Scribe Notes

VerifyY. July 11, 2011.

Refactoring

Refactoring:

- Restructuring or rearranging code, preferably in a series of *small steps*.
- Improves maintainability, addresses code smells.
- Semantics of code *is not changed*.
    - Re-running *unit tests* after refactoring demonstrates that the code still works.
- Common refactoring techniques are defined.
    - http://www.refactoring.com/
    - **http://sourcemaking.com/** (Highly recommended.)
    - “Refactoring: Improving the Design of Existing Code” by Martin Fowler (1999).
    - “Refactoring to Patterns” by Joshua Kerievsky (2004).

Primitive refactoring:

- Basic transformations (cannot be subdivided).
- eg., ‘Move Method,’ ‘Rename Method,’ ‘Add Class,’ ‘Extract Method,’ ‘Pull up Method.’

Composite refactoring:

- Complex transformations, composed of other refactoring steps.
- eg., ‘Move Method To Visitor,’ ‘Add Template Method.’

When to refactor:

- To improve maintainability or clarity, or to address code smells
- ONLY if you can refactor without breaking code
- NOT if the code will never change (stable)
- NOT if the code belongs to someone else

Model (High-level) refactoring

- More risky than low-level refactoring
- A code smell does not necessarily imply a design problem
- More abstract, less detail

Code (Low-level) refactoring

- Highly detailed
- Unit testing

Refactoring to Patterns

Refactoring to Patterns:

- Composite refactorings that introduce a design pattern.
- More high level.
- See “Refactoring to Patterns” by Joshua Kerievsky (2004).
    - Replace Constructors with Creation methods
    - Encapsulate Classes with Factory
    - Introduce Polymorphic Creation with Factory Method
    - Replace Conditional Logic with Strategy
    - Form Template Method
    - Compose Method
    - Replace Implicit Tree with Composite
    - Encapsulate Composite with Builder
    - Move Accumulation to Collecting Parameter
    - Extract Composite, Replace one/many with Composite
    - Replace Conditional Dispatcher with Command
    - Extract Adapter, Unify Interfaces with Adapter
    - Replace Type Code with Class
    - Replace State-Altering Conditionals with State
    - Introduce Null Object
    - Inline Singleton, Limit Instantiation with Singleton
    - Replace Hard-Coded Notifications with Observer
    - Move Embellishment to Decorator, Unify Interfaces, Extract Parameter
    - Move Creation Knowledge to Factory
    - Move Accumulation to Visitor
    - Replace Implicit Language with Interpreter
- Are these refactorings themselves patterns?
    - Depends how you define a pattern: If a patten is “a generic solution to a common problem in a specific context,” then yes. The context here is software solution and change, and the solution is the refactorings to patterns.
    - Further evidence that refactorings to patterns are themselves patterns is the similarity in how they are described; both are typically described with a name, application example, motivation, benefits, liabilities, mechanics, and a detailed example.
- Can look up a list of refactorings to resolve a particular identified code smell.
    - Example: Conditional Complexity
        - Replace conditional logic with *Strategy*
        - Move embellishment to *Decorator*
        - Replace state-altering conditional with *State*
        - (Introduce *Null Object*)
    - Example: Duplicate Code
        - Form *Template Method*
        - Introduce polymorphic creation with *Factory Method*
        - (Chain Constructors)

- Extract *Composite*
- Unify interfaces with *Adapter*

Example: Remove Duplicate Code

a) Suppose both instances are in a single class, and affects at most one variable

- Use Extract Method

Before	public void checkoutBook(Book b) {
	  checkout.add(b);
	  //Notify library
	  Library lib = b.getLibrary();
	  lib.notifyChange(b);
	}
		public void returnBook(Book b) {
		  checkout.remove(b);
		  //Notify library
		  Library lib = b.getLibrary();
		  lib.notifyChange(b);
		}

After	public void checkoutBook(Book b) {
	  checkout.add(b);
	  notifyLibrary(b);
	}
		public void returnBook(Book b) {
		  checkout.remove(b);
		  notifyLibrary(b);
		}

	private void notifyLibrary(Book b) {
	  Library lib = b.getLibrary();
	  lib.notifyChange(b);
	}	

b) Suppose the duplicated code fragment is in sibling classes.

- Use *Extract Method*, then *Pull Up Method*

Before	class Patron extends Person {
	  [...]
	  public void checkoutBook(Book b) {
	  checkout.add(b);
	  //Notify library
	  Library lib = b.getLibrary();
	  lib.notifyChange(b);
	  }
	}
		class Visitor extends Person {
		  [...]
		  public void returnBook(Book b) {
		  checkout.remove(b);
		  //Notify library
		  Library lib = b.getLibrary();
		  lib.notifyChange(b);
		  }
		}

After	class Patron extends Person {
	  [...]
	  public void checkoutBook(Book b) {
	  checkout.add(b);
	  notifyLibrary(b);
	  }
	}
		class Visitor extends Person {
		  [...]
		  public void returnBook(Book b) {
		  checkout.remove(b);
		  notifyLibrary(b);
		  }
		}

	class Person {	

	[...]
	protected void notifyLibrary(Book b) {
	Library lib = b.getLibrary();
	lib.notifyChange(b);
	}
	}	

c) The duplicated code fragment occurs in unrelated classes.

- Use Extract Method, and
  - Use one classes as a component of the other
  - Invoke the method from the other class
    - UnrelatedClass.commonMethod(params);
  - Move the method to a third class, and call it from both original classes.
    - UtilityClass.commonMethod(params);

d) The duplicated code affects more than one variable

- Proceed as usual, however:
  - Use output parameters (language permitting)
  - Create and return a composite data structure

e) The duplicated code is similar but not identical

- Use *Form Template Method*
  - Define outline of algorithm, deferring some steps to subclasses.
  - Subclasses can refine algorithm without changing structure.
  - Causes inverted control structure (superclass calls subclass).
  - Relies on inheritance
    - *Strategy* uses delegation instead

Example: Big Fish and Little Fish

Fish move around randomly in an ocean.

- A big fish can move to where a little fish is (and eat it)
- A little fish will not move to a big fish

Before	class LittleFish extends Fish {
	[...]
	public void move() {
	Direction dir = randomDir();
	Location loc =
	getLocationFromPos(dir);
	if (!loc.hasBigFish()) {
	move(loc);
	}
	}
		class BigFish extends Fish {
		[...]
		public void move() {
		Direction dir = randomDir();
		Location loc =
		getLocationFromPos(dir);
		move(loc);
		}
		}

	} 	

After	class LittleFish extends Fish {
	  [...]
	  public void okayToMove(Location loc) {
	  return !loc.hasBigFish();
	  }
	}
	
		class BigFish extends Fish {
		  [...]
		  public void okayToMove(
		  Location loc) {
		  return true;
		  }
		}

	abstract class Fish {
	  [...]
	  protected abstract boolean okayToMove(
	  Location loc);
	
	  public void move() {
	  Direction dir = randomDir();
	  Location loc =
	  getLocationFromPos(dir);
	  if (okayToMove(loc)) {
	  move(loc);
	  }
	  }
	}	

Example: Replace Constructor with Creation (Factory) Methods

- Problem: Having many constructors makes it hard to tell which constructor to call during development
    - Cannot assign meaningful names to constructors.
- Solution: Use other (non-constructor) factory methods to create objects.

Before	class Loan {
	  public Loan(notional, outstanding, customerRating, expiry) {
	  // Chained constructor
	  this(NullStrategy.instance(), notional, outstanding, customerRating, expiry);
	  }
	  public Loan(notional, outstanding, customerRating, expiry, maturity) {...}
	  public Loan(capitalStrategy, outstanding, customerRating, expiry) {...}
	  public Loan(capitalStrategy, outstanding, customerRating, expiry, maturity) {...}
	  public Loan(type, capitalStrategy, outstanding, customerRating, expiry) {...}
	  public Loan(type, capitalStrategy, outstanding, customerRating, expiry, maturity)
	{
	  … }

After	class Loan {
	  private Loan(type, capitalStrategy, notional, outstanding, customerRating, expiry,
	  maturity) { … }

	public Loan newTermLoan(notional, outstanding, customerRating, expiry) {
	  return newTermLoanWithStrategy(NullStrategy.instance(), notional,
	  outstanding, customerRating, expiry);
	}
	public Loan newTermLoanWithStrategy(capitalStrategy, notional, outstanding,
	  customerRating, expiry) { … }
	public Loan newRevolver(notional, outstanding, customerRating, expiry) { … }
	public Loan newRevolverWithStrategy(capitalStrategy, notional, outstanding,
	  customerRating, expiry) { … }
	public Loan newRCTL(notional, outstanding, customerRating, expiry, maturity) {
	  … }
	public Loan newRCTLWithStrategy(capitalStrategy, notional, outstanding,
	  customerRating, expiry, maturity) { … }

- Examples:
  - In Java, XML DOM parsers use abstract factory for node creation. This allows for switching between vendors.
  - Similarly, Java SQL interfaces use abstract factory to get their implementations from a particular vendor.
- Advantages:
  - Better communicates what kinds of instances are available
  - Makes it easier to find unused creation code
  - Bypasses constructor limitations, such as inability to have two constructors with the same argument signatures.
- Disadvantage:
  - Makes creation non-standard.

Refactoring Tools

- Eclipse
  - Built in primitive refactorings.
  - API for extending and customising refactoring.
- RefractorIT
  - Large set of refactorings.
  - Can detect code smells using computed design metrics (coupling, cohesion, etc.)
  - Eclipse plugin.
- Borland Together
  - Famous for design level refactoring.
  - Expensive.
- Also tools for visual studio.

Example Problems

1. What step is common in every refactoring?

2. Which of the following is not true of the *Template Method* pattern:
    a) The control structure is inverted
    b) The algorithm is refined by subclasses
    c) The pattern relies on inheritance
    d) None of the above

3. Name two advantages of replacing calls to constructors to calls to factory methods.

4. What general problems does a refactoring address? In what general situations should you not apply a refactoring despite it's ability to address these problems?

5. Briefly describe two refactorings (other than *Form Template Method*) that introduce patterns to the code.

6. Name a refactoring that would be appropriate for the following code. Give the resultant code after the refactoring has been performed.

Before	class HTTPSConnection extends HTTPConnection {
	  [...]
	  public void executeConnection(String ip) {
	  Connection c = new Connection (ip);
	  c.initConnection();
	  c = createTLSWrapper(c);
	  c.sendData(data);
	  c.close();
	  }
	}
	
	class HTTPConnection {
	  [...]
	  public void executeConnection(String ip) {
	  Connection c = new Connection (ip);
	  c.initConnection();
	  c.sendData(data);
	  c.close();
	  }
	}

Answers

1. The last step; re-run unit tests to provide evidence that the refactoring did not change code

semantics.

2. d) None of the above.

3.

- Better communicates what kinds of instances are available
- Makes it easier to find unused creation code
- Bypasses constructor limitations, such as inability to have two constructors with the same argument signatures.

4. A refactoring improves code maintainability and clarity, and address code smells. You should not apply a refactoring if the code is stable (will not be changed otherwise), or if the code 'belongs' to someone else.

5. Any of:

- Replace Constructors with Creation methods
- Encapsulate Classes with Factory
- Introduce Polymorphic Creation with Factory Method
- Replace Conditional Logic with Strategy
- Form Template Method
- Compose Method
- Replace Implicit Tree with Composite
- Encapsulate Composite with Builder
- Move Accumulation to Collecting Parameter
- Extract Composite, Replace one/many with Composite
- Replace Conditional Dispatcher with Command
- Extract Adapter, Unify Interfaces with Adapter
- Replace Type Code with Class
- Replace State-Altering Conditionals with State
- Introduce Null Object
- Inline Singleton, Limit Instantiation with Singleton
- Replace Hard-Coded Notifications with Observer
- Move Embellishment to Decorator, Unify Interfaces, Extract Parameter
- Move Creation Knowledge to Factory
- Move Accumulation to Visitor
- Replace Implicit Language with Interpreter

6. Use *Form Template Method*.

After	class HTTPSConnection extends HTTPConnection {
	[...]
	protected Connection performFurtherConnectionInit(Connection c) {
	return createTLSWrapper(c);
	}

```
}

class HTTPConnection {
    [...]
    public void executeConnection(String ip) {
        Connection c = new Connection (ip);
        c.initConnection();
        c = performFurtherConnectionInit(c);
        c.sendData(data);
        c.close();
    }
    protected Connection performFurtherConnectionInit(Connection c) {
        return c;
    }
}
```