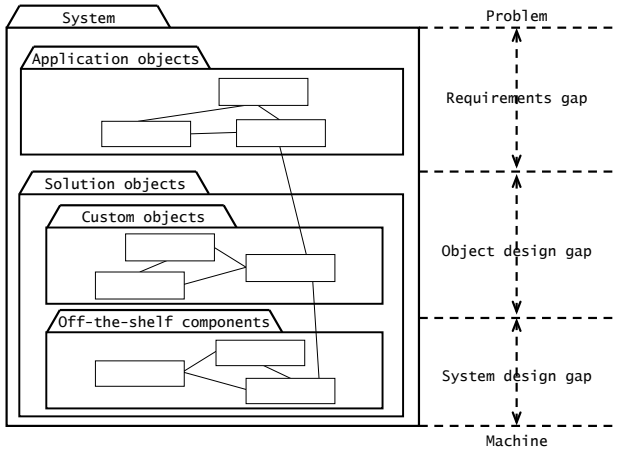
Chapter 8:
Object design: Reusing Pattern Solutions

CS 4354
Summer II 2014

Jill Seaman

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.

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 object design, as part of their processes

Specification Inheritance and 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 (or backwards).
  - ✦example: Set implemented by inheriting from Hashtable

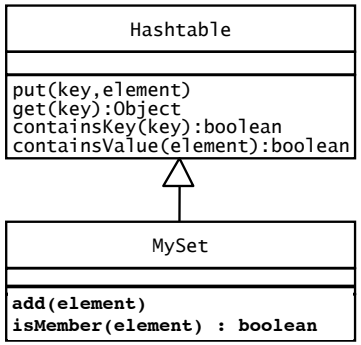
Java Hashtable

- **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**
  - ✦add(element)
    Adds the specified element to this set if it is not already present
  - ✦isMember(element):boolean
    Returns true if the element is in the set, else false.

Set implemented by extending Hashtable

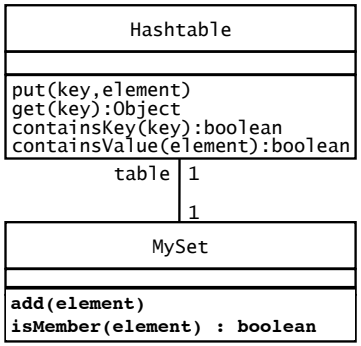



```
// Set implemented using inheritance
class MySet extends Hashtable {
  MySet() { ...
  }
  void add(Object element) {
    if (!containsKey(element)){
      put(element, this);
    }
  }
  boolean isMember(Object element){
    return containsKey(element);
  }
}
```


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 add(Object element) {
    if (!containsValue(element)){
      table.put(element,this);
    }
  }
  boolean isMember(Object element) {
    return (table.containsKey(element));
  }
}
```


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)

Design Patterns

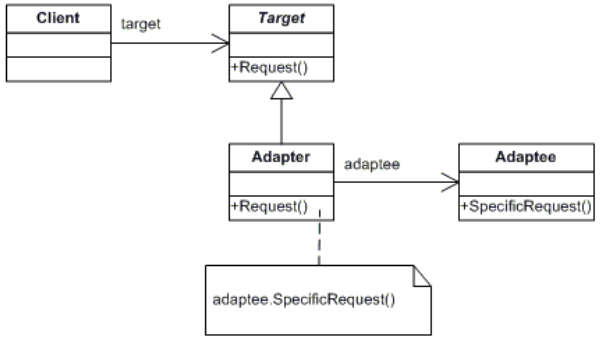
- In object-oriented development, **Design Patterns** are solutions that developers have refined over time to solve a range of recurring problems.
- A design pattern has four elements
  - ✦A **name** that uniquely identifies the pattern from other patterns.
  - ✦A **problem description** that describes 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 denote the classes that collaborate in a design pattern:
  - ✦The **client class** accesses 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. Often the pattern contains many of these.
  - ✦The **extender class** specializes an implementor class to provide different implementation of the pattern. These usually represent future classes anticipated by the developer.
- Common tradeoff: Simple architecture vs extensibility
  - ✦ Agile methods: use refactoring to adopt patterns when need arises (**not** anticipating change like the book describes).

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 pre-existing legacy class) and performs any necessary conversion.



Example: Sorting Strings in a java Array

- Array.sort method (not shown) 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 via delegation

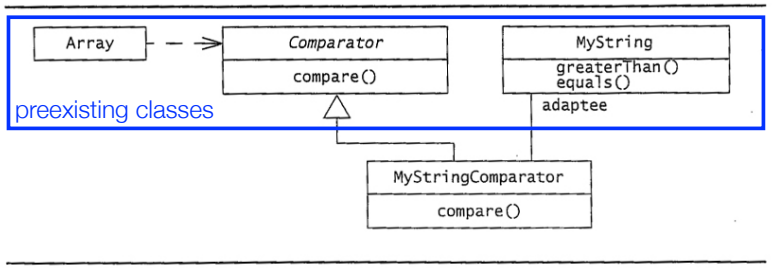


Figure 8-8 Applying the Adapter design pattern for sorting Strings in an Array (UML class diagram). See also source code in Figure 8-9.

Adapter Pattern example: Sorting strings

```
package mine;
// Existing Target interface
interface Comparator {
    int compare (Object o1, Object o2);
}
// Existing Client
class Array {
    public static void sort (Object [] a, Comparator c) {
        //implementation hidden
    }
}
// Existing Adaptee class (legacy)
class MyString  {
    String s;
    public MyString(String x)
    { s = x; }
    public boolean equals (Mystring s1)
    { // implementation hidden }
    boolean greaterThan (MyString s1)
    { // implementation hidden }
}
```


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 ) ;
    }
}
```

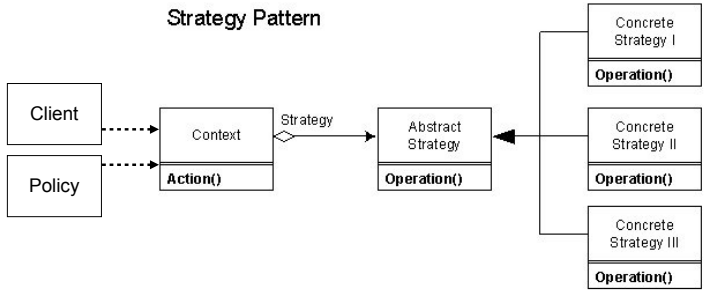
Adapter Pattern: consequences

- 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?

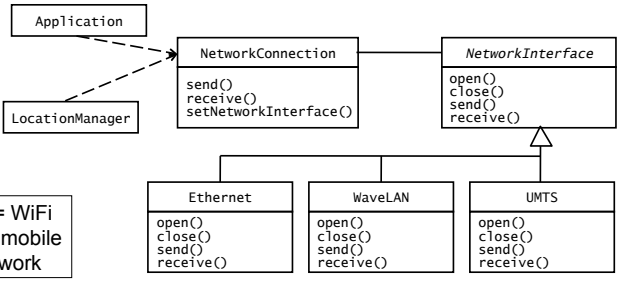
Encapsulating Context with the **Strategy Pattern**

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 (not shown)
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);
   }
}
```

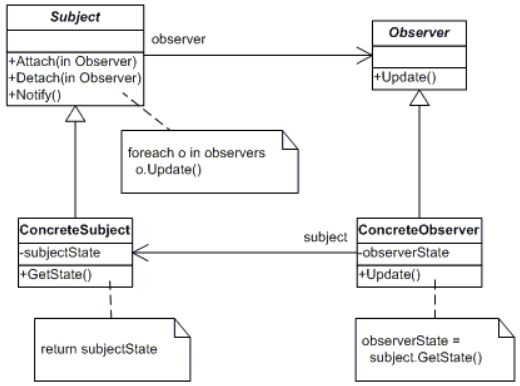

Strategy Pattern: consequences

- 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

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 attach() 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 Observer.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. (The subject and observers 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.