**Creational design pattern (ABSPF)**

Abstract Factory: Set of methods to make various objects.

Builder: Make and return one object various ways.

Factory method: Methods to make and return components of one object various ways.

Prototype Design pattern: Make new objects by cloning the objects which you set as prototypes.

Singleton: A class distributes the only instance of itself.

**Structural Design pattern (ABCDP2F)**

Adapter: A class extends another class , takes in an object , and Makes the taken object behave like the extended class.

Bridge: A abstraction and implementation are in different hierarchies.

Composite: Assemble groups of objects with the same signature.va

Decorator: One class takes in another class, both of which extend the same abstract class and adds functionality.

Façade: One class has a method that performs a complex process calling several other classes.

Flyweight: The reusable and variable parts of a class are broken into two classes to save resources.

Proxy: one class controls the creation of and access to objects in another class.

**Behavioral design pattern**

Chain of responsibility: A method called in one class can move up a hierarchy to find an object that can properly execute the method.

Command: An object encapsulates everything needed to execute a method in another object.

Interpreter: Define a macro language and syntax, parsing input into objects which perform the correct operations.

Mediator: An object distributes the communication between 2 or more objects.

Memento: An object stores another object state.

Observer: An objects notifies other objects if it changes.

State: An object appears to change its class when the class it passes through to switches itself for a related class.

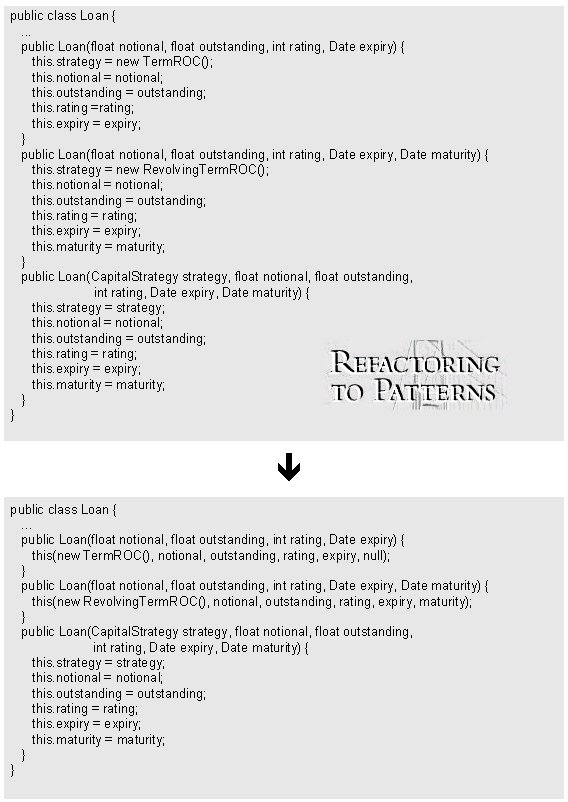
Strategy: An object controls which of a family of methods is called. Each method is in its own class that extends a common base class.

Template: An Abstract class defines various methods and has one non-overridden method which calls the various methods.

Visitor: One or more related classes have the same method, which calls a method specific for themselves in another class.

**Chain Constructors**

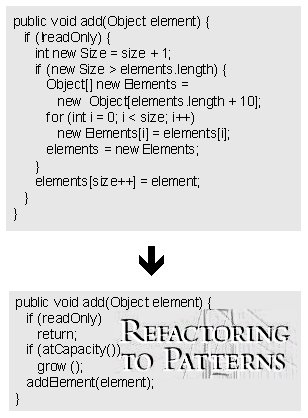
*You have Constructors that contain duplicate code.*

**Chain the constructors together to obtain the least duplicate code.**

# Compose Method

*You can't rapidly understand a method's logic.*

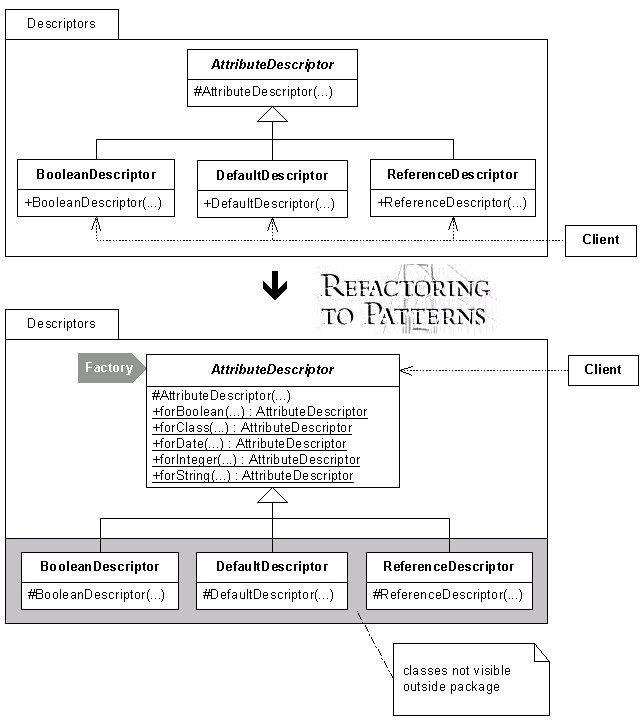
**Transform the logic into a small number of  
intention-revealing steps at the same level of detail.**



# Encapsulate Classes with Factory

*Clients directly instantiate classes that reside  
in one package and implement a common interface.*

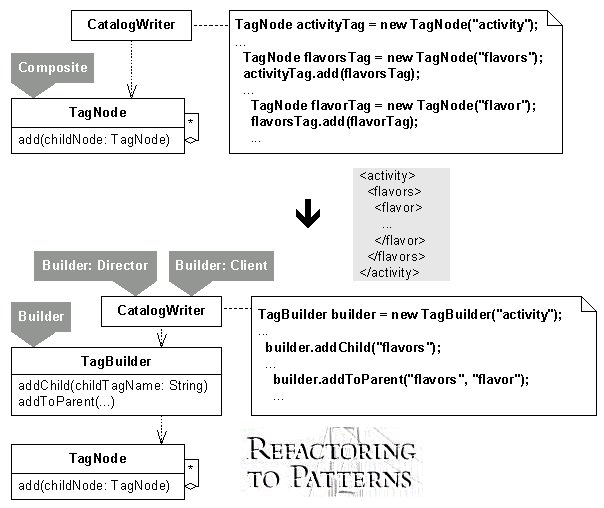
**Make the class constructors non-public and let clients  
create instances of them using a Factory.**



# Encapsulate Composite with Builder

*Building a Composite is repetitive, complicated or error-prone.*

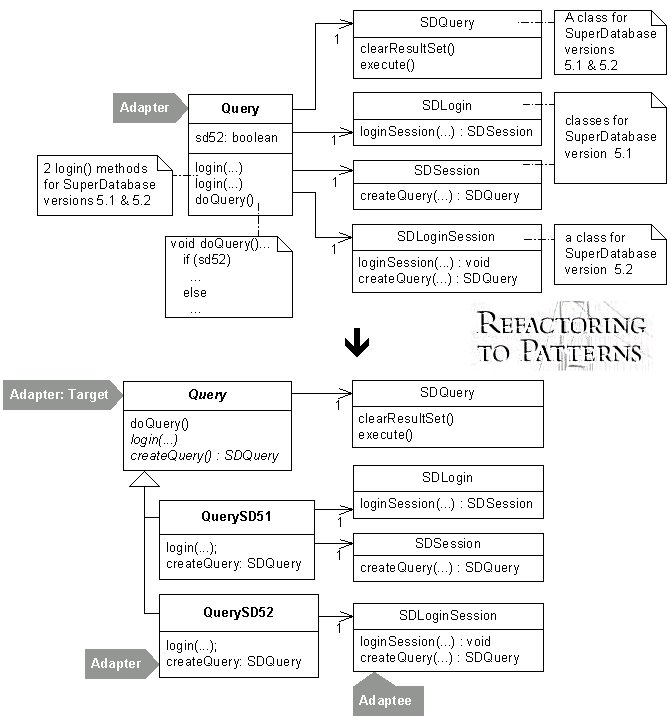
**Simplify the build by letting a Builder handle the details.**



# Extract Adapter

*One class adapts multiple versions of a  
component, library, API or other entity.*

**Extract an Adapter for a single version of the  
component, library, API or other entity.**

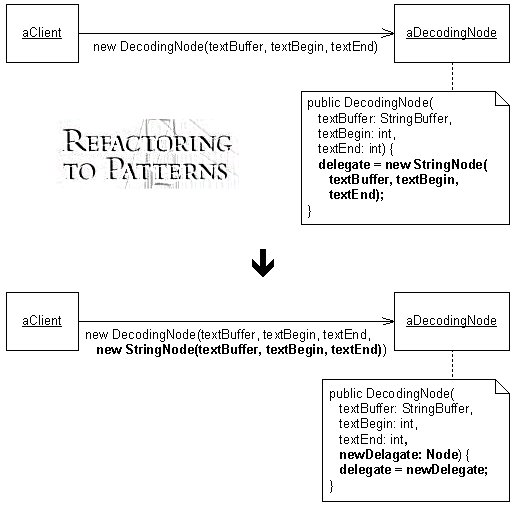


|  |  |
| --- | --- |
| Extract Composite *Subclasses in a hierarchy implement the same Composite.*  **Extract a superclass that implements the Composite.**  https://industriallogic.com/xp/refactoring/images/extractComposite.jpg |  |

# Extract Parameter

*A method or constructor assigns a field  
to a locally instantiated value.*

**Assign the field to a parameter supplied by a client  
by extracting one half of the assignment statement to a parameter.**

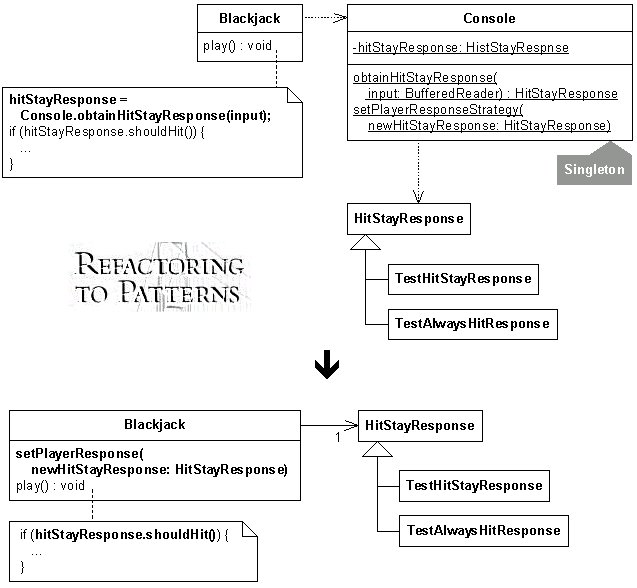


|  |  |
| --- | --- |
| Form Template Method *Two methods in subclasses perform similar steps in the same order, yet the steps are different.*  **Generalize the methods by extracting their steps into methods with identical signatures, then pull up the generalized methods to form a Template Method.**  https://industriallogic.com/xp/refactoring/images/formTemplateMethod.jpg |  |

# Inline Singleton

*Code needs access to an object but  
doesn't need a global point of access to it.*

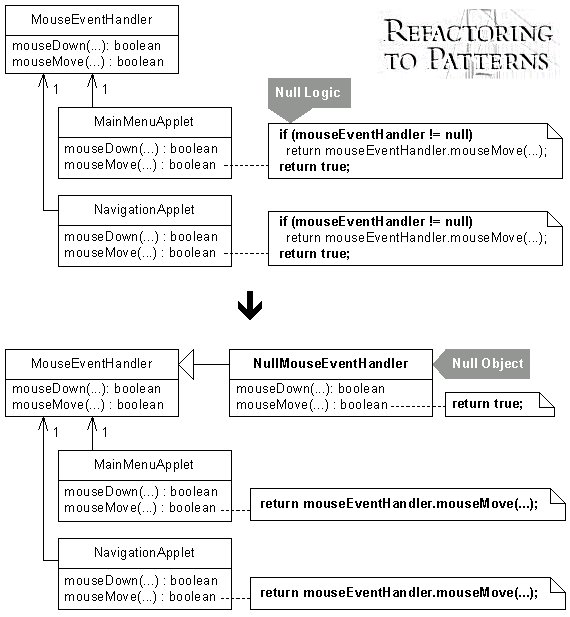
**Move the Singleton's features to a class that stores and  
provides access to the object. Delete the Singleton.**



# Introduce Null Object

*Logic for dealing with a null field or variable  
is duplicated throughout your code.*

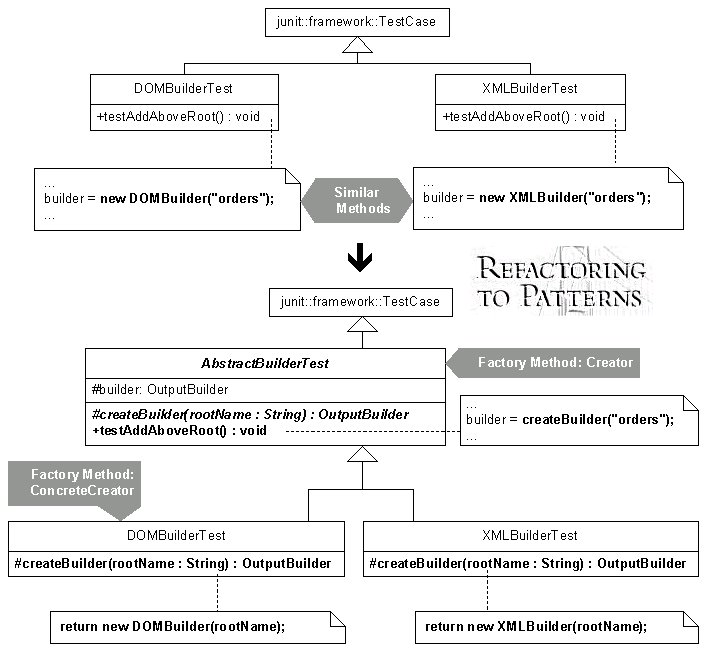
**Replace the null logic with a Null Object: an object  
that provides the appropriate null behavior.**



# Introduce Polymorphic Creation with Factory Method

*Classes in a hierarchy implement a method similarly  
except for an object creation step.*

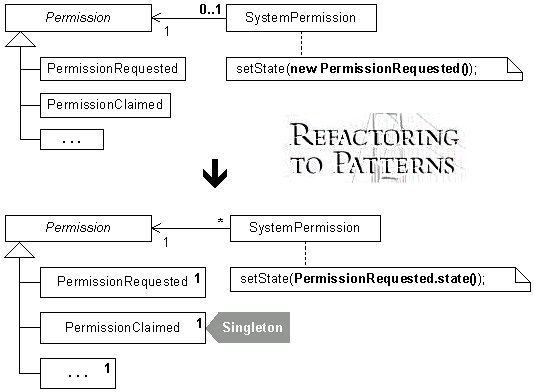
**Make a single superclass version of the method that  
calls a Factory Method to handle the instantiation.**



# Limit Instantiation with Singleton

*Your code creates multiple instances of an object and  
that uses too much memory or slows system performance.*

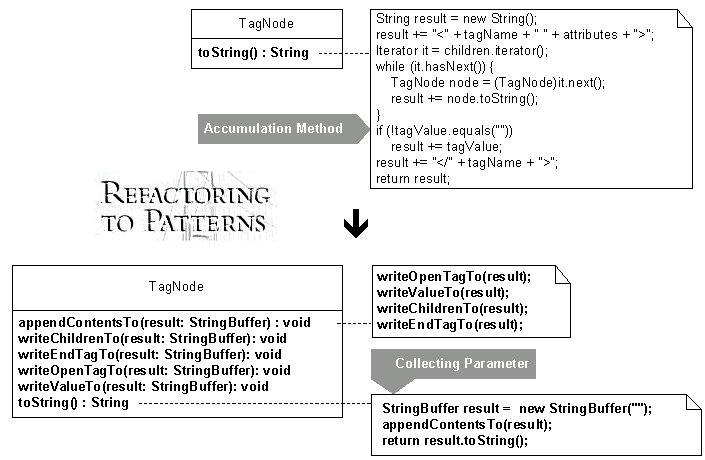
**Replace the multiple instances with a Singleton.**



# Move Accumulation to Collecting Parameter

*You have a single bulky method  
that accumulates information to a local variable.*

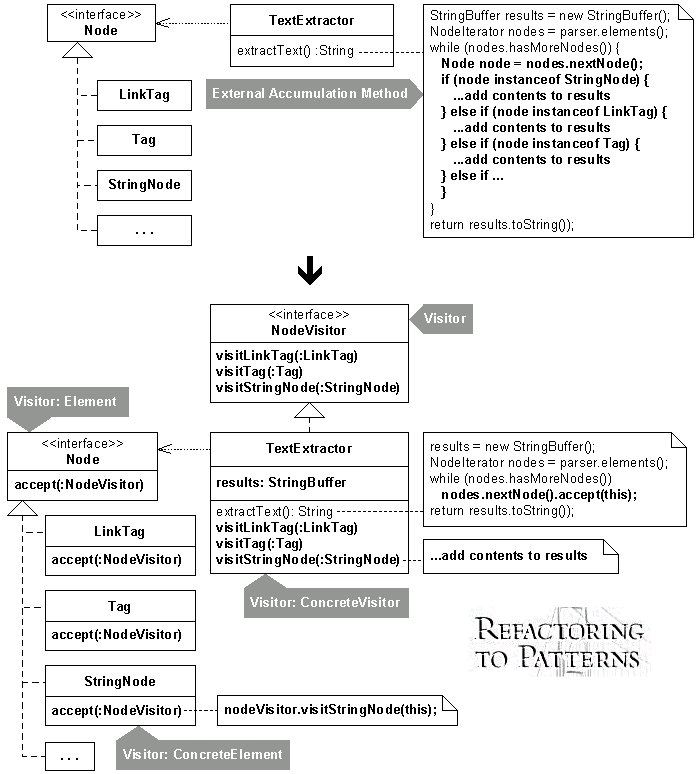
**Accumulate results to a Collecting Parameter  
that gets passed to extracted methods.**



# Move Accumulation to Visitor

*A method accumulates information from heterogeneous classes.*

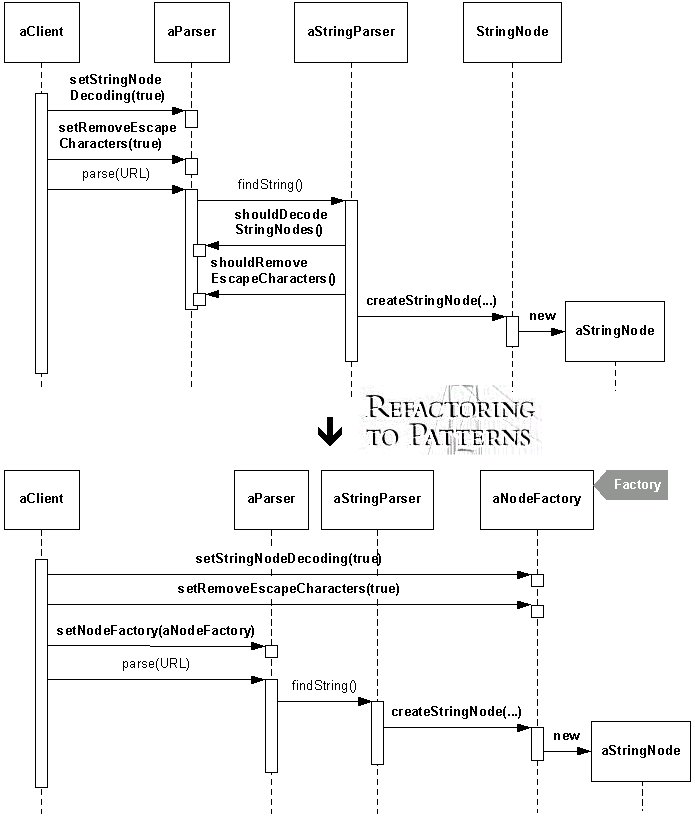
**Move the accumulation task to a Visitor that can  
visit each class to accumulate the information.**



# Move Creation Knowledge to Factory

*Data and code used to instantiate a class  
is sprawled across numerous classes.*

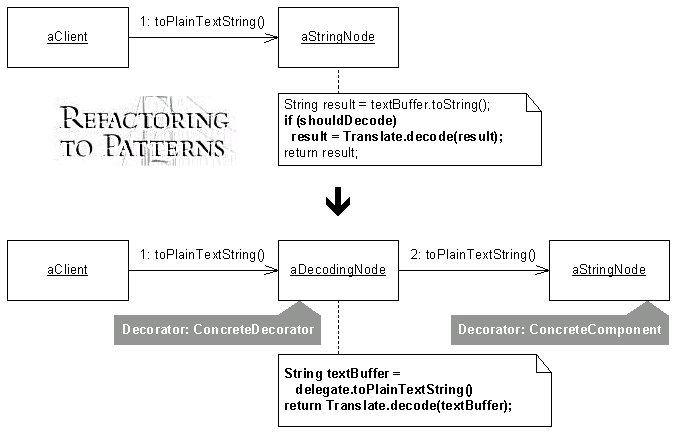
**Move the creation knowledge into a single Factory class.**



# Move Embellishment to Decorator

*Code provides an embellishment to a class' core responsibility.*

**Move the embellishment code to a Decorator.**

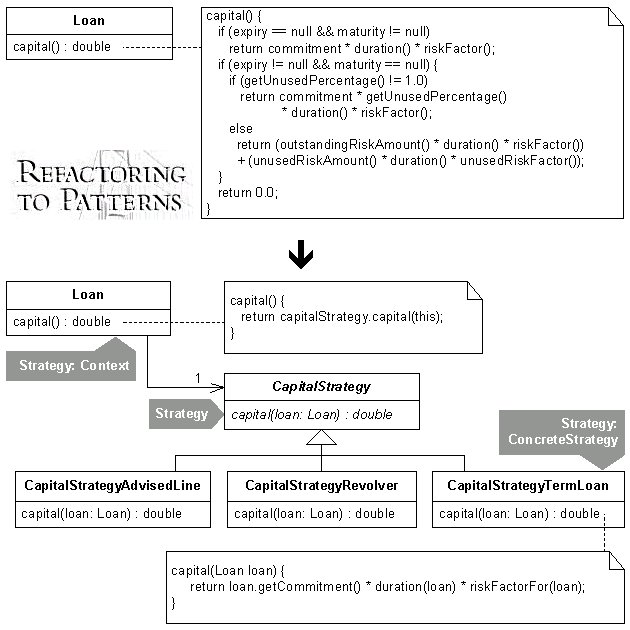


|  |  |
| --- | --- |
| Replace Conditional Dispatcher with Command *Conditional logic is used to dispatch requests and execute actions.*  **Create a Command for each action. Store the Commands in a collection and replace the conditional logic with code to fetch and execute Commands.**  https://industriallogic.com/xp/refactoring/images/conditionDispatcherWithCommand.jpg |  |

# 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.**

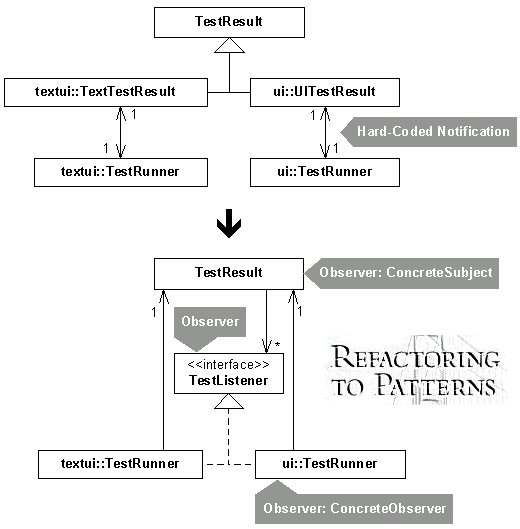


|  |  |
| --- | --- |
| Replace Constructors with Creation Methods *Constructors on a class make it hard to decide which constructor to call during development.*  **Replace the constructors with intention-revealing Creation Methods that return object instances.**  https://industriallogic.com/xp/refactoring/images/multipleconstructorwithcreator.jpg |  |

# Replace Hard-Coded Notifications with Observer

*Subclasses are hard-coded to notify a single instance of another class.*

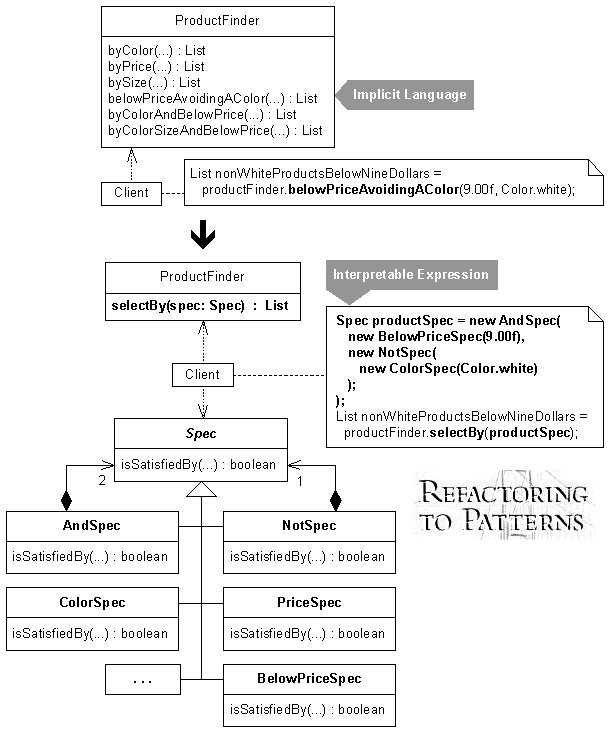
**Remove the subclasses by making their superclass capable of notifying  
one or more instances of any class that implements an Observer interface.**



# Replace Implicit Language with Interpreter

*Numerous methods on a class combine elements of an implicit language.*

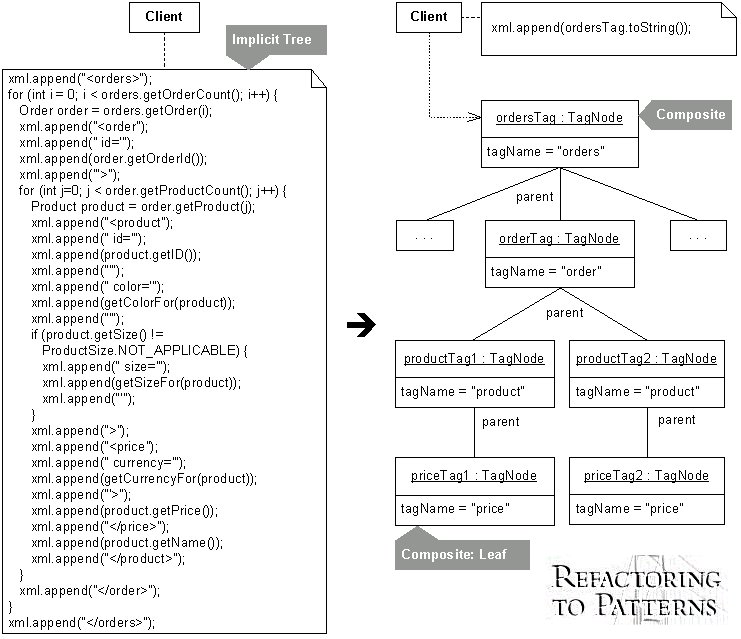
**Define classes for elements of the implicit language so that  
instances may be combined to form interpretable expressions.**



# Replace Implicit Tree with Composite

*You implicitly form a tree structure, using a  
primitive representation, such as a String.*

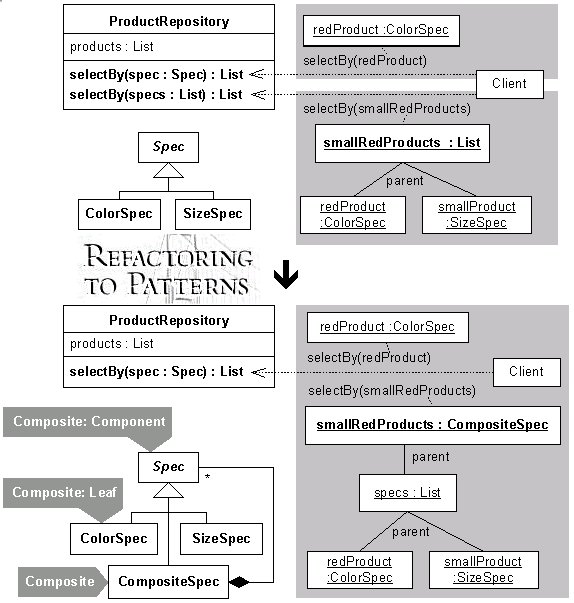
**Replace your primitive representation with a Composite.**



# Replace One/Many Distinctions with Composite

*A class processes single and multiple objects  
using separate pieces of code.*

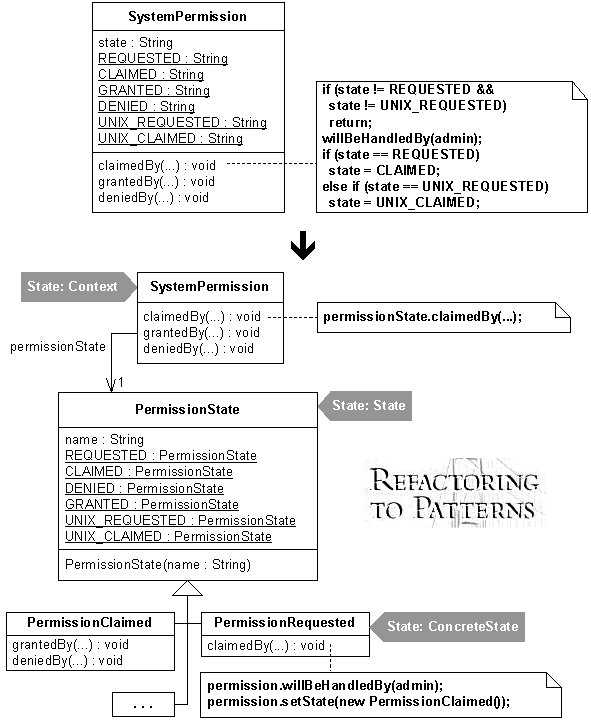
**Use a Composite to produce one piece of code  
capable of handling single or multiple objects.**



# Replace State-Altering Conditionals with State

*The conditional expressions that control  
an object's state transitions are complex.*

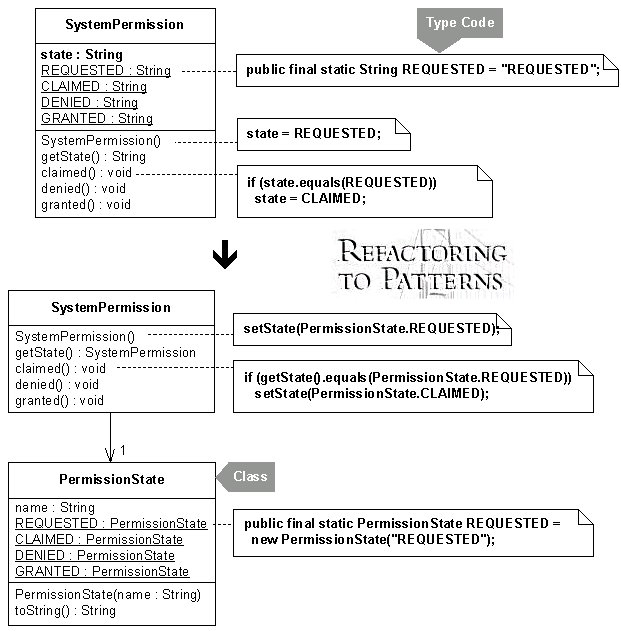
**Replace the conditionals with State classes that handle  
specific states and transitions between them.**



# Replace Type Code with Class

*A field's type (e.g. a String or int) fails to protect it  
from unsafe assignments and invalid equality comparisons.*

**Constrain the assignments and equality comparisons  
by making the type of the field a class.**



|  |  |
| --- | --- |
| Unify Interfaces *You need a superclass and/or interface to have the same interface as a subclass.*  **Find all public methods on the subclass that are missing on the superclass/interface. Add copies of these missing methods to the superclass, altering each one to perform null behavior.**  https://industriallogic.com/xp/refactoring/images/commonInterface.jpg |  |

# Unify Interfaces with Adapter

*Clients interact with two classes, one of which has a preferred interface.*

**Unify the interfaces with an Adapter.**

