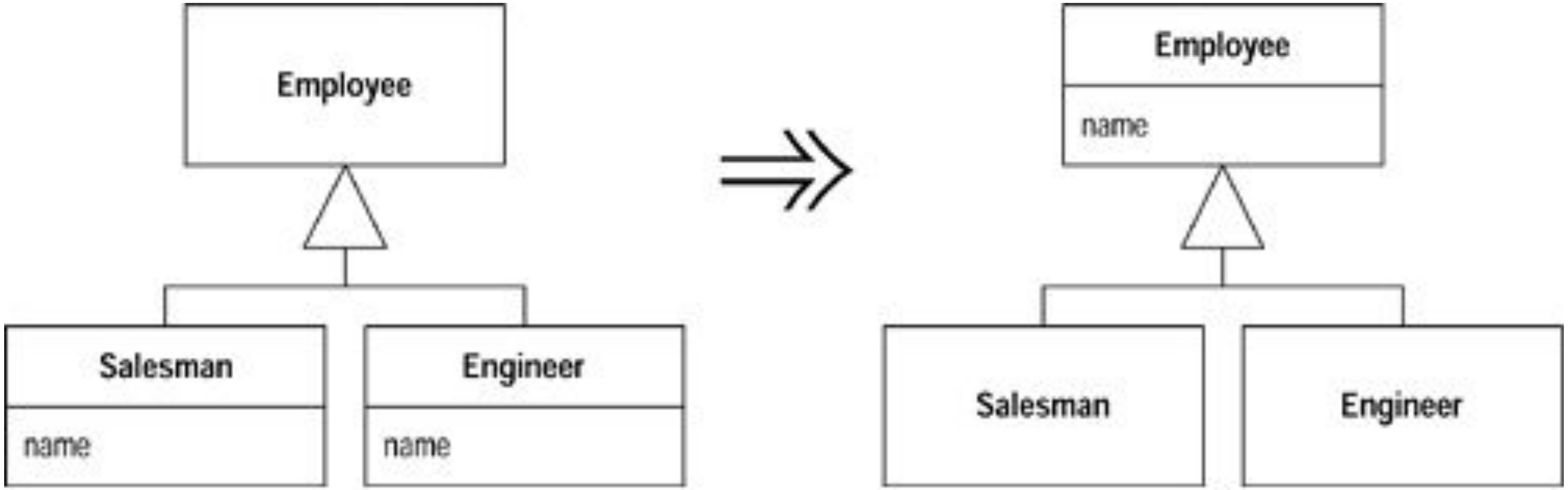
```
for(int i :
asList(1,3,5,10,15))
stack.push(i);

v/s

for(int i=0;i<5;i++){
   stack.push(asList(i));
}
```

```
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);
```

Duplicated Code - Pull Up Field

- We use "Pull Up Field" when multiple subclasses share the same field, and moving it to the superclass eliminates duplicated code while centralizing its management.

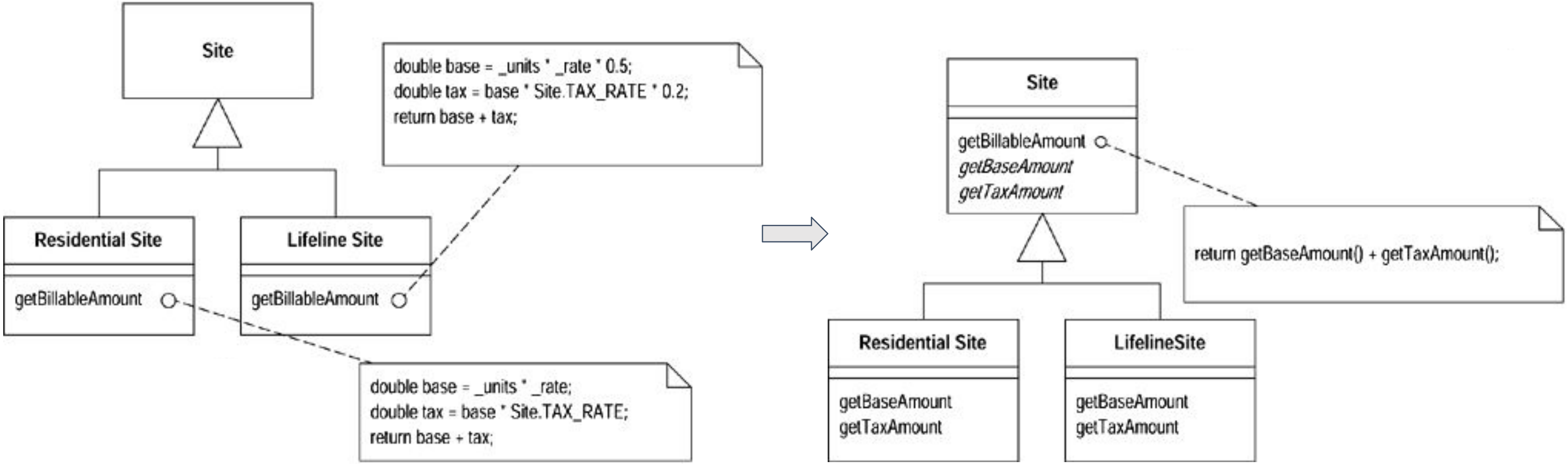
Duplicated Code - Pull Up Field



Duplicated Code - Form Template Method

- We use "Form Template Method" when multiple methods in subclasses share similar steps with minor variations, allowing us to move the shared steps to a superclass method and define the variations in the subclasses.

Duplicated Code - Form Template Method



```
double base = _units * _rate * 0.5;
double tax = base * Site.TAX_RATE * 0.2;
return base + tax;
```

```
double base = _units * _rate;
double tax = base * Site.TAX_RATE;
return base + tax;
```

```
return getBaseAmount() + getTaxAmount();
```

Duplicated Code - Substitute Algorithm

- We use "Substitute Algorithm" when a clearer or more efficient algorithm can replace existing code, eliminating duplication and improving readability or performance.

Duplicated Code - 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 ""; }
```

Duplicated Code - Substitute Algorithm

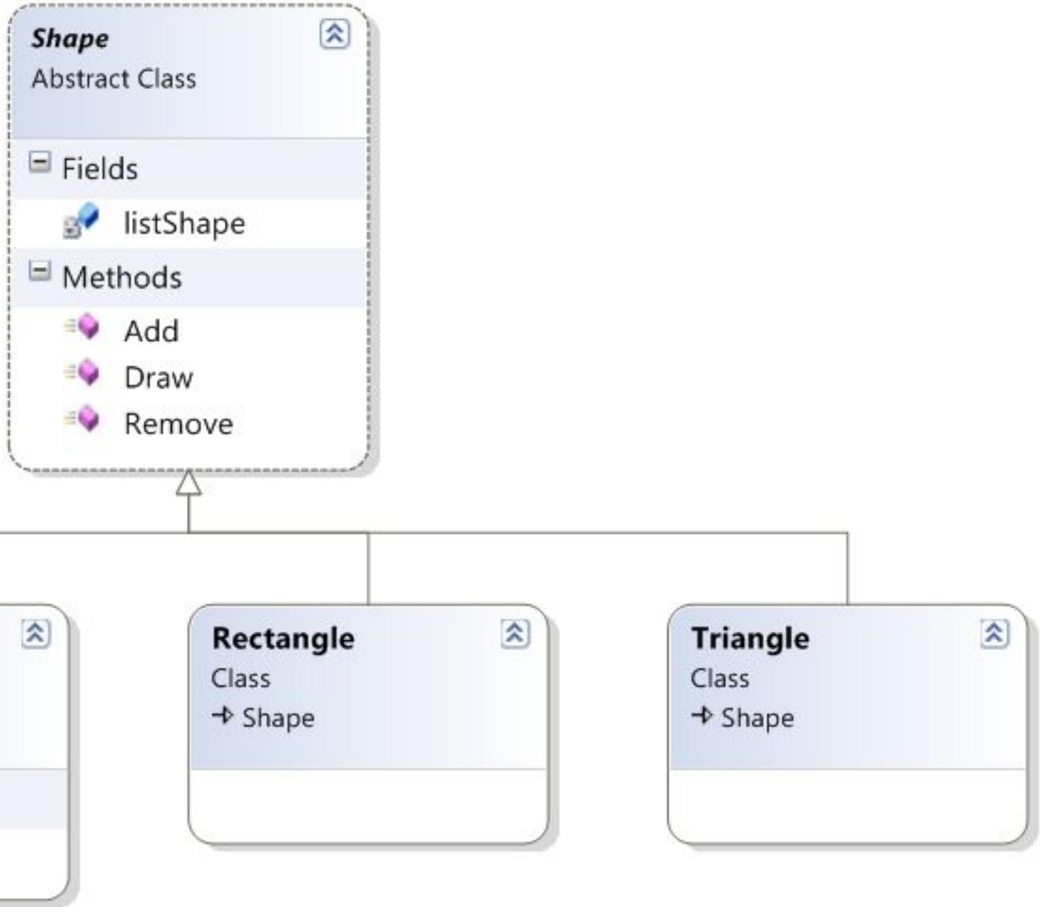
```
String foundPerson(String[] people){
  for (int i = 0; i < people.length; i++) {
   if (people[i].equals ("Don")){
      return "Don";
   }
   if (people[i].equals ("John")){
      return "John";
   }
   if (people[i].equals ("Kent")){
      return "Kent";
   }
  }
  return ""; }
```

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

Code Smell - Refused Bequest

- This code smell 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

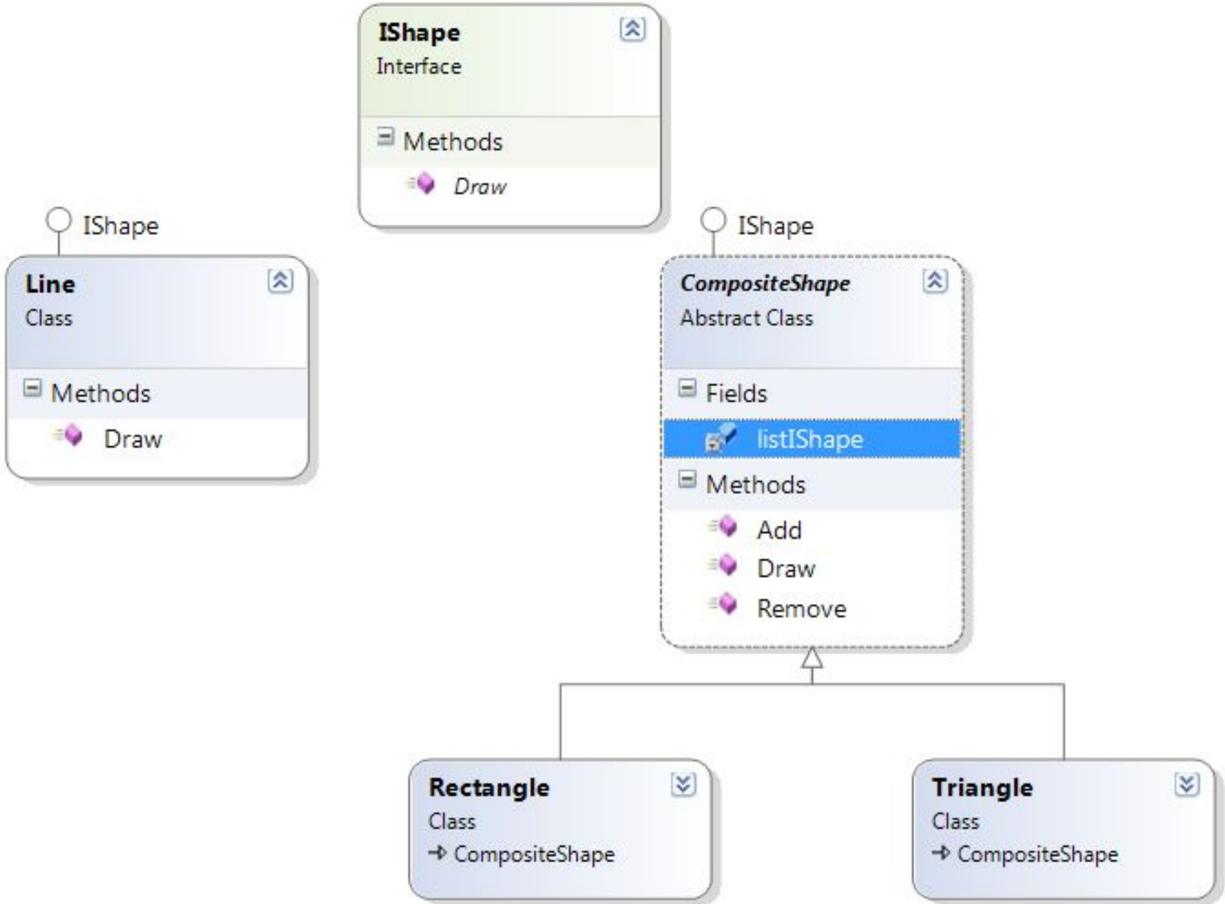


Shape
Abstract Class
Fields
listShape
Methods
Add
Draw
Remove
Line
Class
Shape
Methods
Draw
Rectangle
Class
Shape
Triangle
Class
Shape

Refused Bequest - Push Down Method/Field

- If some method/field in the parent class is only inherited by a subset of children, and the others refuse the bequest, then that method/field should be pushed down to that subset of children.
- If one or more children have the pushed down method/field then we find the code smell of duplicated code, and we should then push up the method/field into a new common parent.

Refused Bequest - Push Down Method/Field



Thank you