Chapter 8:
Object design: Reusing Pattern Solutions (Part II)

CS 4354
Summer II 2014

Jill Seaman

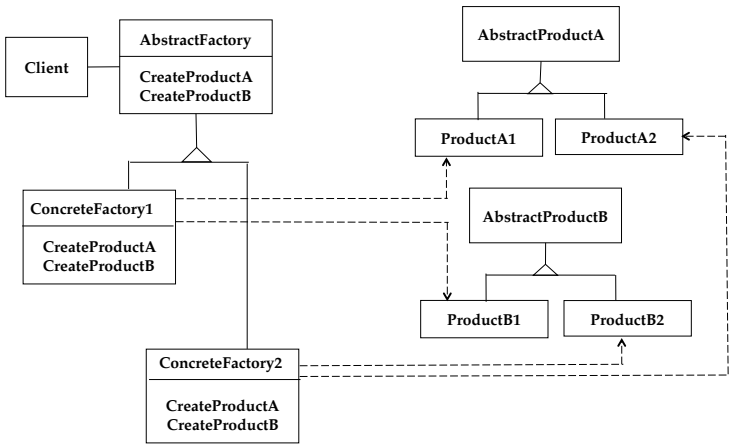
Encapsulating Platforms with the **Abstract Factory Pattern**

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.

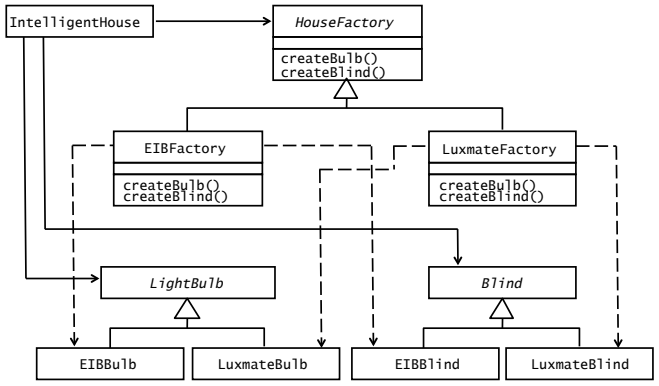
The **Client** depends only on the AbstractProducts and the AbstractFactory classes, making it easy to substitute platforms.

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

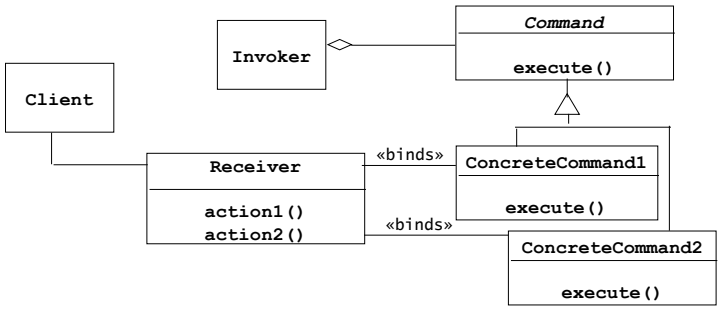
```
class LuxmateFactory extends HouseFactory {
    public LightBulb createBulb() {
        return new LuxmateBulb();
    }
    public Blind createBlind() {
        return new LuxmateBlind();
    }
}
//TBD: LightBulb, EIBBulb, LuxmateBulb
//TBD: Blind, EIBBlind, LuxmateBlind

// 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 families (platforms) is fairly easy.
- Adding new products is somewhat difficult since new realizations for each factory must be created, AbstractFactory must be changed.
  - However, the AbstractProducts are not required to be Abstract. They can represent a default or generic version of each product that can be used for any platform.

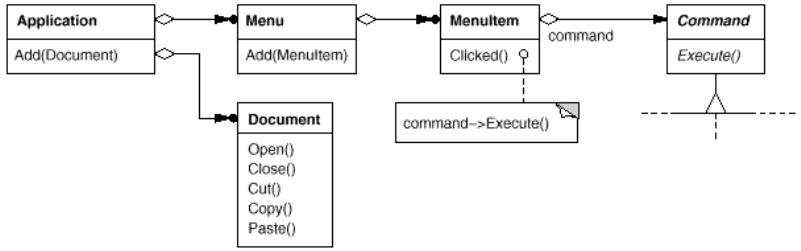
Encapsulating Control Flow with the **Command Pattern**

Name: Command Design Pattern
Problem Description: Encapsulate requests as objects so that they can be executed, undone, logged or queued independently of the request.
Solution: A **Command** abstract class declares the interface for executing an action. **ConcreteCommands** encapsulate an action carried out by a Receiver. The **Client** creates ConcreteCommands and binds them to specific **Receiver** actions. The **Invoker** executes a command, which delegates the execution to an action of the **Receiver**.



Command Pattern: UI Menu

- MenuItem needs to issue messages to objects without knowing anything about the method or the the object.
- When MenuItem is clicked, it will execute it's command, without knowing anything specific about the class or operation that is triggered (Document.Open() perhaps).



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.");
      }
   }
}
```

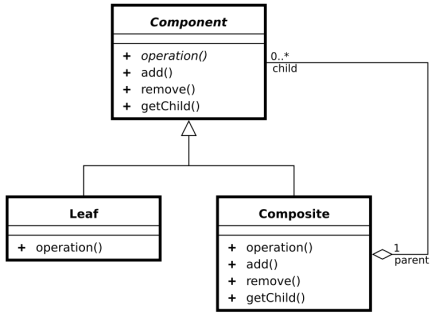
Command Pattern: consequences

- 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 Command Pattern use inheritance? Where does it use delegation?

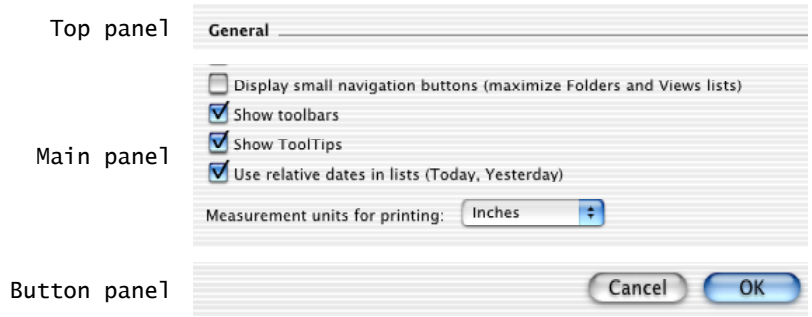
Encapsulating Hierarchies with the **Composite Pattern**

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.



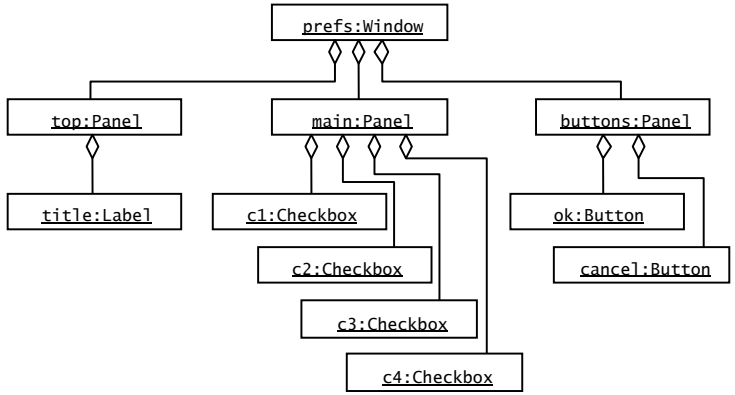
Example: A hierarchy of user interface objects

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



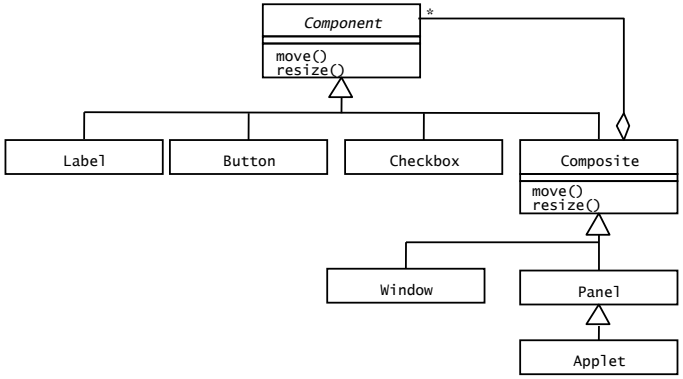
Example: A hierarchy of user interface objects

- 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("   ");  // add 3 spaces
        for (int i = 0; i < m_files.size(); ++i) {
            AbstractFile obj = m_files.get(i);
            obj.ls();
        }
        //remove the 3 spaces:
        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:

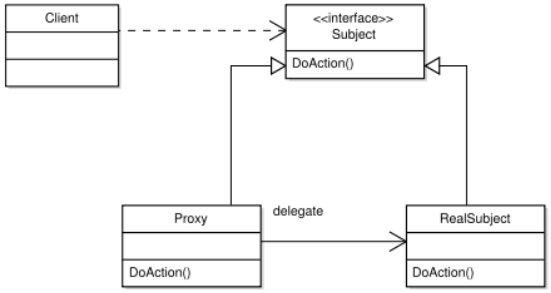
```
dir111
   a
   dir222
      c
      d
      dir333
         e
   b
```

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.

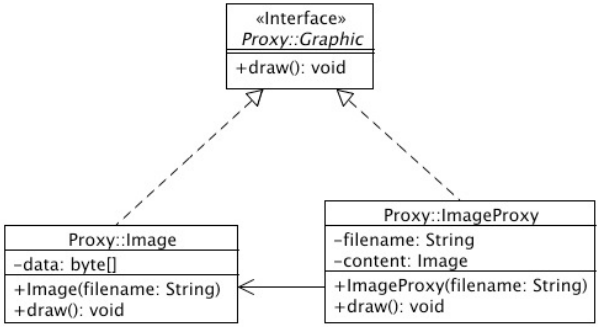
Encapsulating Expensive Objects with the **Proxy Pattern**

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 (content) 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:

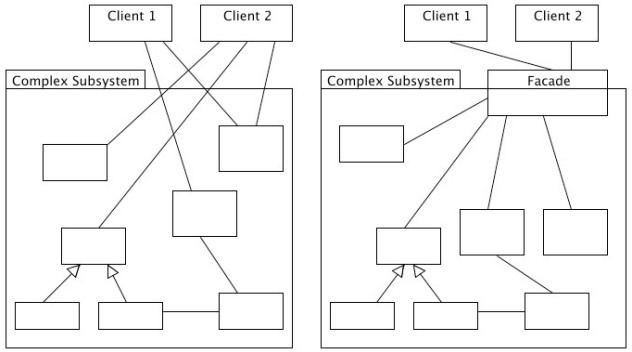
```
public class ImageProxy implements Graphic {

    // 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 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

Encapsulating Subsystems with the **Facade Pattern**

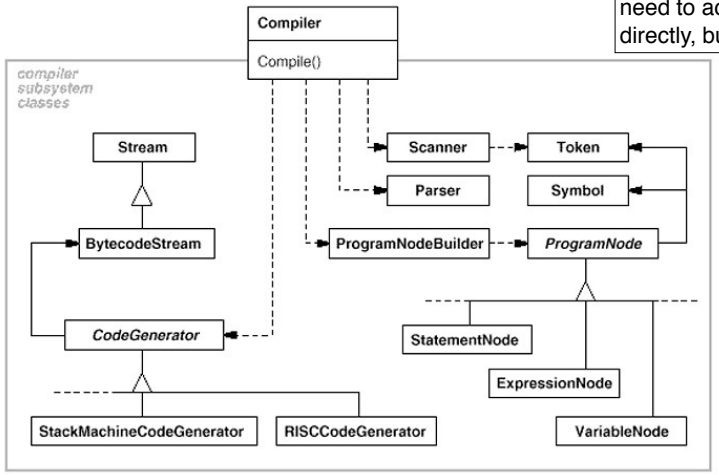
Name: Facade Design Pattern
Problem Description: Reduce coupling between a set of related classes and the rest of the system. Provide a simple interface to a complex subsystem.
Solution: A single **Facade** class implements a high-level interface for a subsystem by invoking the methods of lower-level 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.

Some specialized apps might need to access the classes directly, but most don't.



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.

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 be notified of changes”	Observer
“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

Reuse activity: Identifying off-the-shelf components

- An **application framework** is a reusable partial application that can be specialized to produce custom applications.
- They are targeted to particular technologies, such as data processing, cellular communications, or user interfaces.
  - Java Swing, Mac OSX Cocoa (AppKit) or IOS UIKit.
- They provide reusability and extensibility.
  - **Whitebox frameworks** rely on inheritance and dynamic binding for extensibility. Developers subclass framework base classes and override predefined methods.
  - **Blackbox frameworks** support extensibility by defining interfaces for components that can be plugged into the framework. Developers create components that implement the interface.

Framework vs Class Libraries vs design patterns

- **Frameworks** focus on reuse of concrete designs, algorithms, and implementations
- **Design Patterns** focus on reuse of abstract designs and small collections of cooperating classes.
- **Class Libraries** are less domain specific than frameworks and provide a smaller scope of reuse
  - **examples:** classes for strings, complex numbers, collections, and maps. These can be used across many domains.
- **Components** are self contained instances of classes that are plugged together to form complete applications.