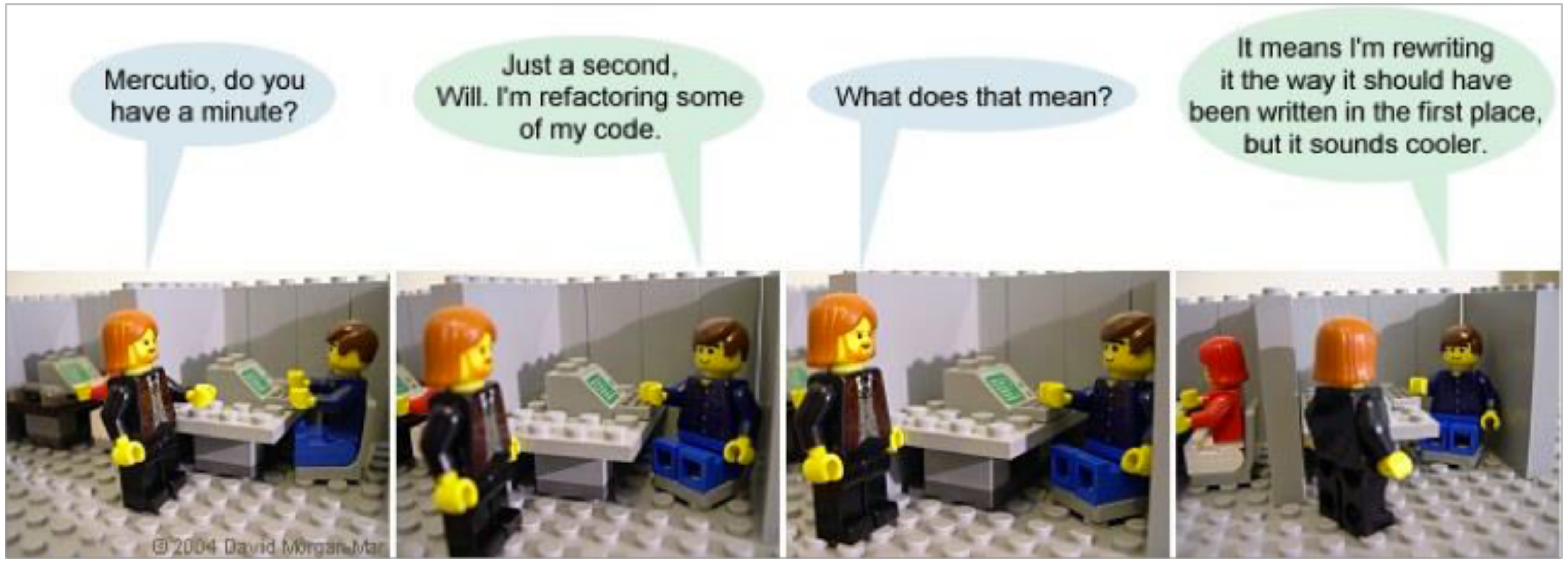
SWEN 6301 Software Construction

Lecture 10: Refactoring

Refactoring

- Modifying software to improve its readability, maintainability, and extensibility *without changing what it actually does*.
  - External behavior does NOT change
  - Internal structure is improved



Refactoring

- It is a disciplined way to clean up code that minimizes the chances of introducing bugs.
- In essence when you refactor you are improving the design of the code after it has been written.
  - In software development, <u>we design first then we code</u>
  - Refactoring is the opposite of this practice: take a bad design, and rework it into well-designed code

Refactoring

- Each step is simple
  - *move* a field from one class to another,
  - *pull* some code out of a method to make into its own method, and
  - *push* some code up or down a hierarchy
- Yet the cumulative effect of these small changes can radically improve the design.

Composing Methods – Extract Method

- You have a code fragment that can be grouped together.
- Turn the fragment into a method whose name explains the purpose of the method.

```
void printOwing(double amount) {
    printBanner();

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

Composing Methods – Extract Method

Motivation

- Method is too long or code that needs a comment to understand its purpose. Then turn that fragment of code into its own method.
- Prefer short, well-named methods for several reasons:
  - First, it increases the chances that other methods can use a method when the method is finely grained.
  - Second, it allows the higher-level methods to read more like a series of comments. Overriding also is easier when the methods are finely grained.

Composing Methods – Extract Method

Mechanics

- Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it).
- Copy the extracted code from the source method into the new target method.
- Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.

Composing Methods – Extract Method

Mechanics (cont.)

- See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.
- Look to see whether any of these local-scope variables are modified by the extracted code.
  - If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned.

Composing Methods – Extract Method

Mechanics (cont.)

- Pass into the target method as parameters local-scope variables that are read from the extracted code.
- Replace the extracted code in the source method with a call to the target method

Composing Methods – Extract Method

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Composing Methods – Extract Method

Example: No Local Variables

- Extract the code that prints the banner. Just cut, paste, and put in a call:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

```
void printBanner() {
    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");
}
```

Composing Methods – Extract Method

Example: Using Local Variables

- The problem is local variables: parameters passed into the original method and temporaries declared within the original method.
- The easiest case with local variables is when the variables are read but not changed.
  - In this case, can just pass them as parameters

Composing Methods – Extract Method

Example: Using Local Variables

- Extract the printing of details with a method with one parameter:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Composing Methods – Extract Method

Example: Using Local Variables

- extract the printing of details with a method with one parameter:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

```
void printDetails (double outstanding) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Composing Methods – Extract Method

Example: Reassigning a Local Variable

- It's the assignment to local variables that becomes complicated. In this case we're only talking about temps.
- For temps that are assigned to, there are two cases:
  - The simpler case is that in which the variable is a temporary variable used only within the extracted code. When that happens, you can move the temp into the extracted code.
  - The other case is use of the variable outside the code. If the variable is not used after the code is extracted, you can make the change in just the extracted code.

Composing Methods – Extract Method

Example: Reassigning a Local Variable

- If it is used afterward, you need to make the extracted code return the changed value of the variable.

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

Composing Methods – Extract Method

Example: Reassigning a Local Variable

- If it is used afterward, you need to make the extracted code return the changed value of the variable.
- The enumeration variable is used only in the extracted code, so I can move it entirely within the new method.

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}
```

```
double getOutstanding() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}
```

Composing Methods – Extract Method

- **Example: Reassigning a Local Variable**
- Rename the returned value if required:

```
double getOutstanding() {
    Enumeration e = _orders.elements();
    double result = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result = each.getAmount();
    }
    return result;
}
```

Composing Methods – Extract Method

Example: Reassigning a Local Variable

- If something more involved happens to the variable, have to pass in the previous value as a parameter.

```
void printOwing(double previousAmount) {

    Enumeration e = _orders.elements();
    double outstanding = previousAmount * 1.2;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

Composing Methods – Extract Method

Example: Reassigning a Local Variable

- In this case, the extraction would look like this:

```
void printOwing(double previousAmount) {
    double outstanding = previousAmount * 1.2;
    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

double getOutstanding(double initialValue) {
    double result = initialValue;
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }
    return result;
}
```

Composing Methods – Inline Method

- A method's body is just as clear as its name.
- Put the method's body into the body of its callers and remove the method.

```
boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Composing Methods – Inline Method

Motivation

- Use short methods named to show their intention, because these methods lead to clearer and easier to read code.
- But sometimes you do come across a method in which the body is as clear as the name. Or you refactor the body of the code into something that is just as clear as the name.
- Another time to use Inline Method is when you have a group of methods that seem badly factored. You can inline them all into one big method and then reextract the methods.

Composing Methods – Inline Method

Mechanics

- Check that the method is not polymorphic.
  - Don't inline if subclasses override the method; they cannot override a method that isn't there.
- Find all calls to the method.
- Replace each call with the method body.
- Remove the method definition.

Composing Methods – Inline Temp

- You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings.
- Replace all references to that temp with the expression.

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

Composing Methods – Inline Method

Motivation

- Most of the time Inline Temp is used as part of Replace Temp with Query, so the real motivation is there.
- The only time Inline Temp is used on its own is when you find a temp that is assigned the value of a method call.
- Often this temp isn't doing any harm and you can safely leave it there. If the temp is getting in the way of other refactorings, such as Extract Method, it's time to inline it.

Composing Methods – Inline Method

Mechanics

- Declare the temp as final if it isn't already, and compile.
  - This checks that the temp is really only assigned to once.
- Find all references to the temp and replace them with the right-hand side of the assignment.
- Remove the declaration and the assignment of the temp.

Composing Methods – Replace Temp with Query

- You are using a temporary variable to hold the result of an expression.
- Extract the expression into a method.
- Replace all references to the temp with the expression. The new method can then be used in other methods.

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

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

Composing Methods – Replace Temp with Query

Motivation

- The problem with temps is that they are temporary and local. Because they can be seen only in the context of the method in which they are used, temps tend to encourage longer methods, because that's the only way you can reach the temp.
- By replacing the temp with a query method, any method in the class can get at the information. That helps a lot in coming up with cleaner code for the class.

Composing Methods – Replace Temp with Query

Mechanics

- Look for a temporary variable that is assigned to once.
  - If a temp is set more than once consider Split Temporary Variable
- Declare the temp as final.
  - This will ensure that the temp is only assigned to once
- Extract the right-hand side of the assignment into a method.
  - Initially mark the method as private. You may find more use for it later, but you can easily relax the protection later.

Composing Methods – Replace Temp with Query

Example

- Start with a simple method

```
double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

Composing Methods – Replace Temp with Query

Example

- I'm inclined to replace both temps, one at a time.
- Although it's pretty clear in this case, I can test that they are assigned only to once by declaring them as final

```
double getPrice() {
    final int basePrice = _quantity * _itemPrice;
    final double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

Composing Methods – Replace Temp with Query

Example

- Compiling will then alert me to any problems. I do this first, because if there is a problem, I shouldn't be doing this refactoring.
- I replace the temps one at a time. First I extract the right-hand side of the assignment:

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;

    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}

private int basePrice() {
return _quantity * _itemPrice;
}
```

Composing Methods – Replace Temp with Query

Example

- First I replace the first reference to the temp:

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;
    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

Composing Methods – Replace Temp with Query

Example

- Do the next. Also remove the temp declaration:

```
double getPrice() {
    final double discountFactor;
    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice() * discountFactor;
}
```

Composing Methods – Replace Temp with Query

Example

- With that gone, can extract discountFactor in a similar way:

```
double getPrice() {
    final double discountFactor = discountFactor();
    return basePrice() * discountFactor;
}

private double discountFactor() {
    if (basePrice() > 1000) return 0.95;
    else return 0.98;
}
```

Composing Methods – Replace Temp with Query

Example

- See how it would have been difficult to extract discountFactor if I had not replaced basePrice with a query.
- The getPrice method ends up as follows:

```
double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return basePrice * discountFactor;
}
```

```
double getPrice() {
    return basePrice() * discountFactor();
}
```

Composing Methods – Introduce Explaining Variable

- You have a complicated expression.
- Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

```
 if ( (platform.toUpperCase().indexOf("MAC") > -1) &&
      (browser.toUpperCase().indexOf("IE") > -1) &&
       wasInitialized() && resize > 0 )
{
      // do something
}
```

```
   final boolean isMacOs      = platform.toUpperCase().indexOf("MAC") >
-1;
   final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")   >
-1;
   final boolean wasResized  = resize > 0;

   if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
       // do something
   }
```

Composing Methods – Introduce Explaining Variable

Motivation

- Expressions can become very complex and hard to read.
- In such situations temporary variables can be helpful to break down the expression into something more manageable.

Composing Methods – Introduce Explaining Variable

Mechanics

- Declare a final temporary variable, and set it to the result of part of the complex expression.
- Replace the result part of the expression with the value of the temp.
  - If the result part of the expression is repeated, you can replace the repeats one at a time.
- Repeat for other parts of the expression.

Composing Methods – Introduce Explaining Variable

Example

- Start with a simple calculation:

```
double price() {
// price is base price - quantity discount + shipping
return _quantity * _itemPrice -
    Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
    Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Composing Methods – Introduce Explaining Variable

Example

- Simple: it may be, but can make it easier to follow.
- First I identify the base price as the quantity times the item price. I can turn that part of the calculation into a temp:

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Composing Methods – Introduce Explaining Variable

Example

- Quantity times item price is also used later, so can substitute with the temp there as well:

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
        Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
        Math.min(basePrice * 0.1, 100.0);
}
```

Composing Methods – Introduce Explaining Variable

Example

- Next I take the quantity discount:

```
   double price() {
      // price is base price - quantity discount + shipping
      final double basePrice = _quantity * _itemPrice;
      final double quantityDiscount = Math.max(0, _quantity - 500) *
_itemPrice * 0.05;
      return basePrice - quantityDiscount +
         Math.min(basePrice * 0.1, 100.0);
   }
```

Composing Methods – Introduce Explaining Variable

Example

- Finally, I finish with the shipping. As do that, can remove the comment, too, because now it doesn't say anything the code doesn't say:

```
   double price() {
      final double basePrice = _quantity * _itemPrice;
      final double quantityDiscount = Math.max(0, _quantity - 500) *
_itemPrice * 0.05;
      final double shipping = Math.min(basePrice * 0.1, 100.0);
      return basePrice - quantityDiscount + shipping;
   }
```

Composing Methods – Introduce Explaining Variable

Example with Extract Method

- Start again:

```
double price() {
   // price is base price - quantity discount + shipping
   return _quantity * _itemPrice -
      Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
      Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Composing Methods – Introduce Explaining Variable

Example with Extract Method

- continue one at a time, finally get:

```
double price() {
    return basePrice() - quantityDiscount() + shipping();
}

private double quantityDiscount() {
    return Math.max(0, _quantity - 500) * _itemPrice * 0.05;
}

private double shipping() {
    return Math.min(basePrice() * 0.1, 100.0);
}

private double basePrice() {
    return _quantity * _itemPrice;
}
```

Composing Methods – Introduce Explaining Variable

Example with Extract Method

- When to use Introduce Explaining Variable? The answer is when Extract Method is more effort.
- If I'm in an algorithm with a lot of local variables, I may not be able to easily use Extract Method. In this case I use Introduce Explaining Variable to help me understand what is going on.
- As the logic becomes less tangled, I can always use Replace Temp with Query later. The temp also is valuable if I end up having to use Replace Method with Method Object.

Composing Methods – Split Temporary Variable

- You have a temporary variable assigned to more than once, but is not a loop variable nor a collecting temporary variable.
- Make a separate temporary variable for each assignment.

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

Composing Methods – Split Temporary Variable

Motivation

- Temporary variables are made for various uses. Some of these uses naturally lead to the temp's being assigned to several times.
- Loop variables change for each run around a loop (such as the i in for (int i=0; i<10; i++). Collecting temporary variables collect together some value that is built up during the method.
- Many other temporaries are used to hold the result of a long-winded bit of code for easy reference later.
  - These kinds of variables should be set only once. Otherwise, its purpose will be confusing and it will be error-prone.

Composing Methods – Split Temporary Variable

Mechanics

- Change the name of a temp at its declaration and its first assignment.
  - If the later assignments are of the form i = i + some expression, that indicates that it is a collecting temporary variable, so don't split it. The operator for a collecting temporary variable usually is addition, string concatenation, writing to a stream, or adding to a collection.
- Declare the new temp as final.
- Change all references of the temp up to its second assignment.
- Declare the temp at its second assignment
- Repeat in stages, each stage renaming at the declaration, and changing references until the next assignment.

Composing Methods – Split Temporary Variable

Example

- Compute the distance traveled by a car. From a standing start, a car experiences an initial force.
- After a delayed period a secondary force kicks in to further accelerate the haggis.

Composing Methods – Split Temporary Variable

Example

- Using the common laws of motion, I can compute the distance traveled as follows:

```
double getDistanceTravelled (int time) {
    double result;
    double acc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * acc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;

     if (secondaryTime > 0) {
         double primaryVel = acc * _delay;
         acc = (_primaryForce + _secondaryForce) / _mass;
         result +=  primaryVel * secondaryTime + 0.5 * acc *
secondaryTime * secondaryTime;
     }
     return result;
  }
```

Composing Methods – Split Temporary Variable

Example

- the variable acc is set twice. It has two responsibilities:
  - one to hold the initial acceleration caused by the first force and
  - another later to hold the acceleration with both forces.
- This is what to split.

Composing Methods – Split Temporary Variable

- **Example**
- Start at the beginning by changing the name of the temp and declaring the new name as final. Then change all references to the temp from that point up to the next assignment.

```
    double getDistanceTravelled (int time) {
        double result;
        final   double primaryAcc  = _primaryForce / _mass;
        int primaryTime = Math.min(time, _delay);
        result = 0.5 * primaryAcc * primaryTime * primaryTime;
        int secondaryTime = time - _delay;
        if (secondaryTime > 0) {
            double primaryVel = primaryAcc * _delay;
            double acc = (_primaryForce + _secondaryForce) / _mass;
            result +=  primaryVel * secondaryTime + 0.5 * acc *
secondaryTime * secondaryTime;
        }
        return result;
    }
```

Composing Methods – Split Temporary Variable

- **Example**
- Choose the new name to represent only the first use of the temp. I make it final to ensure it is only set once
- Then, declare the original temp at its second assignment.

Composing Methods – Split Temporary Variable

- **Example**
- Continue on the second assignment of the temp. This removes the original temp name completely, replacing it with a new temp named for the second use.

```
    double getDistanceTravelled (int time) {
        double result;
        final double primaryAcc = _primaryForce / _mass;
        int primaryTime = Math.min(time, _delay);
        result = 0.5 * primaryAcc * primaryTime * primaryTime;
        int secondaryTime = time - _delay;
        if (secondaryTime > 0) {
            double primaryVel = primaryAcc * _delay;
            final double secondaryAcc = (_primaryForce +
_secondaryForce) / _mass;
        result +=  primaryVel * secondaryTime + 0.5 *
                   secondaryAcc * secondaryTime * secondaryTime;
    }
    return result;
}
```

Composing Methods – Remove Assignments to Parameters

- The code assigns to a parameter.
- Use a temporary variable instead.

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
```

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
```

Composing Methods – Remove Assignments to Parameters

Motivation

- If you pass in an object named foo, in the parameter, assigning to the parameter means to change foo to refer to a different object.
- The reason don't like this comes down to lack of clarity and to confusion between pass by value and pass by reference

```
void aMethod(Object foo) {
  foo.modifyInSomeWay();          // that's OK
  foo = anotherObject;            // trouble and despair will follow
```

Composing Methods – Remove Assignments to Parameters

Mechanics

- Create a temporary variable for the parameter.
- Replace all references to the parameter, made after the assignment, to the temporary variable.
- Change the assignment to assign to the temporary variable.

Composing Methods – Remove Assignments to Parameters

Example

- Start with the following simple routine:

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    if (quantity > 100) inputVal -= 1;
    if (yearToDate > 10000) inputVal -= 4;
    return inputVal;
}
```

Composing Methods – Remove Assignments to Parameters

Example

- Replacing with a temp leads to

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```

Composing Methods – Remove Assignments to Parameters

Example

- You can enforce this convention with the final keyword:

```
    int discount (final int inputVal, final int quantity, final int
yearToDate) {
        int result = inputVal;
        if (inputVal > 50) result -= 2;
        if (quantity > 100) result -= 1;
        if (yearToDate > 10000) result -= 4;
        return result;
    }
```

Composing Methods – Remove Assignments to Parameters

- **What to print?**

```
  class Param {
    public static void main(String[] args) {
        int x = 5;
        triple(x);
        System.out.println ("x after triple: " + x);
    }
    private static void triple(int arg) {
        arg = arg * 3;
        System.out.println ("arg in triple: " + arg);
    }
}
```

Composing Methods – Remove Assignments to Parameters

- **Pass by value vs. Pass by reference**

```
class Param {
  public static void main(String[] args) {
      int x = 5;
      triple(x);
      System.out.println ("x after triple: " + x);
  }
  private static void triple(int arg) {
      arg = arg * 3;
      System.out.println ("arg in triple: " + arg);
  }
}
```

```
arg in triple: 15
x after triple: 5
```

Composing Methods – Replace Method with Method Object

- You have a long method that uses local variables in such a way that you cannot apply Extract Method.
- Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.

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

Composing Methods – Replace Method with Method Object

Motivation

- The difficulty in decomposing a method lies in local variables. If they are rampant, decomposition can be difficult.
- Using Replace Temp with Query helps to reduce this burden, but occasionally you may find you cannot break down a method that needs breaking.
- In this case you reach deep into the tool bag and get out your method object

Composing Methods – Replace Method with Method Object

Mechanics

- Create a new class, name it after the method.
- Give the new class a final field for the object that hosted the original method (the source object) and a field for each temporary variable and each parameter in the method.
- Give the new class a constructor that takes the source object and each parameter.
- Give the new class a method named "compute."

Composing Methods – Replace Method with Method Object

Mechanics (cont.)

- Copy the body of the original method into compute. Use the source object field for any invocations of methods on the original object.
- Replace the old method with one that creates the new object and calls compute.
- Because all the local variables are now fields, you can freely decompose the method without having to pass any parameters.

Composing Methods – Replace Method with Method Object

- **Example**
- A proper example of this requires a long chapter, so showing this refactoring for a method that doesn't need it.

```
Class Account
  int gamma (int inputVal, int quantity, int yearToDate) {
      int importantValue1 = (inputVal * quantity) + delta();
      int importantValue2 = (inputVal * yearToDate) + 100;
      if ((yearToDate - importantValue1) > 100)
          importantValue2 -= 20;
      int importantValue3 = importantValue2 * 7;
      // and so on.
      return importantValue3 - 2 * importantValue1;
  }
```

Composing Methods – Replace Method with Method Object

Example

- To turn this into a method object, I begin by declaring a new class. I provide a final field for the original object and a field for each parameter and temporary variable in the method.

```
class Gamma...
  private final Account _account;
  private int inputVal;
  private int quantity;
  private int yearToDate;
  private int importantValue1;
  private int importantValue2;
  private int importantValue3;
```

Composing Methods – Replace Method with Method Object

Example

- Add a constructor:

```
    Gamma (Account source, int inputValArg, int quantityArg, int
yearToDateArg) {
        _account = source;
        inputVal = inputValArg;
        quantity = quantityArg;
        yearToDate = yearToDateArg;
    }
```

Composing Methods – Replace Method with Method Object

Example

- Now can move the original method over; need to modify any calls of features of account to use the _account field

```
int compute () {
    importantValue1 = (inputVal * quantity) + _account.delta();
    importantValue2 = (inputVal * yearToDate) + 100;
    if ((yearToDate - importantValue1) > 100)
        importantValue2 -= 20;
    int importantValue3 = importantValue2 * 7;
    // and so on.
    return importantValue3 - 2 * importantValue1;
}
```

Outline

- Composing Methods
- Simplifying Conditional Expressions

Simplifying Conditional Expressions – Decompose Conditional

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);
```

Simplifying Conditional Expressions – Decompose Conditional

Motivation

- As with any large block of code, you can make your intention clearer by decomposing it and replacing chunks of code with a method call named after the intention of that block of code.
- With conditions you can receive further benefit by doing this for the conditional part and each of the alternatives.
- This way you highlight the condition and make it clearly what you are branching on.
- You also highlight the reason for the branching.

Simplifying Conditional Expressions – Decompose Conditional

Mechanics

- Extract the condition into its own method.
- Extract the then part and the else part into their own methods.

Simplifying Conditional Expressions – Decompose Conditional

Example

- Calculating the charge for something that has separate rates for winter and summer:

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

Simplifying Conditional Expressions – Decompose Conditional

- **Example**
- Extract the conditional and each leg as follows:

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

private boolean notSummer(Date date) {
    return date.before (SUMMER_START) || date.after(SUMMER_END);
}

private double summerCharge(int quantity) {
    return quantity * _summerRate;
}

private double winterCharge(int quantity) {
    return quantity * _winterRate + _winterServiceCharge;
}
```

Simplifying Conditional Expressions – Consolidate Conditional Expression

- You have a sequence of conditional tests with the same result.
- Combine them into a single conditional expression and extract it.

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // compute the disability amount
```

⇓

```
double disabilityAmount() {
    if (isNotEligableForDisability()) return 0;
    // compute the disability amount
```

Simplifying Conditional Expressions – Consolidate Conditional Expression

- ***Example: Ands***
- How to convert into And?

```
if (onVacation())
    if (lengthOfService() > 10)
        return 1;
return 0.5;
```

```
return (onVacation() && lengthOfService() > 10) ? 1 : 0.5;
```

```
double getPayAmount() {
  double result;
  if (_isDead) result = deadAmount();
  else {
      if (_isSeparated) result = separatedAmount();
      else {
          if (_isRetired) result = retiredAmount();
          else result = normalPayAmount();
      };
  }
return result;
};
```

$$\Downarrow$$

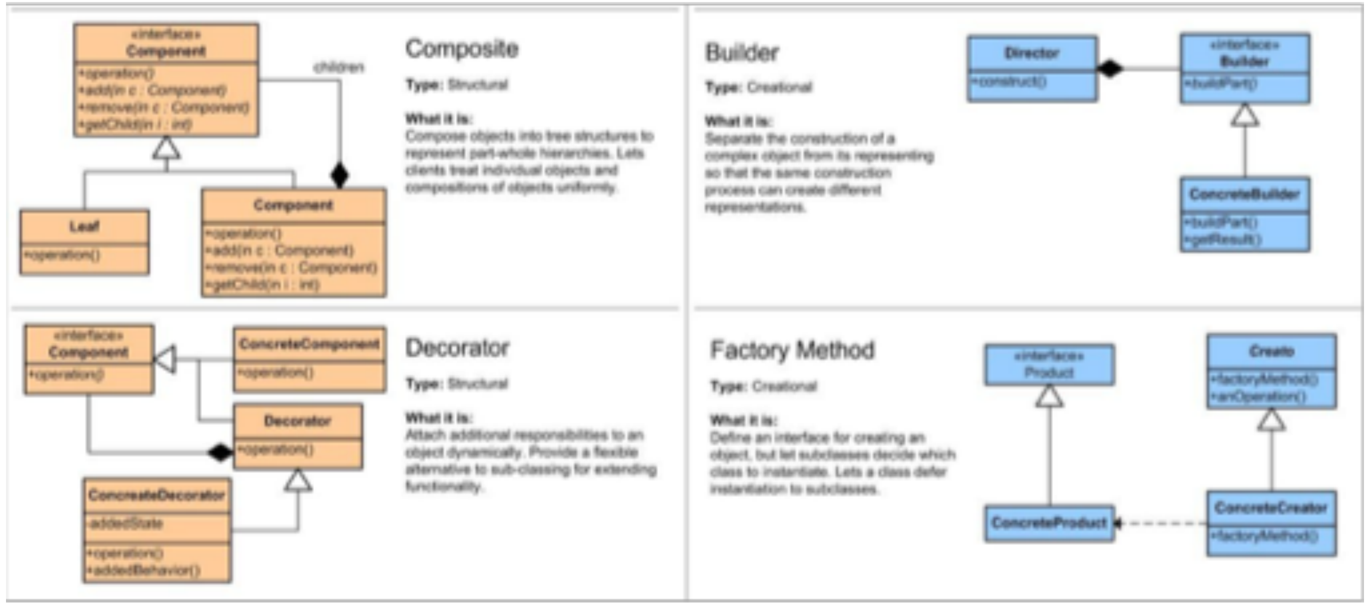
```
double getPayAmount() {
  if (_isDead) return deadAmount();
  if (_isSeparated) return separatedAmount();
  if (_isRetired) return retiredAmount();
  return normalPayAmount();
};
```

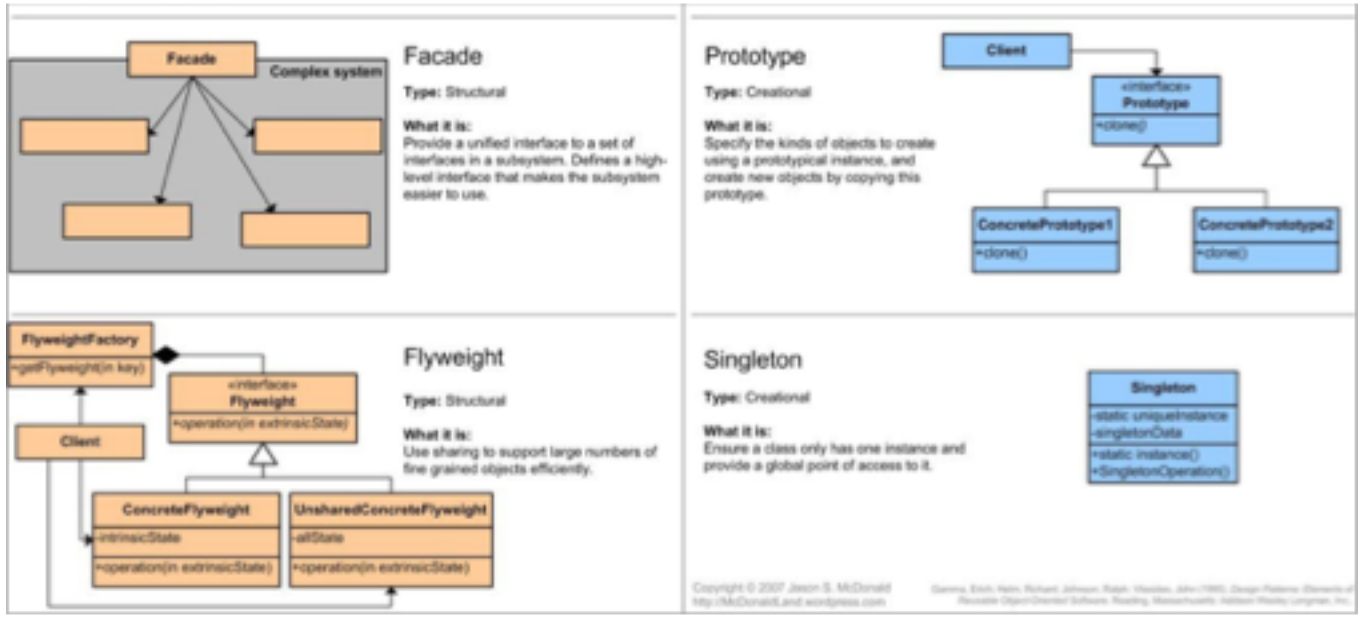
```java
public double getAdjustedCapital() {
  double result = 0.0;
  if (_capital > 0.0) {
    if (_intRate > 0.0 && _duration > 0.0) {
      result = (_income / _duration) * ADJ_FACTOR;
    }
  }
  return result;
}
```

```java
public double getAdjustedCapital() {
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    return (_income / _duration) * ADJ_FACTOR;
}
```

Design Heuristics: **Look for Common Design Patterns**

Design patterns provide the cores of ready-made solutions that can be used to solve many of software's most common problems





Design Heuristics: **Look for Common Design Patterns**

Design patterns provide the cores of ready-made solutions that can be used to solve many of software's most common problems

Pattern	Description
Abstract Factory	Supports creation of sets of related objects by specifying the kind of set but not the kinds of each specific object.
Adapter	Converts the interface of a class to a different interface.
Bridge	Builds an interface and an implementation in such a way that either can vary without the other varying.
Composite	Consists of an object that contains additional objects of its own type so that client code can interact with the top-level object and not concern itself with all the detailed objects.
Decorator	Attaches responsibilities to an object dynamically, without creating specific subclasses for each possible configuration of responsibilities.
Facade	Provides a consistent interface to code that wouldn't otherwise offer a consistent interface.
Factory Method	Instantiates classes derived from a specific base class without needing to keep track of the individual derived classes anywhere but the Factory Method.
Iterator	A server object that provides access to each element in a set sequentially.
Observer	Keeps multiple objects in synch with one another by making an object responsible for notifying the set of related objects about changes to any member of the set.
Singleton	Provides global access to a class that has one and only one instance.
Strategy	Defines a set of algorithms or behaviors that are dynamically interchangeable with each other.
Template Method	Defines the structure of an algorithm but leaves some of the detailed implementation to subclasses.

Reduce complexity by providing ready-made abstractions

Reduce errors by institutionalizing details of common solutions

Provide heuristic value by suggesting design alternatives

Streamline communication by moving the design dialog to a higher level

One potential trap with patterns is **force-fitting code to use a pattern**. In some cases, shifting code slightly to conform to a well-recognized pattern will ***improve*** understandability of the code. But if the code has to be shifted too far, forcing it to look like a standard pattern can sometimes ***increase complexity***.

Another potential trap with patterns is **feature-itis**: using a pattern because of a desire to try out a pattern rather than because the pattern is an appropriate design solution.

https://refactoring.guru/design-patterns

つづく