# Some Design Patterns

Alexandre Bergel
Nancy Hitschfeld
11/11/2020


Design Patterns: Elements of Reusable Object-Oriented Software

*Erich Gamma, Richard Helm, Ralph Johnson, John M. Vlissides, 1994*

# Outline

1. Adapter
2. Proxy
3. Observer
4. State

# 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

## 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

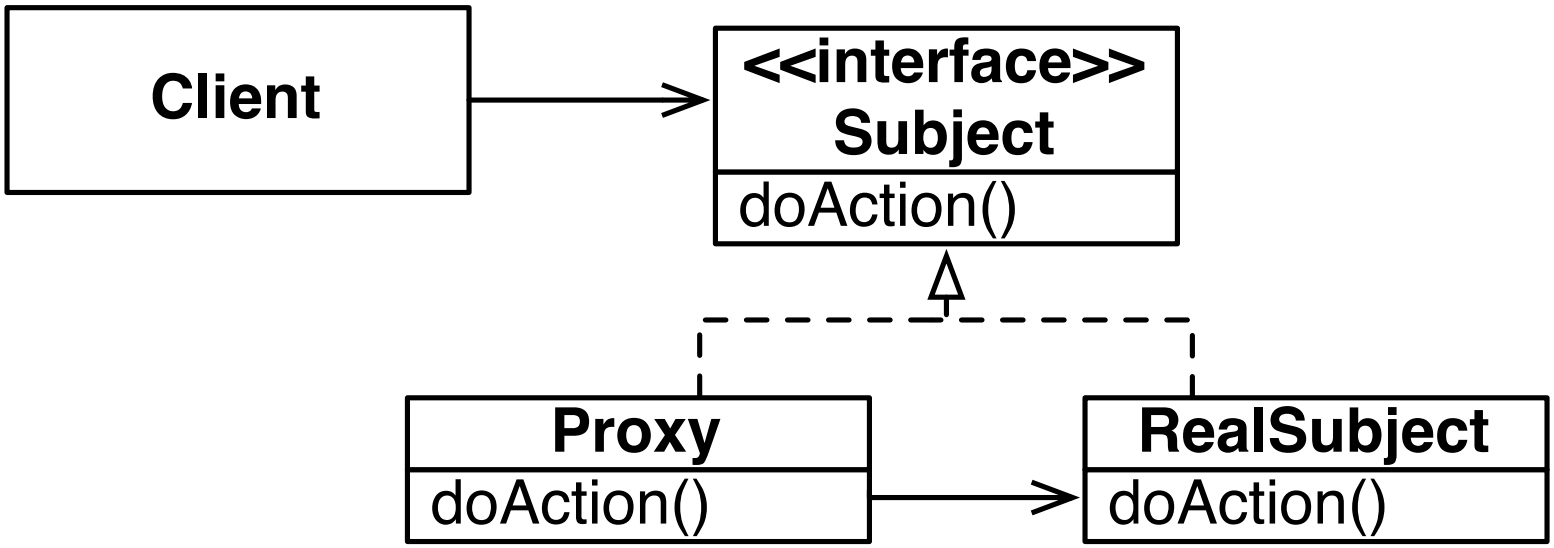
How do you hide the complexity of accessing objects that require pre- or post-processing?

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

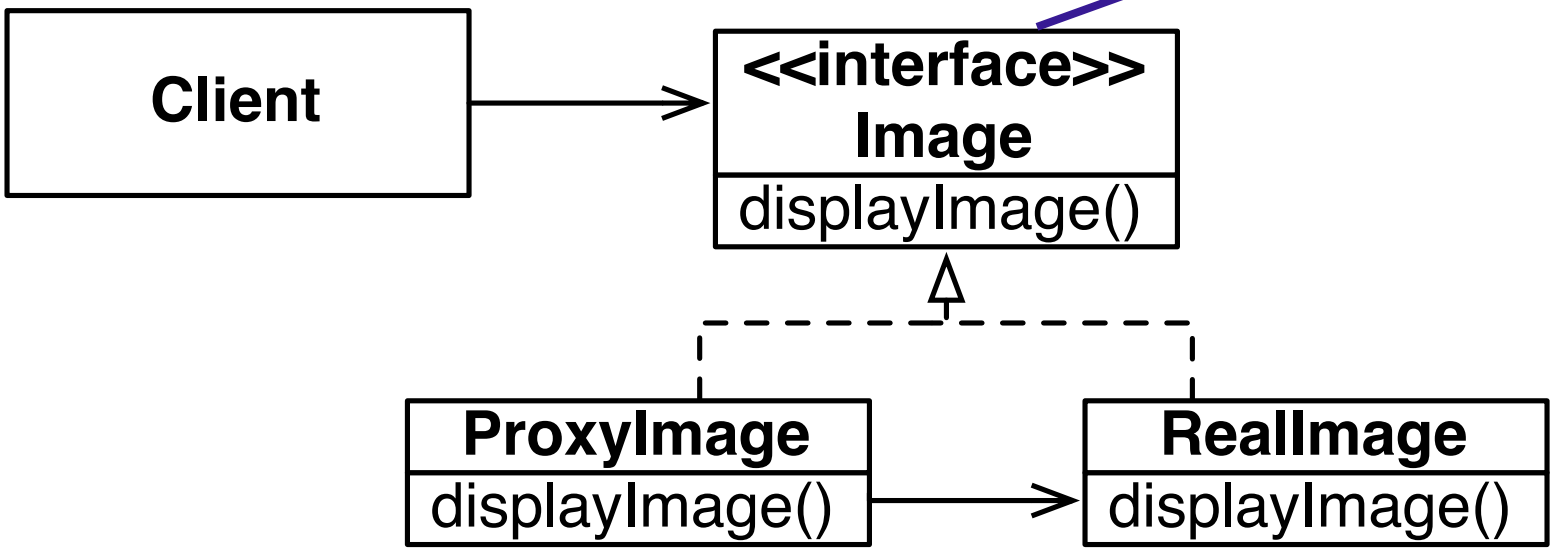
*A proxy provides the same interface as the object that it controls access to*

# Proxy Pattern - UML



# Proxy Pattern - Example

```
public interface Image {
   public void displayImage();
}
```



# 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

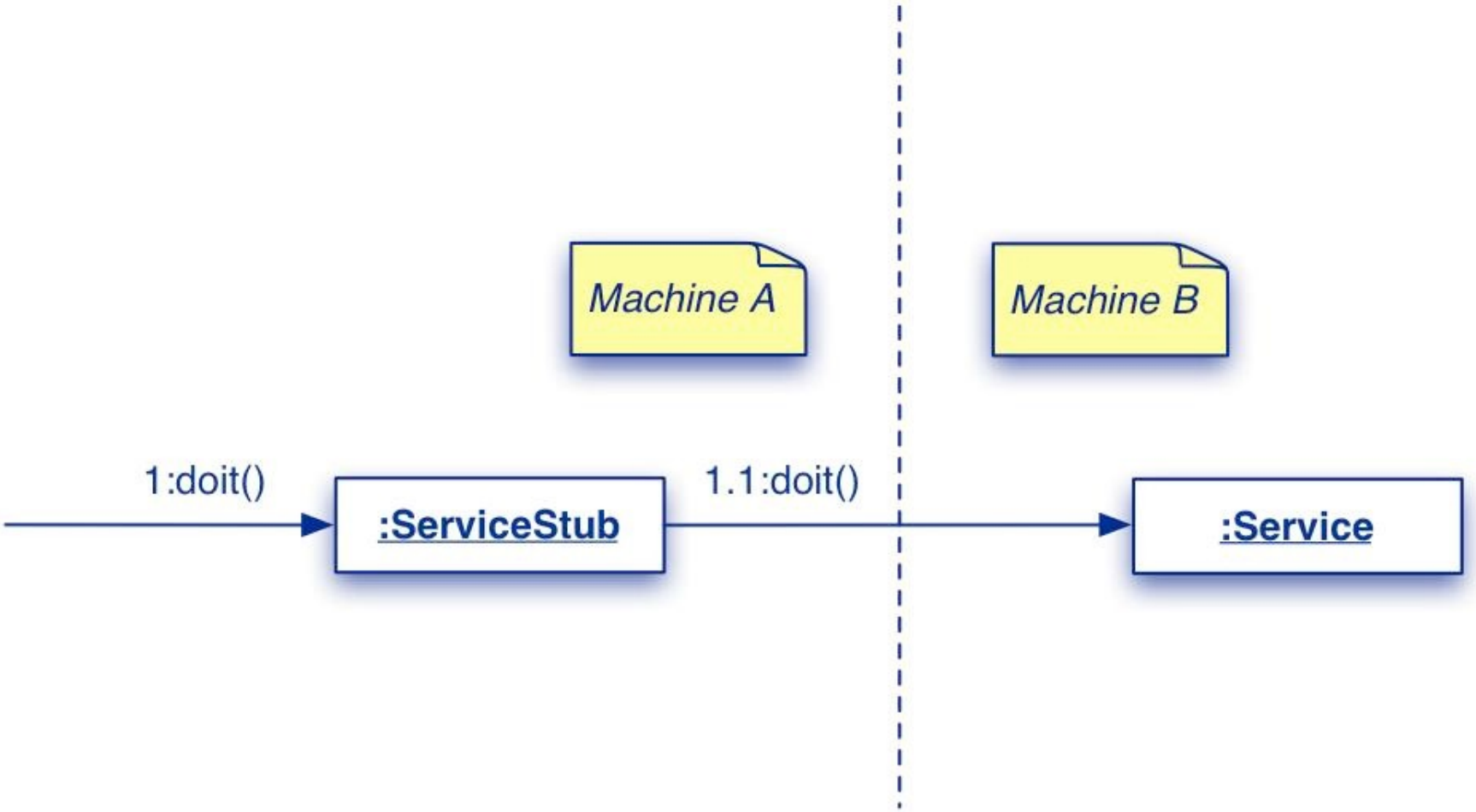
A Java “stub” for a remote object accessed by Remote Method Invocation (RMI)

## 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 remote access example



# Libraries for proxies

Java offers facilities to dynamically create type safe proxies

https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html

# Remote object interaction

Remote Method Invocation (RMI) was popular some years ago

These days, *RESTful web services* are the way to go

- Use the HTTP protocol for communicating
- The client can be written in other language than Java
- Easy to put in place

# 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*!

```
package obs;

import java.util.Observable;          //Observable is here
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class EventSource extends Observable implements Runnable {
    public void run() {
        try {
            final InputStreamReader isr = new InputStreamReader( System.in );
            final BufferedReader br = new BufferedReader( isr );
            while( true ) {
                String response = br.readLine();
                setChanged();
                notifyObservers( response );
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
package obs;

import java.util.Observable;
import java.util.Observer;  /* this is Event Handler */

public class ResponseHandler implements Observer {
    private String resp;
    public void update (Observable obj, Object arg) {
        if (arg instanceof String) {
            resp = (String) arg;
            System.out.println("\nReceived Response: "+ resp );
        }
    }
}
```

```
package obs;

public class MyApp {
    public static void main(String args[]) {
        System.out.println("Enter Text >");

        // create an event source - reads from stdin
        final EventSource evSrc = new EventSource();

        // create an observer
        final ResponseHandler respHandler = new ResponseHandler();

        // subscribe the observer to the event source
        evSrc.addObserver( respHandler );

        // starts the event thread
        Thread thread = new Thread(evSrc);
        thread.start();
    }
}
```

# Problem with Observer/Observable

As you can see, we need to use an `instanceof` and a downcast

This is a significant problem that was addressed since Java 9

You need to use the class `PropertyChangeSupport` and the interface `PropertyChangeListener` instead

In Java 9 and later

```
public class EventSource implements Runnable {

    private PropertyChangeSupport changes;

    public EventSource() {
        changes = new PropertyChangeSupport(this);
    }
    public void addObserver(ResponseHandler resp) {
        changes.addPropertyChangeListener(resp);
    }

    public void run() {
        try {
            // We declare two constant in the code
            // the final keyword forbid from later assignment
            final InputStreamReader isr = new InputStreamReader( System.in );
            final BufferedReader br = new BufferedReader( isr );
            while( true ) {
                String response = br.readLine();
                if(response.equals("quit")) return;

                changes.firePropertyChange(new PropertyChangeEvent(this, "entered
text", null, response));
            }
        }
        catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

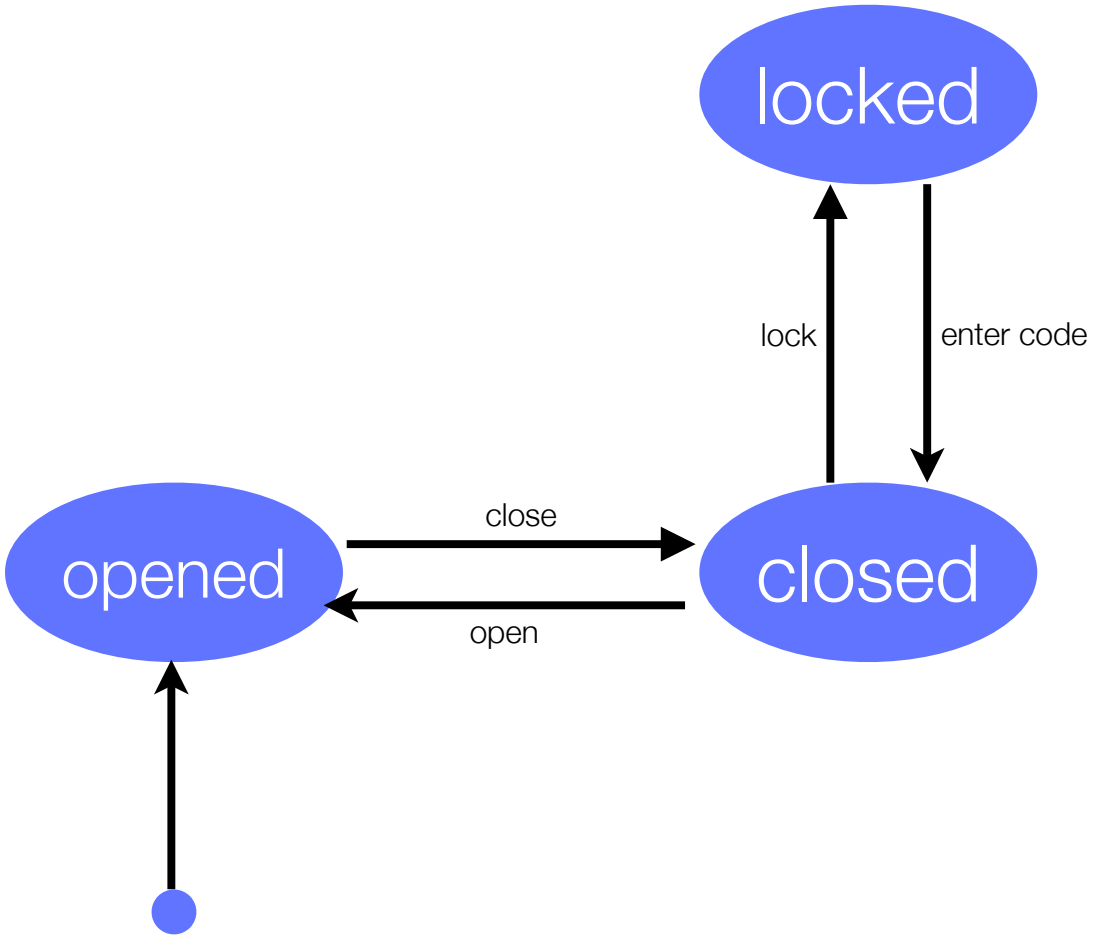
# In Java 9 and later

```
public class ResponseHandler implements PropertyChangeListener {
    @Override
    public void propertyChange(PropertyChangeEvent evt) {
        System.out.println("\nReceived Response: "+ evt.getNewValue() );
    }
}
```

# In Java 9 and later

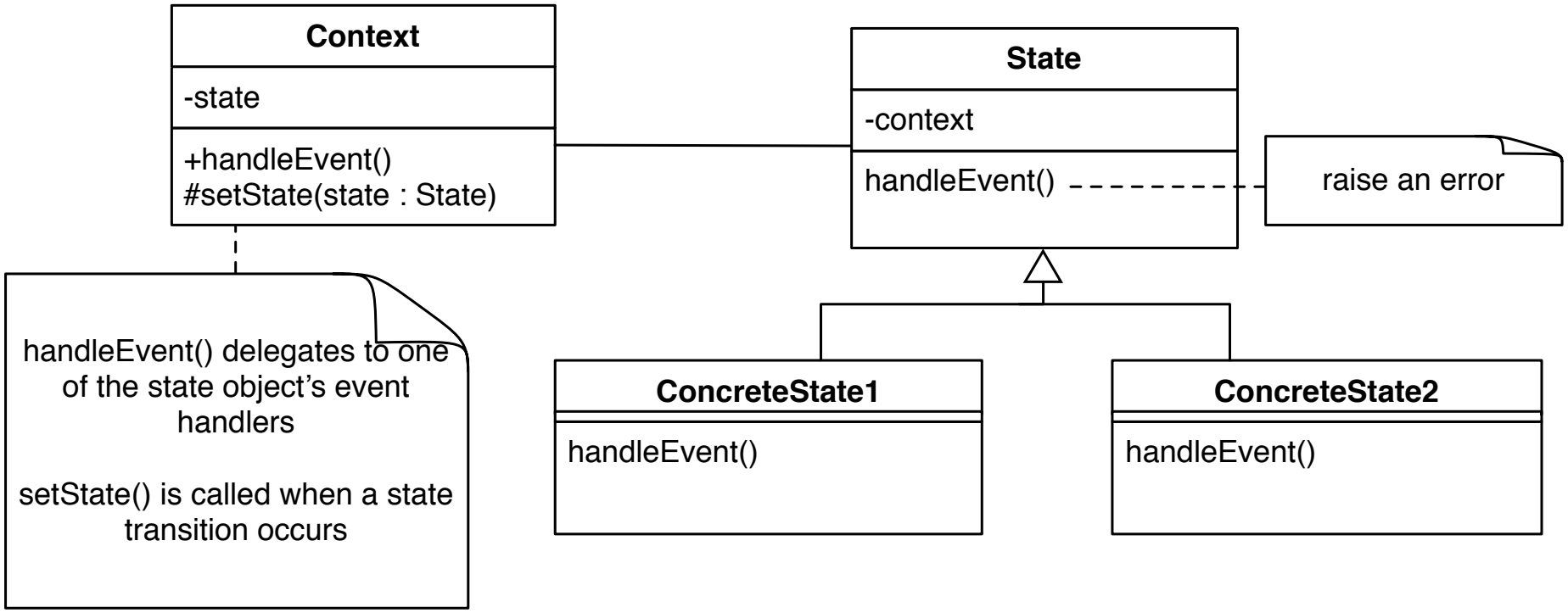
```
public class MyApp {
    public static void main(String args[]) {
        System.out.println("Enter Text >");

        // create an event source - reads from stdin
        final EventSource evSrc = new EventSource();

        // create an observer
        final ResponseHandler respHandler = new ResponseHandler();

        // subscribe the observer to the event source
        evSrc.addObserver(respHandler);

        // starts the event thread
        Thread thread = new Thread(evSrc);
        thread.start();
    }
```

# Handling States



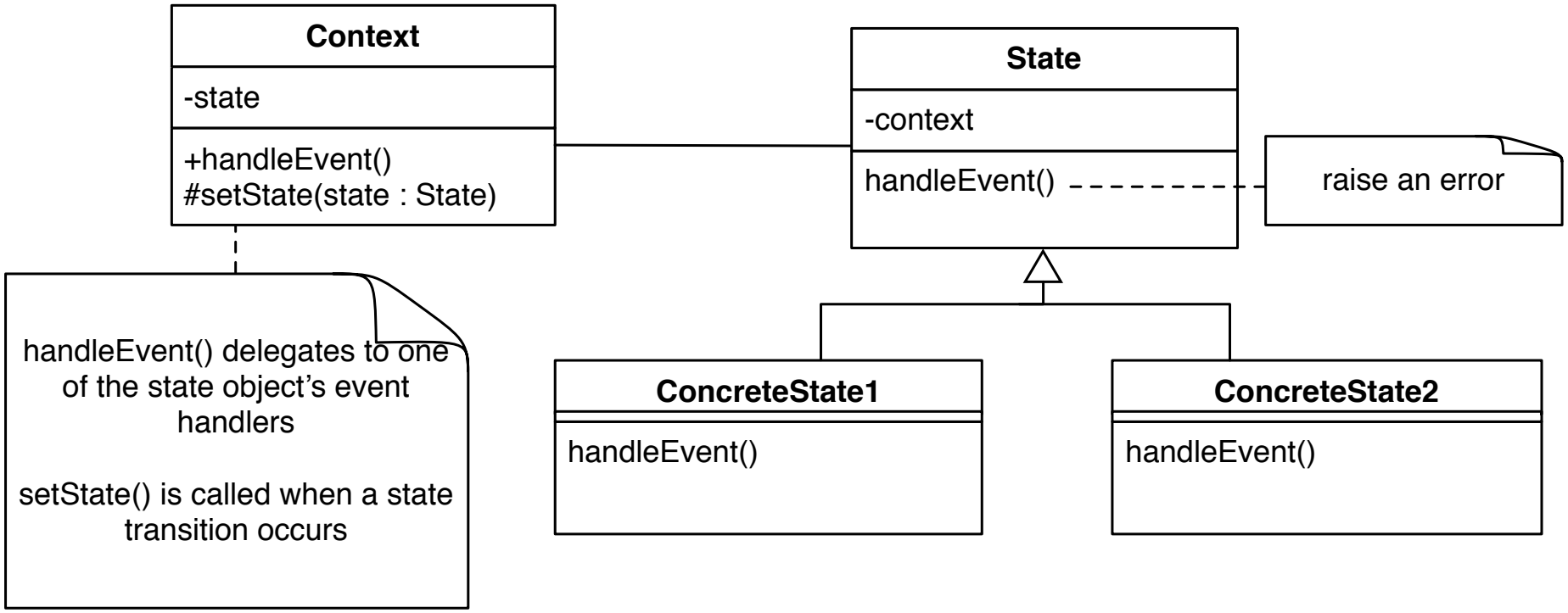
```
while ((line = in.readLine()) != null) {
   if (line.equals("open")){
      changeState(CLOSED, OPENED);
   }
   if (line.equals("close")){
      changeState(OPENED, CLOSED);
   }
   if (line.equals("lock")){
      changeState(CLOSED, LOCKED);
   }
   if (line.equals("unlock")){
      changeState(LOCKED,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

# State Pattern - UML



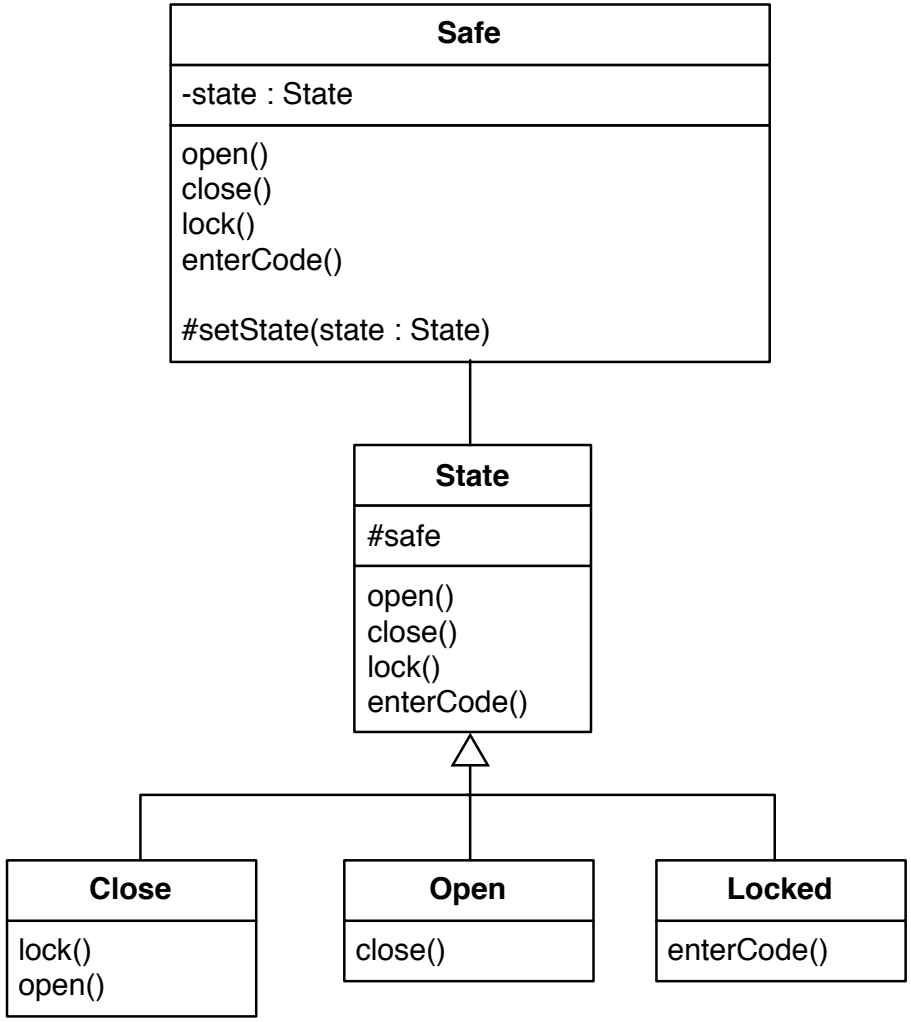

Context may be in a state ConcreteState1 or ConcreteState2

# State Pattern - UML




Variant using an interface for State may be found in the literature. However, using an interface leads to code duplication.

# State Pattern Example



# Each state is a separate object

```
public class Safe {
    private State state;

    public Safe () {
        this.setState(new Open());
    }

    void setState(State aState) {
        state = aState;
        state.setSafe(this);
    }

    public void open() { state.open(); }
    public void close() { state.close(); }
    public void lock() { state.lock(); }
    public void enterCode() { state.enterCode(); }

    public boolean isOpen() { return state.isOpen(); }
    public boolean isClosed() { return state.isClosed(); }
    public boolean isLocked() { return state.isLocked(); }
}
```

# Each state is a separate object

```
public class State {
    private Safe safe;

    public void setSafe(Safe safe) {
        this.safe = safe;
    }

    protected void changeState(State state) {
        safe.setState(state);
    }

    void error() { throw new AssertionError("Wrong state"); }

    void open() { error(); }
    void close() { error(); }
    void enterCode() { error(); }
    void lock() { error(); }

    public boolean isOpen() { return false; }
    public boolean isClosed() { return false; }
    public boolean isLocked() { return false; }
}
```

# Each state is a separate object

```
public class Open extends State {
     void close() { this.changeState(new Close()); }
     public boolean isOpen() { return true; }
}

public class Close extends State {
     void open() { this.changeState(new Open()); }
     void lock() { this.changeState(new Locked()); }
     public boolean isClosed() { return true; }
}

public class Locked extends State {
     void enterCode() { this.changeState(new Close()); }
     public boolean isLocked() { return true; }
}
```

# Each state is a separate object

```
public class SafeTest {
     private Safe safe;

     @Before
     public void setUp() {
          safe = new Safe();
     }

     @Test
     public void testCreation() {
          assertTrue(safe.isOpen());
          assertFalse(safe.isClosed());
          assertFalse(safe.isLocked());
          safe.close();
          assertFalse(safe.isOpen());
          assertTrue(safe.isClosed());
          assertFalse(safe.isLocked());
     }
     …
}
```

# 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

# What you should know!

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

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

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

Can you give example for which it is beneficial to use a factory pattern?

# Can you answer these questions?

What *patterns* do you use when you program?

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

When should you use an *Adapter* instead of modifying the interface that does not fit?

In which situations the *factory pattern* is not appropriate?

# License




Original version of this lecture from Oscar Nierstrasz, Uni - Bern