Software Reengineering
Refactoring To Patterns

Martin Pinzger & Andy Zaidman
Delft University of Technology

TU Delft

SERG

Outline

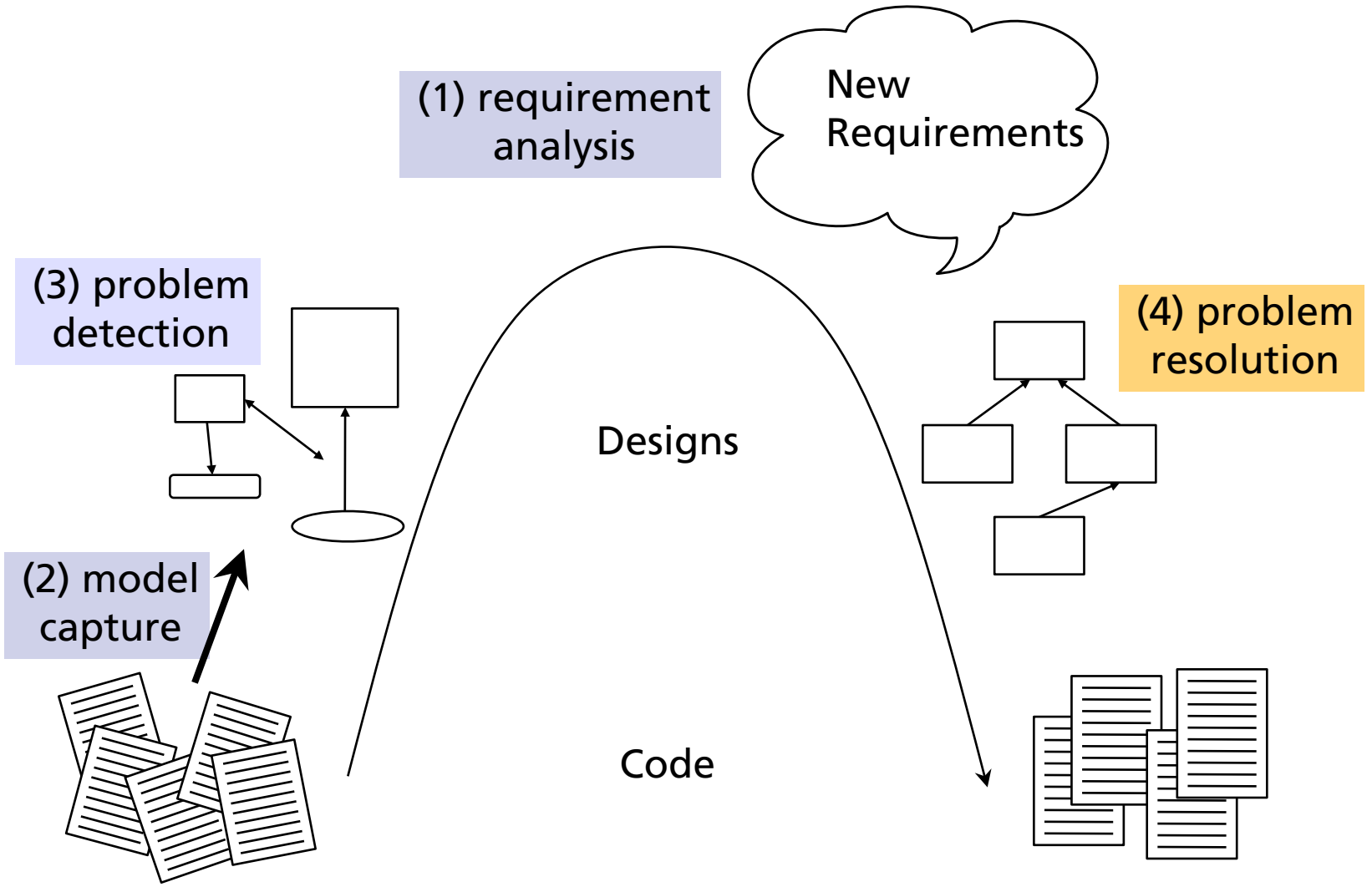
Introduction

Design Patterns

Refactoring to Patterns

Conclusions

The Reengineering Life-Cycle



Object-Oriented Design Patterns

„Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.“

[Gamma, Helm, Johnson, Vlissides 1995]

Design Patterns Idea

Reoccurring design problems ➪ idea:

Do not solve the problem again

... use an existing pattern that can be parameterized

from which concrete solutions can be derived

Reuse design knowledge

Vocabulary for communicating design

Elements of Design Patterns

Pattern name

- Design Vocabulary

Problem

- When to apply the pattern?
- List of preconditions

Solution

- Abstract description of a design problem and how a general arrangements of classes and objects solves it

Consequences

- Impact on flexibility, extensibility, portability, etc.

Describing Design Patterns

Elements of the description

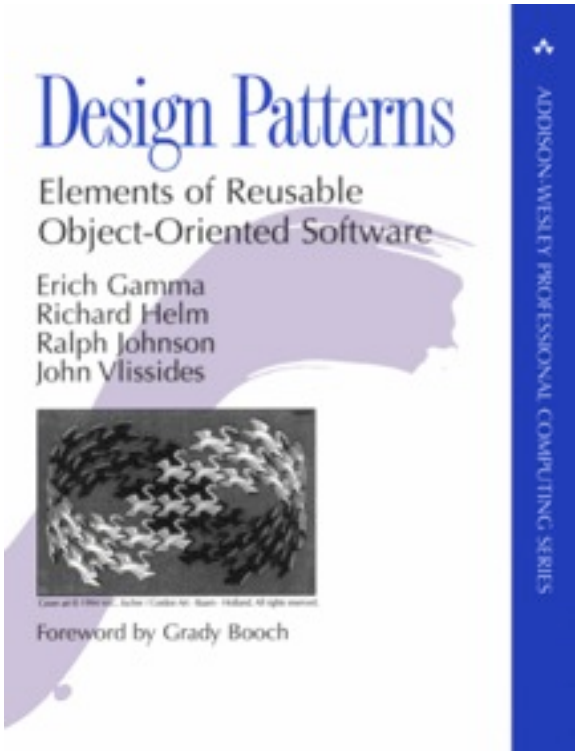
- Pattern name and classification
- Intent, Also known as, Motivation
- Applicability
- Structure, Participants, Collaborations
- Consequences
- Implementation
- Sample code, Known uses, Related patterns

Design Patterns (GoF) Classification

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter
				Template Method
	Object	Abstract Factory	Adapter (object)	Chain of Responsibility
		Prototype	Bridge	Command
		Builder	Composite	Iterator
		Singleton	Decorator	Mediator
			Facade	Memento
			Flyweight	Observer
			Proxy	State
				Strategy
				Visitor

Additional Reading

Design Patterns: Elements of Reusable Object-Oriented Software
Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides



Online: http://sourcemaking.com/design_patterns

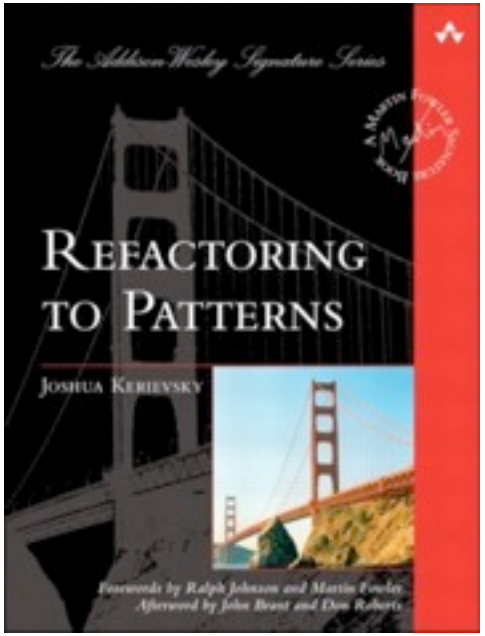
Refactoring to Patterns

Design Patterns and Refactorings

"There is a natural relation between patterns and refactoring. Patterns are where you want to be; refactorings are ways to get there from somewhere else."

[Fowler 1999]

Refactoring to Patterns: Book



A set of composite refactorings to refactor towards design patterns

Joshua Kerievsky, Addison-Wesley, 2005

Some patterns online:
http://www.informit.com/articles/article.aspx?p=1398607

Overview of Refactorings

Creation

Move functionality to create instances of complex classes to Factory and Builder classes

Simplification

Simplify the source code by introducing strategy, state, commands, composites, and decorators

Generalization

Transform specific code into general-purpose code by introducing Template Method, Composite, Observer, Adapter, and Interpreter

Overview of Refactorings (cont.)

Protection

Protect existing code from modifications by introducing a Singleton and Null Object

Accumulation

Accumulating information by introducing a Visitor

Refactoring to Patterns: Examples

Factory Method

- Introduce Polymorphic Creation with Factory Method

State

- Factor out State

Observer

- Replace Hard-Coded Notifications with Observer

Example 1: Creation

Introduce Polymorphic Creation with Factory Method

Example BuilderTest

```
public class DOMBuilderTest extends TestCase {
    private OutputBuilder builder;
    public void testAddAboveRoot() {
        String invalidResult = “<orders> ... </orders>”;
        builder = new DOMBuilder(“orders”);
        builder.addBelow(“order”);
        try {
            builder.addAbove(“customer”);
        } catch (RuntimeException re) {
            fail(“Message”, re);
        }
    }
}

public class XMLBuilderTest extends TestCase {
    private OutputBuilder builder;
    public void testAddAboveRoot()
        // the same
        builder = new XMLBuilder(“orders”);
        // the same
    }
}
```

Introduce Factory Method

Problem

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

Solution

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

Factory Method Pattern

Intent

Define an interface for creating an object, but let subclasses decide which class to instantiate.

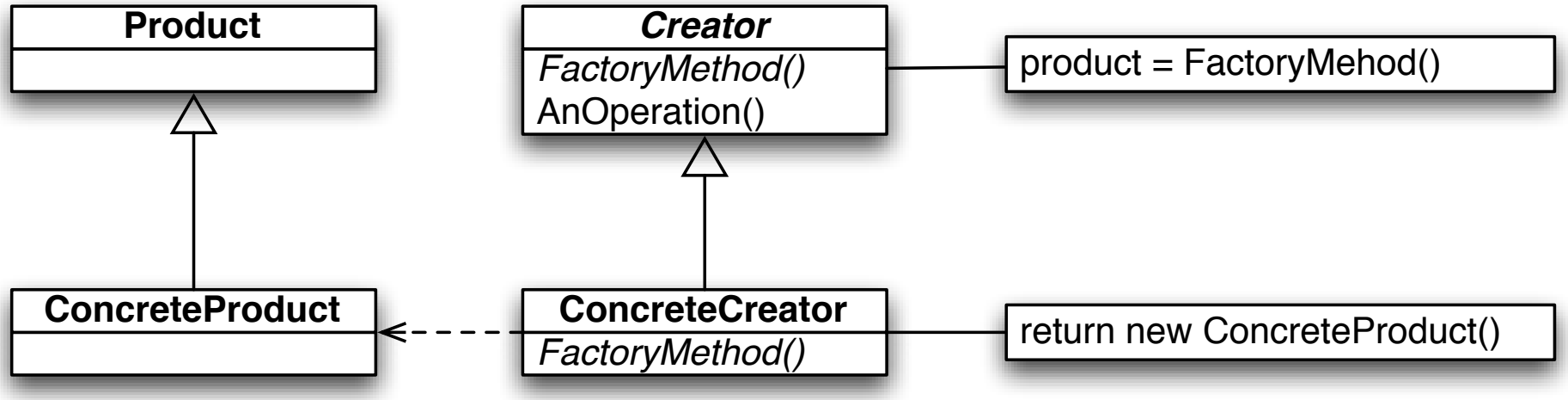
Motivation

Encapsulate the knowledge of which subclass to create to a factory method

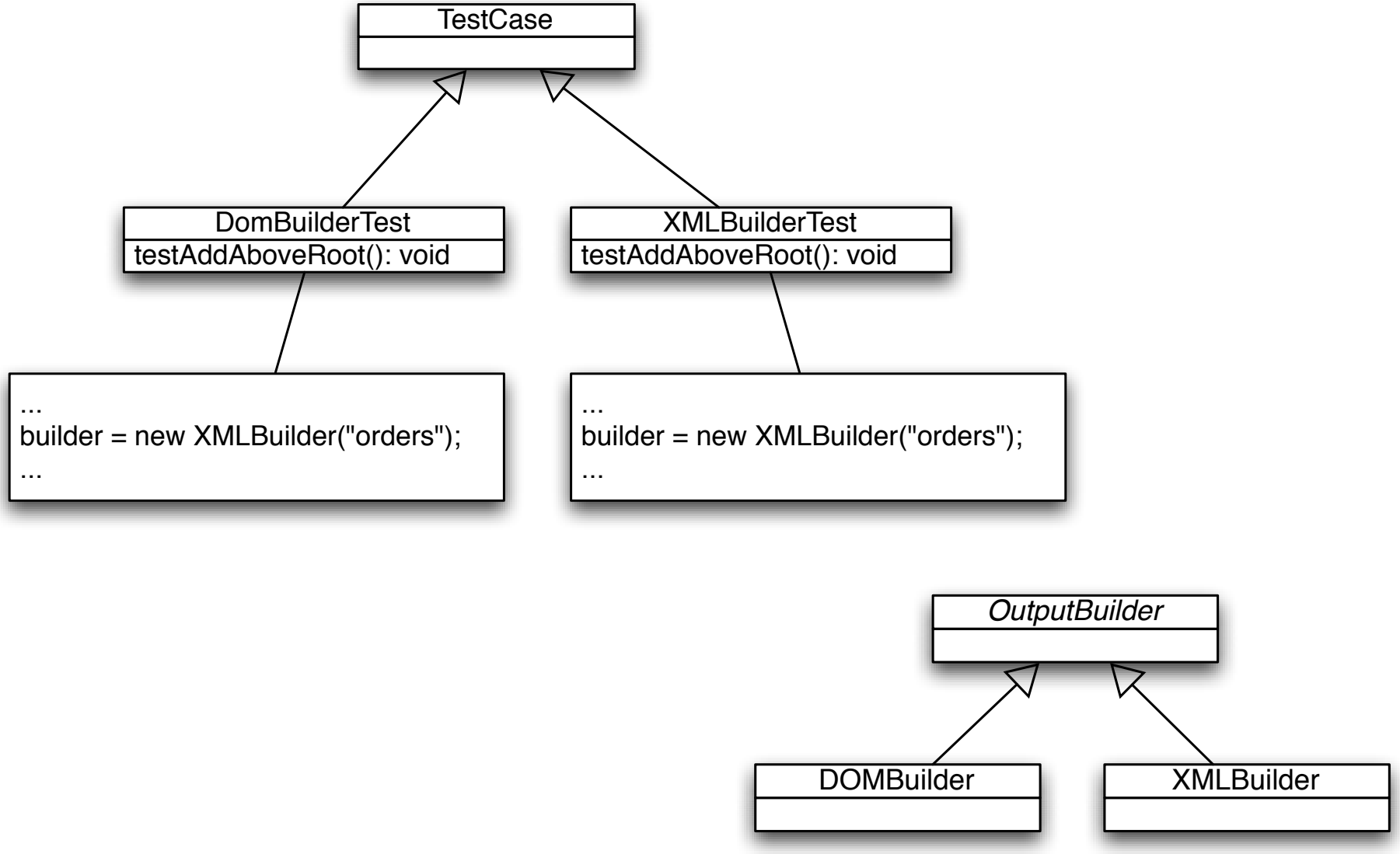
Applicability

Class wants its subclasses to specify the objects it creates

Factory Method Pattern: Structure



Example BuilderTest: Class Diagram



Introduce Factory Method: Mechanics

Extract Method on creation code

- Repeat in sibling subclasses

Extract Superclass to create Creator

Pull Up Method and Form Template Method of testAddAboveRoot()

Add abstract Factory Method

- Implement concrete factory method in subclasses

Extract Method on creation code

```
public class DOMBuilderTest extends TestCase {
    private OutputBuilder builder;
    private OutputBuilder createBuilder(String rootName) {
        return new DOMBuilder(“orders”);
    }
    public void testAddAboveRoot() {
        String invalidResult = “<orders> ... </orders>”;
        builder = createBuilder(“orders”);
        builder.addBelow(“order”);
        ...
    }
}
public class XMLBuilderTest extends TestCase {
    private OutputBuilder builder;
    private OutputBuilder createBuilder(String rootName) {
        return new XMLBuilder(“orders”);
    }
    public void testAddAboveRoot()
        ...
        builder = createBuilder(“orders”);
        ...
    }
}
```

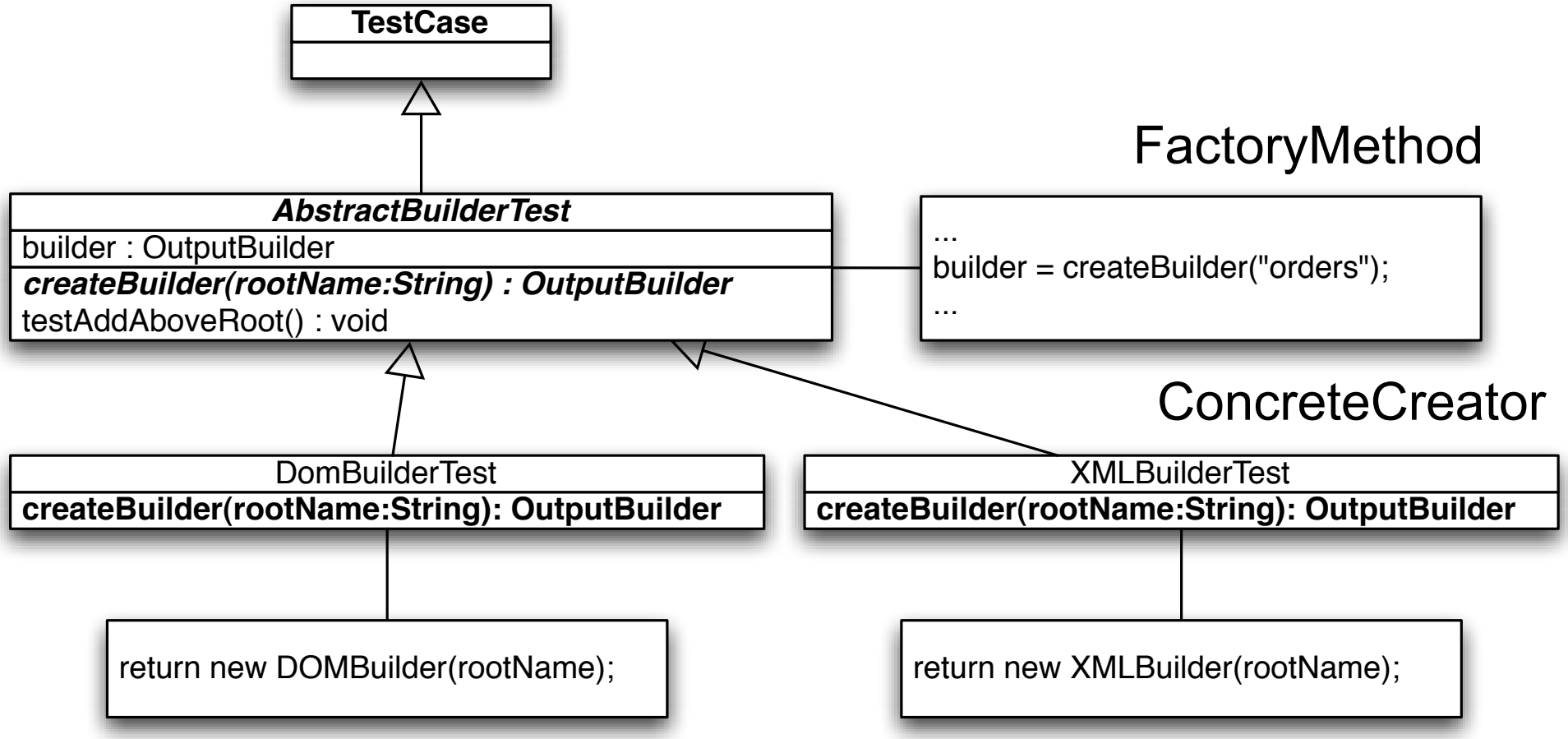
Extract Superclass

```
public abstract class AbstractBuilderTest extends TestCase {

}



public class DOMBuilderTest extends AbstractBuilderTest {
    ...
}

public class XMLBuilderTest extends AbstractBuilderTest {
    ...
}
```

Form Template Method and Pull Up Field

```
public abstract class AbstractBuilderTest extends TestCase {
    protected OutputBuilder builder;

    protected abstract OutputBuilder createdBuilder (String rootName);
}
```

Result Refactoring BuilderTest



Benefits and Liabilities

+ Reduces duplication resulting from a custom object creation step

+ Effectively communicates where creation occurs and how it may be overridden

+ Enforces what type a class must implement to be used by a Factory Method

- May require you to pass unnecessary parameters to some Factory Method implementers

Example 2: Simplicfication

Factor out State

Example SystemPermission

```
public class SystemPermission...
  private SystemProfile profile;
  private SystemUser requestor;
  private SystemAdmin admin;
  private boolean isGranted;
  private String state;

  public final static String REQUESTED = "REQUESTED";
  public final static String CLAIMED = "CLAIMED";
  public final static String GRANTED = "GRANTED";
  public final static String DENIED = "DENIED";

  public SystemPermission(SystemUser requestor, SystemProfile profile) {
    this.requestor = requestor;
    this.profile = profile;
    state = REQUESTED;
    isGranted = false;
    notifyAdminOfPermissionRequest();
  }
  ...
```

Example SystemPermission (cont.)

```
  ...
  public void claimedBy(SystemAdmin admin) {
    if (!state.equals(REQUESTED)) return;
    willBeHandledBy(admin);
    state = CLAIMED;
  }

  public void deniedBy(SystemAdmin admin) {
    if (!state.equals(CLAIMED)) return;
    if (!this.admin.equals(admin)) return;
    isGranted = false;
    state = DENIED;
    notifyUserOfPermissionRequestResult();
  }

  public void grantedBy(SystemAdmin admin) {
    if (!state.equals(CLAIMED)) return;
    if (!this.admin.equals(admin)) return;
    state = GRANTED;
    isGranted = true;
    notifyUserOfPermissionRequestResult();
  }
}
```

Factor out State

Problem

How to make a class extensible whose behavior depends on a complex evaluation of its state?

Solution

Eliminate complex conditional code over an object's state by applying the State Pattern

State Pattern

Intent

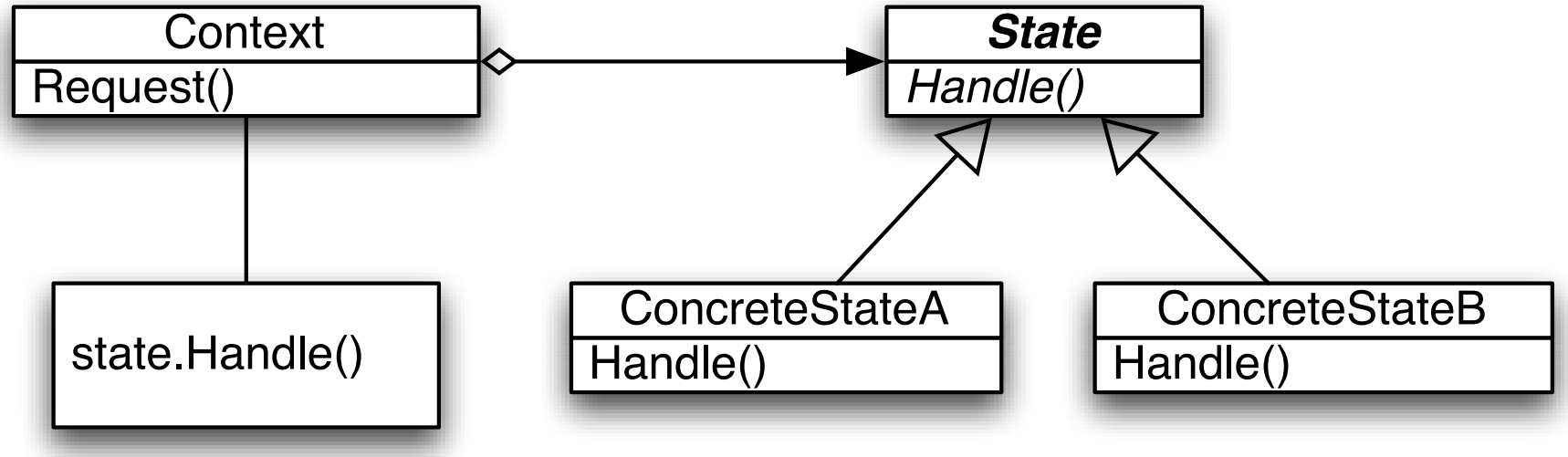
Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

Applicability

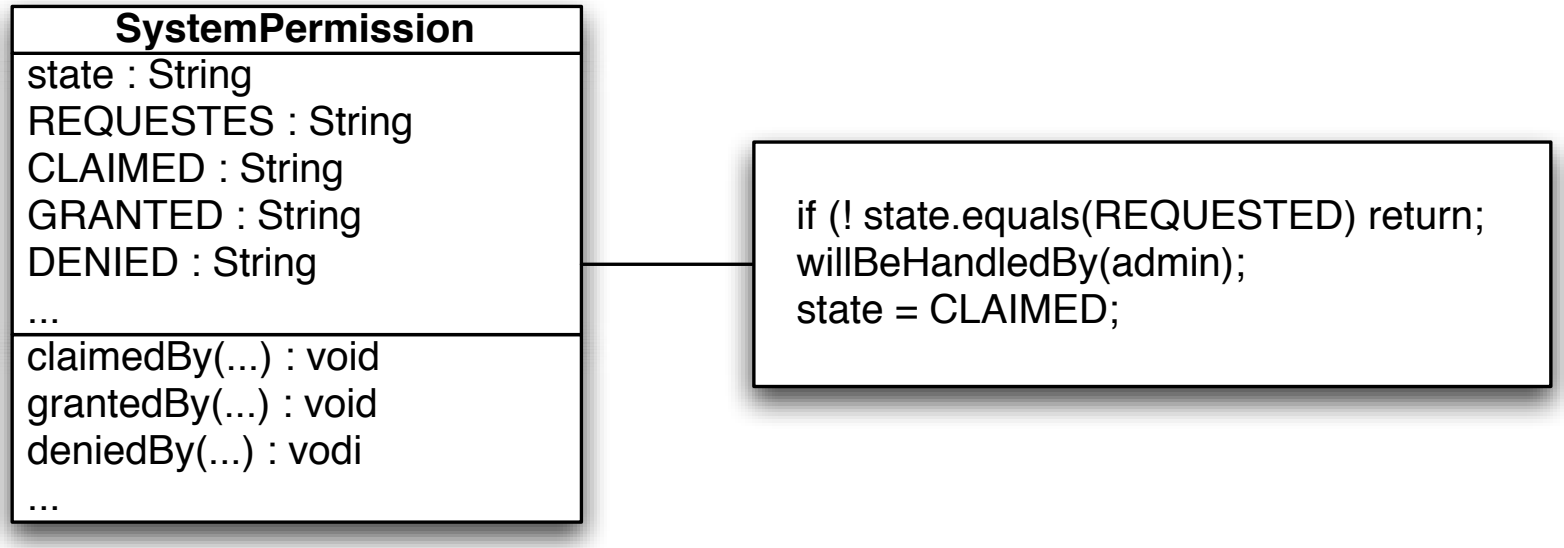
An object's behavior depends on its state, and it must change its behavior at run-time depending on that state

Operations have large, multipart conditional statements that depend on the object's state.

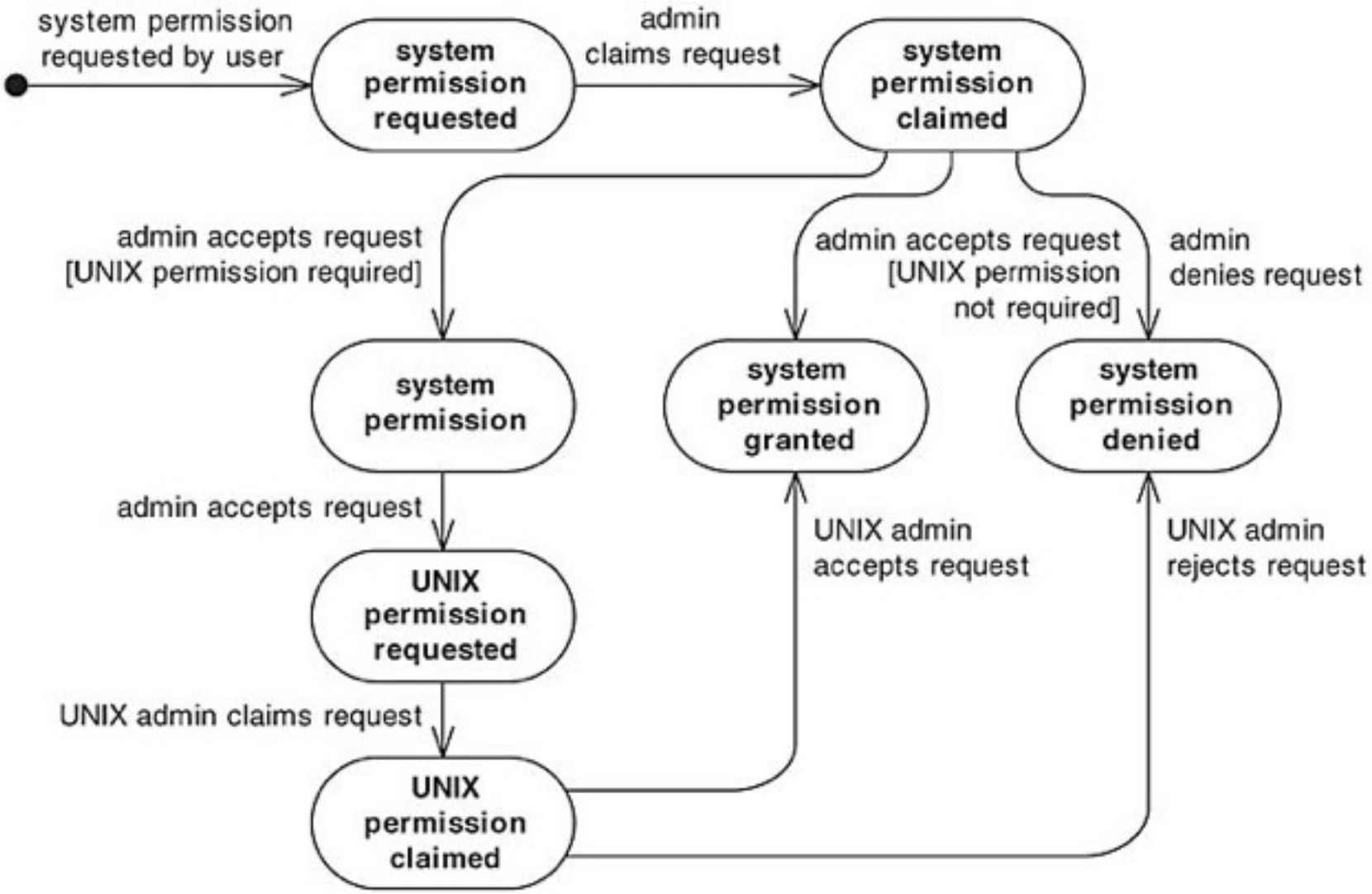
State Pattern: Structure



Example SystemPermission: Class Diagram



Change: Adding two more states



Replace Conditionals with State: Mechanics

Replace Type Code with Class

On the original state field in the context class

Extract Subclass

To produce one subclass per constant (state) and declare state superclass as abstract

Move Method

On context class methods that change the value of original state variable

Replace Type Code with State Class

```
public class PermissionState {
  private String name;

  private PermissionState(String name) {
    this.name = name;
  }

  public final static PermissionState REQUESTED = new PermissionState("REQUESTED");
  public final static PermissionState CLAIMED = new PermissionState("CLAIMED");
  public final static PermissionState GRANTED = new PermissionState("GRANTED");
  public final static PermissionState DENIED = new PermissionState("DENIED");
  public final static PermissionState UNIX_REQUESTED =
    new PermissionState("UNIX_REQUESTED");
  public final static PermissionState UNIX_CLAIMED =
    new PermissionState("UNIX_CLAIMED");

  public String toString() {
    return name;
  }
}
```

Replace Type Code with State Class (cont.)

```
public class SystemPermission {
  private PermissionState permissionState;

  public SystemPermission(SystemUser requestor, SystemProfile profile) {
    ...
    setState(PermissionState.REQUESTED);
    ...
  }

  private void setState(PermissionState state) {
    permissionState = state;
  }

  public void claimedBy(SystemAdmin admin) {
    if (!getState().equals(PermissionState.REQUESTED)
     && !getState().equals(PermissionState.UNIX_REQUESTED))
       return;
    ...
  }
  ...
}
```

Extract Subclass for each State (Constant)

```
public abstract class PermissionState {
  private String name;
  private PermissionState(String name) {
    this.name = name;
  }
  public final static PermissionState REQUESTED = new PermissionRequested();
  ...
}

public class PermissionRequested extends PermissionState {
  public PermissionRequested() {
    super(“REQUESTED”);
  }
}
public class PermissionClaimed extends PermissionState { ... }
public class PermissionDenied extends PermissionState { ... }
public class PermissionGranted extends PermissionState { ... }
public class UnixPermissionRequested extends PermissionState { ... }
public class UnixPermissionClaimed extends PermissionState { ... }
```

Move State Trans. Logic to State Class

```
public abstract class PermissionState...
  public void claimedBy(SystemAdmin admin, SystemPermission permission) {
    if (!permission.getState().equals(REQUESTED) &&
        !permission.getState().equals(UNIX_REQUESTED))
      return;
    permission.willBeHandledBy(admin);
    if (permission.getState().equals(REQUESTED))
      permission.setState(CLAIMED);
    else if (permission.getState().equals(UNIX_REQUESTED)) {
      permission.setState(UNIX_CLAIMED);
    }
  }
```

Move State Trans. Logic (cont.)

```
public class SystemPemission {
  ...
  void setState(PermissionState state) { // now has package-level visibility
    permissionState = state;
  }

  public void claimedBy(SystemAdmin admin) {
    state.claimedBy(admin, this);
  }

  void willBeHandledBy(SystemAdmin admin) {
    this.admin = admin;
  }
}
```

Move State Trans. Logic to Subclasses

```
public class PermissionRequested extends PermissionState {
  ...
  public void claimedBy(SystemAdmin admin, SystemPermission permission) {
    permission.willBeHandledBy(admin);
    permission.setState(CLAIMED);
  }
}
```

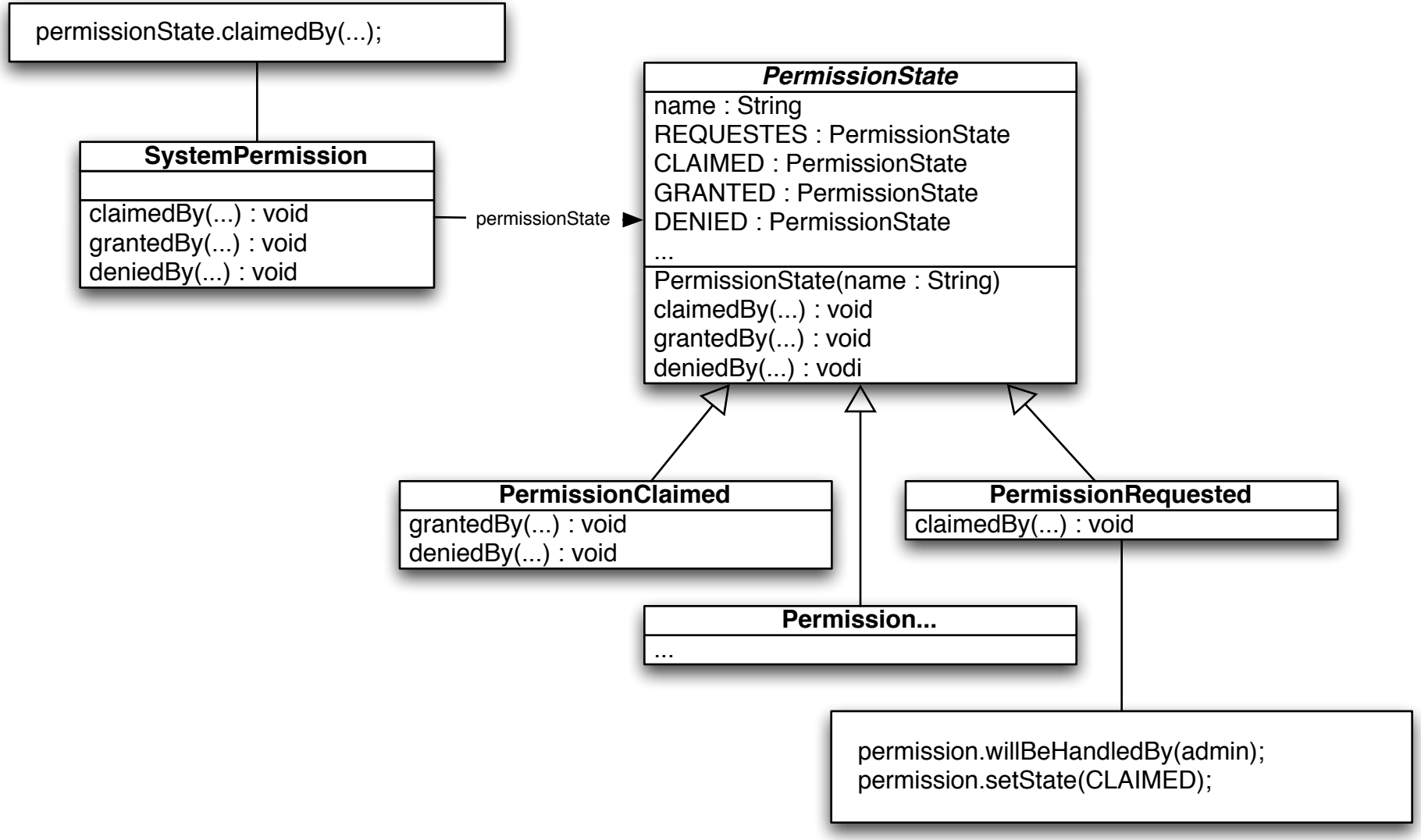
Move State-Transition to Subclasses (cont.)

```
public class PermissionClaimed extends PermissionState...
  public void deniedBy(SystemAdmin admin, SystemPermission permission) {
    if (!permission.getAdmin().equals(admin))
      return;
    permission.setIsGranted(false);
    permission.setIsUnixPermissionGranted(false);
    permission.setState(DENIED);
    permission.notifyUserOfPermissionRequestResult();
  }

  public void grantedBy(SystemAdmin admin, SystemPermission permission) {
    if (!permission.getAdmin().equals(admin))
      return;
    if (permission.getProfile().isUnixPermissionRequired()
        && !permission.isUnixPermissionGranted()) {
      permission.setState(UNIX_REQUESTED);
      permission.notifyUnixAdminsOfPermissionRequest();
      return;
    }
    permission.setState(GRANTED);
    permission.setIsGranted(true);
    permission.notifyUserOfPermissionRequestResult();
  }
```

Move State-Transition to Subclasses (cont.)

```
public abstract class PermissionState {
  public String toString();
  public void claimedBy(SystemAdmin admin, SystemPermission permission) {}
  public void deniedBy(SystemAdmin admin, SystemPermission permission) {}
  public void grantedBy(SystemAdmin admin, SystemPermission permission) {}
}
```

Result Refactoring SystemPermission



Benefits and Liabilities

+ Reduces or removes state-changing conditional logic

+ Simplifies complex state-changing logic

+ Provides a good bird's-eye view of state-changing logic

- Complicates design when state transition logic is already easy to follow

Example 3: Generalization

Replace Hard-Coded Notifications with Observer

Example: TestResult

```
public class UITestResult extends TestResult {
  private TestRunner runner;
  UITestResult(TestRunner runner) {
    this.runner = runner;
  }
  public synchronized void addFailure(Test test, Throwable t) {
    super.addFailure(test, t);
    runner.addFailure(this, test, t); // notification of TestRunner
  }
}
public class TestRunner extends Frame {
  private TestResult testResult;
  protected TestResult createdTestResult() {
    return new UITestResult(this);
  }
  public synchronized void runSuite() {
    testResult = createTestResult();
    testSuite.run(testResult);
  }
  public void addFailure(TestResult result, Test test, Throwable t) {
    // display the test result
  }
}
```

Replace Notifications with Observer

Problem

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

Solution

Remove the subclass by making their superclass capable of notifying one or more instances of any class that implements an observer interface

Observer Pattern

Intend

Maintain a dependency between a central object (Subject) and multiple dependent objects (Observers)

Motivation

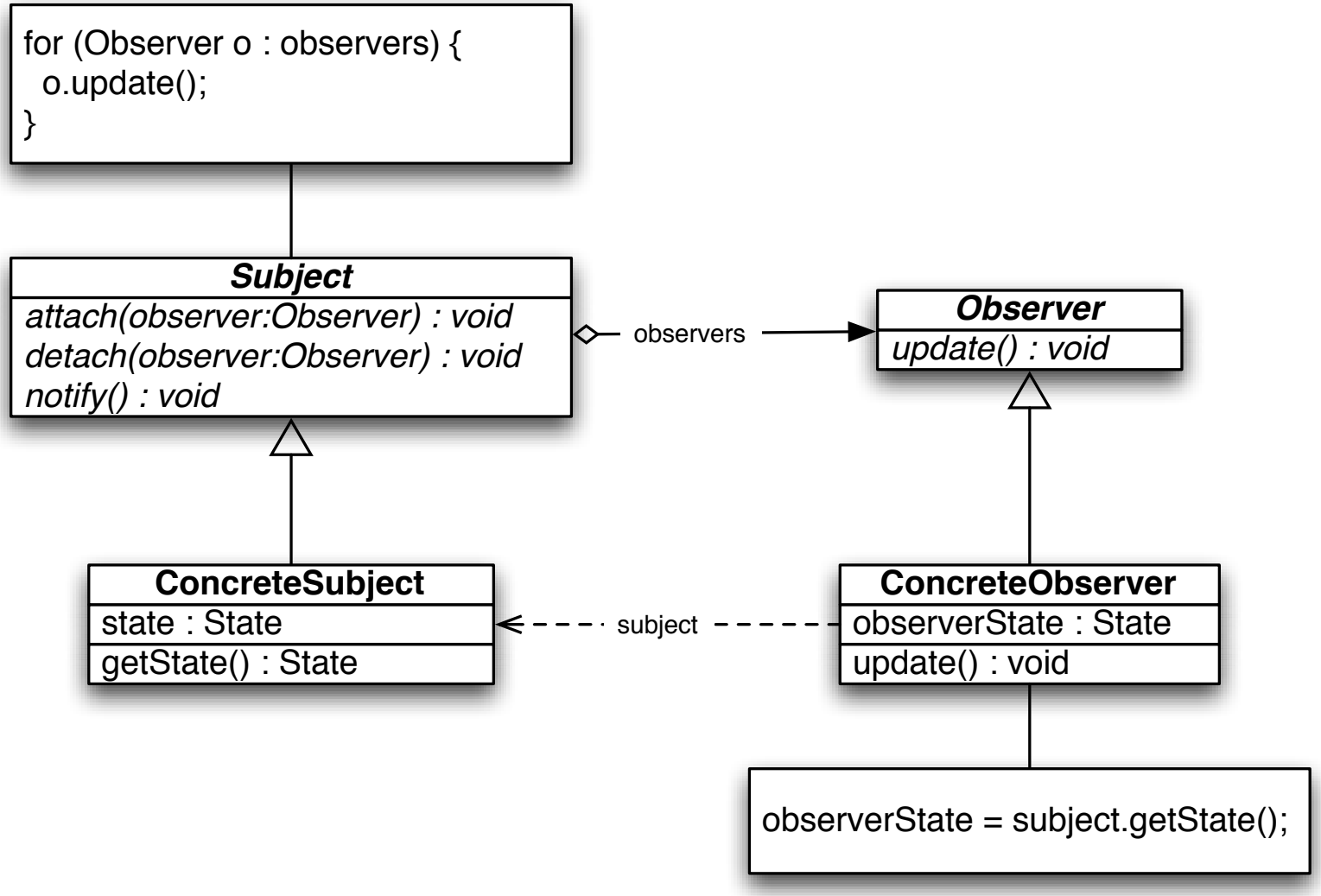
Decouple a subject from its observers

Applicability

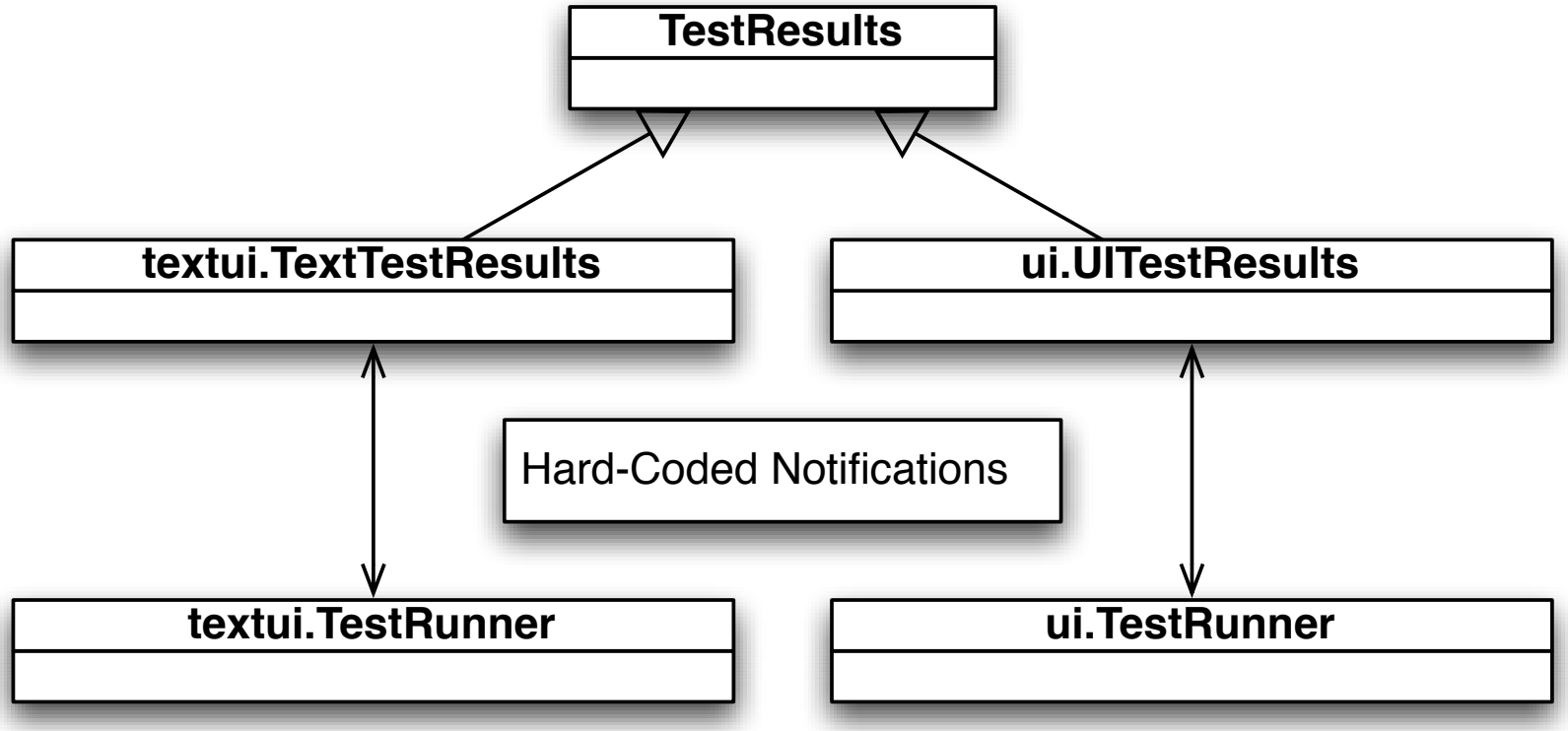
When an instance must notify more than one receiver instance,

E.g., when there are various views (Observers) on the same model instance (Subject)

Observer Pattern: Structure



Example TestResult: Class Diagram



Repl. Notific. with Observer: Mechanics

Move custom behavior to receiver

Extract Interface on a receiver to produce an observer interface

- Only methods called by its notifier

Make every receiver implement the observer interface

Pull Up Method on notification methods in notifier classes

Update notification implementation

Add collection to subject and update observer to register and communicate with the subject

Move Custom Behavior to Receiver

```
package textui;
public class TextTestResult extends TestResult {
  private TestRunner runner;
  TextTestResult(TestRunner runner) {
    this.runner = runner;
  }
  public synchronized void addError(Test test, Throwable t) {
    super.addError(test, t);
    runner.addError(this, test, t);
  }
}
```

```
package textui;
public class TestRunner ...
  protected TextTestResult createdTestResult() {
    return new TextTestResult(this);
  }

  public void addError(TestResult result, Test test, Throwable t) {
    System.out.println("E");
  }
}
```

Extract Observer Interface

```
public class TextTestResult extends TestResult ...
  public synchronized void addError(Test test, Throwable t) {
    super.addError(test, t);
    runner.addError(this, test, t);
  }
  public synchronized void addFailure(Test test, Throwable t) {
    super.addFailure(test, t);
    runner.addFailure(this, test, t);
  }
  public synchronized void startTest(Test test) {
    super.startTest(test);
    runner.startTest(this, test);
  }
}
public interface TestListener {
  public void addError(TestResult testResult, Test test, Throwable t);
  public void addFailure(TestResult testResult, Test test, Throwable t);
  public void startTest(TetResult testResult, Test test);
  public void endTest(TestResult testResult, Test test);
}
public class TestRunner implements TestListener ...
  public void endTest(TestResult testResult, Test test) { }
}
```

Make Receiver Implement the Interface

```
package ui;
public class TestRunner extends Frame implements TestListener {
  ...
}


package ui;
public class UITestResult extends TestResult ...
  protected TestListener runner;

  UITestResult(TestListener runner) {
    this.runner = runner;
  }
}

package textui;
public class TextTestResult extends TestResult ...
  protected TestListener runner;

  TextTestResult(TestListener runner) {
    this.runner = runner;
  }
}
```

Pull Up Methods in Observers

```
public class TestResult ...
  protected TestListener runner;
  public TestResult() {
    failures = new ArrayList<TestFailure>();
    errors = new ArrayList<TestError>();
    runTests = 0;
    stop = false;
  }
  public TestResult(TestListener runner) {
    this();
    this.runner = runner;
  }
  public synchronized void addError(Test test, Throwable t) {
    errors.add(new TestError(test, t));
    runner.addError(this, test, t);
  }
  ...
}
```

```
package ui;
public class UITestResult extends TestResult { }

package textui;
public class TextTestResult extends TestResult { }
```

Update Observers to Work with Subject

```
package textui;
public class TestRunner implements TestListener ...
  protected TestResult createTestResult() {
    return new TestResult(this);
  }
}

package ui;
public class TestRunner implements TestListener ...
  protected TestResult createTestResult() {
    return new TestResult(this);
  }
  public synchronized void runSuite() {
    ...
    TestResult result = createTestResult();
    testSuite.run(testResult);
}
```

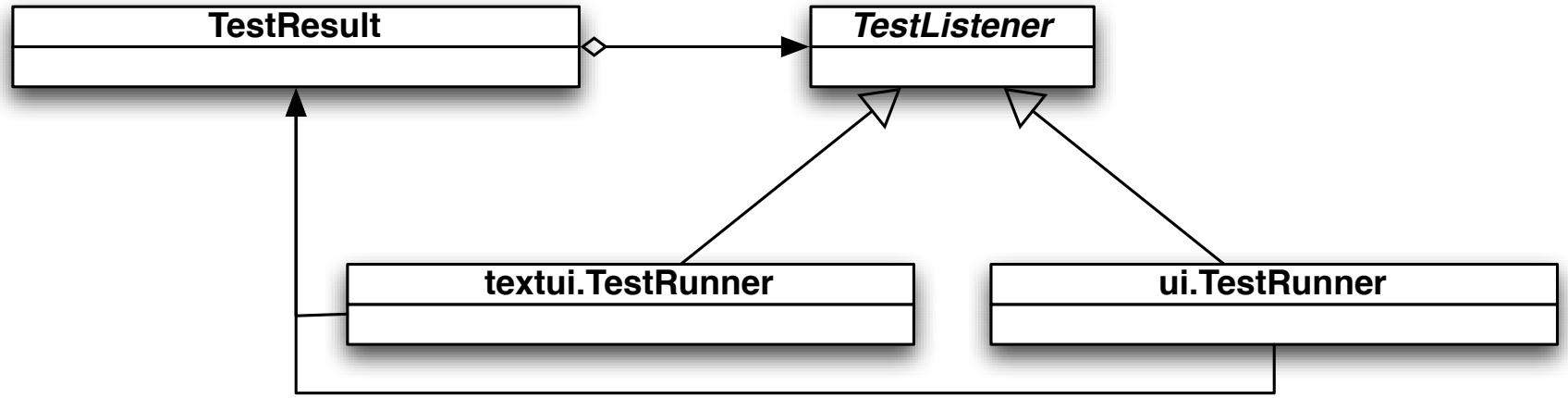
Update Subject to Notify many Observers

```
public class TestResult ...
  protected List<TestListener> observers = new ArrayList<TestListener>();

  public void addObserver(TestListener observer) {
    observers.add(observer);
  }

  public synchronized void addError(Test test, Throwable t) {
    errors.add(new TestError(test, t));
    for (TestListener observer : observers) {
      observer.addError(this, test, t);
    }
  }
  ...
}
```

Update Observer to Register with Subject

```
package textui;
public class TestRunner implements TestListener ...
  protected TestResult createTestResult() {
    TestResult testResult = new TestResult(this);
    testResult.addObserver(this);
    return testResult;
  }
}
```

Result: Refactoring TestResult



Benefits and Liabilities

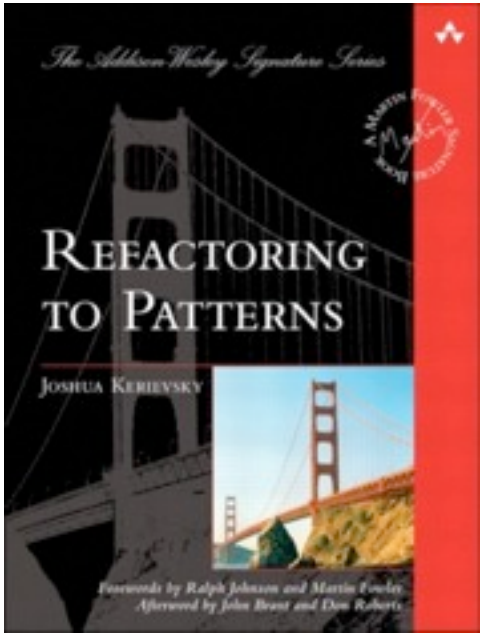
+ Loosely couples a subject with its observers

+ Supports one or many observers

- Complicate design
  - When a hard-coded notification will suffice
  - When you have cascading notifications

- May cause memory leaks when observers aren't removed from their subjects

More Refactorings To Patterns



A set of composite refactorings to refactor towards design patterns

Joshua Kerievsky, Addison-Wesley, 2005

Some patterns online:
http://www.informit.com/articles/article.aspx?p=1398607

Conclusions

Refactoring to Patterns

- Known and tested solutions for similar design problems
- Encapsulates and simplifies logic
- Increases extensibility (interfaces, loose coupling)

But, don’t overdo it

- Only use a pattern when it (really) makes sense