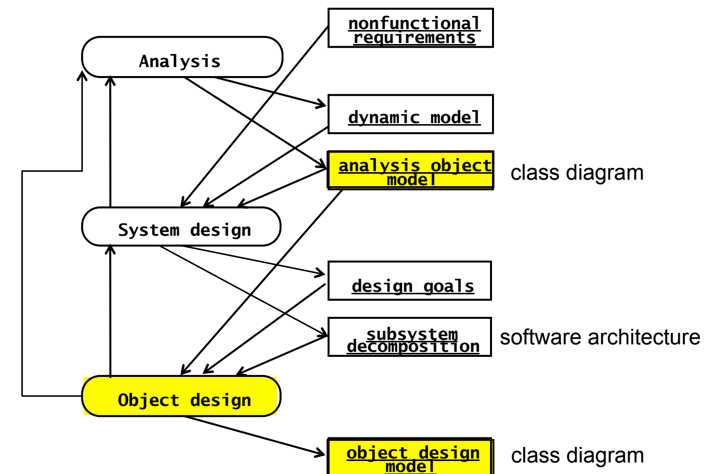
Chapter 8:
Object design: Reusing Pattern Solutions

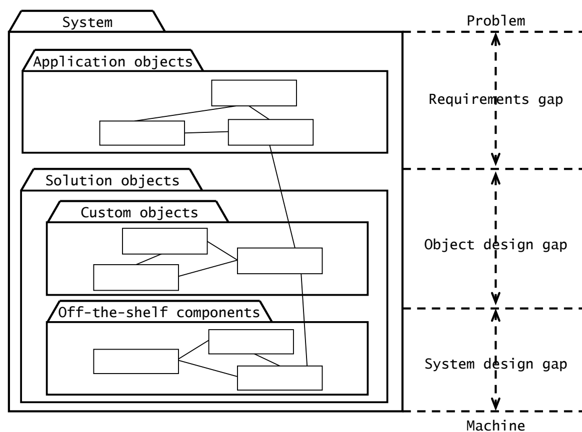
CS 4354
Fall 2012

Jill Seaman

Review: Activities and products of System design



Object Design: closing the gap



Object design closes the gap between application objects identified during requirements and off-the-shelf components selected during system design.

Main activities of Object Design

- **Reuse**: Developers identify off-the-shelf components and design patterns to make use of existing solutions Chapter 8
- **Interface specification**: Developers precisely describe class interface(s) to represent each subsystem interface.
  - ✦The subsystem API Chapter 9
- **Restructuring**: Developers transform the object design model to improve its understandability and extensibility.
  - ✦In order to meet design goals. Chapter 10
- **Optimization**: Developers transform the object design model to address performance criteria.
  - ✦Such as response time or memory utilization. Chapter 10

8.3 Reuse Concepts
8.3.1 Application Objects and Solution Objects

- **Application Objects,** also called “domain objects”, represent concepts of the domain that are relevant to the system.
  - ✦Primarily entity objects, identified during analysis.
  - ✦Independent of any system.
- **Solution Objects** represent components that do not have a counterpart in the application domain, such as persistent data stores, user interface objects, or middleware.
  - ✦Includes boundary and control objects, identified during analysis.
  - ✦More solution objects are identified during system design and object design, as part of their proecesses

More Reuse Concepts
8.3.2 Specification Inheritance+Implementation Inheritance

- **Specification Inheritance** is the classification of concepts into type hierarchies
  - ✦Conceptually, subclass is a specialization of its superclass.
  - ✦Conceptually, superclass is a generalization of all of its subclasses.
- **Implementation Inheritance** is the use of inheritance for the sole purpose of reusing code (from the superclass).
  - ✦the generalization/specialization relationship is usually lacking.
  - ✦example: Set implemented by inheriting from Hashtable

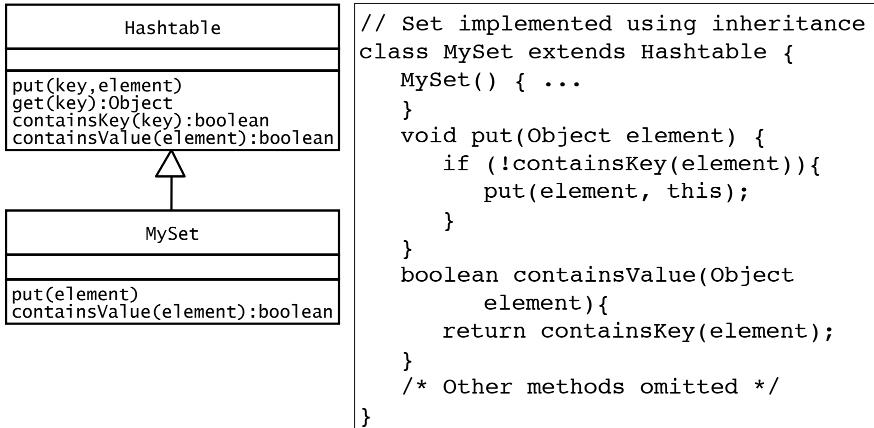
Java Hashtable

- **Description:** This class implements a hashtable, which maps keys to values.
  - ✦Any non-null object can be used as a key or as a value.
- **Hashtable methods**
  - ✦put(key,element)
    Maps the specified key to the specified value in this hashtable.
  - ✦get(key) : Object
    Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
  - ✦containsKey(key): boolean
  - ✦containsValue(element):boolean

Set

- The interface to be implemented:
- **Description:** A collection that contains no duplicate element.
- **Set methods**
  - ✦put(element)
    Adds the specified element to this set if it is not already present
  - ✦containsValue(element):boolean
    Returns true if the element is in the set, else false.

Set implemented by extending Hashtable

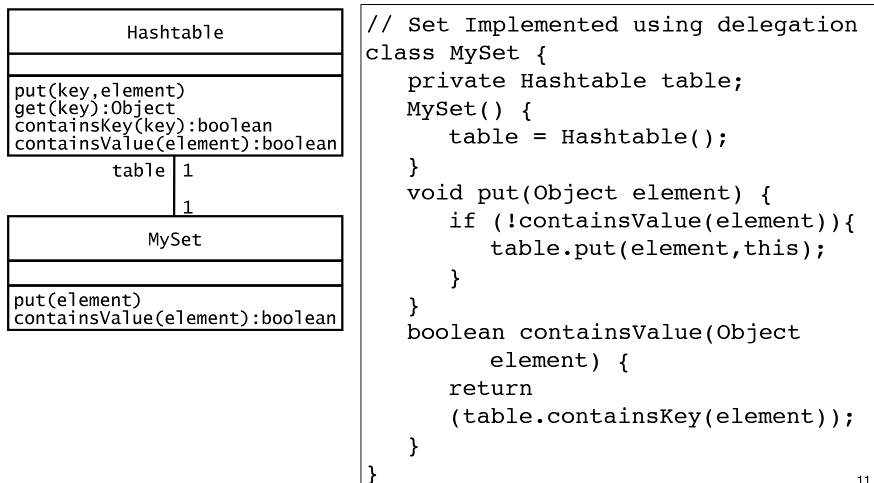



```
// Set implemented using inheritance
class MySet extends Hashtable {
   MySet() { ...
   }
   void put(Object element) {
      if (!containsKey(element)){
         put(element, this);
      }
   }
   boolean containsValue(Object
         element){
      return containsKey(element);
   }
   /* Other methods omitted */
}
```


Evaluation of the inheritance version

- **Good:** code reuse
- **Bad:** Set is not a specialization of Hashtable
  - ✦it inherits methods that don't make sense for it:
    put(key, element), containsKey()
    Potential problem: a client class uses these methods on MySet, and then MySet is re-implemented by inheriting from some other class (like List).
  - ✦it doesn't work as a Hashtable
    It cannot be used correctly as a special kind of Hashtable (ie passed to a function that takes Hashtable as an argument)
    Specifically containsValue() will not work as expected.
- **Liskov Substitution Property:** if S is a subclass of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program. [Wikipedia]

Set implemented using composition/delegation




```
// Set Implemented using delegation
class MySet {
   private Hashtable table;
   MySet() {
      table = Hashtable();
   }
   void put(Object element) {
      if (!containsValue(element)){
         table.put(element,this);
      }
   }
   boolean containsValue(Object
         element) {
      return
      (table.containsKey(element));
   }
}
```


8.3.3 Delegation

- **Delegation:** A special form of composition
  - ✦One class (A) contains a reference to another (B) (via member variable)
  - ✦A implements its operations by calling methods on B.
    (Methods may have different names)
  - ✦Makes explicit the dependencies between A and B.
- Addresses problems of implementation inheritance:
  - ✦Extensibility (allowing for change to implementation)
    Internal representation of A can be changed without impacting clients of A (methods of B are not exposed via A like they would be in inheritance)
  - ✦Subtyping
    A is not a special case of B so it cannot be accidentally used as a special kind of B. (Does not violate LSP, because it does not apply)

8.3.5 Design Patterns

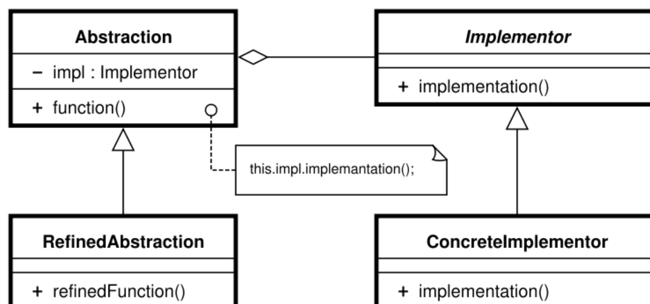
- In object-oriented development, **Design Patterns** are solutions that developers have refined over time to solve a range of recurring problems.
- A design pattern has four elements
  - ✦A **name** that uniquely identifies the pattern from other patterns.
  - ✦A **problem description** that describes the situation in which the pattern can be used. [They usually address modifiability and extensibility design goals.]
  - ✦A **solution** stated as a set of collaborating classes and interfaces.
  - ✦A **set of consequences** that describes the trade-offs and alternatives to be considered with respect to the design goals being addressed.

Design Patterns

- The following terms are used to describe the class that collaborate in a design pattern:
  - ✦The **client class** access the pattern classes
  - ✦The **pattern interface** is the part of the pattern that is visible to the client class (might be an interface or abstract class).
  - ✦The **implementor class** provides low level behavior of the pattern, usually more than one.
  - ✦The **extender class** specializes an implementor class to provide different implementation of ht pattern. Usually represent future classes anticipated by the developer.
- Tradeoff: Simple architecture vs anticipating change (extensibility)
  - ✦ Agile methods: use refactoring to adopt patterns when need arises

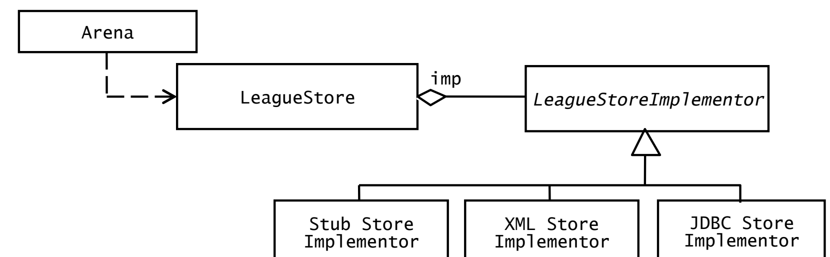
8.4.1 Encapsulating Data Stores with the Bridge Pattern

Name: Bridge Design Pattern
Problem Description: Decouple an interface from an implementation so that the two can vary independently.
Solution: Abstraction is visible to the Client. **Abstraction** maintains a reference to its corresponding **Implementor** instance



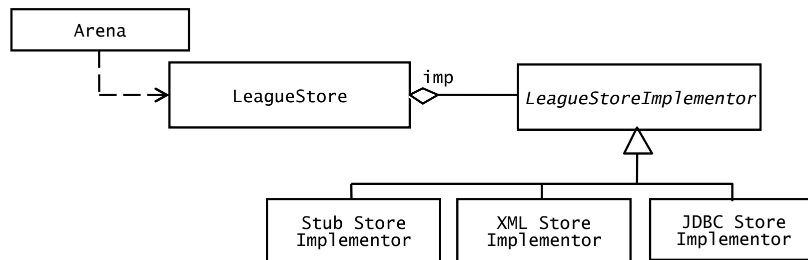
Example: Abstracting database vendors

- Arena is the client, LeagueStore is the abstraction class
- LeagueStoreImplementor is the common interface for various implementations of the storage.
  - ✦Stub is for the prototype, XML is for a flat file system, JDBC is for a database.



Example: Abstracting database vendors

- LeagueStore has a different interface from LeagueStoreImplementor
  - ✦LeagueStore may provide higher level functions
  - ✦LeagueStoreImplementor may provide lower level functionality
- LeagueStore can have its own subclasses



Bridge Pattern example: Shapes

```
/** Implementor */
interface DrawingAPI {
  public void drawCircle(double x, double y, double radius);
}

/** ConcreteImplementor 1/2 */
class DrawingAPI1 implements DrawingAPI {
  public void drawCircle(double x, double y, double radius) {
    System.out.printf("API1.circle at %f:%f radius %f\n", x, y, radius);
  }
}

/** ConcreteImplementor 2/2 */
class DrawingAPI2 implements DrawingAPI {
  public void drawCircle(double x, double y, double radius) {
    System.out.printf("API2.circle at %f:%f radius %f\n", x, y, radius);
  }
}
```


Bridge Pattern example: Shapes

```
/** Abstraction */
interface Shape {
   public void draw();
   public void resizeByPercentage(double pct);
}

/** Refined Abstraction */
class CircleShape implements Shape {
  private double x, y, radius;
  private DrawingAPI drawingAPI;

  public CircleShape(double x, double y, double radius,
                     DrawingAPI drawingAPI) {
      this.x = x; this.y = y; this.radius = radius;
      this.drawingAPI = drawingAPI;
   }
   public void draw() {
      drawingAPI.drawCircle(x, y, radius);
   }
   public void resizeByPercentage(double pct) {
      radius *= pct;
   }
}
```


Bridge Pattern example: Shapes

```
/** Client */
public class BridgePattern {
  public static void main(String[] args) {
    Shape[] shapes = new Shape[2];
    shapes[0] = new CircleShape(1, 2, 3, new DrawingAPI1());
    shapes[1] = new CircleShape(5, 7, 11, new DrawingAPI2());

    for (Shape shape : shapes) {
      shape.resizeByPercentage(2.5);
      shape.draw();
    }
  }
}
```

Output:

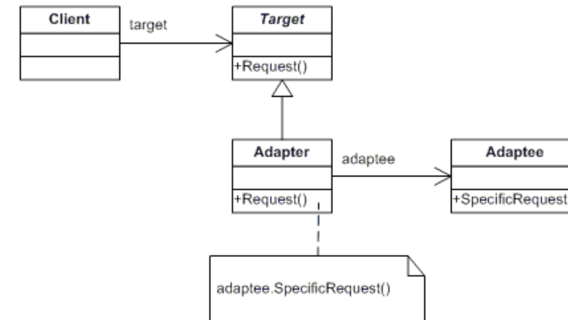
```
API1.circle at 1.000000:2.000000 radius 7.500000
API2.circle at 5.000000:7.000000 radius 27.500000
```

Bridge Pattern: consequences

- Client is shielded from abstract and concrete implementations.
- Interfaces and implementations can be refined independently. The implementation can be chosen (or changed) at runtime
- Improved extensibility: you can extend Abstraction and Implementor hierarchies independently
- Question: Where does the Bridge Pattern use inheritance? Where does it use delegation?

8.4.2 Encapsulating Legacy Components with the Adapter Pattern

Name: Adapter Design Pattern
Problem Description: Convert the interface of a legacy class into a different interface expected by the client, so they can work together without changes.
Solution: Adapter class implements the **Target** interface expected by the client. The **Adapter** delegates requests from the client to the **Adaptee** (the legacy class) and performs any necessary conversion.



Example: Sorting Strings in a java Array

- Array.sort method expects an Array and a Comparator
  - ✦Comparator has a compare() method
  - ✦MyString defines greaterThan() and equals() methods
  - ✦MyStringComparator provides a compare method in terms of the methods in MyString

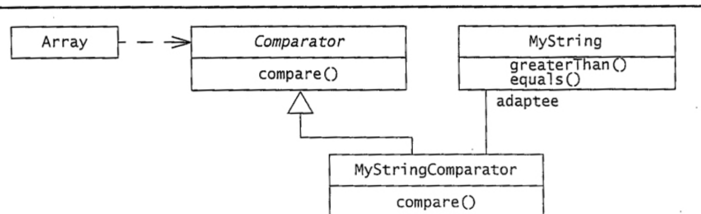


Figure 8-8 Applying the Adapter design pattern for sorting Strings in an Array (UML class diagram). See also source code in Figure 8-9.

Adapter Pattern example: Sorting strings

```
// Existing Target interface
interface Comparator {
   int compare (Object o1, Object o2);
}
// Existing Client
class Array {
   public static void sort (Object [] a, Comparator c) {
      System.out.print("Sorting");
   }
}
// Existing Adaptee class (legacy)
class MyString  {
   String s;
   public MyString(String x)
   { s = x; }
   public boolean equals (Object o)
   { return s.equals(o); }
   boolean greaterThan (MyString s1)
   { return s.equals(s1); }
}
```

Adapter Pattern example: Sorting strings

```
// New Adapter class
class MyStringComparator implements Comparator {
   public int compare(Object o1, Object o2) {
      int result;
      if (((MyString) o1).greaterThan((MyString)o2)) {
         result = 1;
      } else if (((MyString) o1).equals((MyString)o2)) {
         result = 0;
      } else
         result = -1;
      return result;
   }
}
public class AdapterPattern {
   public static void main(String[] args) {
     MyString[] x =  { new MyString ("B"),new MyString ("A") };
     MyStringComparator c = new MyStringComparator();
     Array.sort ( x,c ) ;
   }
}
```

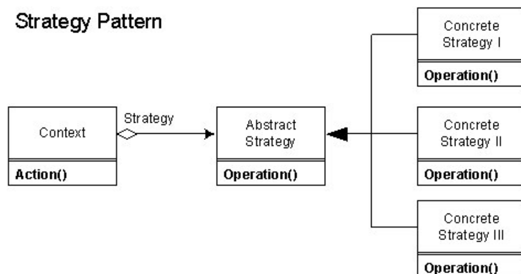

Adapter Pattern: consequences

- Client and Adaptee work together without any modification to either.
- Adapter works with Adaptee and all of its sub classes
- A new Adapter needs to be written for each specialization (subclass) of Target.

- Question: Where does the Adapter Pattern use inheritance? Where does it use delegation?
- How is it different from the Bridge Pattern?

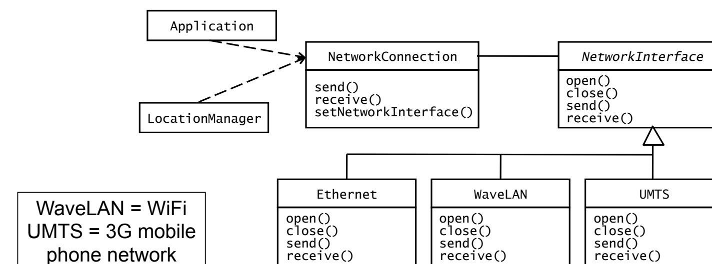
8.4.3 Encapsulating Context with the Strategy Pattern

Name: Strategy Design Pattern
Problem Description: Define a family of algorithms, encapsulate each one, and make them interchangeable. The algorithm is decoupled from the client.
Solution: A Client accesses services provided by a Context. The **Context** is configured to use one of the **ConcreteStrategy** objects (and maintains a reference to it) . The **AbstractStrategy** class describes the interface that is common to all the ConcreteStrategies.



Example: switching between network protocols

- Based on location (available network connections), switch between different types of network connections
  - ✦LocationManager configures NetworkConnection with a concrete NetworkInterface based on the current location
  - ✦Application uses the NetworkConnection independently of concrete NetworkInterfaces (NetworkConnection uses delegation).



WaveLAN = WiFi
UMTS = 3G mobile phone network

Strategy Pattern example: Network protocols

```
// Context Object: Network Connection
public class NetworkConnection {
   private String destination;
   private NetworkInterface intf;
   private StringBuffer queue;

   public NetworkConnect(String destination, NetworkInterface intf) {
      this.destination = destination;  this.intf = intf;
      this.intf.open(destination);
   }
   public void send(byte msg[]) {
      queue.concat(msg);
      if (intf.isReady()) {
         intf.send(queue);
         queue.setLength(0);
      }
   }
   public byte[] receive () {
      return intf.receive();
   }
   public void setNetworkInterface(NetworkInterface newIntf) {
      intf.close()
      newIntf.open(destination);
      intf = newIntf;
}  }
```


Strategy Pattern example: Network protocols

```
//Abstract Strategy,
//Implemented by EthernetNetwork, WaveLanNetwork, and UMTSNetwork
interface NetworkInterface {
   void open(String destination);
   void close();
   byte[] receive();
   void send(StringBuffer queue);
   bool isReady();
}
//LocationManager: decides on which strategy to use
public class LocationManager {
   private NetworkConnection networkConn;

   // called by event handler when location has changed
   public void doLocation() {
      NetworkInterface networkIntf;
      if (isEthernetAvailable())
         networkIntf = new EthernetNetwork();
      else if (isWaveLANAvailable())
         networkIntf = new WaveLanNetwork();
      else if (isUMTSAvailable())
         networkIntf = new UMTSNetwork();
      networkConn.setNetworkInterface(networkIntf);
   }
}
```


Strategy Pattern: consequences

- ConcreteStrategies can be substituted transparently from Context.
- Client (or Policy) decides which Strategy is best, given current circumstances
- New algorithms can be added without modifying Context or Client

- Question: How is Strategy pattern different from the Bridge Pattern?
  - ✦Bridge is a structural pattern, Strategy is a behavioral pattern
  - ✦Bridge implementations are subsystems, Strategies encapsulate algorithms
  - ✦Bridge implementation often created at system initialization, Strategies created on the fly.
  - ✦Bridge Abstraction sets up its own implementation, Strategy context is usually configured by another object (Policy).

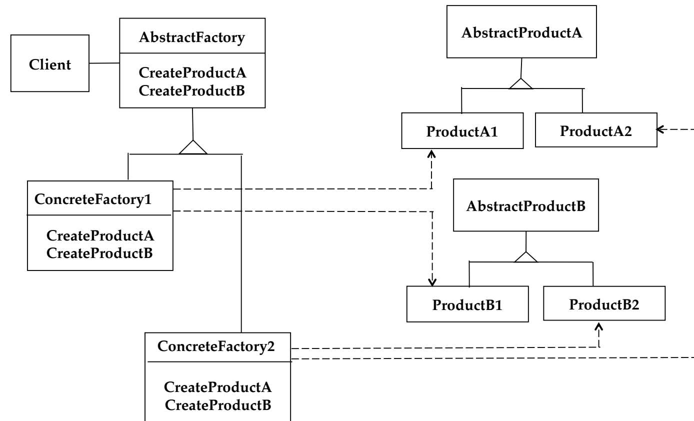
8.4.4 Encapsulating Platforms with the Abstract Factory Pattern

Name: Abstract Factory Design Pattern
Problem Description: Shield the client from different platforms that provide different implementations for the same set of concepts.

Solution:
A platform is represented as a set of **AbstractProducts**, each representing a concept (class) that is supported by all platforms.
An **AbstractFactory** class declares the operations for creating each individual product.
A specific platform is then realized by a **ConcreteFactory** and a set of **ConcreteProducts** (one for each AbstractProduct).
A ConcreteFactory depends only on its related ConcreteProducts.

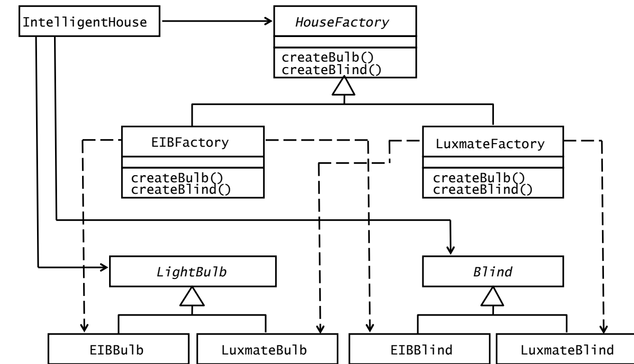
The **Client** depends only on the AbstractProducts and the AbstractFactory classes, making it easy to substitute platforms.

The Abstract Factory Pattern (solution diagram)



Example:
A Facility Management System for a House

- Devices from the two manufacturers (EIB and Luxmate) are NOT interoperable.



Abstract Factory Pattern example: IntelligentHouse

```
abstract class HouseFactory {
    public static HouseFactory getFactory() {
        int man = readFromConfigFile("MANUFACTURER_TYPE");
        if (man == 0)
            return new EIBFactory();
        else
            return new LuxmateFactory();
    }
    public abstract LightBulb createBulb();
    public abstract Blind createBlind();
}

class EIBFactory extends HouseFactory {
    public LightBulb createBulb() {
        return new EIBBulb();
    }
    public Blind createBlind() {
        return new EIBBlind();
    }
}
```


Abstract Factory Pattern example: IntelligentHouse

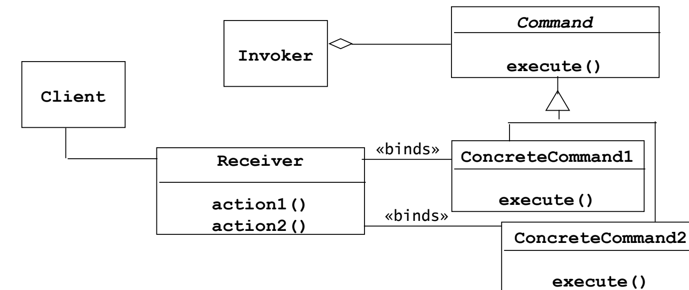
```
class LuxmateFactory extends HouseFactory {
    public LightBulb createBulb() {
        return new LuxmateBulb();
    }
    public Blind createBlind() {
        return new LuxmateBlind();
    }
}
//TBD: LightBulb, EIBBulb, LuxmateBulb
//TBD: Blind, EIBBlind, LuxmateBlind

// IntelligentHouse is not aware of EIB or Luxmate
public class IntelligentHouse {
    public static void main(String[] args) {
        HouseFactory factory = HouseFactory.getFactory();
        LightBulb bulb = factory.createBulb();
        bulb.switchOn();
    }
}
```

Abstract Factory Pattern: consequences

- Client is shielded from concrete product classes.
- Substituting families at runtime is possible
- Adding new products is difficult since new realizations for each factory must be created, AbstractFactory must be changed.

8.4.5 Encapsulating Control Flow with the Command Pattern

Name: Command Design Pattern
Problem Description: Encapsulate requests so that they can be executed, undone, or queued independently of the request.
Solution: A **Command** abstract class declares the interface supported by all ConcreteCommands. **ConcreteCommands** encapsulate a service to be applied to a Receiver. The **Client** creates ConcreteCommands and binds them to specific **Receivers**. The **Invoker** actually executes or undoes a command, which delegates the execution to an action of the **Receiver**.



Command Pattern example: Light switch

```
/* The Command interface */
public interface Command {
   void execute();
}

/* The Invoker class */
public class Switch {
   private List<Command> history = new ArrayList<Command>();
   public void storeAndExecute(Command cmd) {
      this.history.add(cmd); // optional
      cmd.execute();
   }
}

/* The Receiver class */
public class Light {
   public void turnOn() {
      System.out.println("The light is on");
   }
   public void turnOff() {
      System.out.println("The light is off");
   }
}
```


Command Pattern example: Light switch

```
/* The Command for turning on the light - ConcreteCommand #1 */
public class FlipUpCommand implements Command {
   private Light theLight;
   public FlipUpCommand(Light light) {
      this.theLight = light;
   }
   public void execute(){
      theLight.turnOn();
   }
}

/* The Command for turning off the light - ConcreteCommand #2 */
public class FlipDownCommand implements Command {
   private Light theLight;
   public FlipDownCommand(Light light) {
      this.theLight = light;
   }
   public void execute() {
      theLight.turnOff();
   }
}
```

Command Pattern example: Light switch

```
/* The test class or client */
public class PressSwitch {
   public static void main(String[] args){
      Light lamp = new Light();
      Command switchUp = new FlipUpCommand(lamp);
      Command switchDown = new FlipDownCommand(lamp);
      Switch s = new Switch();
      try {
         if (args[0].equalsIgnoreCase("ON")) {
            s.storeAndExecute(switchUp);
         }
         else if (args[0].equalsIgnoreCase("OFF")) {
            s.storeAndExecute(switchDown);
         }
         else
            System.out.println("Argument \"ON\" or \"OFF\" is required.");
      } catch (Exception e) {
         System.out.println("Arguments required.");
      }
   }
}
```

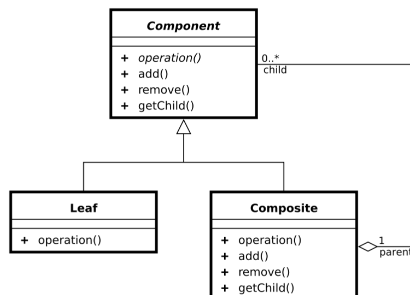

Command Pattern: consequences

- The object of the command (Receiver) and the algorithm of the command (ConcreteCommand) are decoupled.
- Invoker is shielded from specific commands.
- ConcreteCommands are objects. They can be created and stored.
- New ConcreteCommands can be added without changing existing code.

- Question: Where does the Adapter Pattern use inheritance? Where does it use delegation?

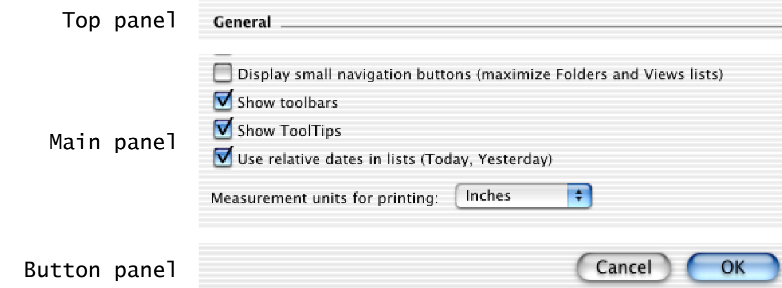
8.4.6 Encapsulating Hierarchies with the Composite Pattern

Name: Composite Design Pattern
Problem Description: Represent a hierarchy of variable width and depth so that leaves and composites can be treated uniformly through a common interface.
Solution: The **Component** interface specifies the services that are shared among Leaf and Composite (operation()). A **Composite** has an aggregation association with Components and implements each service by iterating over each contained Component. The **Leaf** services do most of the actual work.



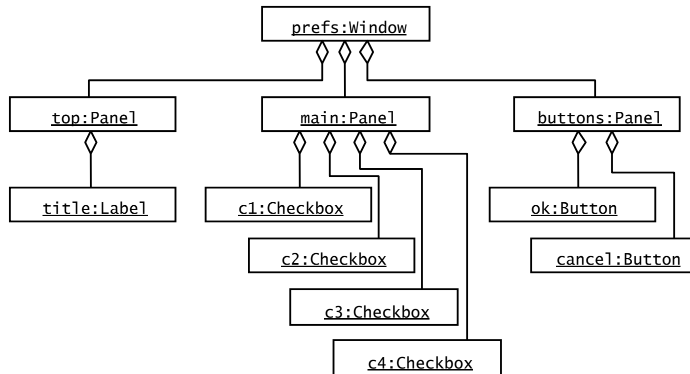
Example: A hierarchy of user interface objects

- Anatomy of a preference dialog. Aggregates, called Panels, are used for grouping user interface objects that need to be resized and moved together.



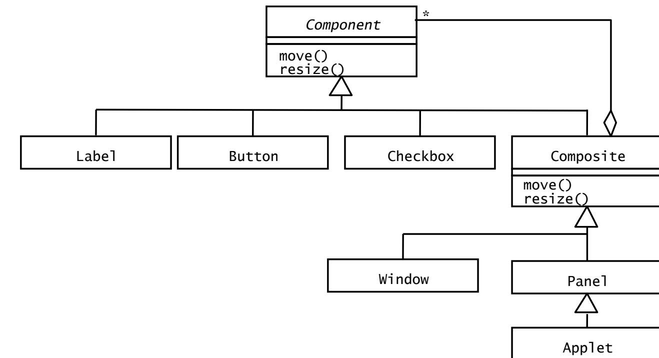
Example: A hierarchy of user interface objects

- An object diagram (it contains instances, not classes) of the previous example:



Example: A hierarchy of user interface objects

- A class diagram, for user interface widgets



Composite Pattern example: File system

```
//Component Node, common interface
interface AbstractFile {
    public void ls();
}

// File implements the common interface, a Leaf
class File implements AbstractFile {
    private String m_name;
    public File(String name) {
        m_name = name;
    }
    public void ls() {
        System.out.println(CompositeDemo.g_indent + m_name);
    }
}
```


Composite Pattern example: File system

```
// Directory implements the common interface, a composite
class Directory implements AbstractFile {
    private String m_name;
    private ArrayList<AbstractFile> m_files = new ArrayList<AbstractFile>();
    public Directory(String name) {
        m_name = name;
    }
    public void add(AbstractFile obj) {
        m_files.add(obj);
    }
    public void ls() {
        System.out.println(CompositeDemo.g_indent + m_name);
        CompositeDemo.g_indent.append("   ");
        for (int i = 0; i < m_files.size(); ++i) {
            AbstractFile obj = m_files.get(i);
            obj.ls();
        }
        CompositeDemo.g_indent.setLength(CompositeDemo.g_indent.length() - 3);
    }
}
```

Composite Pattern example: File system

```
public class CompositeDemo {
    public static StringBuffer g_indent = new StringBuffer();

    public static void main(String[] args) {
        Directory one = new Directory("dir111"),
                  two = new Directory("dir222"),
                  thr = new Directory("dir333");
        File a = new File("a"), b = new File("b"),
             c = new File("c"), d = new File("d"), e = new File("e");
        one.add(a);
        one.add(two);
        one.add(b);
        two.add(c);
        two.add(d);
        two.add(thr);
        thr.add(e);
        one.ls();
    }
}
```

Output:

```
dir111
   a
   dir222
      c
      d
      dir333
         e
   b
```

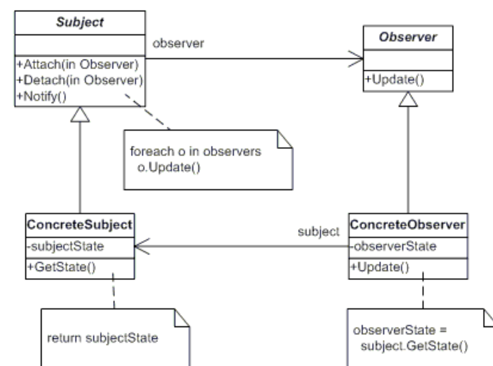

Composite Pattern: consequences

- Client uses the same code for dealing with Leaves or Composites
- Leaf-specific behavior can be modified without changing the hierarchy
- New classes of leaves (and composites) can be added without changing the hierarchy
- Could make your design too general. Sometimes you want composites to have only certain components. May have to add your own run-time checks.

A.7 Decoupling Entities from Views with the Observer Pattern

Name: Observer Design Pattern
Problem Description: Maintain consistency across the states of one Subject and many Observers.

Solution: The **Subject** maintains some state. One or more **Observers** use the state maintained by the Subject. Observers invoke the subscribe() method to register with a Subject. Each **ConcreteObserver** defines an update() method to synchronize its state with the Subject. Whenever the state of the Subject changes, it invokes its notify method, which iteratively invokes each Subscriber.update() method.



Observer Pattern: Java support

- We could implement the Observer pattern “from scratch” in Java. But Java provides the Observable/Observer classes as built-in support for the Observer pattern.
- The java.util.Observer interface is the Observer **interface**. It must be implemented by any observer class. It has one method.
  - void **update** (Observable o, Object arg)
    This method is called whenever the observed object is changed.
    Observable o is the object it is observing.
    Object arg, if not null, is the changed object.

Observer Pattern: Java support

- The java.util.Observable class is the base Subject **class.** Any class that wants to be observed extends this class.
  - public synchronized void **addObserver**(Observer o)
    Adds an observer to the set of observers of this object
  - protected synchronized void **setChanged**()
    Indicates that this object has changed
  - public void **notifyObservers**(Object arg)
  - public void **notifyObservers**()
    If this Observable object has changed, then notify all of its observers. Each observer has its update() method called with this Observable object and the arg argument. The arg argument can be used to indicate which attribute of this Observable object has changed.

Observer Pattern example:

```
import java.util.Observable;

/* A subject to observe! */
public class ConcreteSubject extends Observable {
   private String name;
   private float price;
   public ConcreteSubject(String name, float price) {
       this.name = name;
       this.price = price;
       System.out.println("ConcreteSubject created: " + name + " at " + price);
   }
   public String getName() {return name;}
   public float getPrice() {return price;}
   public void setName(String name) {
        this.name = name;
        setChanged();
        notifyObservers(name);
   }
   public void setPrice(float price) {
       this.price = price;
       setChanged();
       notifyObservers(new Float(price));
   }
}
```


Observer Pattern example:

```
import java.util.Observable;
import java.util.Observer;

//An observer of name changes.
public class NameObserver implements Observer {
    private String name;

    public NameObserver() {
        name = null;
        System.out.println("NameObserver created: Name is " + name);
    }

    public void update(Observable obj, Object arg) {
        if (arg instanceof String) {
            name = (String) arg;
            System.out.println("NameObserver: Name changed to " + name);
        } else {
            System.out.println("NameObserver: Some other change to subject!");
        }
    }
}
```


Observer Pattern example:

```
import java.util.Observable;
import java.util.Observer;

//An observer of price changes.
public class PriceObserver implements Observer {
    private float price;

    public PriceObserver() {
        price = 0;
        System.out.println("PriceObserver created: Price is " + price);
    }

    public void update(Observable obj, Object arg) {
        if (arg instanceof Float) {
            price = ((Float) arg).floatValue();
            System.out.println("PriceObserver: Price changed to " + price);
        } else {
            System.out.println("PriceObserver: Some other change to subject!");
        }
    }
}
```

Observer Pattern example:

```
//Test program for ConcreteSubject, NameObserver and PriceObserver
public class TestObservers {
  public static void main(String args[]) {
    // Create the Subject and Observers.
    ConcreteSubject s = new ConcreteSubject("Corn Pops", 1.29f);
    NameObserver nameObs = new NameObserver();
    PriceObserver priceObs = new PriceObserver();
    // Add those Observers!
    s.addObserver(nameObs);
    s.addObserver(priceObs);
    // Make changes to the Subject.
    s.setName("Frosted Flakes");
    s.setPrice(4.57f);
    s.setPrice(9.22f);
    s.setName("Sugar Crispies");
  }
}
```

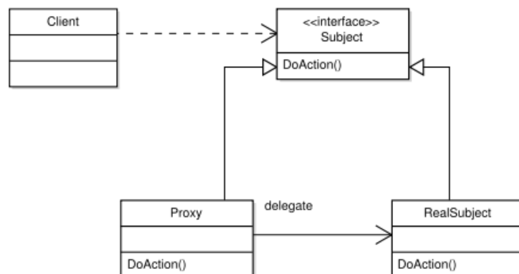
```
ConcreteSubject created: Corn Pops at 1.29
NameObserver created: Name is null
PriceObserver created: Price is 0.0
PriceObserver: Some other change to subject!
NameObserver: Name changed to Frosted Flakes
PriceObserver: Price changed to 4.57
NameObserver: Some other change to subject!
PriceObserver: Price changed to 9.22
NameObserver: Some other change to subject!
PriceObserver: Some other change to subject!
NameObserver: Name changed to Sugar Crispies
```


Observer Pattern: consequences

- Decouples a Subject from the Observers. Subject knows only that it contains a list of Observers, each with an update() method. They can belong to different layers.
- Observers can change or be added without changing Subject.
- Observers can ignore notifications (decision is not made by Subject).
- Can result in many spurious broadcasts (and calls to getState()) when the state of a Subject changes.

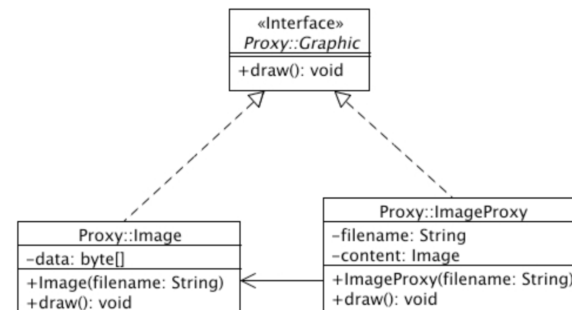
A.8 Encapsulating Expensive Objects with the Proxy Pattern

Name: Proxy Design Pattern
Problem Description: Improve the performance or security of a system by delaying expensive computations, using memory only when needed, or checking access before loading an object into memory.
Solution: The **Proxy** class acts on behalf of a **RealSubject** class. Both classes implement the same **Subject** interface. They Proxy stores a subset of the attributes of the RealSubject. The Proxy handles certain requests completely, whereas others are delegated to the RealSubject.



Example: Delayed loading of image content

- ImageProxy contains the filename of the image. Its reference to the Image can be null until the draw method is called. Then it creates the Image object using the filename.



Proxy Pattern example:

```
public interface Graphic {

      // a method used to draw the image
      public void draw();
}

public class Image implements Graphic {

    private byte[] data;

    public Image(String filename) {
        // Load the image
        data = loadImage(filename);
    }

    public void draw() {
        // Draw the image
        drawToScreen(data);
    }
}
```


Proxy Pattern example:

```
public class ImageProxy implements Graphic {

    // Variables to hold the concrete image
    private String filename;
    private Image content;

    public ImageProxy(String filename) {
        this.filename = filename;
        content = null;
    }

    // on a draw-request, load the concrete image
    //   if we haven't done it until yet.
    public void draw() {
        if (content == null) {
            content = new Image(filename);
        }
        // Forward to the Concrete image.
        content.draw();
    }
}
```


Proxy Pattern: consequences

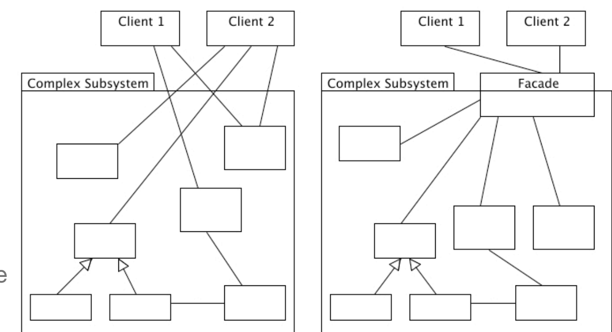
- Adds a level of indirection between Client and RealSubject
  - Can hide the fact that an object is not stored locally
  - Can create a complete object on demand
  - Can make sure caller has access permissions before performing request.
- Note the use of delegation

A.6 Encapsulating Subsystems with the Facade Pattern

Name: Facade Design Pattern

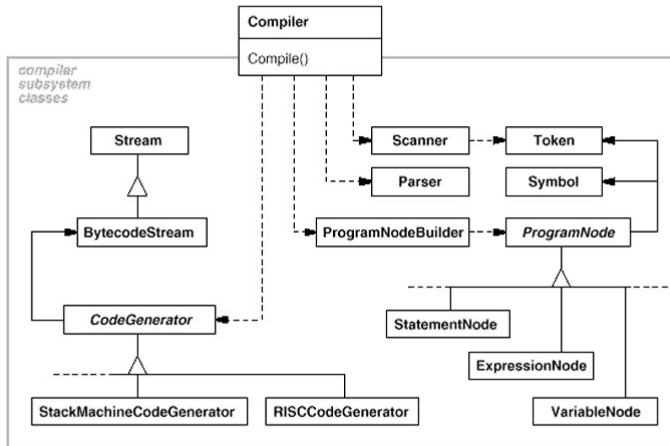
Problem Description: Reduce coupling between a set of related classes and the rest of the system. Provide a simple interface to a complex subsystem.

Solution: A single **Facade** class implements a high-level interface for a subsystem by invoking the methods of lower-level classes. A Facade is opaque in the sense that a caller does not access the lower-level classes directly. The use of Facade patterns recursively yields a layered system.



Example: Compiler subsystem

- Compiler class is a facade hiding the Scanner, Parser, ProgramNodeBuilder and CodeGenerator.



Facade Pattern: consequences

- Shields a client from the low-level classes of a subsystem.
- Simplifies the use of a subsystem by providing higher-level methods.
- Promotes "looser" coupling between subsystems.

- Note the use of delegation to reduce coupling.

8.4.7 Heuristics for Selecting Design Patterns

- Use key phrases from design goals to help choose pattern

Phrase	Design Pattern
"Manufacturer independence" "Platform independence"	Abstract Factory
"Must comply with existing interface" "Must reuse existing legacy component"	Adapter
"Must support future protocols"	Bridge
"All commands should be undoable" "All transactions should be logged"	Command
"Must support aggregate structures" "Must allow for hierarchies of variable depth and width"	Composite
"Policy and mechanisms should be decoupled" "Must allow different algorithms to be interchanged at runtime"	Strategy