Lecture 22: Refactoring to Patterns

Kenneth M. Anderson

Object-Oriented Analysis and Design

CSCI 6448 - Spring Semester, 2005

Credit and Goals

- Credit where credit is due
  - Some of the material for this lecture is taken from “Refactoring to Patterns” by Joshua Kerievsky; as such some of this material is copyright © Pearson Education, Inc., 2005
- Goals of this Lecture
  - Present the idea of refactoring to patterns
  - Cover several examples

Refactoring to Patterns

- Refactoring is the process of transforming code such that functionality is maintained while improving the code's structure
  - Refactoring advocates small/safe transformations easy to learn/apply
- Design Patterns are solutions to recurring design problems that can be "rendered" into code in a straightforward way
  - In existing software systems, design patterns also transform code
    - i.e., the code was in state A before the pattern is applied
    - and in an improved state, state B, after the pattern is applied
- As such, design patterns represent "targets" for refactoring
  - Not only improving a code's structure but adding a time-tested solution to a common design problem to the code

Refactoring Directions

- Viewed in this way, a refactoring can be viewed as taking code to, towards, or away from a particular design pattern
- For example, some refactorings replace one pattern with another
  - Such a refactoring simultaneously moves away from the original pattern and to the new pattern
    - An example is a refactoring called "Move Accumulation to Visitor" which replaces the use of the Iterator pattern with the use of the Visitor pattern
- The towards direction is interesting; this occurs when you start a refactoring that leads to a particular design pattern but you only complete a few of the steps. The author, Joshua Kerievsky, states
  - "... this book contains numerous refactorings that provide acceptable design improvements whether you go towards or all the way to [the target pattern]."

Refactoring Directions, continued

- As an example, the refactoring "Move Embellishment to Decorator" has as one of its early steps "Replace Conditional with Polymorphism"
  - You may decide that the benefits of performing just that step is "enough" for your current situation and stop
    - One factor that will contribute to this decision is how much code needs to change to complete the rest of the pattern
- The next step in that refactoring is "Replace Inheritance with Delegation"
  - This, again, may provide just enough improvement to the code that you decide that going "all the way" to the Decorator pattern is not necessary
- You can always return to the code later to complete the refactoring

Examples

- Replace Conditional Logic With Strategy
- Replace Implicit Tree with Composite
- Move Embellishment to Decorator

- All of these refactorings go to their respective Patterns

Replace Conditional Logic with Strategy

- Conditional logic in a method controls which of several variants of a calculation are executed. Create a Strategy for each variant and make the method delegate the calculation to a Strategy instance
  - Strategy is a design pattern that separates an object and its behavior for a particular method (the behavior is put into its own object; the original object delegates to this new object)
- Mechanics
  - Create a strategy class; name it after the behavior being performed by the calculation; optionally add the word “Strategy” to the class name
  - Apply Move Method to move the calculation method to the strategy; the original method now delegates to this new method (compile/test)
  - Allow clients of the original class to choose a strategy (compile/test)
  - Apply “Replace Conditional With Polymorphism” to produce strategy subclasses that remove the conditional logic from the original method

Example

- Consider a Loan class that needs to calculate capital

```
public class Loan...
   public double capital() {
      if (expiry == null && maturity != null)
         return commitment * duration() * riskFactor();
      if (expiry != null && maturity == null) {
         if (getUnusedPercentage() != 1.0)
            return commitment * getUnusedPercentage() * duration() *
            riskFactor();
         else
            return (outstandingRiskAmount() * duration() *
            riskFactor()) +
                        (unusedRiskAmount() * duration() *
            unusedRiskFactor());
      }
      return 0.0;
   }
```

Create a strategy class

- We are strategizing the capital method, so we create the following:

```
public class CapitalStrategy {
   public double capital() {
      return 0.0;
   }
}
```

- Recall that refactoring advocates taking small, safe steps
- Now, we will use “Move Method” to move the capital() method from Loan to CapitalStrategy resulting in...

Move Method

- The biggest change is the addition of the loan parameter

```
public class CapitalStrategy...
   public double capital(Loan loan) {
      if (loan.getExpiry() == null && loan.getMaturity() != null)
         return loan.getCommitment() * loan.duration() *
         loan.riskFactor();
      if (loan.getExpiry() != null && loan.getMaturity() == null) {
         if (loan.getUnusedPercentage() != 1.0)
           return loan.getCommitment() * loan.getUnusedPercentage() *
           loan.duration() * loan.riskFactor();
         else
           return (loan.outstandingRiskAmount() * loan.duration() *
           loan.riskFactor()) +(loan.unusedRiskAmount() *
           loan.duration() * loan.unusedRiskFactor());
      }
      return 0.0;
   }
```

Move Method

- To complete, the Move Method refactoring, we delegate the original method's behavior to the newly created method in CapitalStrategy

```
public class Loan...
   public double capital() {
      return new CapitalStrategy().capital(this);
   }
}
```

- Now, we transform Loan to have a strategy instance variable and allow clients to configure it

```
public class Loan...
   private CaptialStrategy capitalStrategy;
   public Loan(..., CapitalStrategy strategy) {
      ...
      capitalStrategy = strategy;
   }
   public double capital() { return capitalStrategy.capital(this);}
}
```

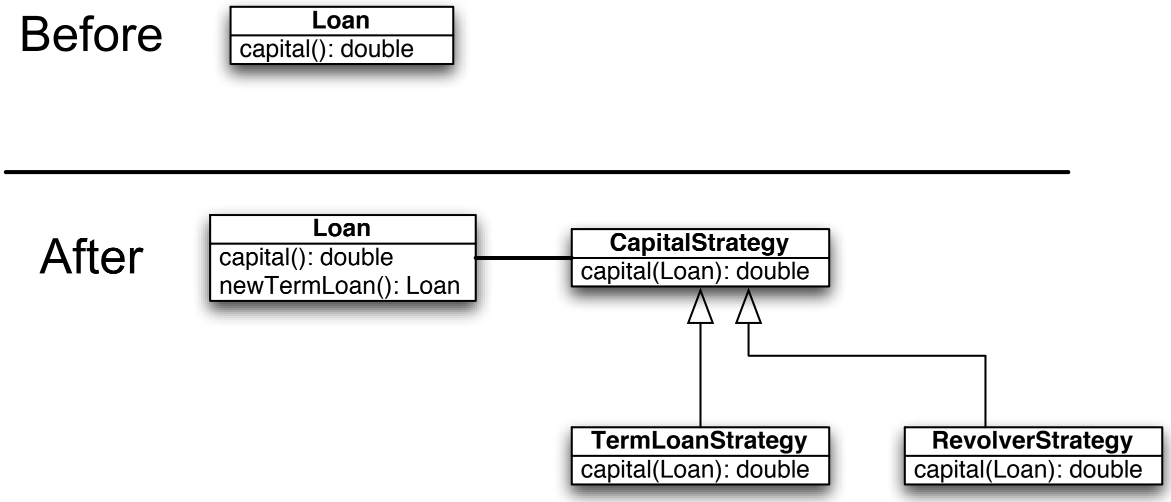

Replace Conditional with Polymorphism

- The next step is to create subclasses of CapitalStrategy that deal with the various branches of the original conditional
  - For instance the first branch (null expiry date, non-null maturity date) deals with Term loans; as such one of our subclasses will look like this

```
public class TermLoanStrategy extends CapitalStrategy {
   public double capital(Loan loan) {
      return loan.getCommitment() * loan.duration() *
      loan.riskFactor();
   }
}
```

  - The last step is to add factory methods to Loan to create Loan objects configured with the correct strategies, for instance
    - `public class Loan...`
      - `public static Loan newTermLoan(...) {`
        - `return new Loan(..., new TermLoanStrategy()); }`

Structure Before/After

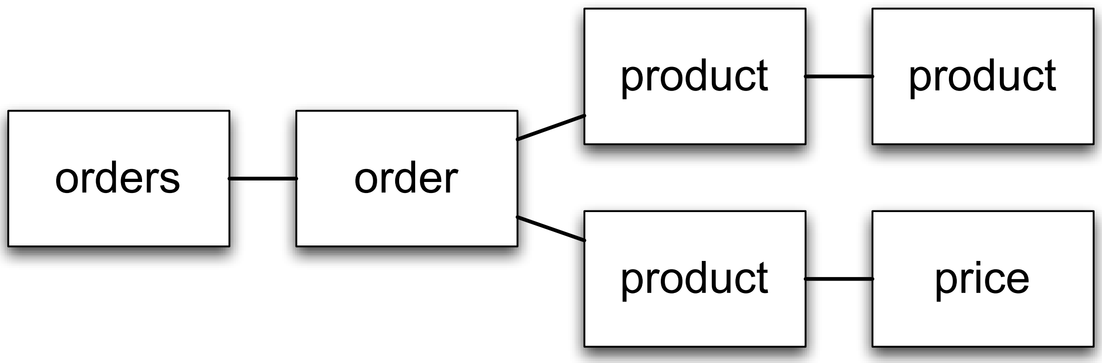


Replace Implicit Tree with Composite

- Description
  - You implicitly form a tree structure, using a primitive representation, such as a String. Replace your primitive representation with a Composite
- Example

```
String expectedResult =
  "<orders>" +
    "<order id='321'>" +
      "<product id='f1234' color='red' size='medium'>" +
        "<price currency='USD'>8.95</price>" +
          "Fire Truck</product>" +
      "<product id='p1112' color='red'>" +
        "<price currency='USD'>230.0</price>" +
          "Toy Porshe Convertible</product>" +
    "</order>" +
  "</orders>";
```

String as Tree



Mechanics

- Identify an implicit leaf, a part of the implicit tree that could be modeled with a new class. Create a leaf node class with instance variables for keeping track of the implicit leaf's contents and attributes; Compile and Test
- Replace every occurrence of the implicit leaf with an instance of the new leaf node; Compile and Test
- Repeat steps 1 and 2 for any additional implicit leafs
- Identify an implicit parent and create a parent node class for it; it needs to implement the "child management" functions of the Composite pattern; Compile and Test
- Replace every occurrence of the implicit parent with an instance of the new parent node; Compile and Test
- Repeat steps 4 and 5 until done

Example

- I will not show the entire example in this lecture
  - But I will show you enough to get you started
- Assume code like this exists to create (a portion of) our XML string:

```
private void writePriceTo(StringBuffer xml, Product product) {
   xml.append("<price");
   xml.append(" currency='");
   xml.append(product.getCurrency());
   xml.append("'>");
   xml.append(product.getPrice());
   xml.append("</price>");
}
```

- This code writes out the <price> tag portions of the string we saw previously

Generic Node Class

- Looking at our implicit tree string, we notice that each tag has
  - a name
  - an optional number of attributes
  - an optional number of children
  - an optional value
- Knowing this, we will design a generic TagNode class (using Test-Driven Design, for instance) that can handle these characteristics
- A portion of a test for this class might look like

```
TagNode priceTag = new TagNode("price");
priceTag.addAttribute("currency", "USD");
priceTag.addValue("8.95");
assertEquals("<price currency=...", priceTag.toString());
```

TagNode Class

```
public class TagNode {
   private String name = "";
   private String value = "";
   private StringBuffer attributes;
   public TagNode(String name) {
      this.name = name;
      attributes = new StringBuffer("");
   }
   public void addAttribute(String name, String value) {
      attributes.append(" "+attribute+"='"+value+"'");
   }
   public void addValue(String vlaue) {
      this.value = value;
   }
   public String toString() {
      return "<"+name+attributes+">"+value+"</"+name+">";
   }
}
```


Update writePriceTo

- Now that we have a class for the price "implict leaf" we can update the code that creates that portion of the string

```
private void writePriceTo(StringBuffer xml, Product product) {
   TagNode priceNode = new TagNode("price");
   priceNode.addAttribute("currency", product.getCurrency());
   priceNode.addValue(product.getPrice());
   xml.append(priceNode.toString());
}
```

- This class handles all of our implicit leaves; and it can handle our implicit parents too, if we add child management functions to it
  - This is a case where a single node plays all of the roles in the Composite pattern: Component, Leaf, and Composite

Updates to TagNode

- We need to add a collection class to hold a node's children

```
private List children;
```

- We need to add a method to get a list of our children

```
private List children() {
   if (children == null) {
      children = new LinkedList();
   }
   return children;
}
```

- Note: this is an example of "lazy creation" with respect to an instance variable; `children` remains null until the first time we ask for a list of a node's children; we do not initialize the instance variable until we need it

Updates to TagNode continued

- We need a method to add a child to a node

```
public void add(TagNode child) {
   children().add(child);
}
```

- Finally, we need to modify the `toString()` method to handle a node's children

```
public String toString() {
   String result =  "<"+name+attributes+">"
   Iterator itr = children.iterator();
   while (itr.hasNext()) {
      TagNode node = (TagNode)itr.next();
      result += node.toString();
   result += value;
   result += "</"+name+">";
   return result;
}
```

Update Price method

- We can now update the method that prints our Price info to simply create a priceNode and add it to its parent (in this case a product)

```
private void writePriceTo(TagNode parent, Product product) {
   TagNode priceNode = new TagNode("price");
   priceNode.addAttribute("currency", product.getCurrency());
   priceNode.addValue(product.getPrice());
   parent.add(priceNode);
}
```

- And, we can update the method that previously created the implicit product node to create an actual product node and call the updated method above to get a price node added to it
  - See next slide
  - Note: we have not previously shown this method

Updated Product Method

```
private void writeProductsTo(TagNode orderNode, Order order) {
   for (int j=0; j<order.getProductCount(); j++) {
      Product product = order.getProduct(j);
      TagNode productNode = new TagNode("product");
      productNode.addAttribute("id", product.getId());
      productNode.addAttribute("color", product.getColor());
      ...
      writePriceTo(productTag, product);
      productTag.addValue(product.getName());
      orderNode.add(productTag)
   }
}
```

Repeat until done

- To complete this refactoring, you would create similar methods for order and orders nodes of the tree we showed previously
- Your program now explicitly creates a tree structure using the Composite pattern and can output the XML for that tree with a single call:

```
System.out.println(root.toString());
```

- The advantages of doing this refactoring is that you can now easily add new types of leaf nodes and parent nodes
- Plus, our approach to building the tree allows us to create different XML representations of the tree if needed; we simply build a different type of tree, perhaps using different nodes/attributes

Move Embellishment to Decorator

- Description
  - Code provides an embellishment to a class's core responsibility; Move the embellishment code to a decorator
- Background
  - When adding new features to a system, it is common to add new code to old classes; the new code is said to "embellish" the old code with new functionality
    - The problem with this approach is that the embellishment adds new fields, methods, and logic, all of which exists for special-case behavior
- Motivating Idea
  - Try to place the new functionality in a decorator and then wrap the decorator around the original object at runtime when the new behavior is needed

Litmus Test

- This refactoring should not be used when the target class has a lot of public methods (where “a lot” depends on context)
- The reason?
  - The Decorator pattern requires transparent enclosures: decorators must implement the entire public interface of the target class
- Also, this refactoring is discouraged in situations where client code must be aware of the decorators, that is the client code checks the run-time types of the objects that it points at
  - For instance, beware client code that looks like this:
    - if (variable instance of SomeClass) then
  - If you dynamically wrap an instance of SomeClass with a decorator, the above code will fail

Mechanics

- Identify or create an enclosure type, an interface or class that declares the public methods needed by clients of the target class
- Find the conditional logic that adds the embellishment to the target class and remove that logic by applying “Replace Conditional with Polymorphism”; Compile and Test.
- Step 2 produced one or more subclasses of the embellished class. Transform these subclasses into delegating classes by applying “Replace Inheritance with Delegation”; Compile and Test
- Each delegating class now assigns its delegate to a new instance of the target class; Ensure that this assignment logic exists in the delegating class’s constructor and gets access to the delegate via a parameter; Compile and Test

Example

- Embellishments on StringNode of the HTML Parser project
  - Open Source HTML parser
    - http://sourceforge.net/projects/htmlparser/
- StringNode is used to store text found in HTML files
  - HTML often has text that looks like this:
    - “The Testing & Refactoring Workshop”
  - The string “&” is a character entity that needs to be translated to the character “&” when displayed to a user or otherwise processed by client software
- One of the embellishments to StringNode handled decoding these entity references; another embellishment was stripping escape characters (such as \n, \t, \r, etc.) from StringNodes

Problem

- These embellishments were not implemented as decorators on the StringNode class. Instead, the embellishments were implemented via options on the HTML Parser class and boolean flags within the StringNode class
- Thus, a programmer who wanted StringNodes to be decoded would write code like this:

```
Parser parser = Parser.createParser(...);
parser.setNodeDecoding(true);
```

- When StringNodes were created, they would be passed this flag as a parameter

```
StringNode s = new StringNode(..., parser.shouldDecodeNodes());
```

Problem, continued

- Then, when a StringNode was asked for its contents, it would check whether it should decode the text string before returning it to the client

```
public class StringNode...
   public String toPlainTextString() {
      String result = textBuffer.toString();
      if (shouldDecode)
         result = Translate.decode(result);
      return result;
   }
}
```

- This approach to embellishing StringNode will not scale well; requiring a new flag in toPlainTextString(), a new method in Parser, and a new parameter in StringNode's constructor for each embellishment

Applying the Refactoring

- Identifying an enclosure type
  - The HTML Parser framework had the following class hierarchy
    - Node→AbstractNode→StringNode
  - After analysis, the author selects Node as the enclosure type (the class defining the public interface shared by the target class, StringNode, and our new decorator, DecodingNode)
    - The key factor was finding a class that did not define any instance variables (to avoid having decorators from needlessly inheriting them)
- First Step: create new subclass, DecodingNode
  - Node→AbstractNode→StringNode→DecodingNode
- Second Step: make Decoding Node a delegating class
  - Node→DecodingNode
  - →AbstractNode→StringNode

© University of Colorado, Boulder, 2005

Second Step: Replace Conditional with Polymorphism

- Our "conditional" in this instance is the code that looked like this in `toPlainTextString()`:

```
 if (shouldDecode)
    result = Translate.decode(result);
```

- First, we encapsulate this field within StringNode, like so
  - Change constructor

```
public StringNode(..., boolean shouldDecode) {
   ...
   setShouldDecode(shouldDecode)
```

  - Change toPlainTextString()

```
if (shouldDecode()) {
   result = Translate.decode(result);
```

  - Add instance variable, getter and setter methods (not shown)

```
private boolean shouldDecode;
```


Step 2, continued

- Now we create our subclass

```
public class DecodingNode extends StringNode {
   public DecodingNode(...) {
      super(...);
   }
   protected boolean shouldDecode() {
      return true; -- Decoding Node always decodes
   }
}
```

- We update StringNode to no longer require the shouldDecode parameter to its constructor and update its shouldDecode() method to always return false; we also delete the shouldDecode instance variable and its associated setter method

Step 2, continued

- We now add a factory method to the StringNode class that returns the appropriate object based on a shouldDecode parameter; this method returns a value of type Node, the enclosure type

```
public class StringNode...
   public static Node createStringNode(..., boolean shouldDecode) {
      if (shouldDecode)
         return new DecodingNode(...);
      return new StringNode(...);
   }
}
```


Step 2, continued

- We can now remove the conditional in StringNode's toPlainTextString() and add an overriding version of this method in DecodingNode
  - In StringNode the method goes from this

```
public String toPlainTextString() {
   String result = textBuffer.toString();
   if (shouldDecode()) {
      result = Translate.decode(result);
   return result;
}
```

  - to this

```
public String toPlainTextString() {
   return textBuffer.toString();
}
```

Step 2 completed

- In DecodingNode, we add

```
public String toPlainTextString() {
   return Translate.decode(super.toPlainTextString());
}
```

  - and we can delete the shouldDecode() methods in both classes
- And we are now done with Step 2.
  - We compile and test to make sure that everything still works
    - We failed to show one step, which was having the Parser call the new factory method that we added to StringNode
- We are now ready to convert DecodingNode to a decorator
  - We start by using the refactoring “Replace Inheritance with Delegation”

Step 3: Replace Inheritance wit Delegation

- First, we add a field to DecodingNode that points to itself

```
private Node delegate = this;
```

- The enclosure type is used to set-up the Decorator pattern
- We now replace any calls to StringNode methods with calls to the delegate

```
public class DecodingNode extends StringNode...
   public String toPlainTextString() {
      return Translate.decode(delegate.toPlainTextString());
   }
}
```

  - This code will compile but not run, since it causes an infinite loop; The delegate object currently points to the calling object!

Step 3, continued

- We now break the inheritance relationship between the two classes

```
public class DecodingNode implements Node
```

- DecodingNode now implements the enclosure type interface rather than being a direct subclass of StringNode
  - We do this to keep our factory method code happy!
- We now set up the delegate instance variable to point to an instance of a StringNode

```
public class DecodingNode implements Node...
   private Node delegate = null;
   public DecodingNode(...) {
      delegate = new StringNode(...);
   }
```


Step 3 completed

- In DecodingNode, we now implement all of Node's public methods
  - Each one simply delegates the task to the delegate instance variable

```
public void accept(NodeVisitor visitor) { delegate.accept(visitor); }
```

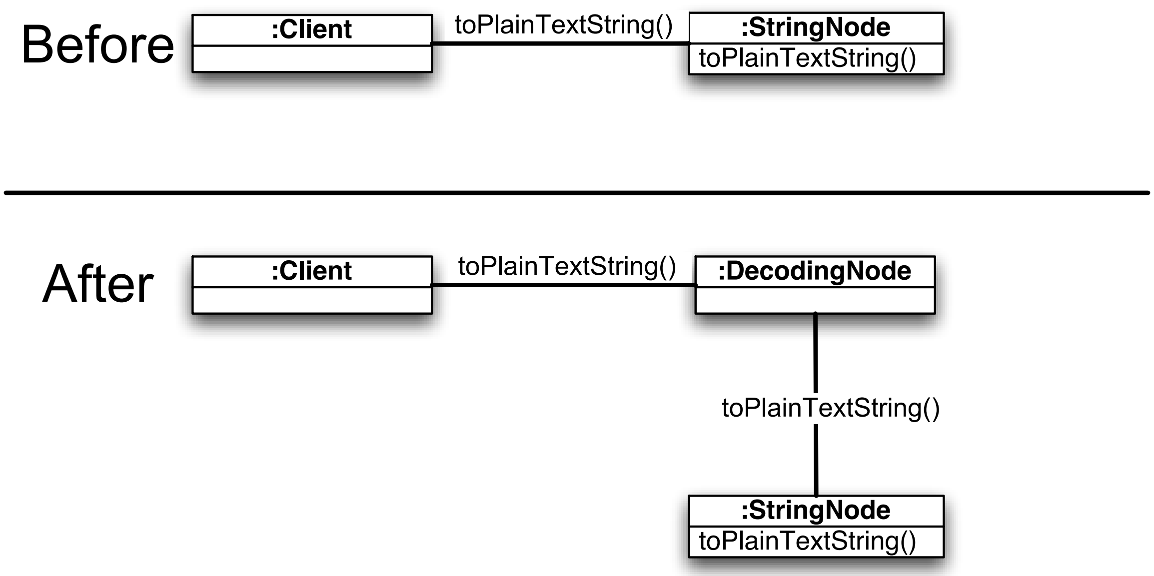
- Finally, to make DecodingNode a decorator, we change its constructor to accept a Node variable to define its delegate

```
public class DecodingNode implements Node...
   public DecodingNode(Node delegate) {
      this.delegate = delegate;}
```

- And we update our factory method to use the new constructor

```
public class StringNode...
   public static Node createStringNode(..., boolean shouldDecode) {
      if (shouldDecode)
         return new DecodingNode(new StringNode(...));
      return new StringNode(...);
   }}
```


Structure before/after



Summary

- Design Patterns can serve as “larger grain” targets for refactoring
  - As we’ve seen, these “larger grain” refactorings often consist of multiple “fine grain” refactorings, each which provide some benefit to the overall code
- This lecture shows how OO techniques build on each other
  - you can take your knowledge of design patterns and look for ways to include them into existing systems
  - you can use your knowledge of refactorings to ensure that these transformations are incremental and safe
    - You ensure safety by writing test cases before the refactoring and making sure that the changes do not break the functionality of the existing system
- What’s Next? Domain-Driven Design