

CS 635 Advanced Object-Oriented Design &  
Programming  
Spring Semester, 2007  
Doc 13 Proxy & State  
Apr 3, 2007

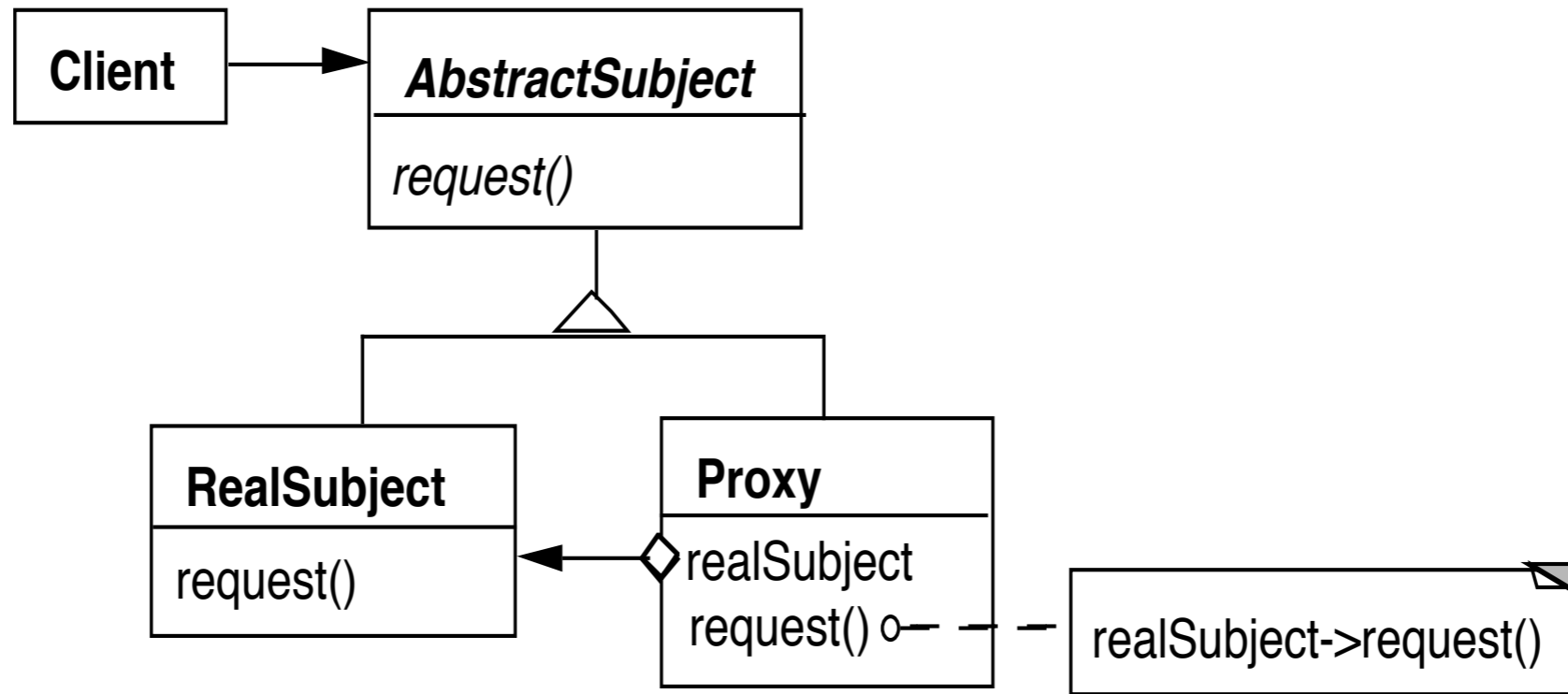
Copyright ©, All rights reserved. 2007 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (<http://www.opencontent.org/opl.shtml>) license defines the copyright on this document.

## References

Design Patterns: Elements of Resuable Object-Oriented Software,  
Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995, pp. 207-218, 305-314

## Proxy (Surrogate)

a person authorized to act on behalf of another



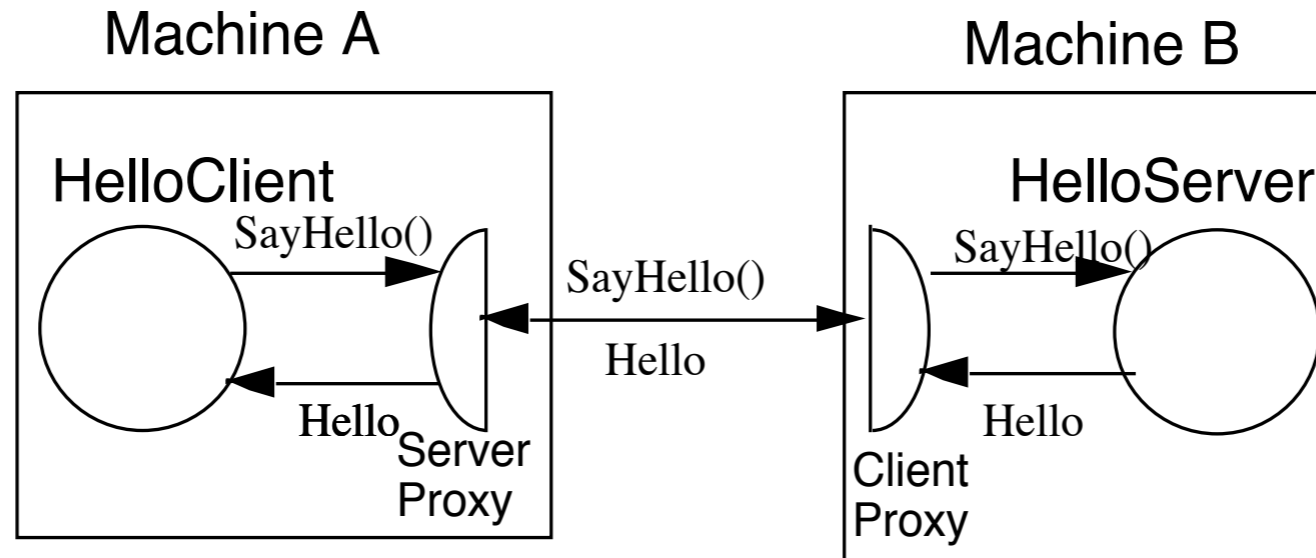
```

class Proxy {
    AbstractSubject realSubject;

    public Foo service(Bar x ) {
        return realSubject(x);
    }
}
  
```

Why do it?

# Remote Proxy



```
String server = getHelloHostAddress( args);  
Hello proxy = (Hello) Naming.lookup( server );  
String message = proxy.sayHello();  
System.out.println( message );
```

# More General Proxy

```
class Proxy {  
    AbstractSubject realSubject;  
  
    public Foo service(Bar x ) {  
        some preprocessing  
        result = realSubject(x);  
        some postprocessing  
    }  
}
```

# Virtual Proxy

Creates/accesses expensive objects on demand



# Java's Synchronized List

```
ArrayList notSafe = new ArrayList();  
List threadSafe = Collections.synchronizedList(notSafe);
```

```
static class SynchronizedList {  
    List list;  
    public Object get(int index) {  
        synchronized(mutex) {return list.get(index);}  
    }  
}
```

# Java's Unmodifiable List

```
ArrayList notSafe = new ArrayList();  
List noChange = Collections.unmodifiableList(notSafe);
```

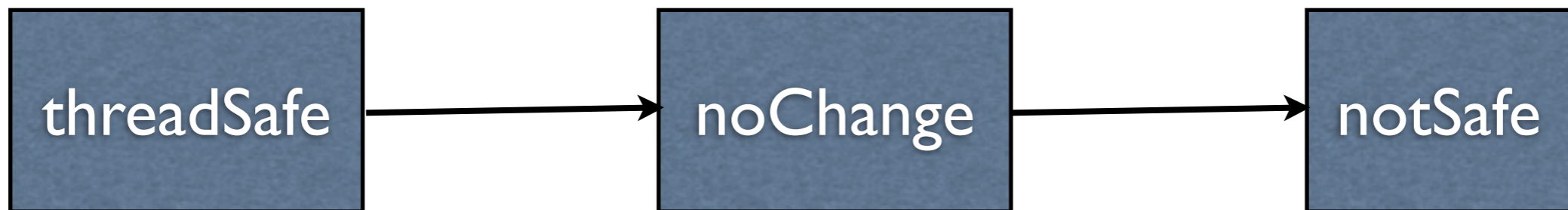
```
static class UnmodifiableList {  
    List list;  
    public Object get(int index) { return list.get(index);}  
  
    public Object set(int index, Object element) {  
        throw new UnsupportedOperationException();  
    }  
}
```

# Proxy or Decorator?

```
ArrayList notSafe = new ArrayList();
```

```
List noChange = Collections.unmodifiableList(notSafe);
```

```
List threadSafe = Collections.synchronizedList(noChange);
```



# Proxy verses Decorator

"Decorators can have similar implementations as proxies"

Proxy controls access to an object

Decorator adds one or more responsibilities to an object

# Smalltalk Proxy Trick

Object subclass: #Proxy

instanceVariableNames: 'target '

classVariableNames: "

poolDictionaries: "

category: 'Whitney-Examples'

## Class Method

on: anObject

^super new target: anObject

## Instance Methods

doesNotUnderstand: aMessage

^target

perform: aMessage selector

withArguments: aMessage arguments

target: anObject

target := anObject

| wrapper |

wrapper := Proxy on: Transcript.

wrapper open.

wrapper show: 'Hi mom'.

| wrapper |

wrapper := Proxy on: 3.

wrapper + 5.

| wrapper |

wrapper := Proxy on: 'Hi '.

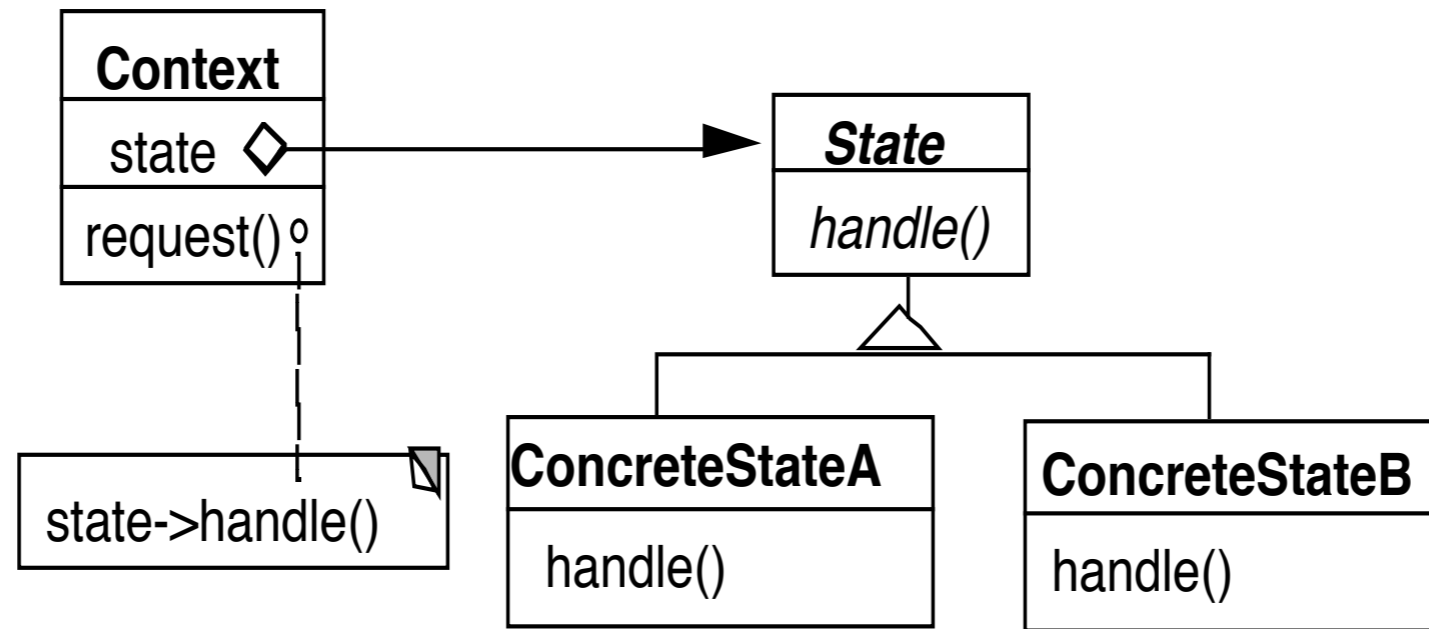
wrapper , ' mom'.

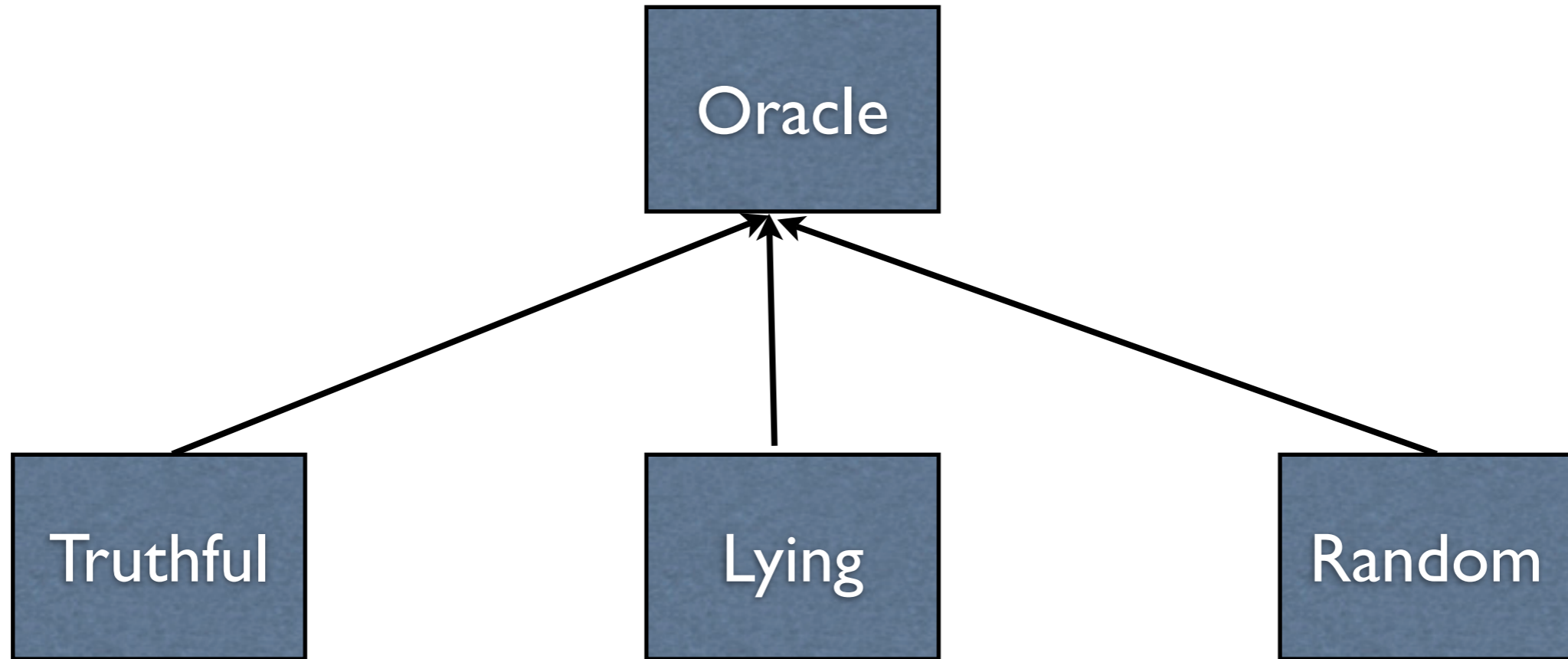
# State Pattern

Allow an object to alter its behavior when its internal state changes

The object will appear to change its class

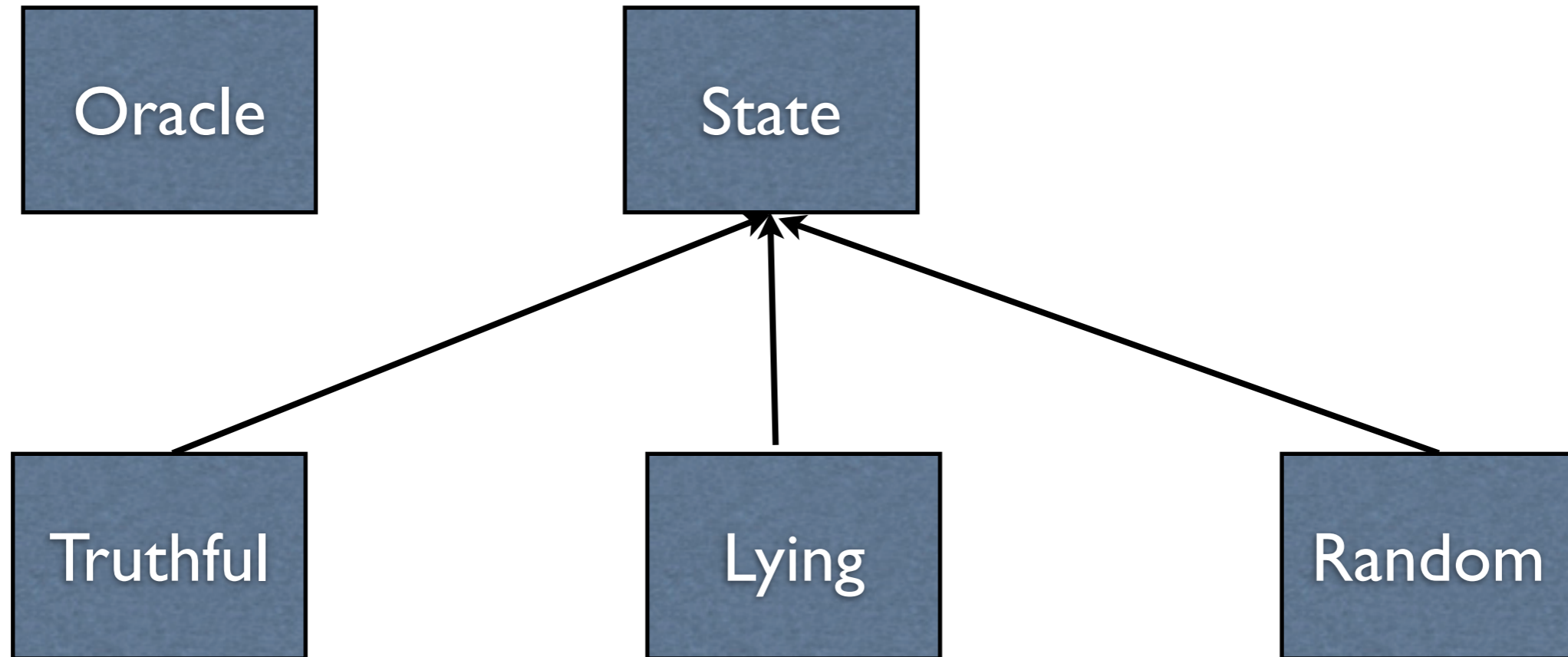
# Structure





```
Oracle seer = new Truthful();  
seer.willThereBeAStrike();  
seer = new Lying();  
seer.willThereBeAStrike();
```





```
class Oracle {
    private State mode = set mode;

    public boolean willThereBeAStrike() {
        return mode.willThereBeAStrike();
    }
}
```

```
Oracle seer = new Truthful();
seer.willThereBeAStrike();
seer.willThereBeAStrike();
```

# Who defines state Transitions - Context

```
class Context {  
    private AbstractState state = new StartState();  
  
    public Bar foo(int x) {  
        int result = state.foo(x);  
        if (someConditionHolds() )  
            state = nextState();  
        return result;  
    }  
}
```

# Who defines state Transitions - States

```
class Context {  
    private AbstractState state = new StartState();  
  
    public void foo(int x) {  
        state = state.foo(x);  
    }  
}
```

What if foo returns a value?

# Who defines state Transitions - States

```
class Context {  
    private AbstractState state = new StartState();  
  
    public int foo(int x) {  
        return state.foo(x, this);  
    }  
  
    protected void setState(AbstractState newState) {  
        state = newState;  
    }  
}
```

# Sharing State Objects

Stateless state

- State objects without fields

- Can be shared by multiple contexts

Can store data in context and pass as arguments

Large number of state transitions can be expensive

- Only create state once & reuse same object

# Changing Class - No Need for Context

Language Dependent Feature

Smalltalk & Lisp

```
class Truthful extends Oracle {  
  
    public boolean foo(int x) {  
        int result = state.foo(x);  
        this.changeClassTo(Random);  
        return result;  
    }  
}
```

# State Verses Strategy

Rate of Change

## Strategy

Context usually contains just one strategy object

## State

Context often changes state objects

# State Verses Strategy

Exposure of Change

## **Strategy**

Strategies all do the same thing

Client do not see change in behavior of Context

## **State**

States act differently

Client see the change in behavior