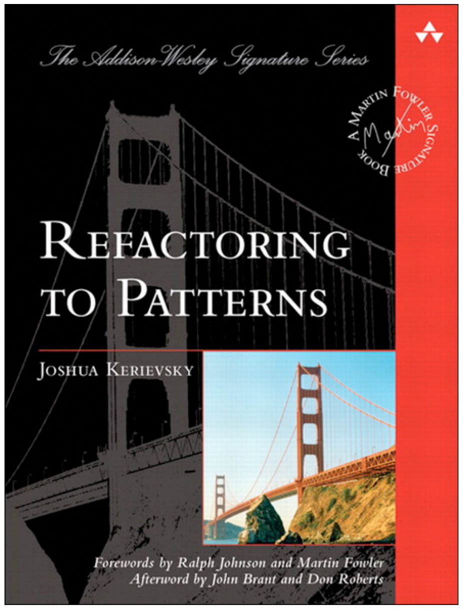

The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Refactoring to Patterns
Joshua Kerievsky
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts

Refactoring to Patterns

Joshua Kerievsky
joshua@industriallogic.com

Hello World - Strategy

```
interface MessageStrategy {
    public void sendMessage();
}

abstract class AbstractStrategyFactory {
    public abstract MessageStrategy createStrategy(MessageBody mb);
}

class MessageBody {
    Object payload;
    public Object getPayload() {
        return payload;
    }
    public void configure(Object obj) {
        payload = obj;
    }
    public void send(MessageStrategy ms) {
        ms.sendMessage();
    }
}
```

Hello World - Factory

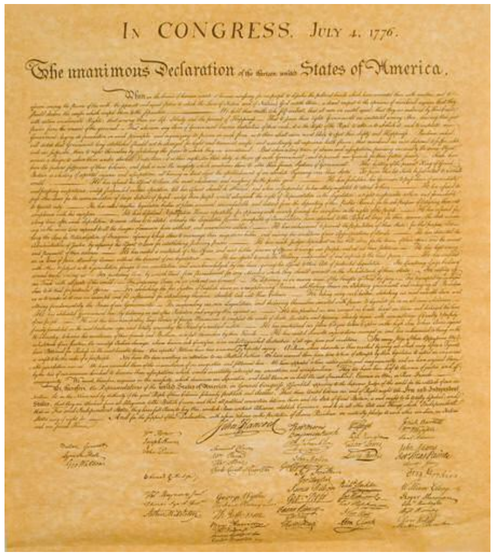
```
class DefaultFactory extends AbstractStrategyFactory {
    private DefaultFactory() {
    }
    static DefaultFactory instance;
    public static AbstractStrategyFactory getInstance() {
        if (instance == null)
            instance = new DefaultFactory();
        return instance;
    }

    public MessageStrategy createStrategy(final MessageBody mb) {
        return new MessageStrategy() {
            MessageBody body = mb;
            public void sendMessage() {
                Object obj = body.getPayload();
                System.out.println(obj);
            }
        };
    }
}
```

Hello World - Main

```
public class HelloWorld {
    public static void main(String[] args) {
        MessageBody mb = new MessageBody();
        mb.configure("Hello World!");
        AbstractStrategyFactory asf = DefaultFactory.getInstance();
        MessageStrategy strategy = asf.createStrategy(mb);
        mb.send(strategy);
    }
}
```

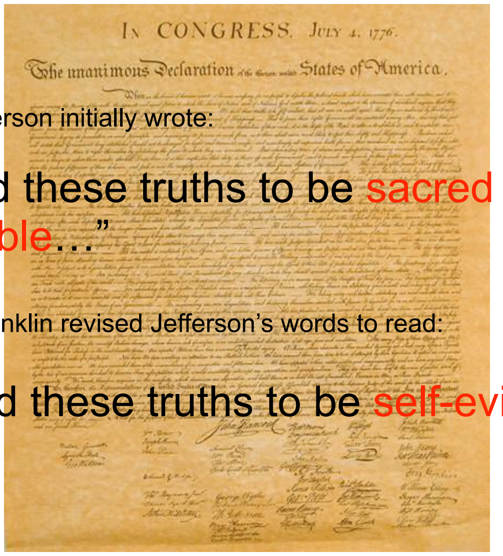
Thanks to Jason Tiscioni for this example.



IN CONGRESS. JULY 4, 1776.

The unanimous Declaration of the thirteen united States of America.

[illegible]



IN CONGRESS. JULY 4, 1776.

The unanimous Declaration of the thirteen united States of America.

[illegible]

Thomas Jefferson initially wrote:

“We hold these truths to be sacred and undeniable…”

Benjamin Franklin revised Jefferson’s words to read:

“We hold these truths to be self-evident…

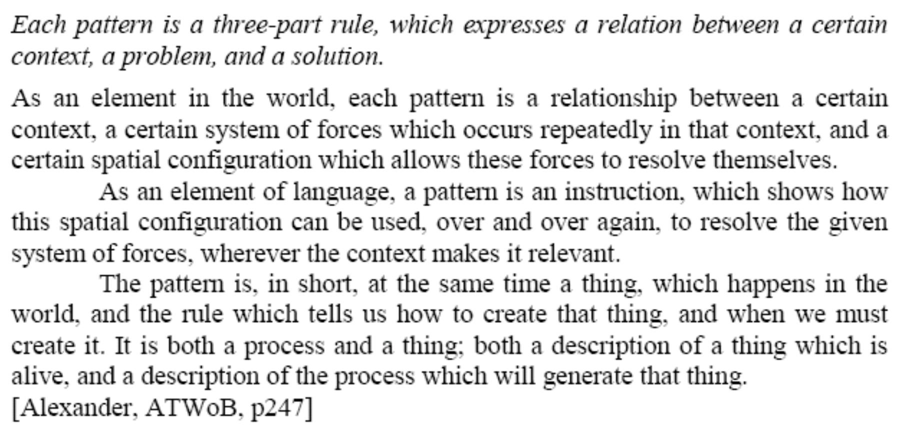
John Thompson, ~~hatter~~, ~~makes~~
~~and~~ ~~sells~~ ~~hats~~ ~~for ready money~~

John Thompson

“All good writing is based upon revision”

Jacques Barzun, *Simple & Direct*, 4th Edition

What Is A Pattern



Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution.

As an element in the world, each pattern is a relationship between a certain context, a certain system of forces which occurs repeatedly in that context, and a certain spatial configuration which allows these forces to resolve themselves.

As an element of language, a pattern is an instruction, which shows how this spatial configuration can be used, over and over again, to resolve the given system of forces, wherever the context makes it relevant.

The pattern is, in short, at the same time a thing, which happens in the world, and the rule which tells us how to create that thing, and when we must create it. It is both a process and a thing; both a description of a thing which is alive, and a description of the process which will generate that thing.

[Alexander, ATWoB, p247]

Algebra & Word Problems

In algebra class, we first learn different manipulations, like:

> “add the same value to both sides of the equation”

> “the commutative property of addition allows us to swap its operands.”

Once we know the manipulations, we’re given word problems:

> “A train leaves New York heading West. . . .”

To solve this problem, you express it in terms of an algebraic equation and then apply the rules of algebra to arrive at an answer.

Refactoring & Patterns

Design patterns are the word problems of the programming world; refactoring is its algebra. After having read Design Patterns [DP], you reach a point where you say to yourself, “If I had only known this pattern, my system would be so much cleaner today.” The book you are holding introduces you to several sample problems, with solutions expressed in the operations of refactoring.

The Algebra of Refactoring

Many people will read this book and try to memorize the steps to implement these patterns. Others will read this book and clamor for these larger refactorings to be added to existing programming tools. Both of these approaches are misguided. The true value of this book lies not in the actual steps to achieve a particular pattern but in understanding the thought processes that lead to those steps. By learning to think in the algebra of refactoring, you learn to solve design problems in behavior-preserving steps, and you are not bound by the small subset of actual problems that this book represents.

Patterns of Refactoring

So take these exemplars that Josh has laid out for you. Study them. Find the underlying patterns of refactoring that are occurring. Seek the insights that led to the particular steps. Don’t use this as a reference book, but as a primer.

Smells

1. Duplicated Code [F]
2. Long Method [F]
3. Conditional Complexity
4. Primitive Obsession
5. Indecent Exposure
6. Solution Sprawl
7. Alternative Classes with Different Interfaces [F]
8. Lazy Class [F]
9. Large Class [F]
10. Switch Statements [F]
11. Combinatorial Explosion
12. Oddball Solution

Deoderizing Refactorings

Smell	Refactoring(s)
Duplicated Code (44) [F]	Form Template Method (174), Introduce Polymorphic Creation with Factory Method (80), Chain Constructors (285), Replace One/Many Distinctions with Composite (189), Extract Composite (181), Unify Interfaces with Adapter (209), Introduce Null Object (254)
Long Method (44) [F]	Compose Method (109), Move Accumulation to Collecting Parameter (263), Replace Conditional Dispatcher with Command (164), Move Accumulation to Visitor (269), Replace Conditional Logic with Strategy (114)

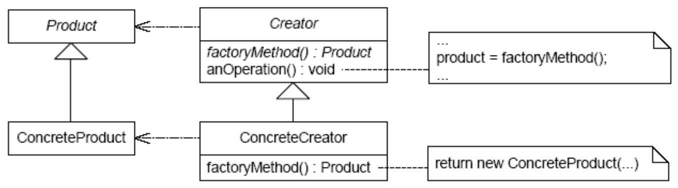
Smell	Refactoring(s)
Conditional Complexity (45)	Replace Conditional Logic with Strategy (114), Move Embellishment to Decorator (126), Replace State-Altering Conditionals with State (144), Introduce Null Object (254)
Primitive Obsession (45) [F]	Replace Type Code with Class (241), Replace State-Altering Conditionals with State (144), Replace Conditional Logic with Strategy (114), Replace Implicit Tree with Composite (154), Replace Implicit Language with Interpreter (227), Move Embellishment to Decorator (126), Encapsulate Composite with Builder (87)
Indecent Exposure (46)	Encapsulate Classes with Factory (73)
Solution Sprawl (46)	Move Creation Knowledge to Factory (63)
Alternative Classes with Different Interfaces (47) [F]	Unify Interfaces with Adapter (209)
Lazy Class (47) [F]	Inline Singleton (102)
Large Class (47) [F]	Replace Conditional Dispatcher with Command (164), Replace State-Altering Conditionals with State (144), Replace Implicit Language with Interpreter (227)
Switch Statements (47) [F]	Replace Conditional Dispatcher with Command (164), Move Accumulation to Visitor (269)
Combinatorial Explosion (47)	Replace Implicit Language with Interpreter (227)
Oddball Solution (48)	Unify Interfaces with Adapter (209)

Benefits of Composite Refactorings

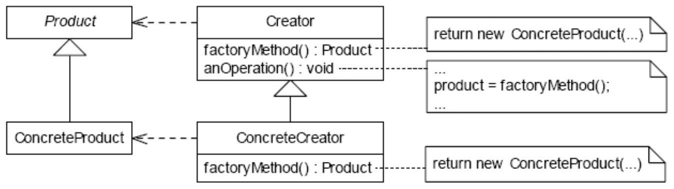
- They provide an overall plan for a refactoring sequence.
- They suggest non-obvious design directions.
- They provide insights into implementing patterns.

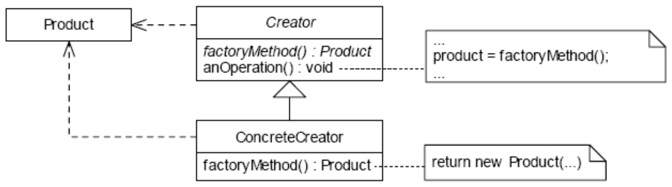
Pattern: Factory Method

Structure:


Product
Creator
factoryMethod() : Product
anOperation() : void
...
product = factoryMethod();
...
ConcreteProduct
ConcreteCreator
factoryMethod() : Product
return new ConcreteProduct(...)

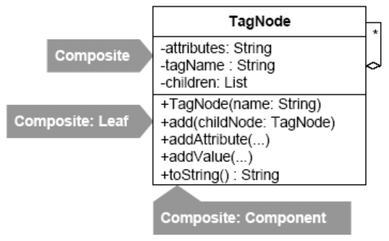
There Are Many Ways To Implement A Pattern!

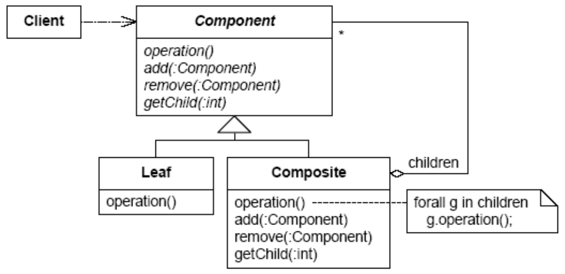

Product
Creator
factoryMethod() : Product
anOperation() : void
return new ConcreteProduct(...)
...
product = factoryMethod();
...
ConcreteProduct
ConcreteCreator
factoryMethod() : Product
return new ConcreteProduct(...)



Pattern: Composite

Sample Structures:





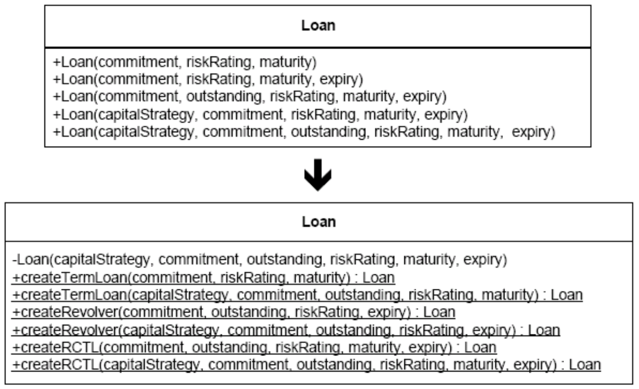
There Are Many Ways To Implement A Pattern

It seems you can't overemphasize that a pattern's Structure diagram is just an example, not a specification. It portrays the implementation we see most often. As such the Structure diagram will probably have a lot in common with your own implementation, but differences are inevitable and actually desirable. At the very least you will rename the participants as appropriate for your domain. Vary the implementation trade-offs, and your implementation might start looking a lot different from the Structure diagram. [Vlissides, C++ Report, April 1998]

Replace Constructors with Creation Methods

Constructors on a class make it hard to decide
which constructor to call during development

Replace the constructors with intention-revealing
Creation Methods that return object instances


Loan
+Loan(commitment, riskRating, maturity)
+Loan(commitment, riskRating, maturity, expiry)
+Loan(commitment, outstanding, riskRating, maturity, expiry)
+Loan(capitalStrategy, commitment, riskRating, maturity, expiry)
+Loan(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry)
Loan
-Loan(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry)
+createTermLoan(commitment, riskRating, maturity) : Loan
+createTermLoan(capitalStrategy, commitment, outstanding, riskRating, maturity) : Loan
+createRevolver(commitment, outstanding, riskRating, expiry) : Loan
+createRevolver(capitalStrategy, commitment, outstanding, riskRating, expiry) : Loan
+createRCTL(commitment, outstanding, riskRating, maturity, expiry) : Loan
+createRCTL(capitalStrategy, commitment, outstanding, riskRating, maturity, expiry) : Loan

Patterns of Refactoring

Automation First

Manual refactorings are dirt roads. Automated refactorings are highways. When deciding how to refactor, look first for the highways.

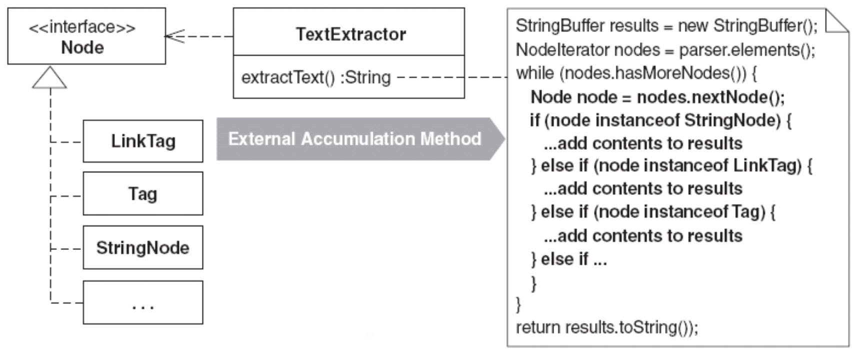
Client First

We like to refactor smelly code – yet we may only see a manual way to refactor. To find a simpler, automated way of refactoring, consider starting with a client of the smelly code.

Move Accumulation to Visitor

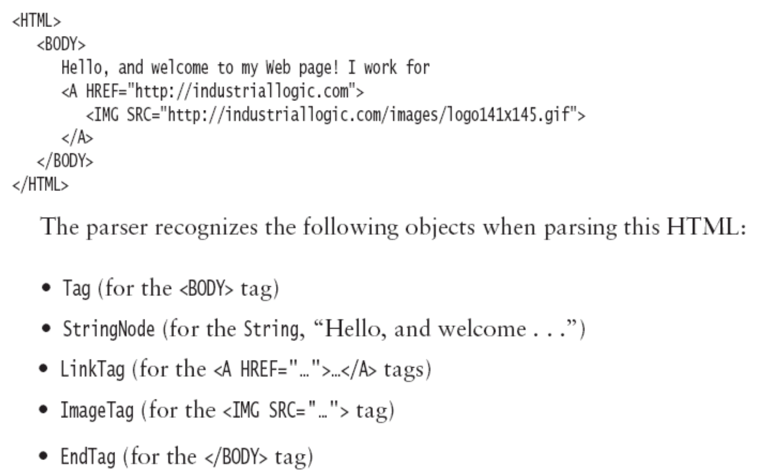
A method accumulates information from heterogeneous classes.

Move the accumulation task to a Visitor that can visit each class to accumulate the information.



```
StringBuffer results = new StringBuffer();
NodeIterator nodes = parser.elements();
while (nodes.hasMoreNodes()) {
  Node node = nodes.nextNode();
  if (node instanceof StringNode) {
    ...add contents to results
  } else if (node instanceof LinkTag) {
    ...add contents to results
  } else if (node instanceof Tag) {
    ...add contents to results
  } else if ...
  }
}
return results.toString());
```

How HTMLParser Works

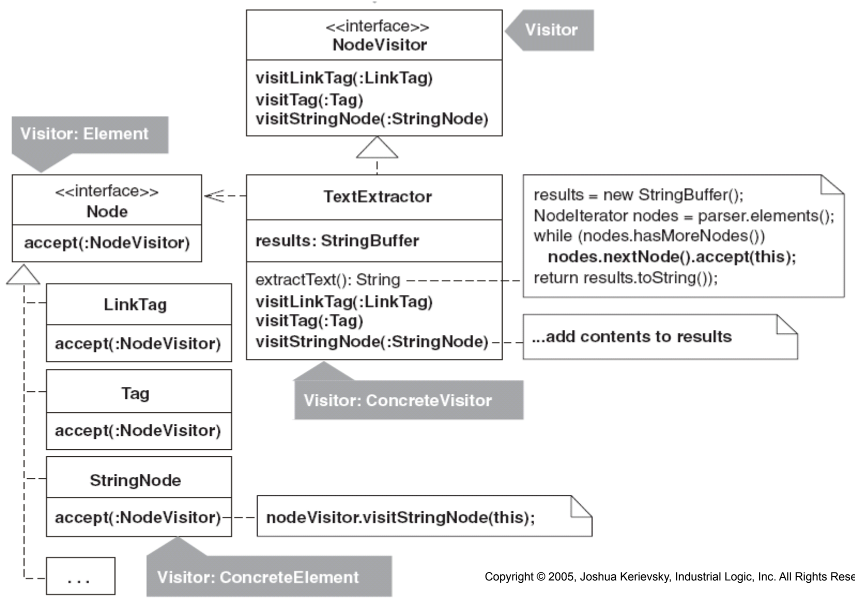


```
<HTML>
   <BODY>
      Hello, and welcome to my Web page! I work for
      <A HREF="http://industriallogic.com">
         <IMG SRC="http://industriallogic.com/images/logo141x145.gif">
      </A>
   </BODY>
</HTML>
```

The parser recognizes the following objects when parsing this HTML:

- `Tag` (for the `<BODY>` tag)
- `StringNode` (for the `String`, "Hello, and welcome . . .")
- `LinkTag` (for the `<A HREF="…">…</A>` tags)
- `ImageTag` (for the `<IMG SRC="…">` tag)
- `EndTag` (for the `</BODY>` tag)

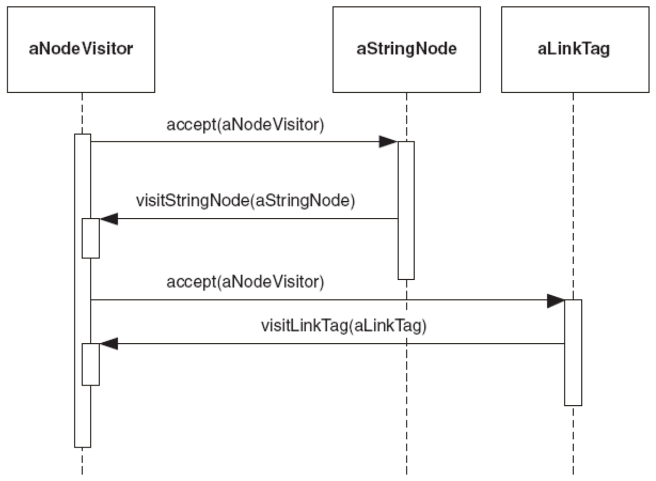
Move Accumulation to Visitor



```
results = new StringBuffer();
NodeIterator nodes = parser.elements();
while (nodes.hasMoreNodes())
  nodes.nextNode().accept(this);
return results.toString());
```

```
nodeVisitor.visitStringNode(this);
```

Move Accumulation to Visitor



Pattern	To	Towards	Away
Adapter	*Extract Adapter* (218), *Unify Interfaces with Adapter* (209)	*Unify Interfaces with Adapter* (209)	
Builder	*Encapsulate Composite with Builder* (87)		
Collecting Parameter	*Move Accumulation to Collecting Parameter* (263)		
Command	*Replace Conditional Dispatcher with Command* (164)	*Replace Conditional Dispatcher with Command* (164)	
Composed Method	*Compose Method* (109)		
Composite	*Replace One/Many Distinctions with Composite* (189), *Extract Composite* (181), *Replace Implicit Tree with Composite* (154)		*Encapsulate Composite with Builder* (87)
Creation Method	*Replace Constructors with Creation Methods* (55)		
Decorator	*Move Embellishment to Decorator* (126)	*Move Embellishment to Decorator* (126)	
Factory	*Move Creation Knowledge to Factory* (63), *Encapsulate Classes with Factory*		

	(73)		
Factory Method	*Introduce Polymorphic Creation with Factory Method* (80)		
Interpreter	*Replace Implicit Language with Interpreter* (227)		
Iterator			*Move Accumulation to Visitor* (269)
Null Object	*Introduce Null Object* (254)		
Observer	*Replace Hard-Coded Notifications with Observer* (200)	*Replace Hard-Coded Notifications with Observer* (200)	
Singleton	*Limit Instantiation with Singleton* (250)		*Inline Singleton* (102)
State	*Replace State-Altering Conditionals with State* (144)	*Replace State-Altering Conditionals with State* (144)	
Strategy	*Replace Conditional Logic with Strategy* (114)	*Replace Conditional Logic with Strategy* (114)	
Template Method	*Form Template Method* (174)		
Visitor	*Move Accumulation to Visitor* (269)	*Move Accumulation to Visitor* (269)	

More Information

- refactoring.com
- industriallogic.com/xp/refactoring
- industriallogic.com/training
- refactoring@yahoogroups.com
- eclipse.org
- intellij.com