CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2018
Doc 16 Singleton, Decorator, Adapter, Proxy
Nov 6, 2018

Mock Objects

Java

Mockito

Swift

https://qualitycoding.org/swift-mock-objects/

Article on creating mock objects in Swift

Python

https://docs.python.org/3/library/unittest.mock.html

Singleton

Warning

Simplest pattern

But has subtle issues particularly in Java

Most controversial pattern

Intent

Ensure a class only has one instance

Provide global point of access to single instance

Singleton

```
public class Counter {
   private int count = 0;
   private static Counter instance;
   private Counter() { }

   public static Counter instance() {
      if (instance == null)
         instance = new Counter();
      return instance;
   }

   public int increase()  {return ++count;}
}
```

One instance

Global access

Some Uses

Java Security Manager

Logging a Server

Null Object

Globals are Evil

Why Singletons Are Controversial(Evil)

Singletons provide global access point for some service

Hidden dependencies

Is there a different design that does not need singletons

Pass a reference

Why Singletons Are Controversial(Evil)

Singletons allow you to limit creation of objects of a class

Should that be the responsibility of the class?

Class should do one thing

Use factory or builder to limit the creation

Why Singletons Are Controversial(Evil)

Singletons tightly couple you to the exact type of the singleton object

No polymorphism

Hard to subclass

Why Singletons Are Controversial(Evil)

Singletons carry state with them that last as long as the program lasts

Persistent state makes testing hard and error prone

Why Singletons Are Controversial(Evil)

A Singleton today is a multiple tomorrow

Singleton pattern makes it hard to change to allow multiple objects

Why Singletons Are Controversial(Evil)

In Java Singletons are a lie

More on this later

Singleton Implementation

Why Not Use This?

```
public class Counter {
    private static int count = 0;

    public static int increase()  {return ++count;}
}
```

Why Not Use This?

```
public class Counter {
   private int count = 0;
   private Counter() { }

   public static Counter instance = new Counter();

   public int increase()  {return ++count;}
}
```

Two Useful Features

Lazy

Only created when needed

Thread safe

Recommended Implementation

```
public class Counter {
   private int count = 0;
   private Counter() { }

   private static class SingletonHolder {
      private final static Counter INSTANCE = new Counter();
   }

   public static Counter instance() {
         return SingletonHolder.INSTANCE;
   }

   public int increase()  {return ++count;}
}
```

Correct but not Lazy

```
public class Counter {
   private int count = 0;
   protected Counter() { }

   private final static Counter INSTANCE = new Counter();

   public static Counter instance() {
         return INSTANCE;
   }

   public int increase()  {return ++count;}
}
```

Lazy, Thread safe with Overhead

```
public class Counter {
    private int count = 0;
    private static Counter instance;
    private Counter() { }

    public static synchronized Counter instance() {
        if (instance == null)
            instance = new Counter();
        return instance();
    }

    public int increase()  {return ++count;}
}
```

Java Templates & Singleton

Does not compile

```
public class TemplateSingleton<Type> {
   Type foo;

   public static TemplateSingleton<Type> instance =
      new TemplateSingleton<Type>();
}
```

When is a Singleton not a Singleton?

When Java Garbage Collects Classes

Singleton class can be garbage collected
Singleton loses any value it had

Solution

Turn off garbage collection of classes (-Xnoclassgc)

Make sure there is always a reference to the class/instance

When Multiple Java Class Loaders are Used

When loaded by two different class loaders there will be two versions of the class

Some servlet engines use different class loader for each servlet

Using custom class loaders can cause this

Purposely Reloading a Java Class

Servlet engines can force a class to be reloaded

Serialize and Deserialize Singleton Object

Serialize the singleton
Deserialize the singleton
You now have two copies

One way to serialize a Java object is using ObjectOutputStream

Ruby Marshal.dump() will not marshal a singleton

Adapter

Adapter

Convert interface of a class into another interface

Use adapter when

- You want to use an existing class but does not have interface on needs
- You want to create a reusable class that works with unrelated or unforseen classes

Address Book & JTable

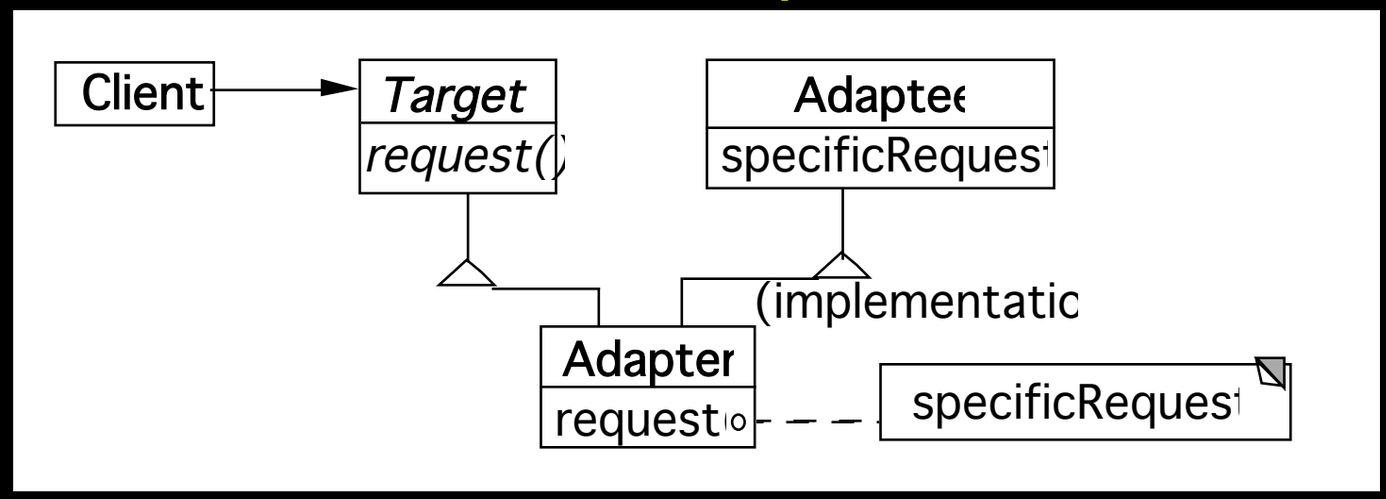
Display an AddressBook object in a JTable

JTables require objects of type TableModel

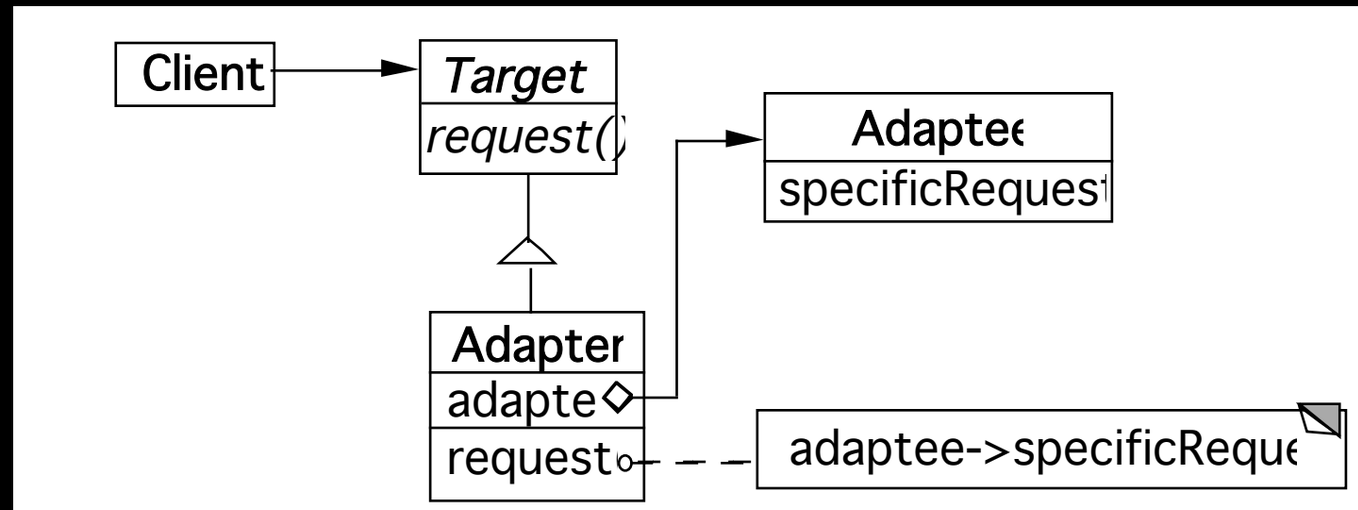
```
public class AddressBook{
    List personList;
    public int getSize(){...}
    public int addPerson(...){...}
    public Person getPerson(...){...}
    ...
}
```

```
public class AddressBookTableAdapter implements TableModel
{
    AddressBook ab;
    public AddressBookTableAdapter( AddressBook ab ){
        this.ab = ab;
    }
//TableModel impl
public getRowCount(){
    ab.getSize();

public Object getValueAt(int rowIndex, int columnIndex) {
    Person requested =
        ad.getPerson(convertRowToName(rowIndex);
return requested.get(convert(columnIndex));
}
```

Class Adapter



Object Adapter



Class Adapter Example

```
class OldSquarePeg {
    public: void squarePegOperation()  { do something }
}

class RoundPeg {
    public:  void virtual roundPegOperation = 0;
}

class PegAdapter: private OldSquarePeg,  public RoundPeg  {
    public:
        void virtual roundPegOperation()  {
            add some corners;
            squarePegOperation();
        }
}

void clientMethod() {
    RoundPeg* aPeg = new PegAdapter();
    aPeg->roundPegOperation();
}
```

Object Adapter

```
class OldSquarePeg{
    public: void squarePegOperation() { do something }
    }

class RoundPeg      {
    public: void virtual roundPegOperation = 0;
    }

class PegAdapter:  public RoundPeg      {
private:
    OldSquarePeg* square;

public:
    PegAdapter()  { square = new OldSquarePeg; }

    void virtual roundPegOperation()      {
        add some corners;
        square->squarePegOperation();
        }
}
```

How Much Adapting does the Adapter do?

Two-way Adapters

```
class OldSquarePeg {
    public:
        void virtual squarePegOperation() { blah }
}

class RoundPeg {
    public:
        void virtual roundPegOperation() { blah }
}

class PegAdapter: public OldSquarePeg, RoundPeg {
    public:
        void virtual roundPegOperation() {
            add some corners;
            squarePegOperation();
        }
        void virtual squarePegOperation() {
            add some corners;
            roundPegOperation();
        }
}
```

Flasher and MouseListener

```
class Flasher
    def toggle()
        @flashing = !@flashing
    end

    def pause()
        #etc
    end

    def resume()
        #etc
    end
end
```

```
class MouseListener
    def mouseClicked(event)
    end

    def mouseEntered(event)
    end

    def mouseExited(event)
    end
end
```

Actions we want

mouse click toggles flasher
mouse enter pauses
mouse exits resumes

Flasher as MouseListener

```
class Flasher
    def toggle()
        @flashing = !@flashing
    end

    def pause()
        #etc
    end

    def resume()
        #etc
    end

    def mouseClicked(event)
        toggle()
    end

    def mouseEntered(event)
        pause()
    end

    def mouseExited(event)
        resume()
    end
end
```

Simple Adapter

```
class Flasher
    def toggle()
        @flashing = !@flashing
    end

    def pause()
        #etc
    end

    def resume()
        #etc
    end
end
```

```
class FlasherAdaptor
    def initialize(aFlasher)
        @flasher = aFlasher
    end

    def mouseClicked(event)
        @flasher.toggle()
    end

    def mouseEntered(event)
        @flasher.pause()
    end

    def mouseExited(event)
        @flasher.resume()
    end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
FlasherAdapter.new(yellowFlasher)
```

A Ruby Adapter - Fowardable

```
class Flasher
    def toggle()
        @flashing = !@flashing
    end

    def pause()
        #etc
    end

    def resume()
        #etc
    end
end
```

```
require 'forwardable'

class FlasherMouseListener
    extend Forwardable

    def initialize()
        @flasher = Flasher.new()
     end
    def_delegator(:@flasher, :toggle, :mouseClick)
    def_delegator(:@flasher, :pause, :mouseEnter)
    def_delegator(:@flasher, :resume, :mouseExit)

end


adaptor = FlasherMouseListener.new()
adaptor.mouseClick()
```

Parameterized Adapter

```
class MouseListenerAdapter
    def initialize(adaptee, clickMethod, enterMethod, exitMethod)
        @adaptee = adaptee
        @clickMethod = clickMethod
        @enterMethod = enterMethod
        @exitMethod = exitMethod
    end

    def mouseClicked(event)
        @adaptee.send(clickMethod)
    end

    def mouseEntered(event)
        @adaptee.send(clickMethod)
    end

    def mouseExited(event)
        @adaptee.send(clickMethod)
    end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
MouseListenerAdapter.new(
                        yellowFlasher,
                        :toggle,
                        :pause,
                        :resume)
```

Better Parameterized Adapter

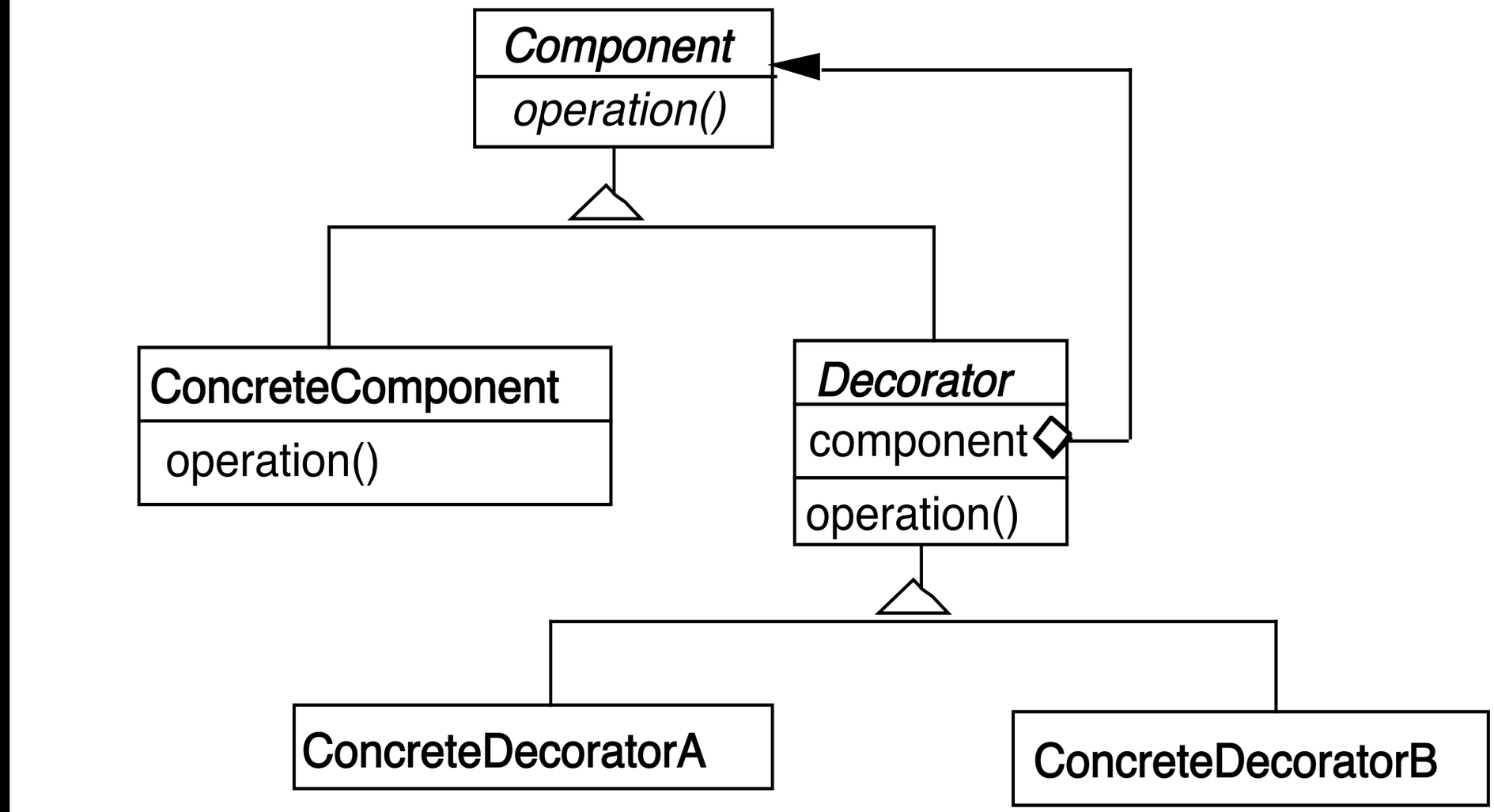
```
class MouseListenerAdapter
    def initialize(adaptee, clickLambda, enterLambda, exitLambda)
        @adaptee = adaptee
        @clickLambda = clickLambda
        @enterLambda = enterLambda
        @exitLambda = exitLambda
    end

    def mouseClicked(event)
        @clickLambda.call(adaptee)
    end

    def mouseEntered(event)
        @enterLambda.call(adaptee)
    end

    def mouseExited(event)
        @exitLambda.call(adaptee)
    end
end
```

```
yellowFlasher = Flasher.new(yellow, fast)
MouseListenerAdapter.new(
        yellowFlasher,
        lambda {|flasher | flasher.toggle()},
        lambda {|flasher | flasher.pause()},
        lambda {|flasher | flasher.resume()})
```

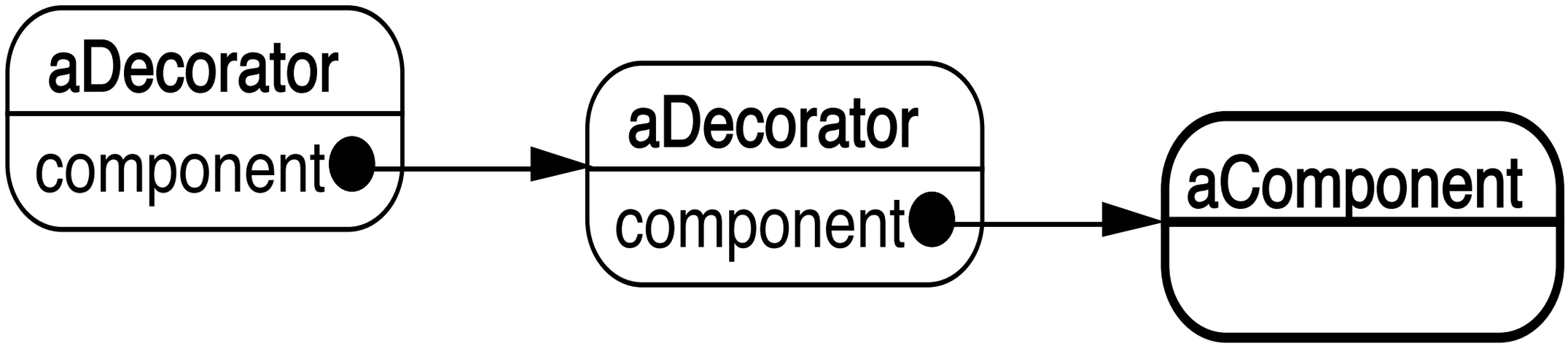
Decorator Pattern

Adds responsibilities to individual objects

Dynamically
Transparently


Component
operation()
ConcreteComponent
operation()
Decorator
component
operation()
ConcreteDecoratorA
ConcreteDecoratorB

Decorator forwards all component operations



Coffee Example from Wikipedia

Compute the cost of coffee

Base Price $1

Cost of Milk $0.50

Cost of Sprinkles $0.20

```
public interface Coffee {
    public double getCost();
    public String getIngredients();
}

public class SimpleCoffee implements Coffee {
    public double getCost() {  return 1;  }

    public String getIngredients() { return "Coffee"; }
}
```

Abstract Decorator

```
public abstract class CoffeeDecorator implements Coffee {
    protected final Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee c) {
        this.decoratedCoffee = c;
    }

    public double getCost() { // Implementing methods of the interface
        return decoratedCoffee.getCost();
    }

    public String getIngredients() {
        return decoratedCoffee.getIngredients();
    }
}
```

Milk

```
class WithMilk extends CoffeeDecorator {
    public WithMilk(Coffee c) {
        super(c);
    }

    public double getCost() {
        return super.getCost() + 0.5;
    }

    public String getIngredients() {
        return super.getIngredients() + ", Milk";
    }
}
```

```
public class Main {
    public static void printInfo(Coffee c) {
        System.out.println("Cost: " + c.getCost() + "; Ingredients: " + c.getIngredients());
    }

    public static void main(String[] args) {
        Coffee c = new SimpleCoffee();
        printInfo(c);

        c = new WithMilk(c);
        printInfo(c);

        c = new WithSprinkles(c);
        printInfo(c);
    }
}
```

Simple Examples

Good for explaining basic concept

But

- Prices change
- New extras
- Seasonal extras

System driven by data

Download information to cash register

Favor Composition over Inheritance

Benefits & Liabilities

Benefits

Simplifies a class
Distinguishes a classes core responsibilities from embellishments

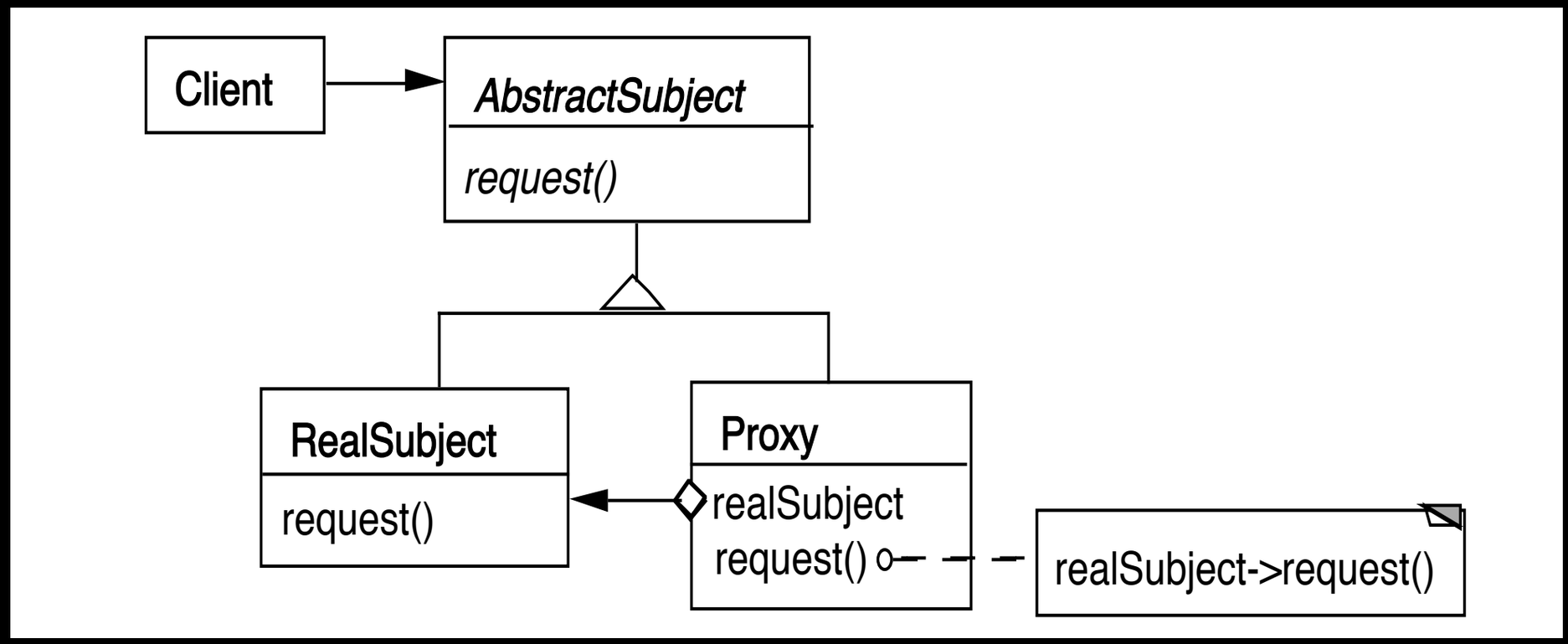
Liabilities

Changes the object identity of a decorated object
Code harder to understand and debug
Combinations of decorators may not work correctly together

Proxy

Proxy (Surrogate)

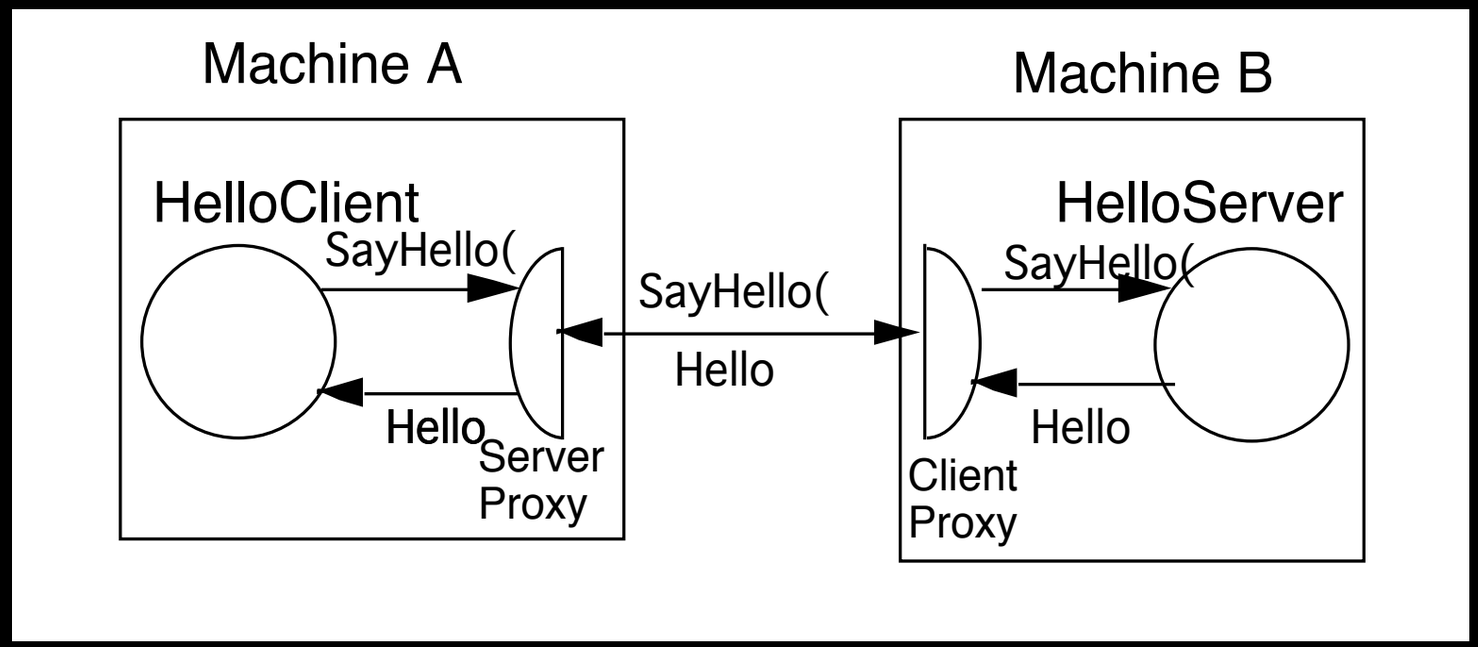
a person authorized to act on behalf of another




```
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

O-R Mapping Layers

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);}
 }
```

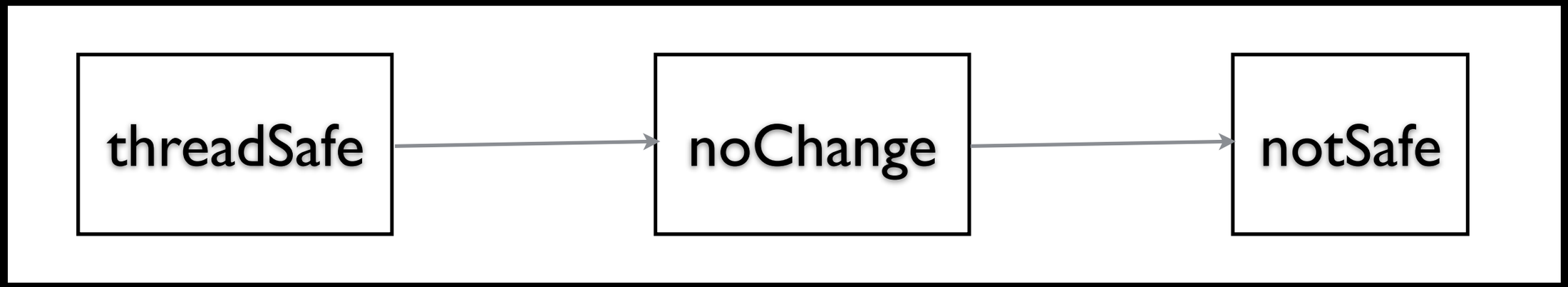
Java's Unmodifiable List

```
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