

Some Design Patterns

Alexandre Bergel
abergel@dcc.uchile.cl
26/05/2010



Design Patterns: Elements of Reusable
Object-Oriented Software
*Erich Gamma, Richard Helm, Ralph Johnson, John
M. Vlissides, 1994*

Adapter Pattern

How do you use a class that provide the right features but the wrong interface?

Introduce an adapter.

An adapter converts the interface of a class into another interface clients expect.

The client and the adapted object remain independent.

An adapter adds an extra level of indirection.

Also known as Wrapper

Adapter Pattern

Examples

A `WrappedStack` adapts `java.util.Stack`, throwing an `AssertionException` when `top()` or `pop()` are called on an empty stack.

An `ActionListener` converts a call to `actionPerformed()` to the desired handler method.

Consequences

The client and the adapted object *remain independent*

An adapter adds *an extra level of indirection*

Adapter Pattern Example

```
class LegacyRectangle implements Shape
{
    public void draw(int x, int y, int w, int h)
    {
        System.out.println("rectangle at (" + x + ', ' + y + ") with
width " + w
        + " and height " + h);
    }
}

class Rectangle implements Shape
{
    private LegacyRectangle adaptee = new LegacyRectangle();
    public void draw(int x1, int y1, int x2, int y2)
    {
        adaptee.draw(Math.min(x1, x2), Math.min(y1, y2), Math.abs(x2 -
x1),
        Math.abs(y2 - y1));
    }
}
```

Proxy Pattern

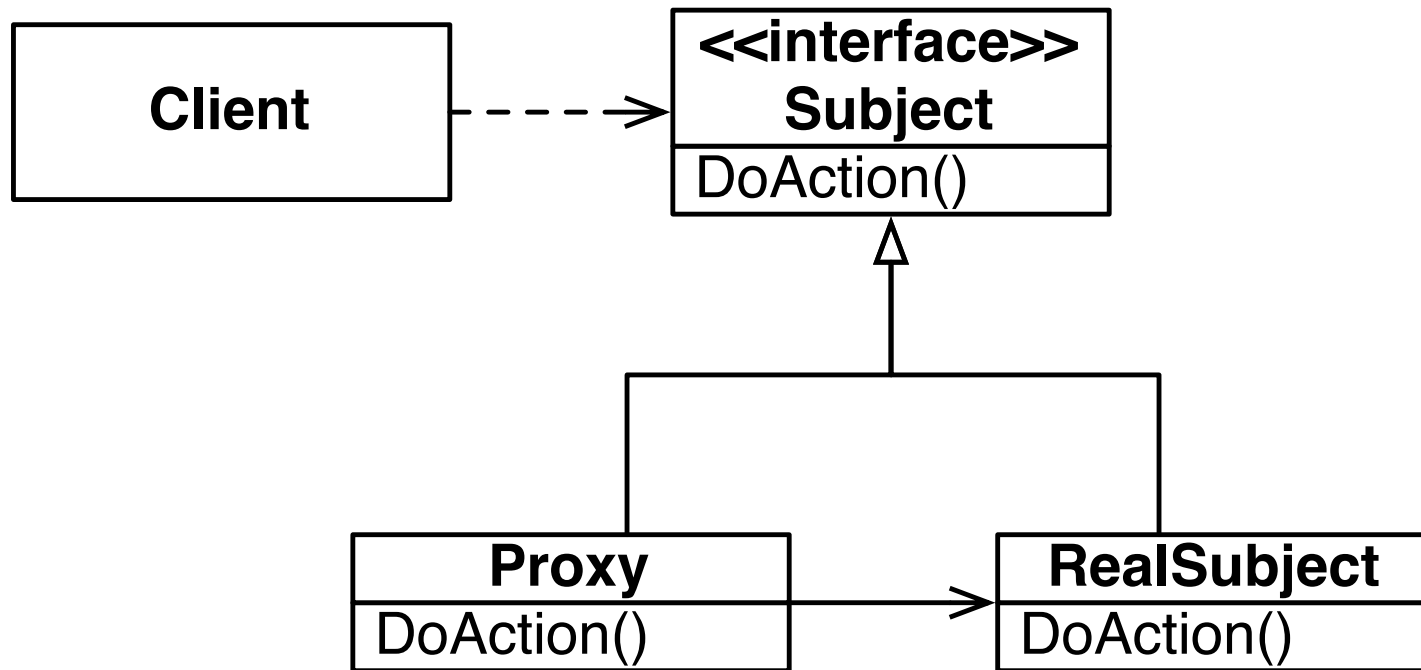
How do you hide the complexity of accessing objects that require pre- or post-processing?

Introduce a proxy to control access to the object

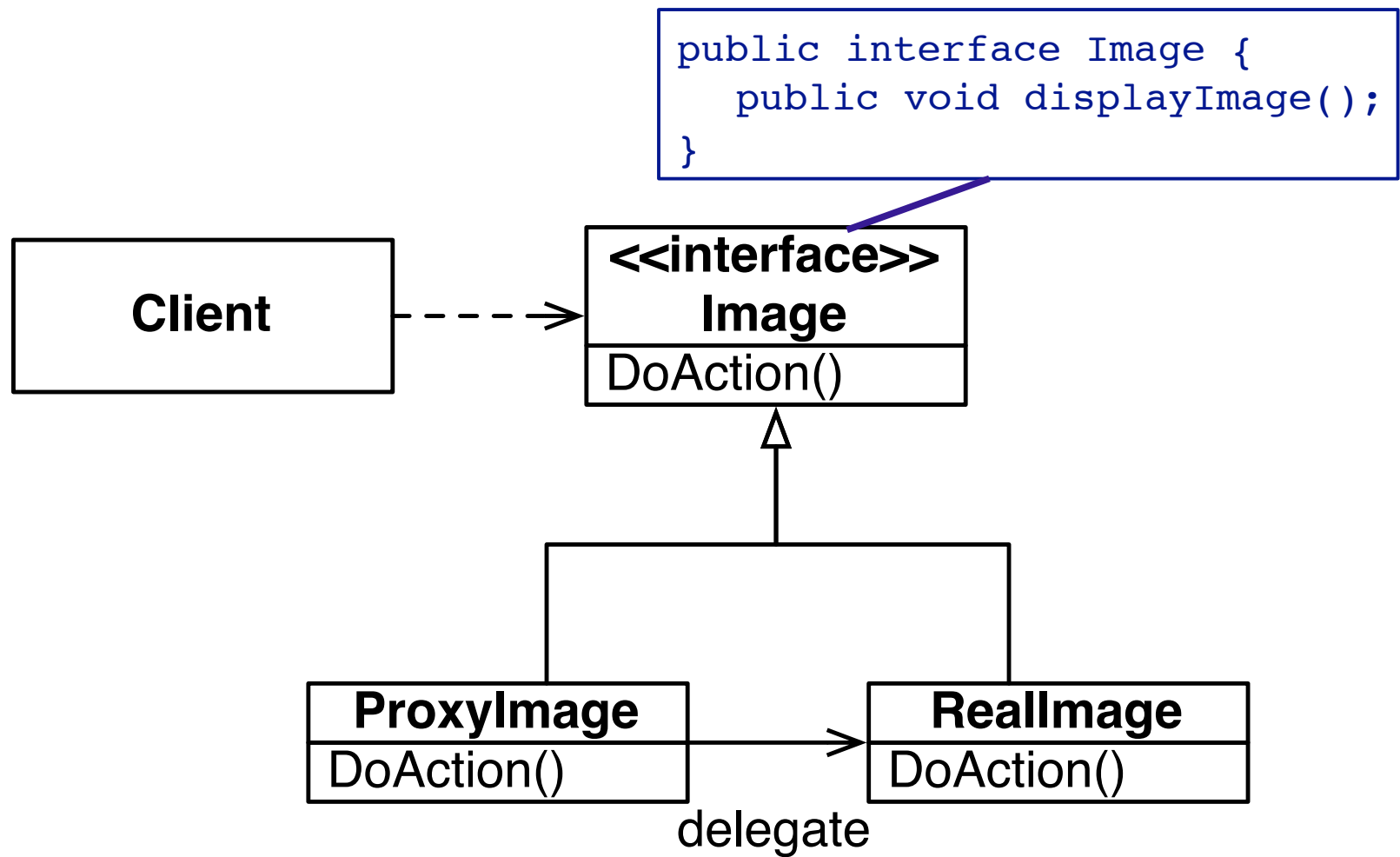
Some services require special pre or post-processing. Examples include objects that reside on a remote machine, and those with security restrictions

A proxy provides the same interface as the object that it controls access to

Proxy Pattern - UML



Proxy Pattern - Example

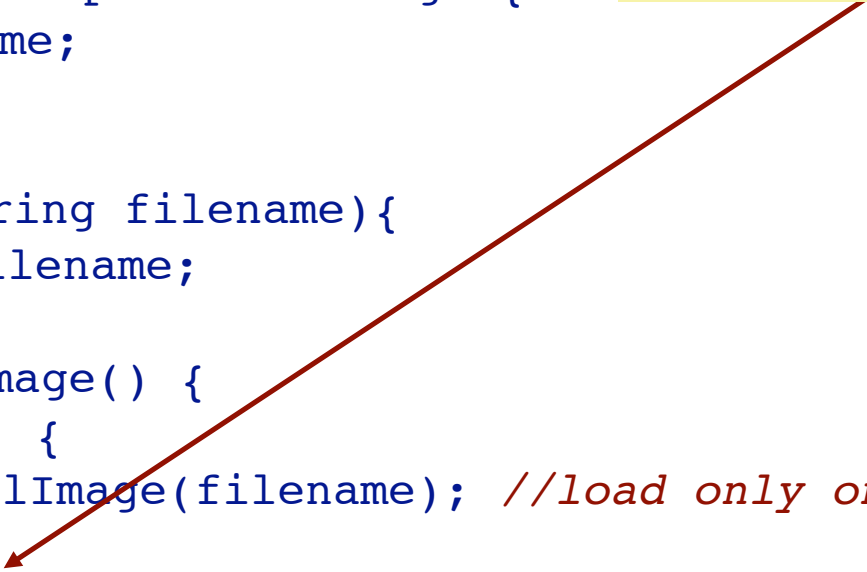


Proxy Pattern - Example

```
public class ProxyImage implements Image {
    private String filename;
    private Image image;

    public ProxyImage(String filename){
        this.filename = filename;
    }
    public void displayImage() {
        if (image == null) {
            image = new RealImage(filename); //load only on demand
        }
        image.displayImage();
    }
}
```

delegate request
to real subject



Proxy Pattern - Example

```
public class RealImage implements Image {
    private String filename;

    public RealImage(String filename) {
        this.filename = filename;
        System.out.println("Loading "+filename);
    }

    public void displayImage() {
        System.out.println("Displaying "+filename);
    }
}
```

Proxy Pattern - Example, the client

```
public class ProxyExample {
    public static void main(String[] args) {

        ArrayList<Image> images = new ArrayList<Image>();
        images.add(new ProxyImage("HiRes_10MB_Photo1"));
        images.add(new ProxyImage("HiRes_10MB_Photo2"));
        images.add(new ProxyImage("HiRes_10MB_Photo3"));

        images.get(0).displayImage();
        images.get(1).displayImage();
        images.get(0).displayImage(); // already loaded
    }
}
```

Proxies are used for remote object access

Example

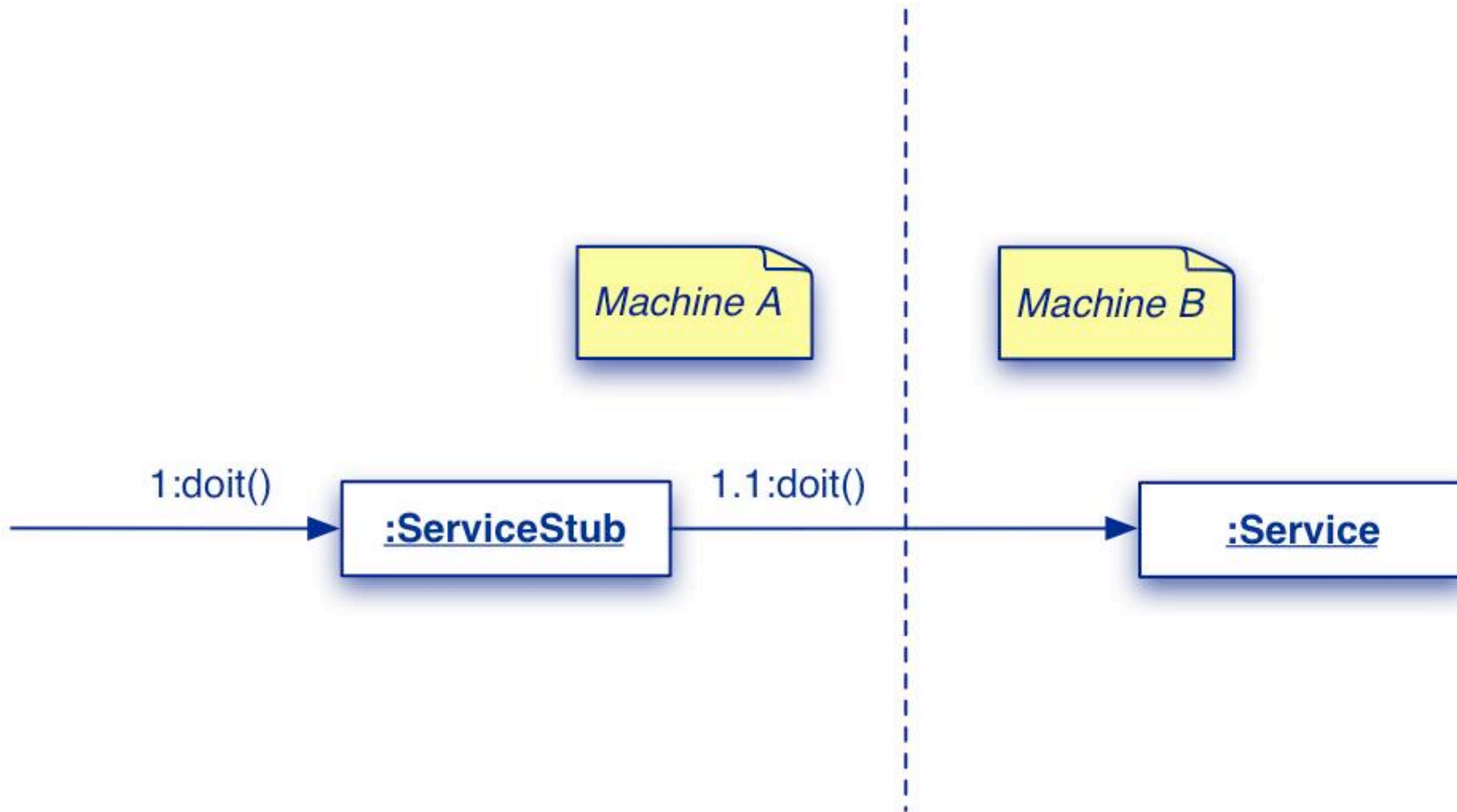
A Java “stub” for a remote object accessed by Remote Method Invocation (RMI).

Consequences

A Proxy decouples clients from servers. A Proxy introduces a level of indirection.

Proxy differs from Adapter in that it does not change the object's interface

Proxy remove access example



Template Method Pattern

How do you implement a generic algorithm, deferring some parts to subclasses?

Define it as a Template Method

A Template Method factors out the common part of similar algorithms, and delegates the rest to:

hook methods that subclasses *may extend*, and

abstract methods that subclasses *must implement*

Template Method Pattern

Example

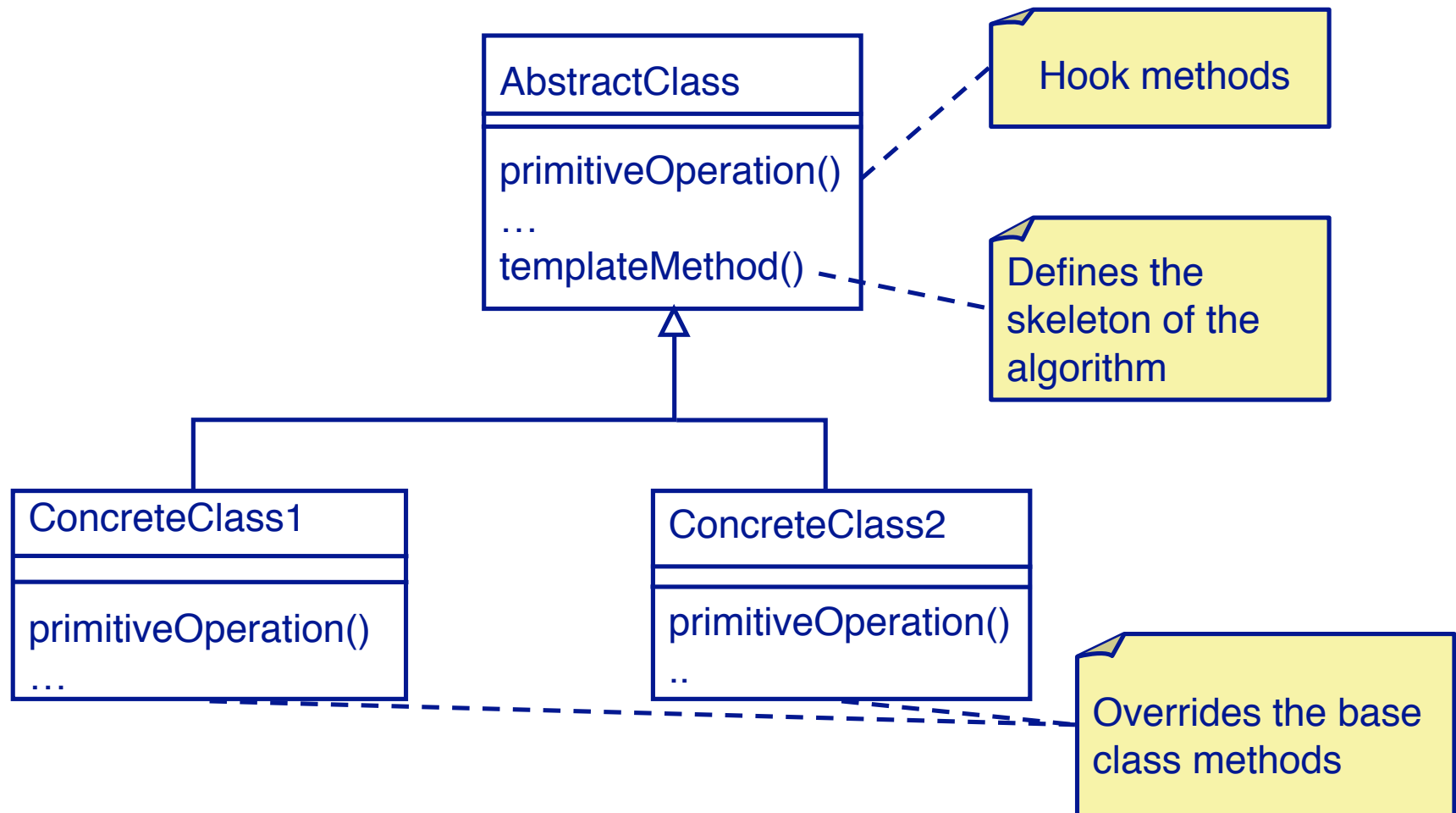
`TestCase.runBare()` is a template method that calls the hook method `setUp()`

Consequences

Template methods lead to an *inverted control structure* since a parent class calls the operations of a subclass and not the other way around.

Template Method is used in most frameworks to allow application programmers to easily extend the functionality of framework classes.

Template Method Pattern - UML



Template Method Pattern - Example

Subclasses of TestCase are expected to *override hook method* `setUp()` and possibly `tearDown()` and `runTest()`

```
public abstract class TestCase implements Test {
    ...
    public void runBare() throws Throwable {
        setUp();
        try { runTest(); }
        finally { tearDown(); }
    }
    protected void setUp() { }           // empty by default
    protected void tearDown() { }
    protected void runTest() throws Throwable { ... }
}
```

Composite Pattern

How do you manage a part-whole hierarchy of objects in a consistent way?

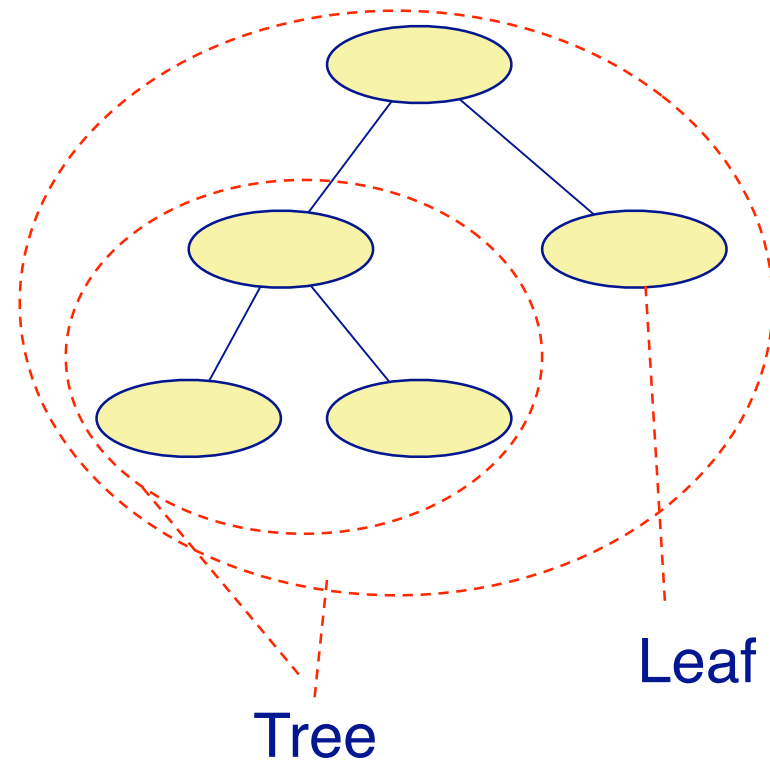
Define a common interface that both parts and composites implement

Typically composite objects will implement their behavior by *delegating to their parts*

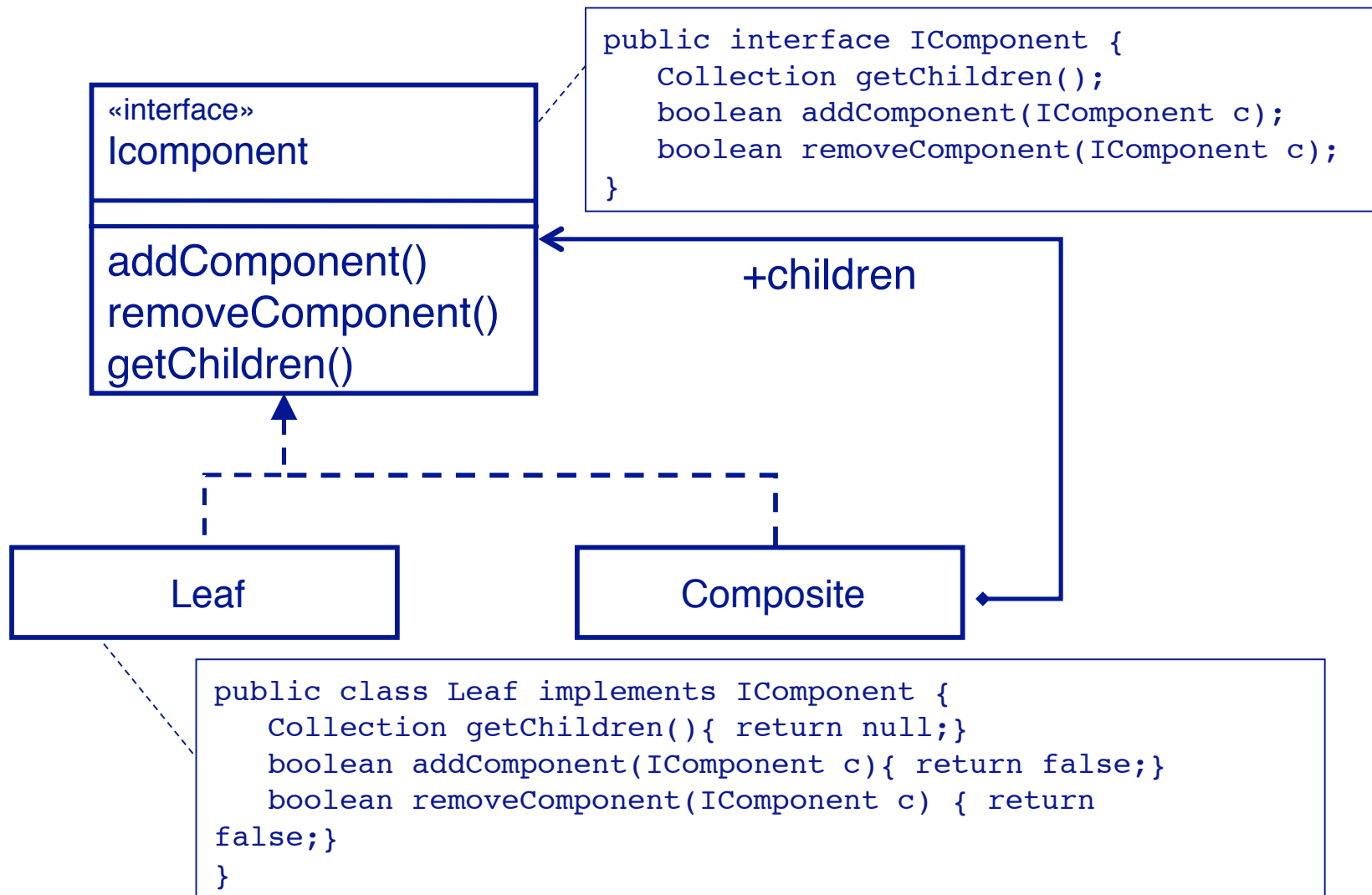
Composite Pattern Example

Composite allows you to treat a single instance of an object the same way as a *group* of objects.

Consider a *Tree*. It consists of *Trees* (subtrees) and *Leaf* objects.



Composite Pattern Example



Composite Pattern Example

```
public class Composite implements IComponent {
    private String id;
    private ArrayList<IComponent> list = new ArrayList<IComponent> ();
    public boolean addComponent(IComponent c) {
        return list.add(c);
    }
    public Collection getChildren() {
        return list;
    }
    public boolean removeComponent(IComponent c) {
        return list.remove(c);
    }
    ...
}
```

Composite Pattern Example - client usage

```
public class CompositeClient {  
    public static void main(String[] args) {  
  
        Composite chile = new Composite("Chile");  
        Leaf santiago = new Leaf("Santiago");  
        Leaf serena = new Leaf("serena");  
        chile.addComponent(santiago);  
        chile.addComponent(serena);  
  
        Composite southAmerica = new Composite("South America");  
        southAmerica.addComponent(chile);  
  
        System.out.println(southAmerica.toString());  
    }  
}
```

Observer Pattern

How can an object inform arbitrary clients when it changes state?

Clients implement a common Observer interface and register with the “observable” object; the object notifies its observers when it changes state

An observable object *publishes* state change events to its *subscribers*, who must implement a common interface for receiving notification

Observer Pattern

Example

A Button expects its observers to implement the ActionListener interface.

(see the Interface and Adapter examples)

Consequences

Notification can be *slow* if there are many observers for an observable, or if observers are themselves observable!

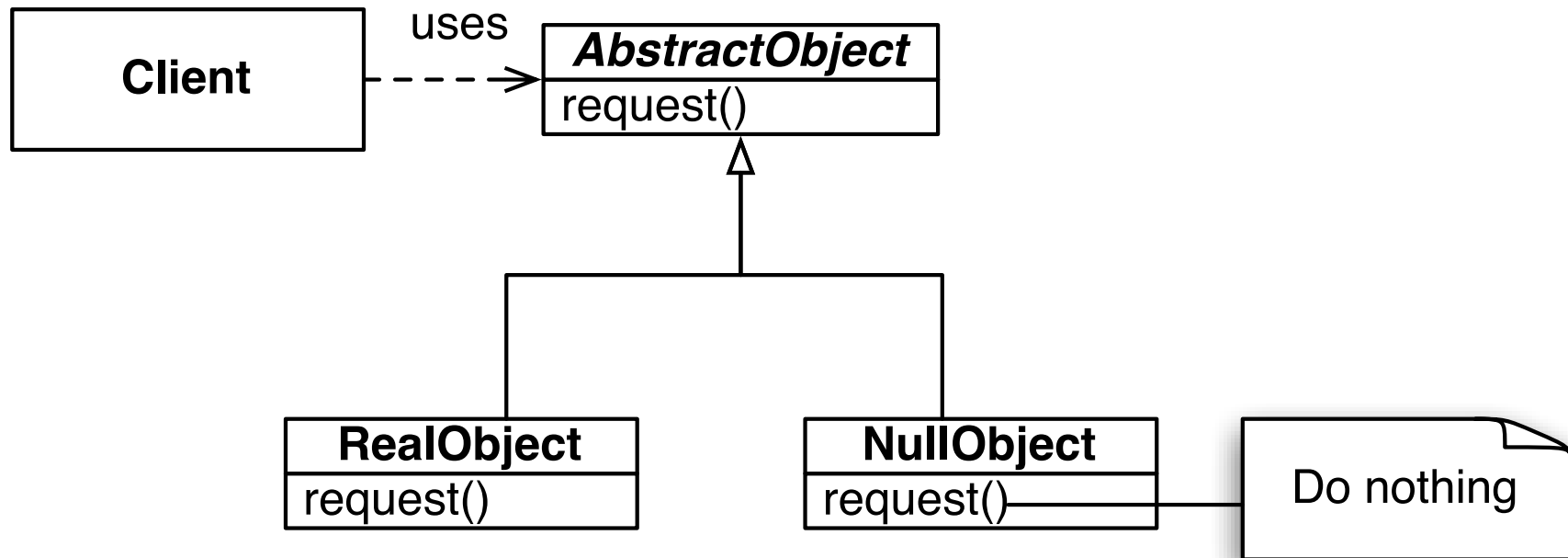
Null Object Pattern

How do you avoid cluttering your code with tests for null object pointers?

Introduce a Null Object that implements the interface you expect, but does nothing

Null Objects may also be Singleton objects, since you never need more than one instance

Null Object Pattern - UML



Null Object

Examples

NullOutputStream extends OutputStream with an empty write() method

Consequences

Simplifies client code

Not worthwhile if there are only few and localized tests for null pointers

Some other Design Patterns...

State

The state pattern is a behavioral design pattern, also known as the objects for states pattern. This pattern is used in to represent the state of an object. This is a clean way for an object to partially change its type at runtime

Decorator

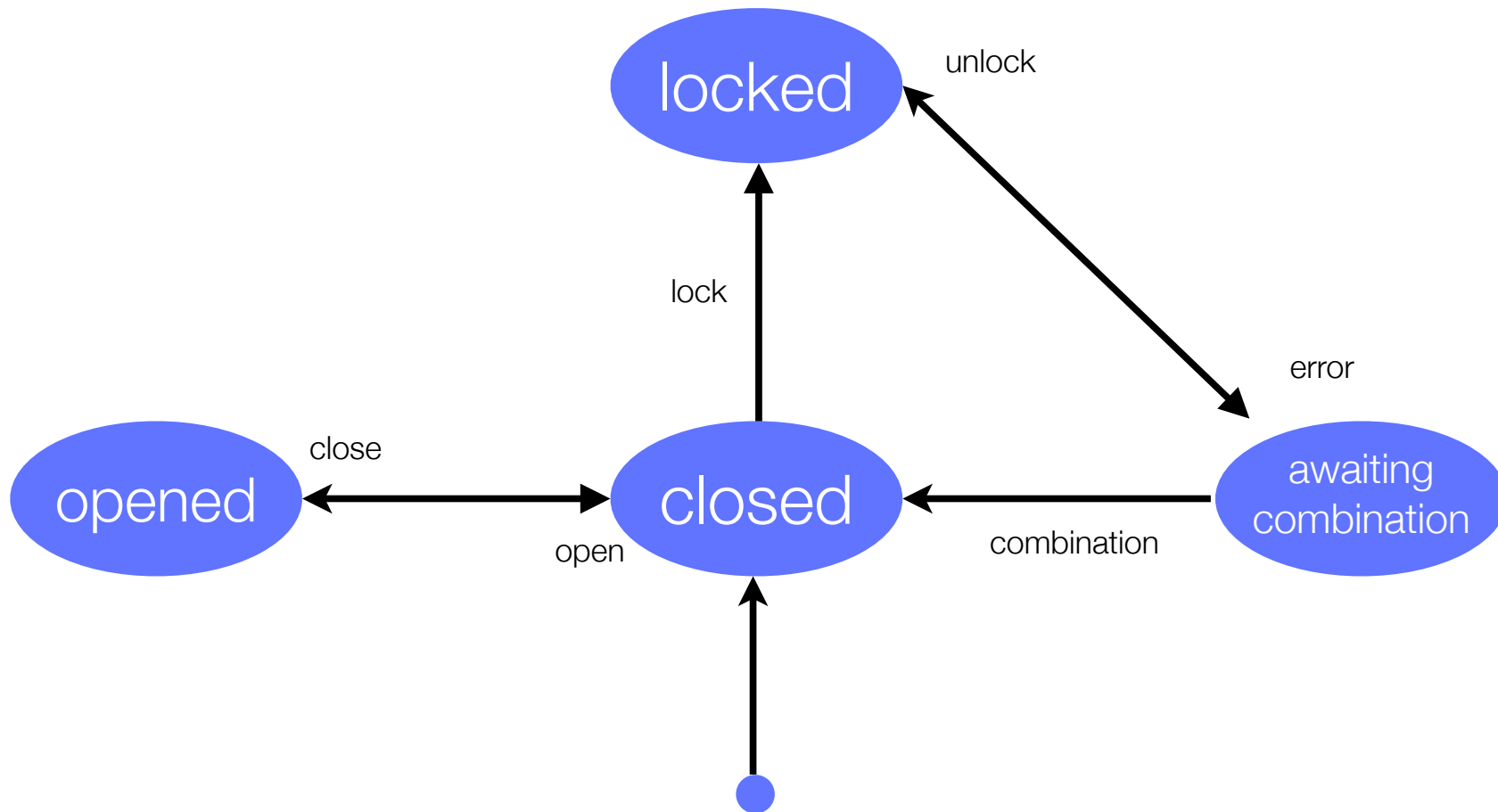
that allows new/additional behaviour to be added to an existing method of an object dynamically.

Some other Design Patterns...

Visitor

a way of separating an algorithm from an object structure. A practical result of this separation is the ability to add new operations to existing object structures without modifying those structures.

Handling States

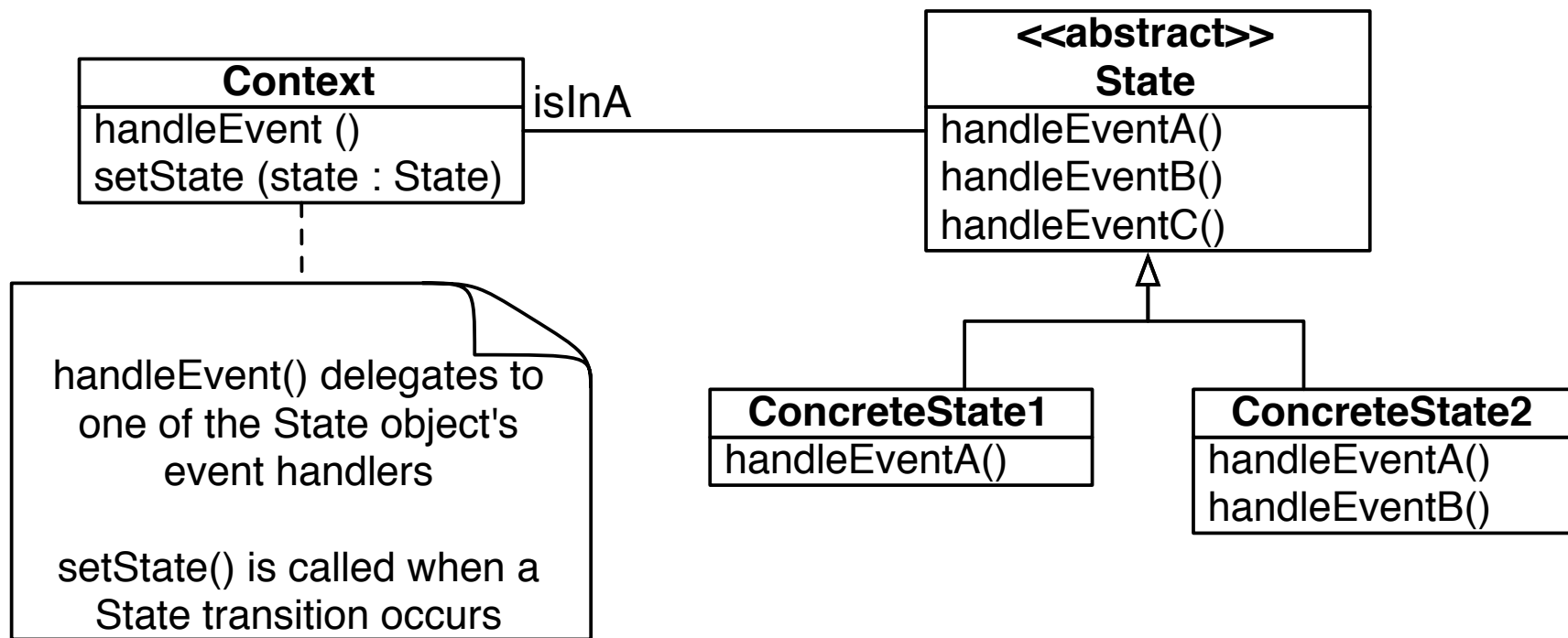


Handling States

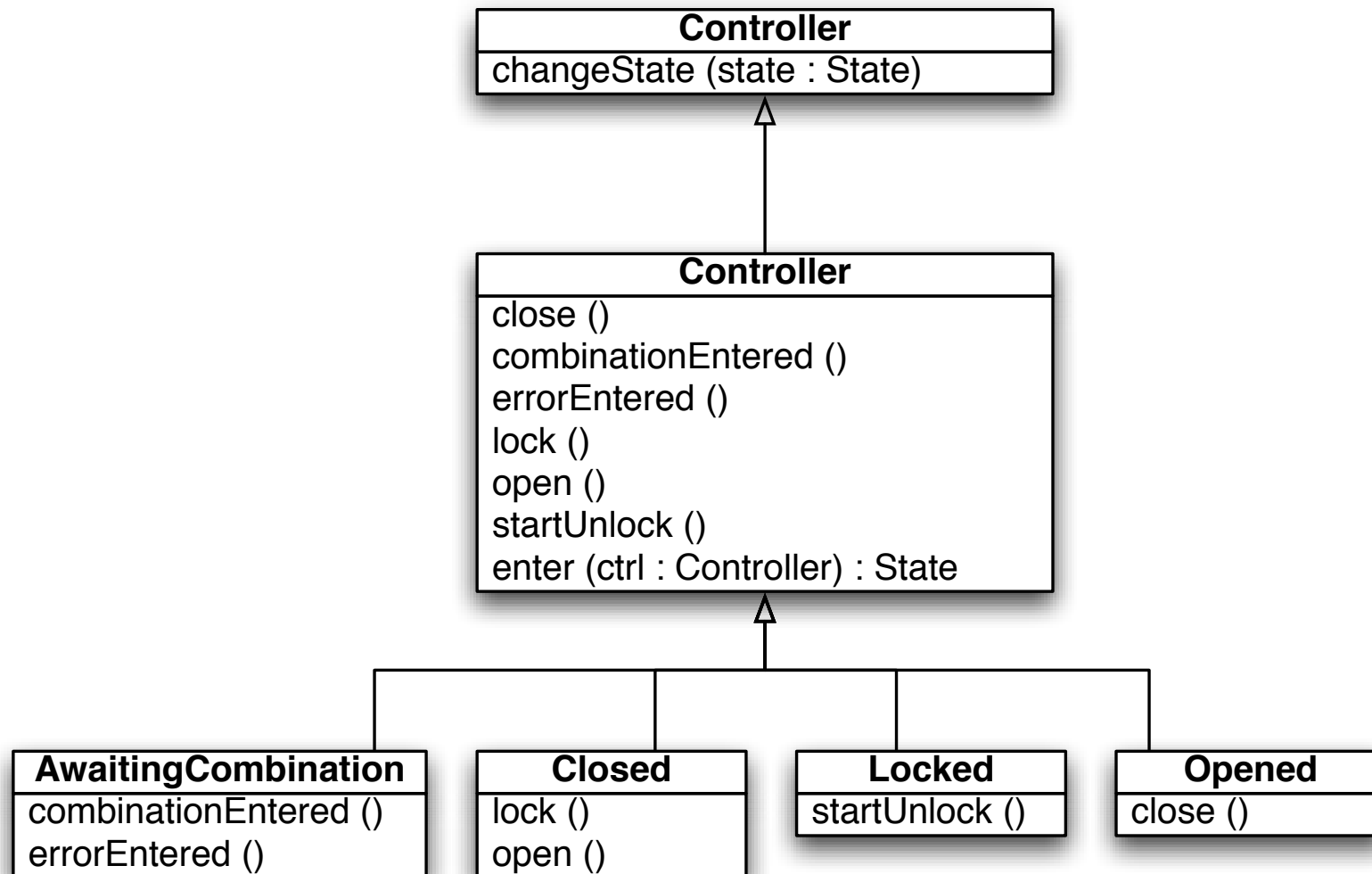
```
while ((line = in.readLine()) != null) {  
    if (line.equals("open")){  
        changeState(CLOSED, OPENED);  
    }  
    if (line.equals("close")){  
        changeState(OPENED, CLOSED);  
    }  
    if (line.equals("lock")){  
        changeState(CLOSED, LOCKED);  
    }  
    if (line.equals("unlock")){  
        changeState(LOCKED, AWAITING_COMBINATION);  
    }  
    if (line.equals("combination")){  
        changeState(AWAITING_COMBINATION, CLOSED);  
    }  
    if (line.equals("error")){  
        changeState(AWAITING_COMBINATION, LOCKED);  
    }  
    this.prompt();  
}
```



State Pattern - UML



State Pattern Example



Each state is a separate object

```
public class Opened extends State {  
    private static Opened instance;  
  
    private Opened(Controller controller){  
        this.controller = controller;  
    }  
  
    public static State enter(Controller controller){  
        if(instance == null) // lazy evaluation  
            instance = new Opened(controller);  
        return instance;  
    }  
  
    public void close() {  
        controller.changeState(Closed.enter(controller));  
    }  
}
```

What Problems do Design Patterns solve?

Patterns:

- document design experience

- enable widespread reuse of software architecture

- improve communication within and across software development teams

- explicitly capture knowledge that experienced developers already understand implicitly

- arise from practical experience

- help ease the transition to object-oriented technology

- facilitate training of new developers

- help to transcend “programming language-centric” viewpoints

Doug Schmidt, CACM Oct 1995

What you should know!

What's wrong with *long methods*? How long should a method be?

What's the difference between a *pattern* and an *idiom*?

When should you use *delegation* instead of *inheritance*?

How does a *Proxy* differ from an *Adapter*?

How can a Template Method help to eliminate duplicated code?

When do I use a Composite Pattern? Do you know any examples from the Frameworks you know?

Can you answer these questions?

What *idioms* do you regularly use when you program? What patterns do you use?

What is the difference between an *interface* and an *abstract class*?

When should you use an *Adapter* instead of modifying the interface that doesn't fit?

Is it good or bad that `java.awt.Component` is an abstract class and not an interface?

Why do the Java libraries use different interfaces for the Observer pattern (`java.util.Observer`, `java.awt.event.ActionListener` etc.)?

License

<http://creativecommons.org/licenses/by-sa/2.5>



Attribution-ShareAlike 2.5

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:



Attribution. You must attribute the work in the manner specified by the author or licensor.



Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.