Object-Oriented Software Engineering
Using UML, Patterns, and Java

Object Design: Reusing Pattern Solutions

Object Design

- Purpose of object design:
  - Prepare for the implementation of the system model based on design decisions
  - Transform the system model (optimize it)
- Investigate alternative ways to implement the system model
  - Use design goals: minimize execution time, memory and other measures of cost.
- Object design serves as the basis of implementation.

Object Design Activities

- During object design, we close the gap between the application objects and the off-the-shelf components by identifying additional solution objects and refining existing objects. Object design include **reuse**, interface specification, restructuring, optimization.

  - Off-the-shelf components identified during system design are used to help in the realization of each subsystem. Class libraries and additional components are selected for basic data structures and services.

    Design patterns are selected for solving common problems and for protecting specific classes from future change.

    Often, components and design patterns need to be adapted before they can be used.

    This is done by wrapping custom objects around them or by refining them using inheritance. During all these activities, the developers are faced with the same buy-versus-build trade-offs they encountered during system design.

Object Design Activities

- During object design, we close the gap between the application objects and the off-the-shelf components by identifying additional solution objects and refining existing objects. Object design include reuse, **interface specification**, restructuring, optimization.

- During this activity, the subsystem services identified during system design are described in terms of class interfaces, including operations, arguments, type signatures, and exceptions. Additional operations and objects needed to transfer data among subsystems are also identified.

  The result of service specification is a complete interface specification for each subsystem. The subsystem service specification is often called subsystem **API** (Application Programmer Interface).

Object Design Activities

- During object design, we close the gap between the application objects and the off-the-shelf components by identifying additional solution objects and refining existing objects. Object design include reuse, interface specification, **restructuring**, optimization.

  - During restructuring, we address design goals such as maintainability, readability, and understandability of the system model.

    Each restructuring activity can be seen as a graph transformation on subsets of a particular model. Typical activities include transforming N-ary associations into binary associations, implementing binary associations as references, merging two similar classes from two different subsystems into a single class, collapsing classes with no significant behavior into attributes, splitting complex classes into simpler ones, and/or rearranging classes and operations to increase the inheritance and packaging.

Object Design Activities

- During object design, we close the gap between the application objects and the off-the-shelf components by identifying additional solution objects and refining existing objects. Object design include reuse, interface specification, restructuring, **optimization**.

  - Optimization activities address performance requirements of the system model. This includes changing algorithms to respond to speed or memory requirements, reducing multiplicities in associations to speed up queries, adding redundant associations for efficiency, rearranging execution orders, adding derived attributes to improve the access time to objects, and opening the architecture, that is, adding access to lower layers because of performance requirements.

An overview of the Object Design

- Conceptually, software system development fills the gap between a given problem and an existing machine.
  - Analysis reduces the gap between the problem and the machine by identifying objects representing problem-specific concepts.
  - System design reduces the gap between the problem and the machine in two ways:
    - First, system design results in a virtual machine that provides a higher level of abstraction than the machine. This is done by selecting off-the-shelf components for standard services such as middleware, user interface toolkits, application frameworks, and class libraries.
    - Second, system design identifies off-the-shelf components for application domain objects such as reusable class libraries of banking objects.

System Development

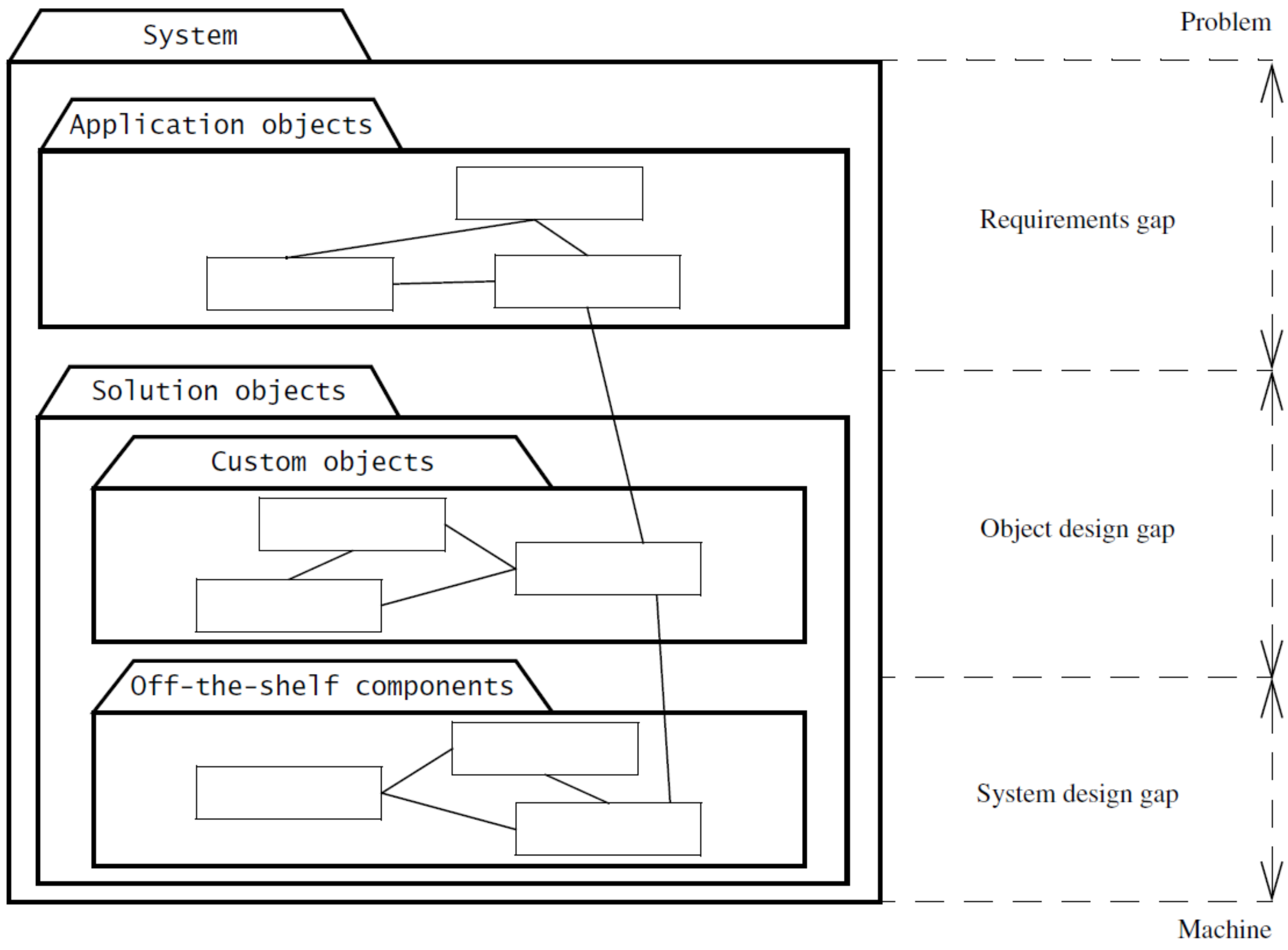


Figure 8-1 Object design closes the gap between application objects identified during requirements and off-the-shelf components selected during system design (stylized UML class diagram).

Customization: Build Custom Objects

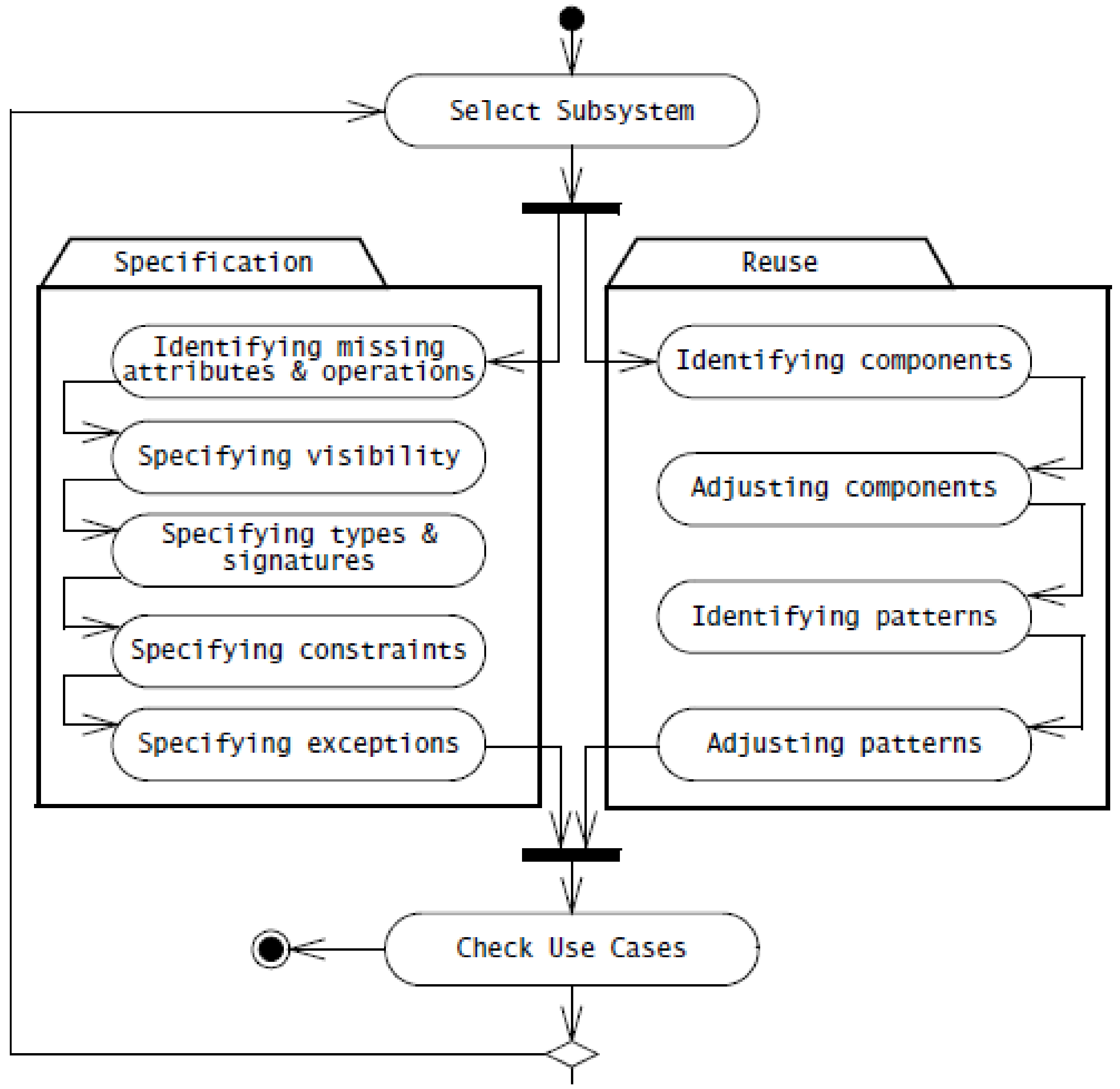
- Problem: Close the object design gap
  - Develop new functionality

- Main goal:
  - Reuse knowledge from previous experience
  - Reuse functionality already available

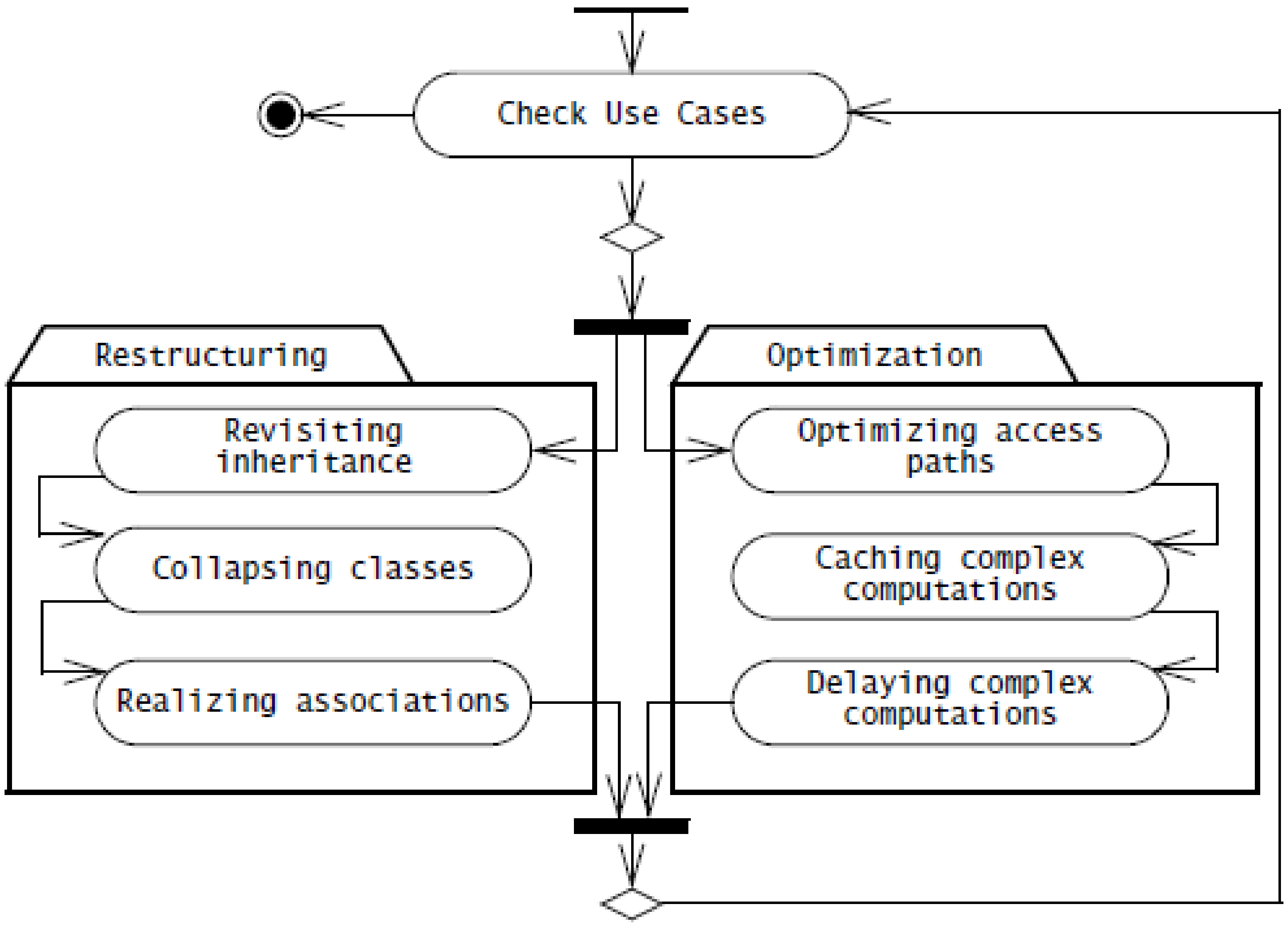
- Composition (also called Black Box Reuse)
  - New functionality is obtained by aggregation
  - The new object with more functionality is an aggregation of existing objects

- Inheritance (also called White-box Reuse)
  - New functionality is obtained by inheritance

Activities of Object Design



Activities of Object Design_2

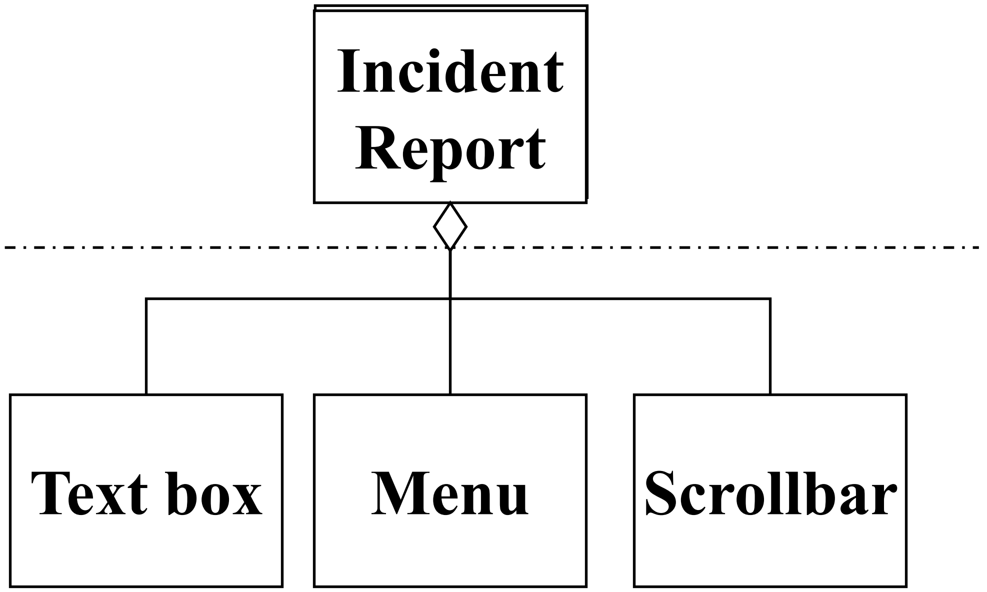


Application Objects and Design Objects

- **Application objects**, also called “domain objects,” represent concepts of the domain that are relevant to the 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.
- **During system design**, we **identify solution objects in terms of software and hardware platforms**.
- **During object design**, we **refine and detail both application and solution objects and identify additional solution objects needed to bridge the object design gap**.

Identification of new Objects during Object Design

Requirements Analysis
(Language of Application Domain)



Object Design
(Language of Solution Domain)

Specification Inheritance and Implementation Inheritance

- **During analysis**, we use **inheritance to classify objects into taxonomies**. This allows us to **differentiate the common behavior** of the general **case, that is, the parent (also called the "base class"), from the behavior that is specific to specialized objects**, that are, the **children classes** (also called the "**derived classes**").

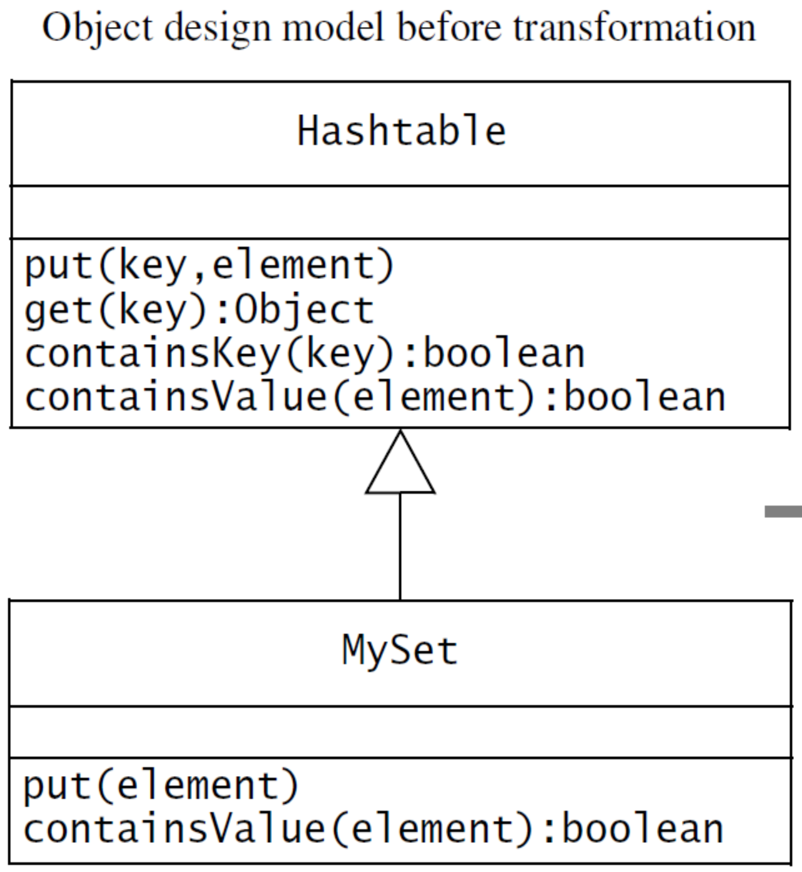
- **During object design, the focus of inheritance is to reduce redundancy and enhance extensibility.** By factoring all redundant behavior into a single parent class, we reduce the risk of introducing inconsistencies during changes (e.g., when repairing a defect) since we must make changes only once for all children classes. By providing abstract classes and interfaces that are used by the application, we can write new specialized behavior.

Specification Inheritance and Implementation Inheritance_2

- Although inheritance can make an analysis model more understandable and an object design model more modifiable or extensible, these benefits do not occur automatically.

- On the contrary, **inheritance is such a powerful mechanism that novice developers often produce code that is more obfuscated and more brittle than if they had not used inheritance in the first place**.

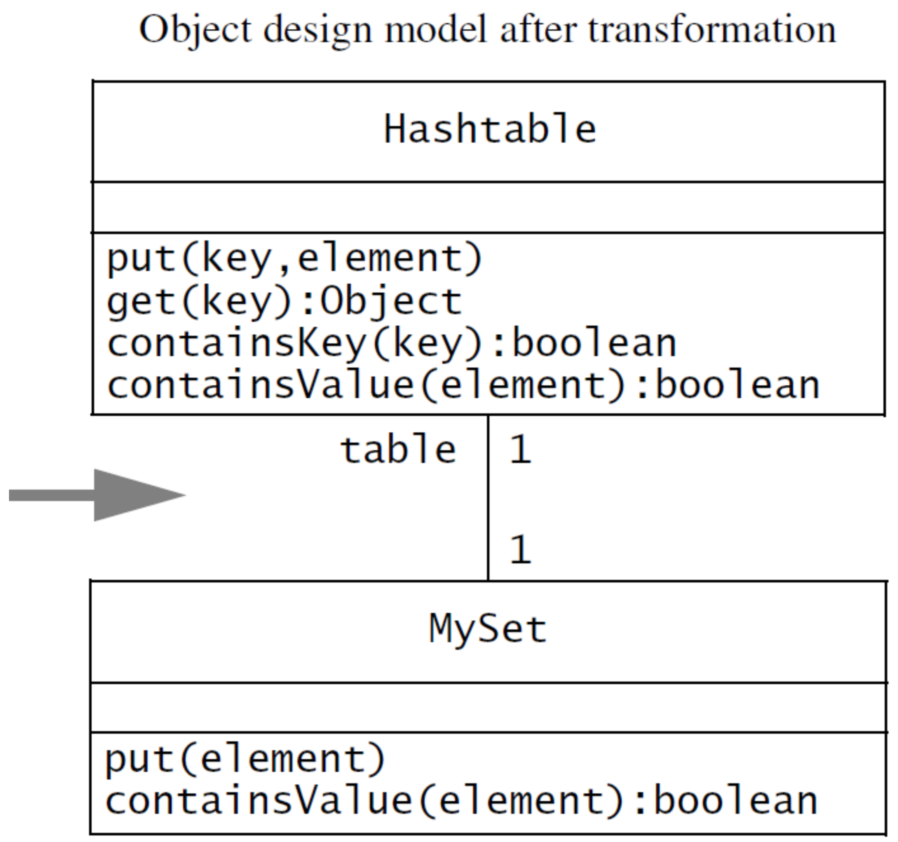
Specification Inheritance and Implementation Inheritance_3


Object design model before transformation
Hashtable
put(key,element)
get(key):Object
containsKey(key):boolean
containsValue(element):boolean
MySet
put(element)
containsValue(element):boolean

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

Figure 8-3 An example of implementation inheritance. A questionable implementation of MySet using implementation inheritance.

Specification Inheritance and Implementation Inheritance_4


Object design model after transformation
Hashtable
put(key,element)
get(key):Object
containsKey(key):boolean
containsValue(element):boolean
table
1
1
MySet
put(element)
containsValue(element):boolean

```
/* Implementation of MySet 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));
}
/* Other methods omitted */
}
```

Figure 8-3 An example of implementation inheritance. An improved implementation using delegation.

Inheritance metamodel

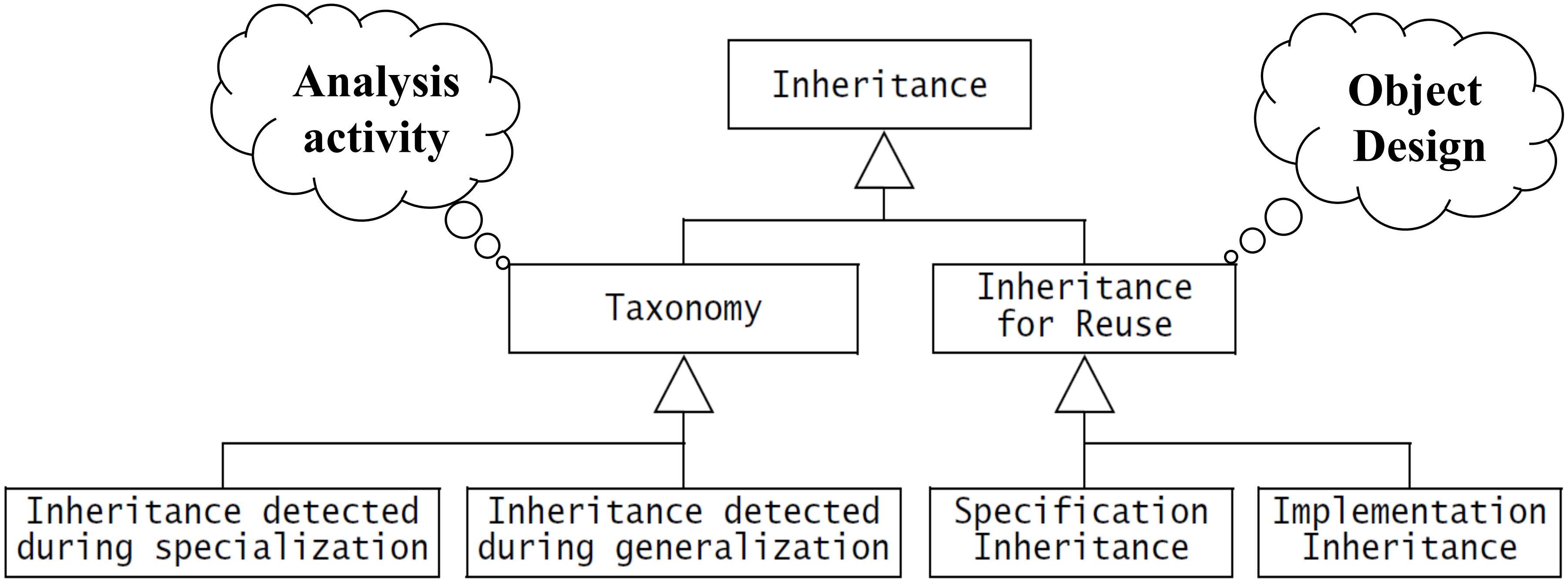

Analysis activity
Inheritance
Object Design
Taxonomy
Inheritance for Reuse
Inheritance detected during specialization
Inheritance detected during generalization
Specification Inheritance
Implementation Inheritance

Figure 8-4 Inheritance meta-model (UML class diagram).

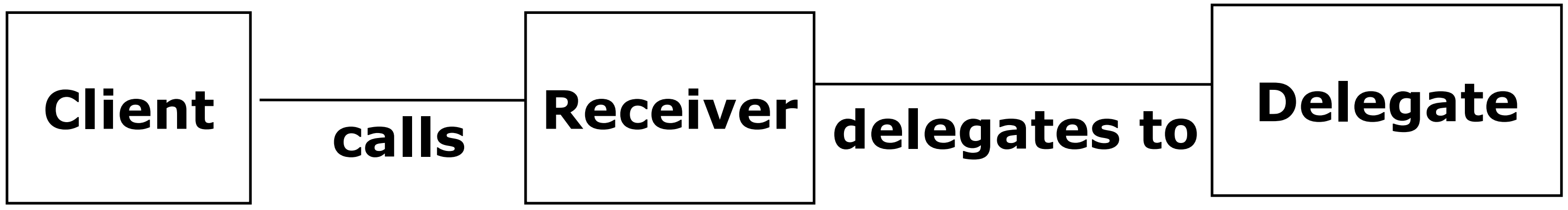
Delegation

- **Delegation** is **the alternative to implementation inheritance** that should be used when reuse is desired.
- A **class is said to delegate** to another class if it implements an operation by referring another operation of the other class.
- **Delegation** makes explicit the dependencies between the reused class and the new class. The right column of Figure 8-3 shows an implementation of MySet using delegation instead of implementation inheritance. The only significant change is the private field table and its initialization in the MySet() constructor.

Delegation

- Delegation is a way of making composition as powerful for reuse as inheritance

- In delegation two objects are involved in handling a request from a Client
    - The Receiver object delegates operations to the Delegate object
    - The Receiver object makes sure, that the Client does not misuse the Delegate object.



Comparison: Delegation v. Inheritance

- Code-Reuse can be done by delegation as well as inheritance
- Delegation
  - Flexibility: Any object can be replaced at run time by another one
  - Inefficiency: Objects are encapsulated
- Inheritance
  - Straightforward to use
  - Supported by many programming languages
  - Easy to implement new functionality
  - Exposes a subclass to details of its super class
  - Change in the parent class requires recompilation of the subclass.

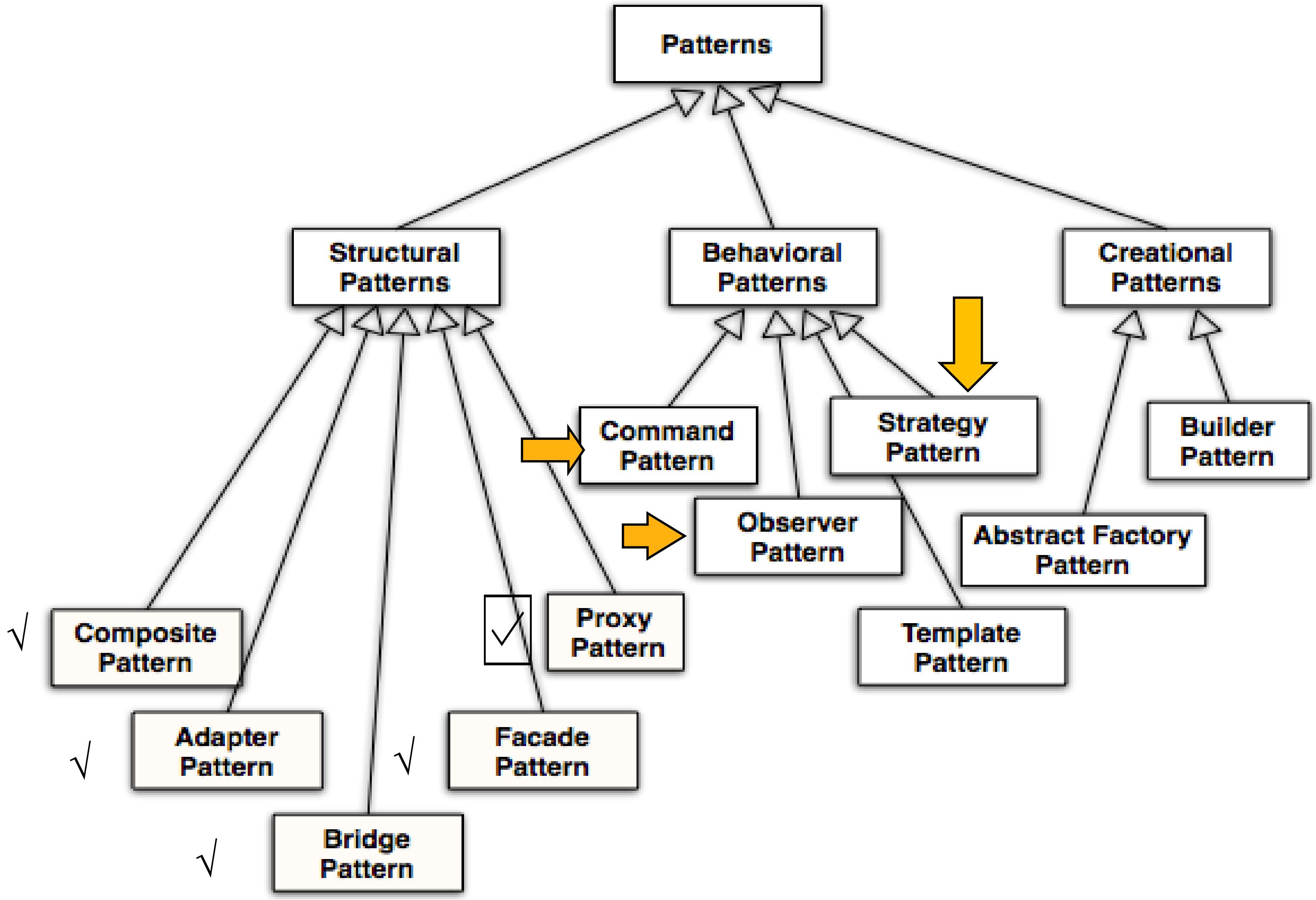
The Liskov Substitution Principle

- If **an object of type S can be substituted in all the places** where an object of type T is expected, then S is a subtype of T.
- An inheritance relationship that complies with the Liskov Substitution Principle is called **strict inheritance**.

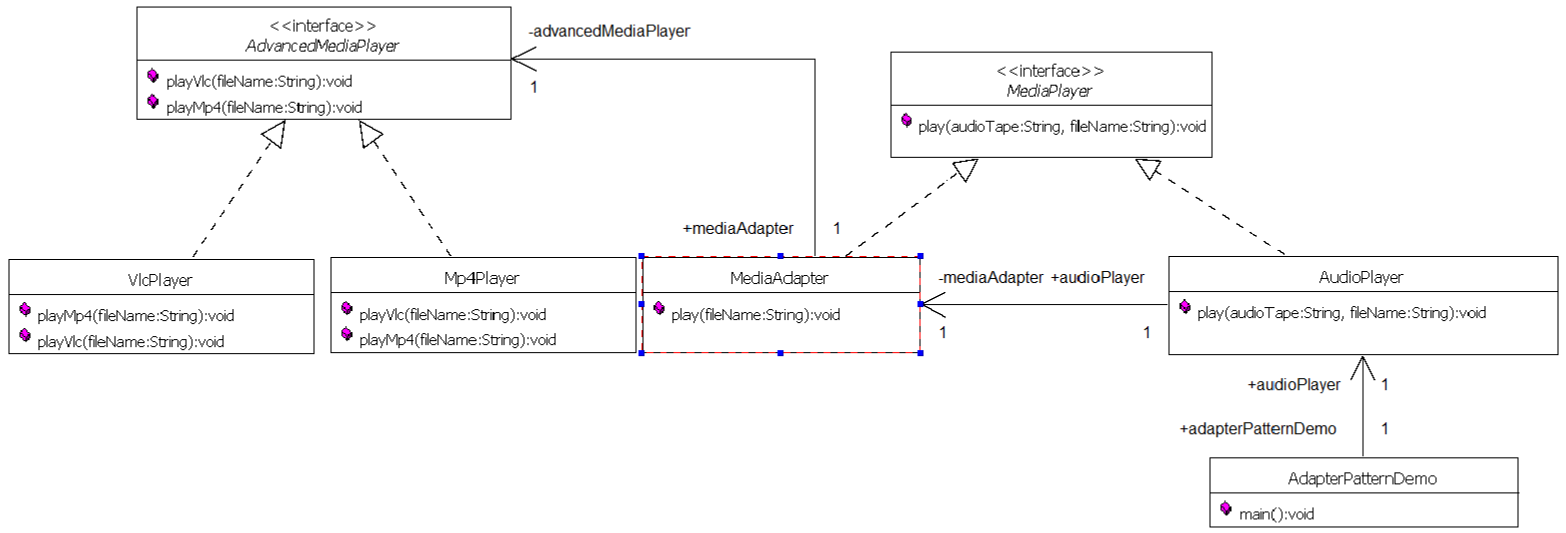
Delegation and Inheritance in Design Patterns

- In object-oriented development, **design patterns** are template solutions that developers have refined over time to solve a range of recurring problems [Gamma et al., 1994]. A design pattern has four elements:
    1. A ***name*** that uniquely identifies the pattern from other patterns.
    2. A ***problem description*** that describes the situations in which the pattern can be used. Problems addressed by design patterns are usually the realization of modifiability and extensibility design goals and nonfunctional requirements.
    3. A ***solution*** stated as a set of collaborating classes and interfaces.
    4. A set of ***consequences*** that describes the trade-offs and alternatives to be considered with respect to the design goals being addressed.

A Taxonomy of Design Patterns



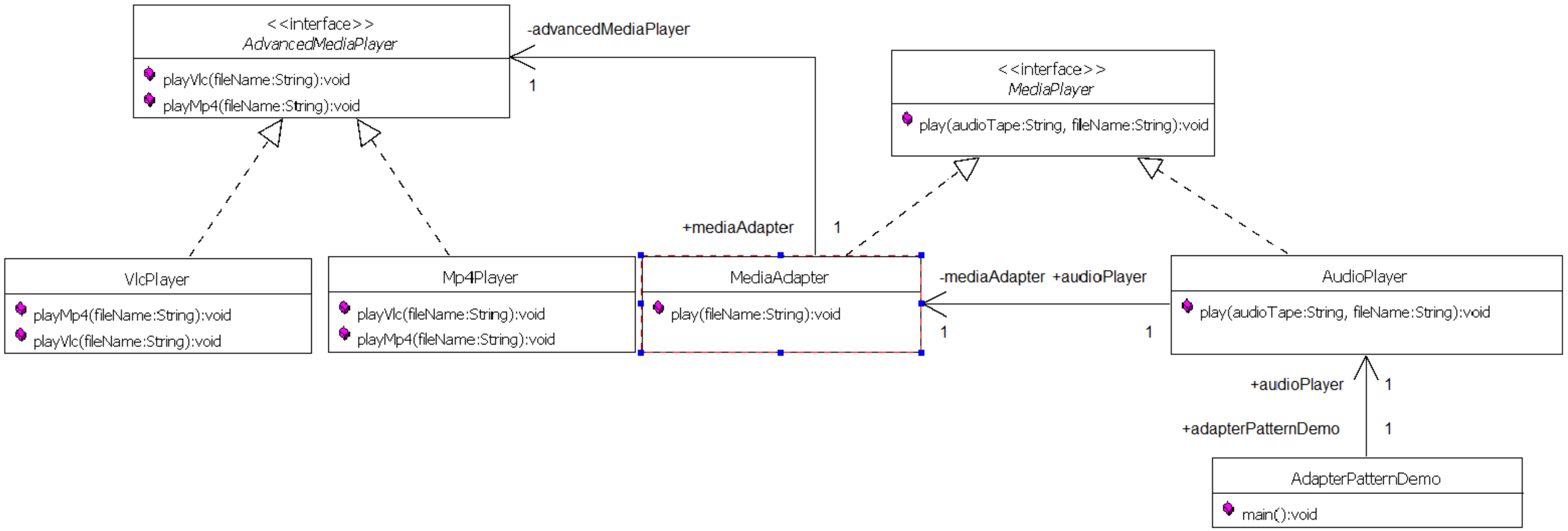
The Adapter Pattern



<<interface>>
AdvancedMediaPlayer
playVlc(fileName:String):void
playMp4(fileName:String):void
-advancedMediaPlayer
1
<<interface>>
MediaPlayer
play(audioTape:String, fileName:String):void
+mediaAdapter
1
VlcPlayer
playMp4(fileName:String):void
playVlc(fileName:String):void
Mp4Player
playVlc(fileName:String):void
playMp4(fileName:String):void
MediaAdapter
play(fileName:String):void
-mediaAdapter
+audioPlayer
1
1
AudioPlayer
play(audioTape:String, fileName:String):void
+audioPlayer
1
+adapterPatternDemo
1
AdapterPatternDemo
main():void

- **Adapter pattern** works as a **bridge between two incompatible** interfaces. This design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.
  - This pattern involves a single class **MediaAdapter** which is responsible to join functionalities of independent or incompatible interfaces: **MediaPlayer** and **AdvancedMediaPlayer**.
  - If AudioPlayer is unable to deliver client's request, then **MediaAdapter** delegates to **AdvancedMediaPlayer** the request.
  - **MediaAdapter** and **AdvancedMediaPlayer** work together without modification.
  - If new AudioTape are needed, the **MediaAdapter** will be updated.

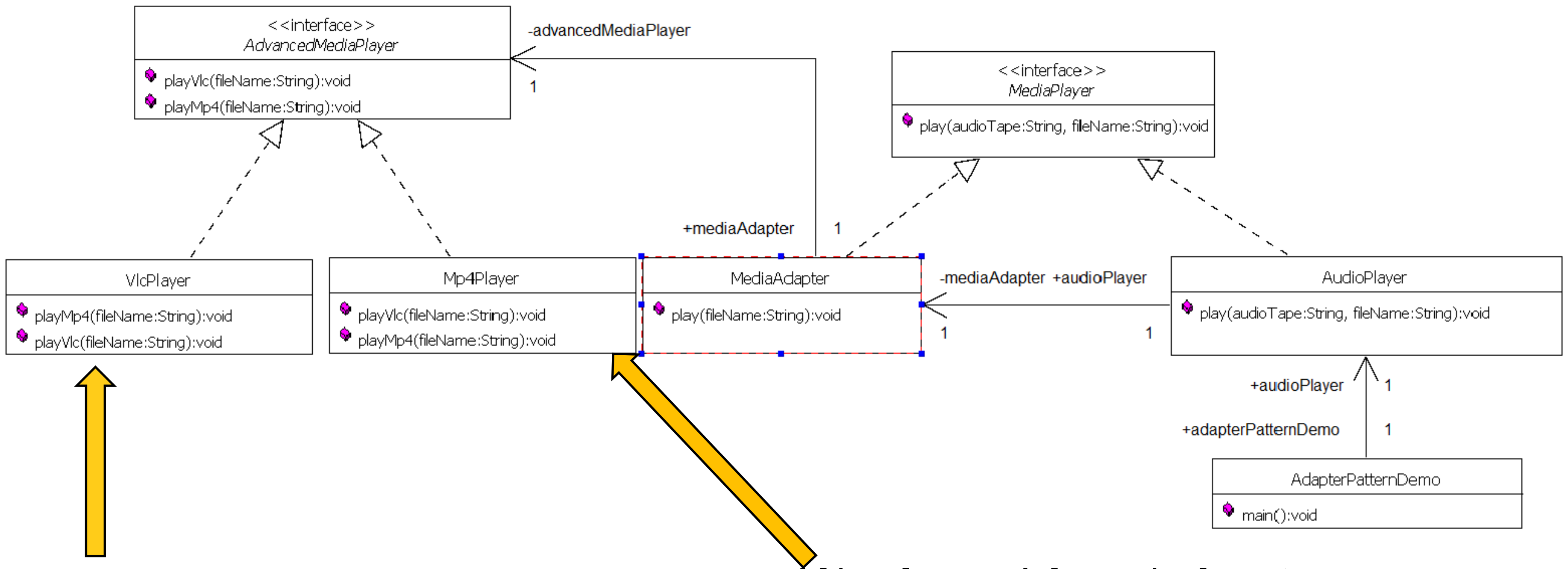
The Adapter Pattern_2



<<interface>>
AdvancedMediaPlayer
playVlc(fileName:String):void
playMp4(fileName:String):void
-advancedMediaPlayer
1
<<interface>>
MediaPlayer
play(audioTape:String, fileName:String):void
+mediaAdapter
1
VlcPlayer
playMp4(fileName:String):void
playVlc(fileName:String):void
Mp4Player
playVlc(fileName:String):void
playMp4(fileName:String):void
MediaAdapter
play(fileName:String):void
-mediaAdapter
+audioPlayer
1
1
AudioPlayer
play(audioTape:String, fileName:String):void
+audioPlayer
1
+adapterPatternDemo
1
AdapterPatternDemo
main():void

```
public interface MediaPlayer {
   public void play(String audioType, String fileName);
}

public interface AdvancedMediaPlayer {
   public void playVlc(String fileName);
   public void playMp4(String fileName);
}
```

The Adapter Pattern_3

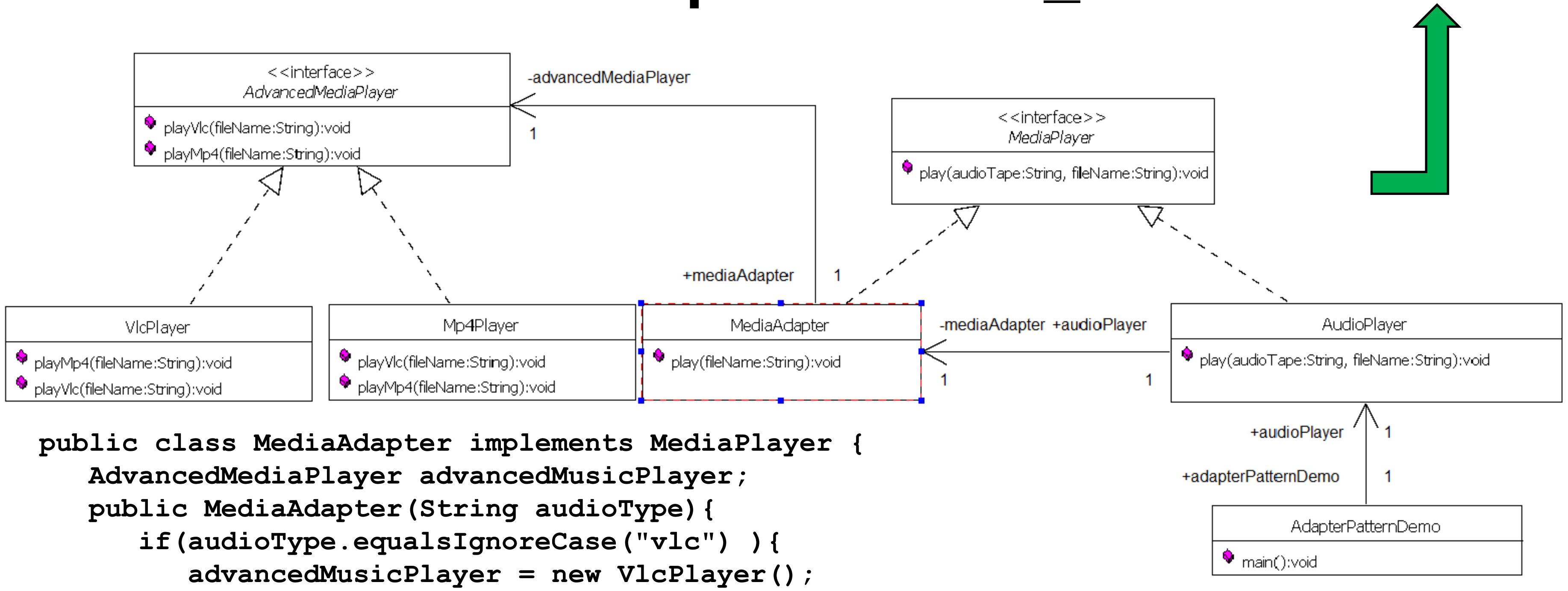


<<interface>>
AdvancedMediaPlayer
playVlc(fileName:String):void
playMp4(fileName:String):void
-advancedMediaPlayer
1
<<interface>>
MediaPlayer
play(audioTape:String, fileName:String):void
+mediaAdapter
1
VlcPlayer
playMp4(fileName:String):void
playVlc(fileName:String):void
Mp4Player
playVlc(fileName:String):void
playMp4(fileName:String):void
MediaAdapter
play(fileName:String):void
-mediaAdapter
+audioPlayer
1
1
AudioPlayer
play(audioTape:String, fileName:String):void
+audioPlayer
1
+adapterPatternDemo
1
AdapterPatternDemo
main():void

```
public class VlcPlayer implements
AdvancedMediaPlayer{
@Override
   public void playMp4(String fileName) {
   }
   @Override
   public void playVlc(String fileName) {
      System.out.println("Playing Vlc file.
Name: "+ fileName);
   }
}
```

```
public class Mp4Player implements
AdvancedMediaPlayer{
@Override
   public void playVlc(String fileName) {
   }
   @Override
   public void playMp4(String fileName) {
      System.out.println("Playing mp4 file.
Name: "+ fileName);
   }
}
```

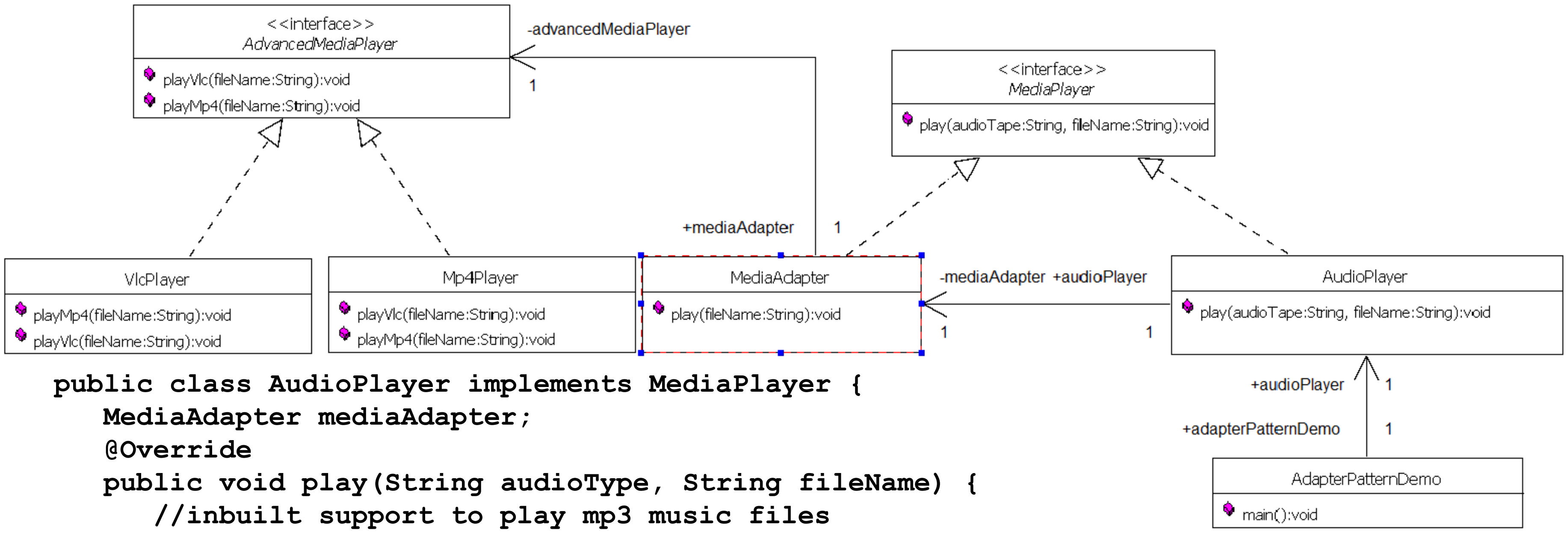
The Adapter Pattern_4



<<interface>>
AdvancedMediaPlayer
playVlc(fileName:String):void
playMp4(fileName:String):void
-advancedMediaPlayer
1
<<interface>>
MediaPlayer
play(audioTape:String, fileName:String):void
+mediaAdapter
1
VlcPlayer
playMp4(fileName:String):void
playVlc(fileName:String):void
Mp4Player
playVlc(fileName:String):void
playMp4(fileName:String):void
MediaAdapter
play(fileName:String):void
-mediaAdapter
+audioPlayer
1
1
AudioPlayer
play(audioTape:String, fileName:String):void
+audioPlayer
1
+adapterPatternDemo
1
AdapterPatternDemo
main():void

```
public class MediaAdapter implements MediaPlayer {
   AdvancedMediaPlayer advancedMusicPlayer;
   public MediaAdapter(String audioType){
      if(audioType.equalsIgnoreCase("vlc") ){
         advancedMusicPlayer = new VlcPlayer();
      }else if (audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer = new Mp4Player();
      }
   }
   @Override
   public void play(String audioType, String fileName) {
      if(audioType.equalsIgnoreCase("vlc")){
         advancedMusicPlayer.playVlc(fileName);
      }
      else if(audioType.equalsIgnoreCase("mp4")){
         advancedMusicPlayer.playMp4(fileName);
      }
   }
}
```

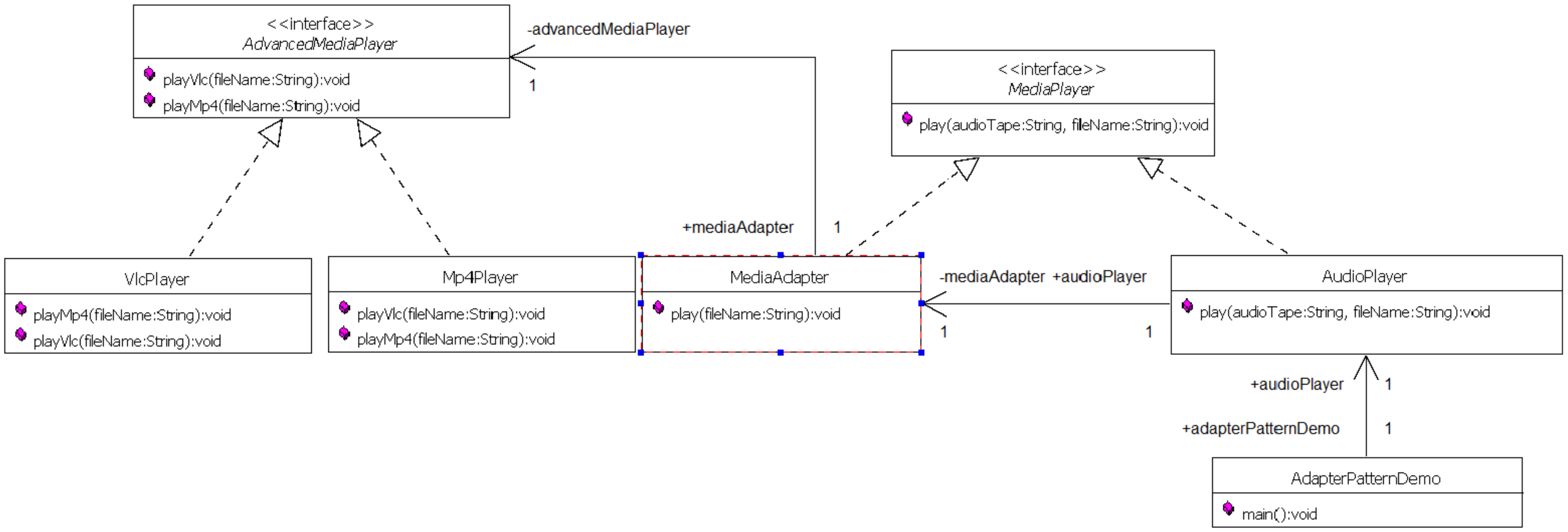
The Adapter Pattern_5



<<interface>>
AdvancedMediaPlayer
playVlc(fileName:String):void
playMp4(fileName:String):void
-advancedMediaPlayer
1
<<interface>>
MediaPlayer
play(audioTape:String, fileName:String):void
+mediaAdapter
1
VlcPlayer
playMp4(fileName:String):void
playVlc(fileName:String):void
Mp4Player
playVlc(fileName:String):void
playMp4(fileName:String):void
MediaAdapter
play(fileName:String):void
-mediaAdapter
+audioPlayer
1
1
AudioPlayer
play(audioTape:String, fileName:String):void
+audioPlayer
1
+adapterPatternDemo
1
AdapterPatternDemo
main():void

```
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;
   @Override
   public void play(String audioType, String fileName) {
      //inbuilt support to play mp3 music files
      if(audioType.equalsIgnoreCase("mp3")){
         System.out.println("Playing mp3 file. Name: " + fileName);

      }
      //mediaAdapter is providing support to play other file formats
      else if(audioType.equalsIgnoreCase("vlc") ||
audioType.equalsIgnoreCase("mp4")){
         mediaAdapter = new MediaAdapter(audioType);
         mediaAdapter.play(audioType, fileName);
      }
      else{
         System.out.println("Invalid media. " + audioType + " format
not supported");
      }
   }
}
```

The Adapter Pattern_6


<<interface>>
AdvancedMediaPlayer
playVlc(fileName:String):void
playMp4(fileName:String):void
-advancedMediaPlayer
1
<<interface>>
MediaPlayer
play(audioTape:String, fileName:String):void
+mediaAdapter
1
VlcPlayer
playMp4(fileName:String):void
playVlc(fileName:String):void
Mp4Player
playVlc(fileName:String):void
playMp4(fileName:String):void
MediaAdapter
play(fileName:String):void
-mediaAdapter
+audioPlayer
1
1
AudioPlayer
play(audioTape:String, fileName:String):void
+audioPlayer
1
+adapterPatternDemo
1
AdapterPatternDemo
main():void

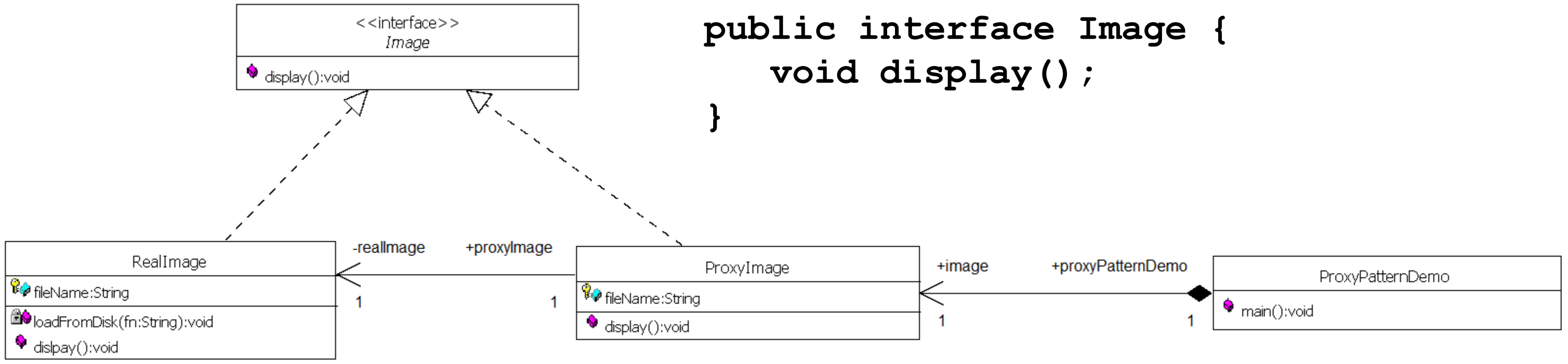
The results obtained after running the AdapterPatternDemo

```
Playing mp3 file. Name: beyond the horizon.mp3
Playing mp4 file. Name: alone.mp4
Playing vlc file. Name: far far away.vlc
Invalid media. avi format not supported
```

The Proxy Pattern

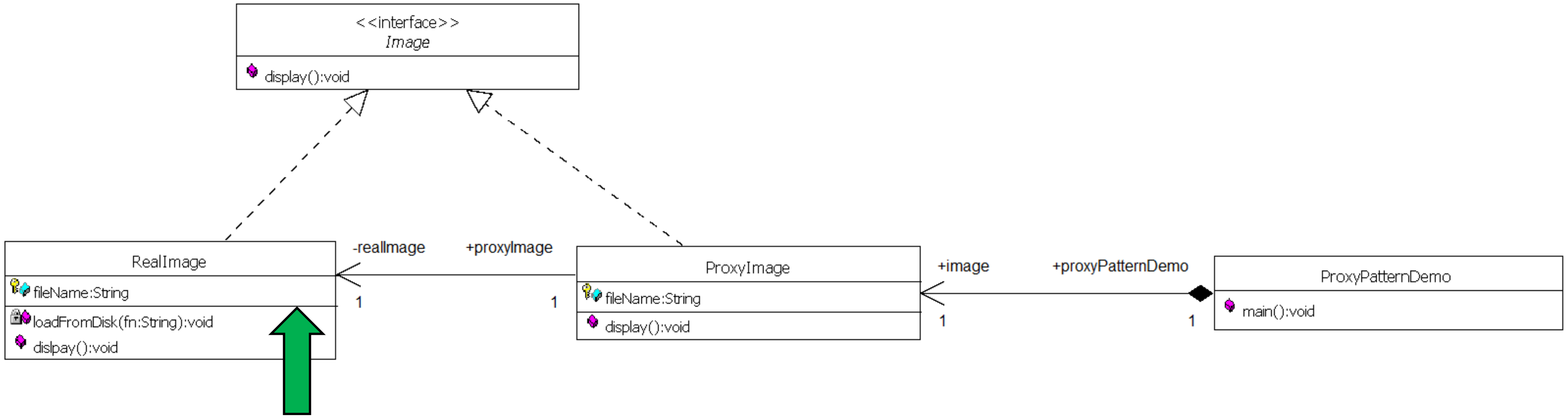
The ProxyObject class acts on behalf of a RealObject class. Both classes implement the same interface. The ProxyObject stores a subset of the attributes of the RealObject.

The ProxyObject handles certain requests completely (e.g., determining the size of an image), whereas others are delegated to the RealObject. After delegation, the RealObject is created and loaded in memory.



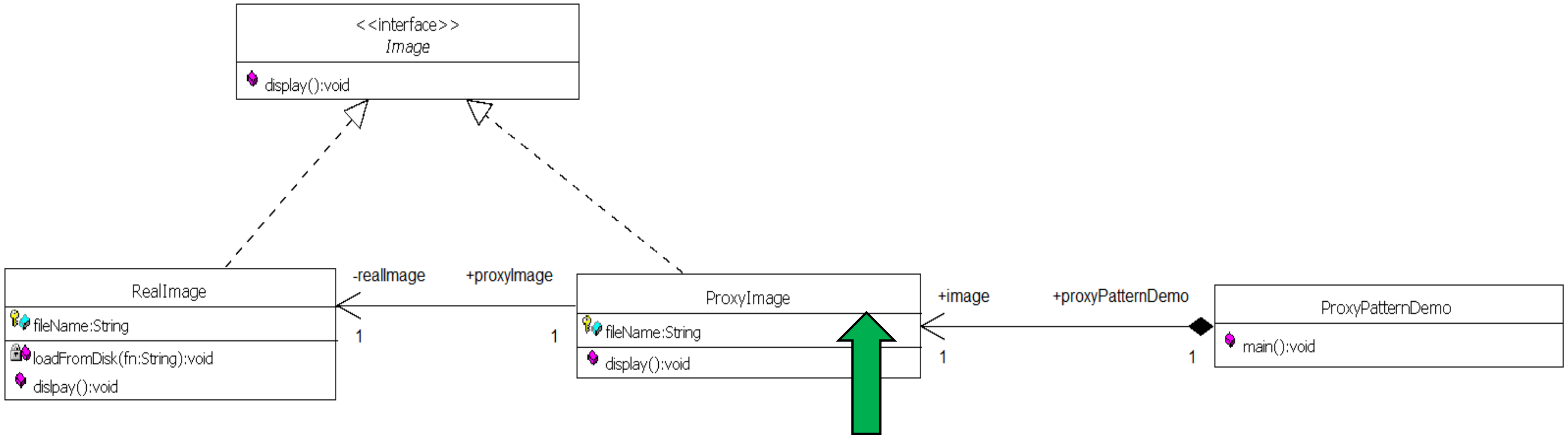
```
public interface Image {
   void display();
}
```

The Proxy Pattern_2



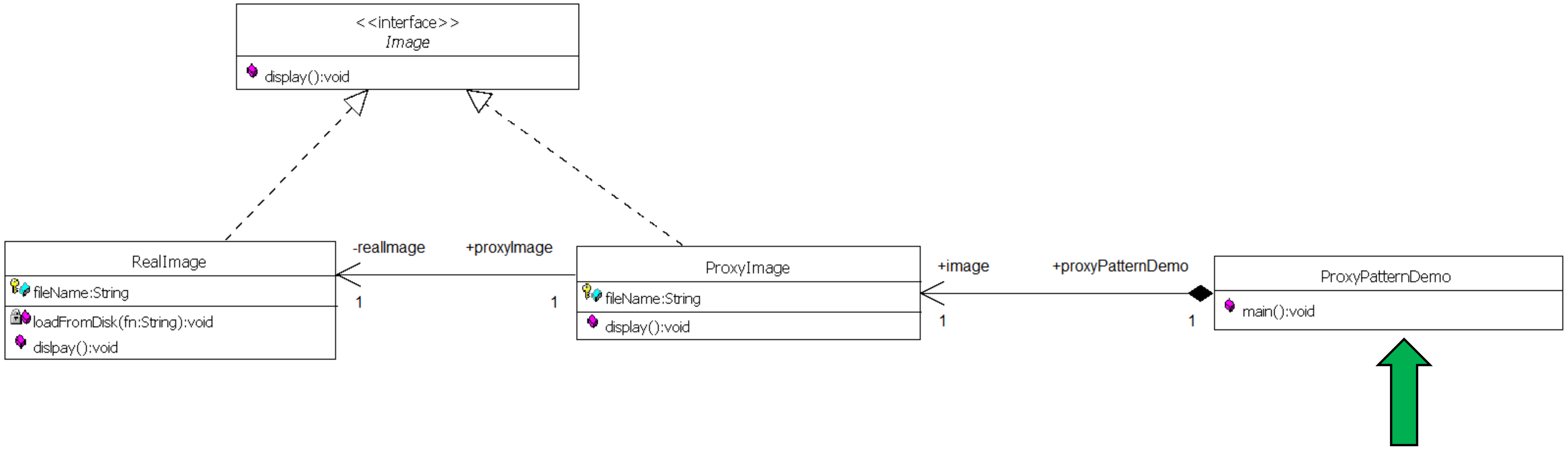
```
public class RealImage implements Image {

   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }

   @Override
   public void display() {
      System.out.println("Displaying " +
fileName);
   }
```

The Proxy Pattern_3



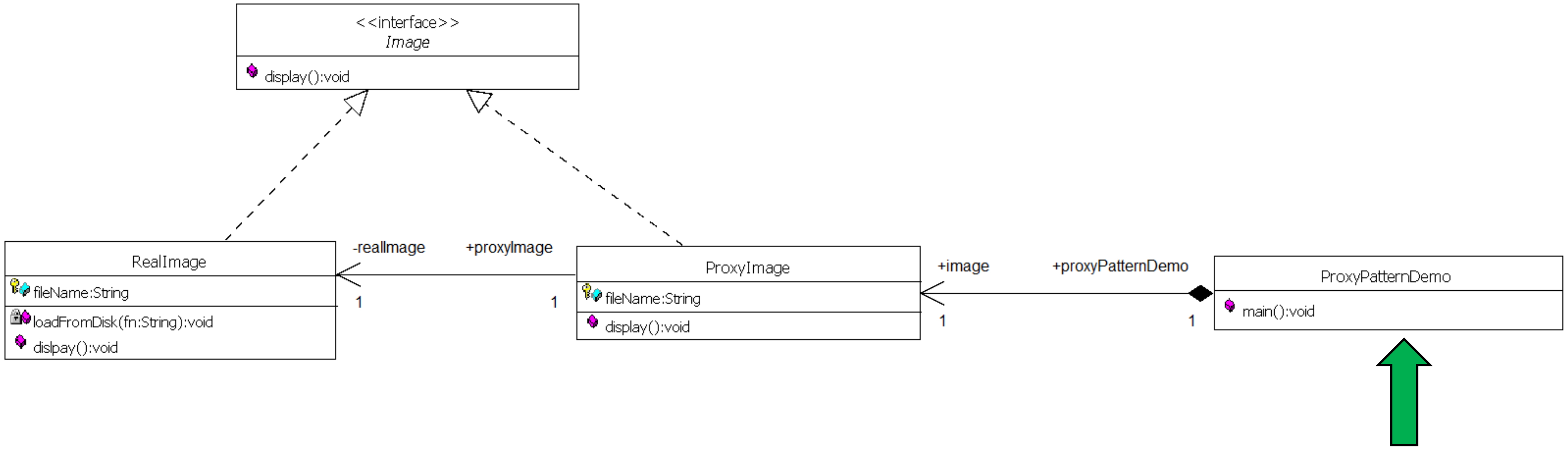
```
public class ProxyImage implements Image{

   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}
```

The Proxy Pattern_4



```
public class ProxyPatternDemo {

   public static void main(String[] args) {
      Image image = new ProxyImage("test_10mb.jpg");

      //image will be loaded from disk
      image.display();
      System.out.println("");

      //image will not be loaded from disk
      image.display();
   }
}
```

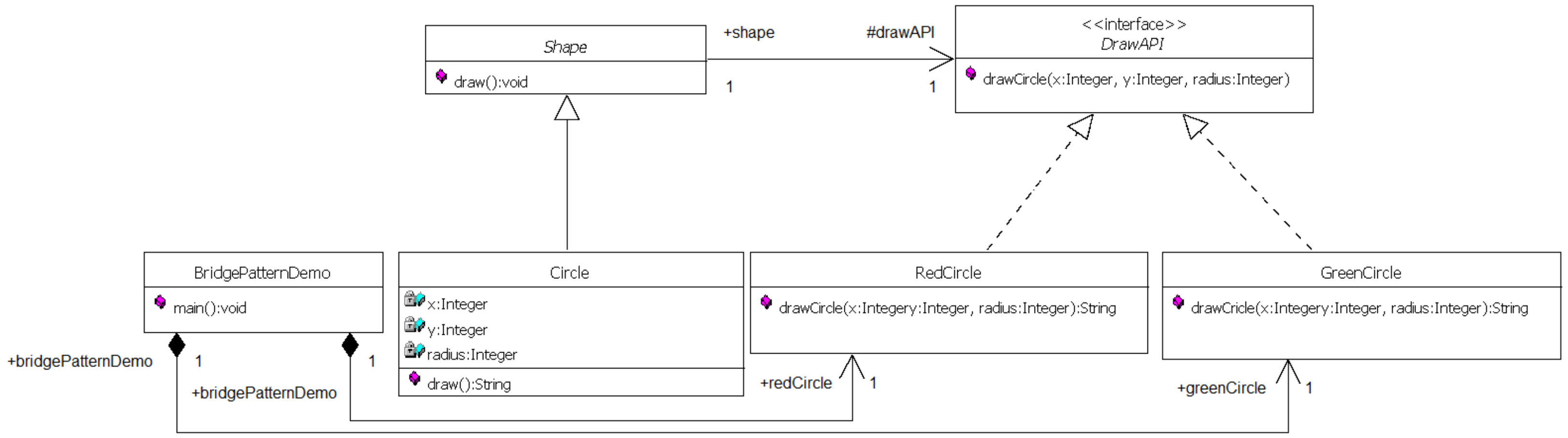
The Proxy Pattern_5



The results obtained after running ProxyPatternDemo

```
Loading test_10mb.jpg
Displaying test_10mb.jpg

Displaying test_10mb.jpg
```

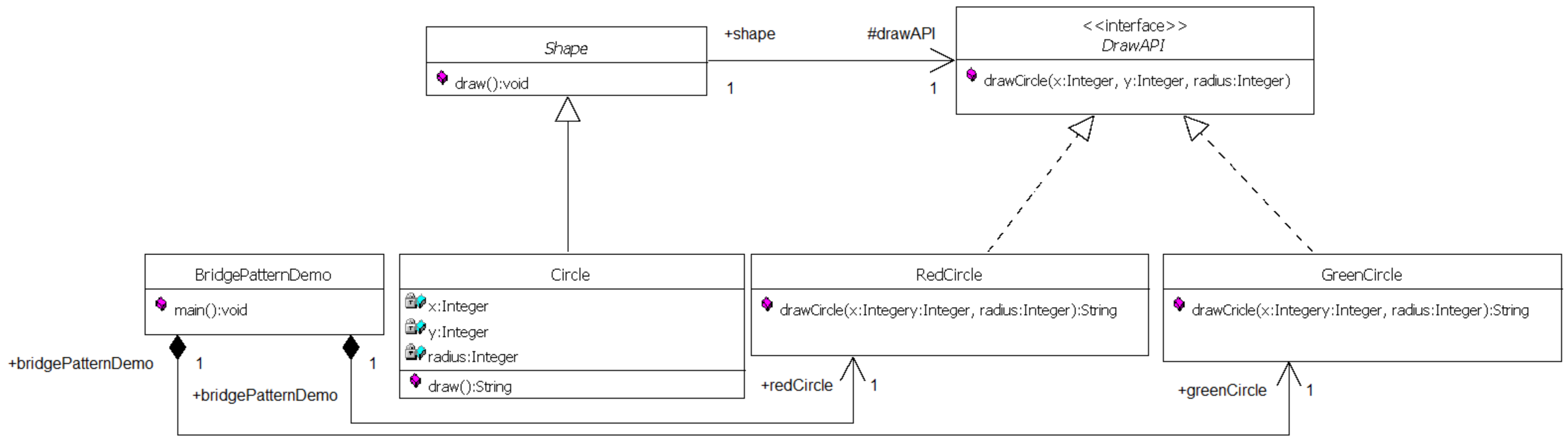
The Bridge Pattern



- Bridge is used when we need to decouple an abstraction from its implementation so that the two can vary independently. The *DrawAPI* interface acta as a bridge implementer.

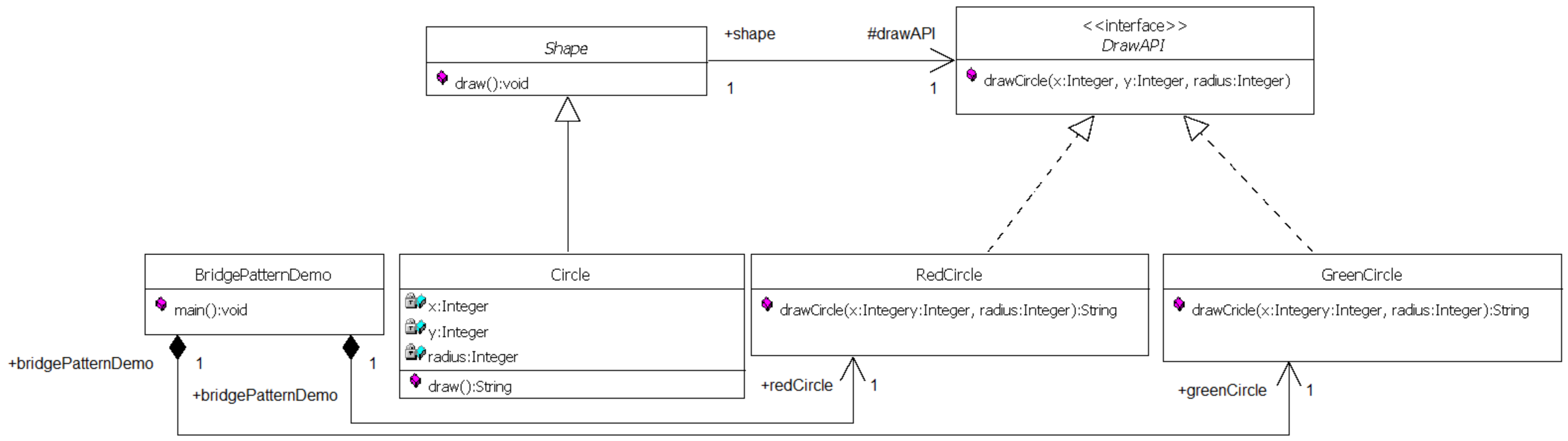
- This pattern involves an interface which acts as a bridge which makes the functionality of concrete classes independent from interface implementer classes. Both types of classes can be altered structurally without affecting each other concrete classes.

The Bridge Pattern_2



```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}

public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: " + radius + ", x: " + x + ", " + y
+ "]");
   }
}

public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: " + radius + ", x: " + x + ", " +
y + "]");
   }
}
```

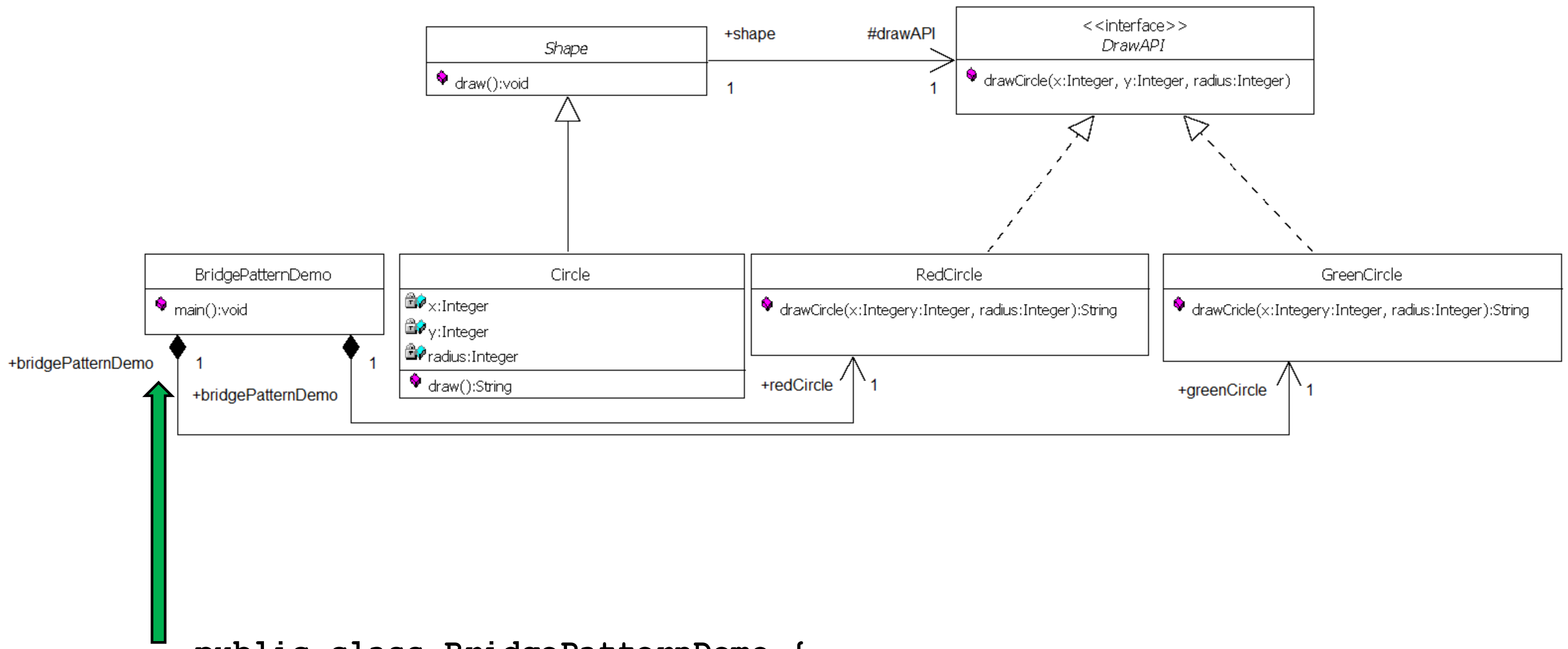
The Bridge Pattern_3



```
public abstract class Shape {
   protected DrawAPI drawAPI;

   protected Shape(DrawAPI drawAPI)
   {
      this.drawAPI = drawAPI;
   }
   public abstract void draw();

}
```

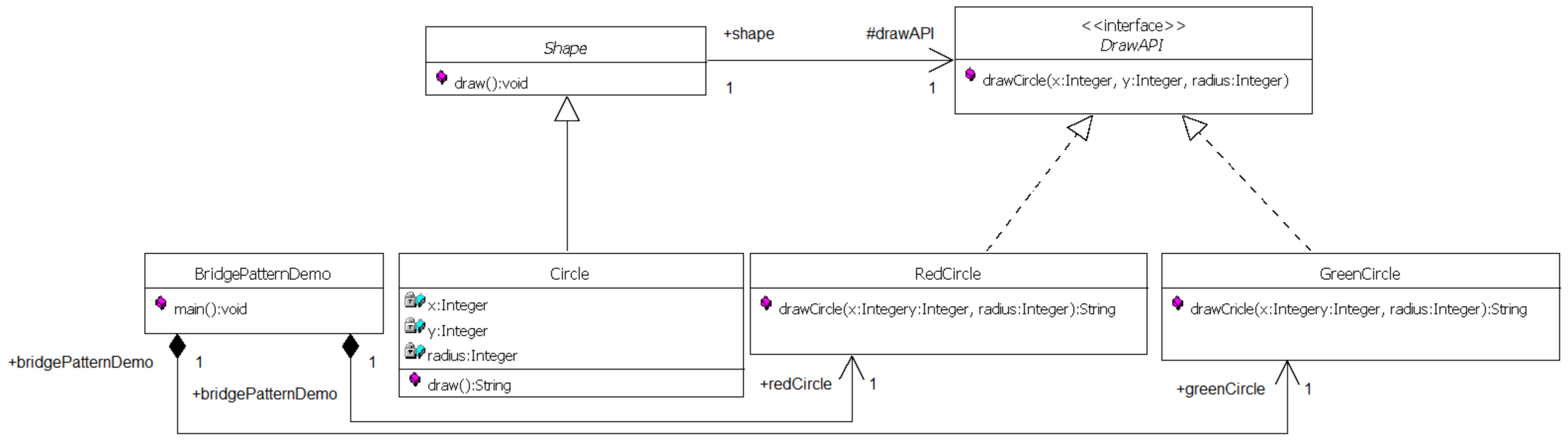
```
public class Circle extends Shape {
   private int x, y, radius;

public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }
   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

The Bridge Pattern_4



```
public class BridgePatternDemo {
   public static void main(String[] args) {
      Shape redCircle = new Circle(100,100, 10, new RedCircle());
      Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

      redCircle.draw();
      greenCircle.draw();
   }
}
```

The Bridge Pattern_5



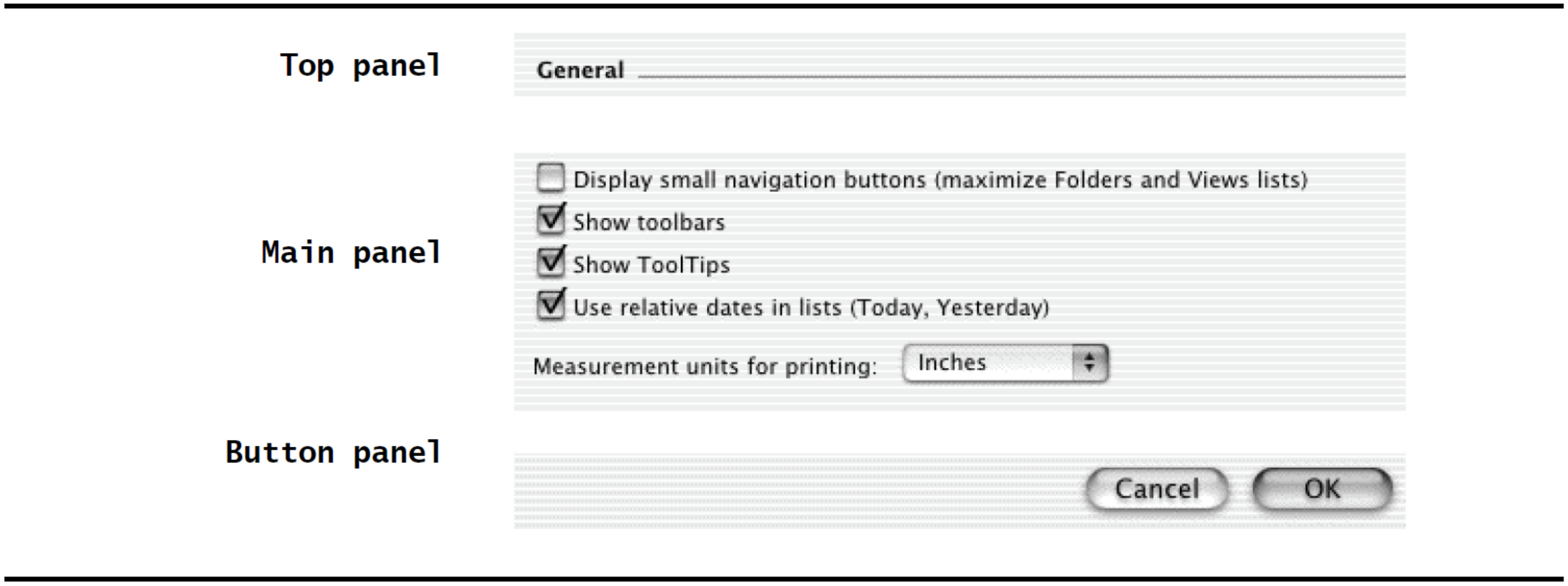
The result of running BridgePatternDemo:

```
Drawing Circle[ color: red, radius: 10, x: 100, 100]
Drawing Circle[  color: green, radius: 10, x: 100, 100]
```

The Composite Pattern

- Modern toolkits enable developers to organize the user interface objects into hierarchies of aggregate nodes, called “panels,” that can be manipulated the same way as the concrete user interface objects.
- For example, our preferences dialog can include a top panel for the title of the dialog and instructions for the user, a center panel containing the checkboxes and their labels, and a bottom panel for the ‘ok’ and ‘cancel’ button.
- Each panel is responsible for the layout of its subpanels, called “children,” and the overall dialog only must deal with the three panels (Figures 8-14 and 8-15).

The Composite Pattern_2



- [ ] Display small navigation buttons (maximize Folders and Views lists)
- [x] Show toolbars
- [x] Show ToolTips
- [x] Use relative dates in lists (Today, Yesterday)

Measurement units for printing: Inches

Figure 8-14 Anatomy of a preference dialog. Aggregates, called “panels,” are used for grouping user interface objects that need to be resized and moved together.

The Composite Pattern_3

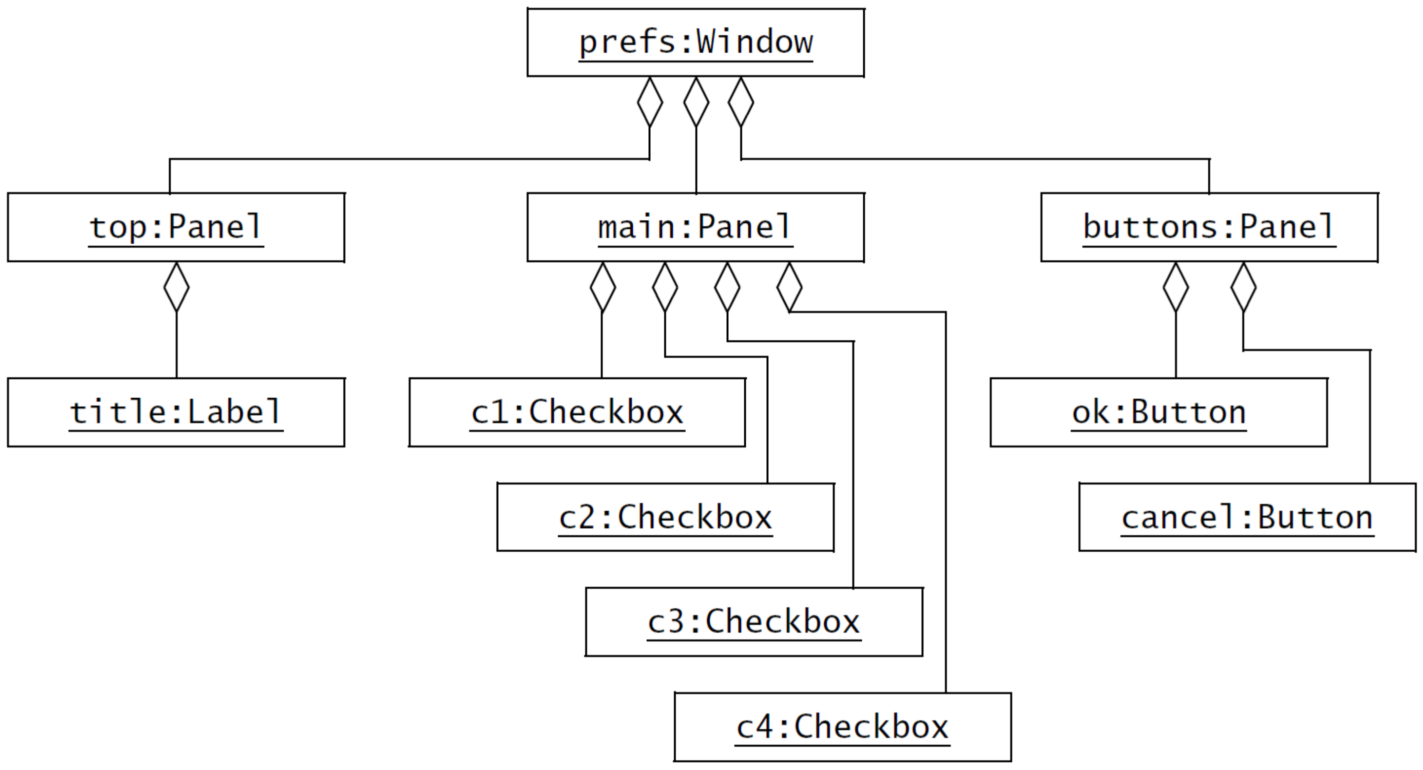


Figure 8-15 UML object diagram for the user interface objects of Figure 8-14.

The Composite Pattern_4

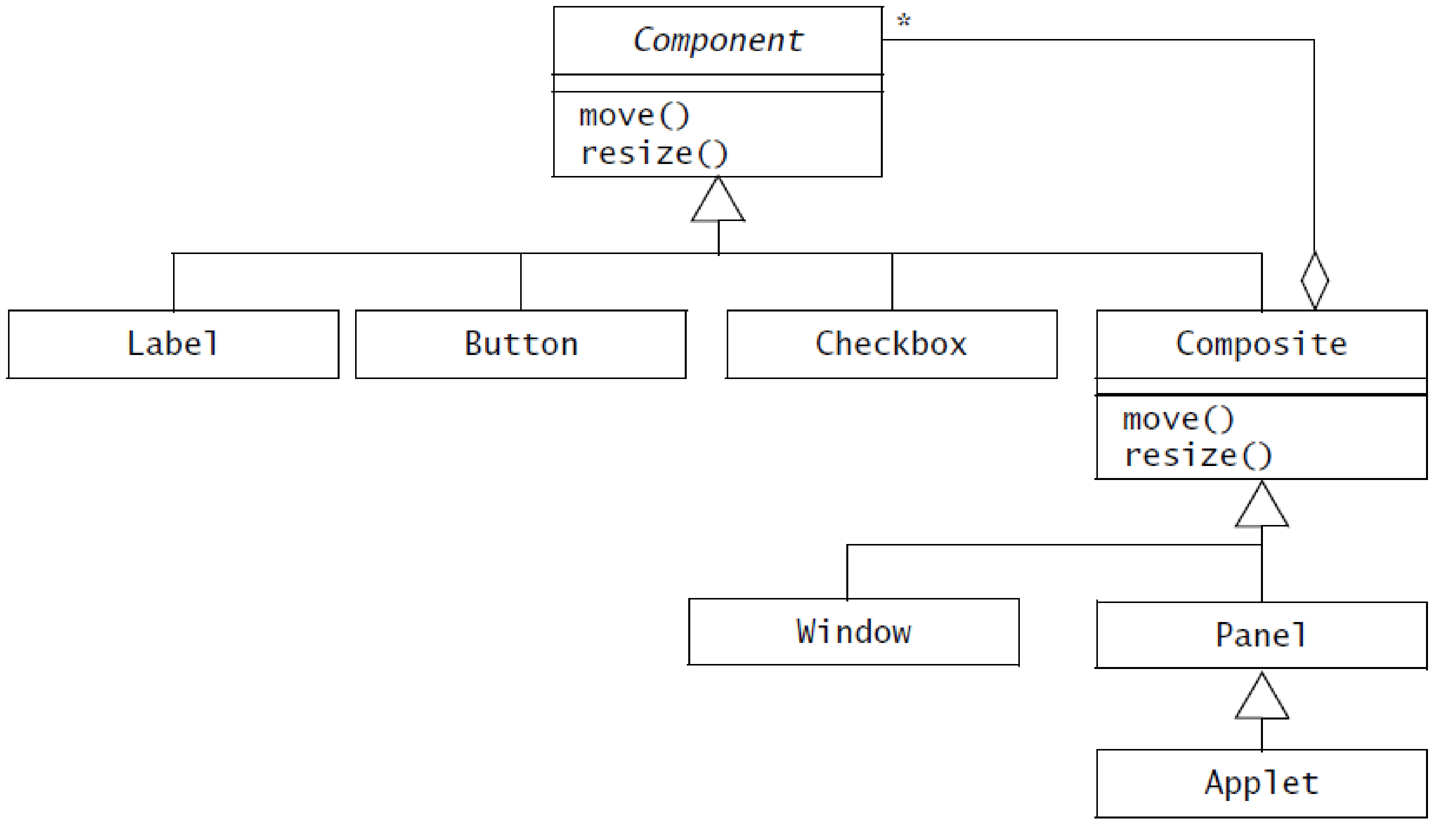
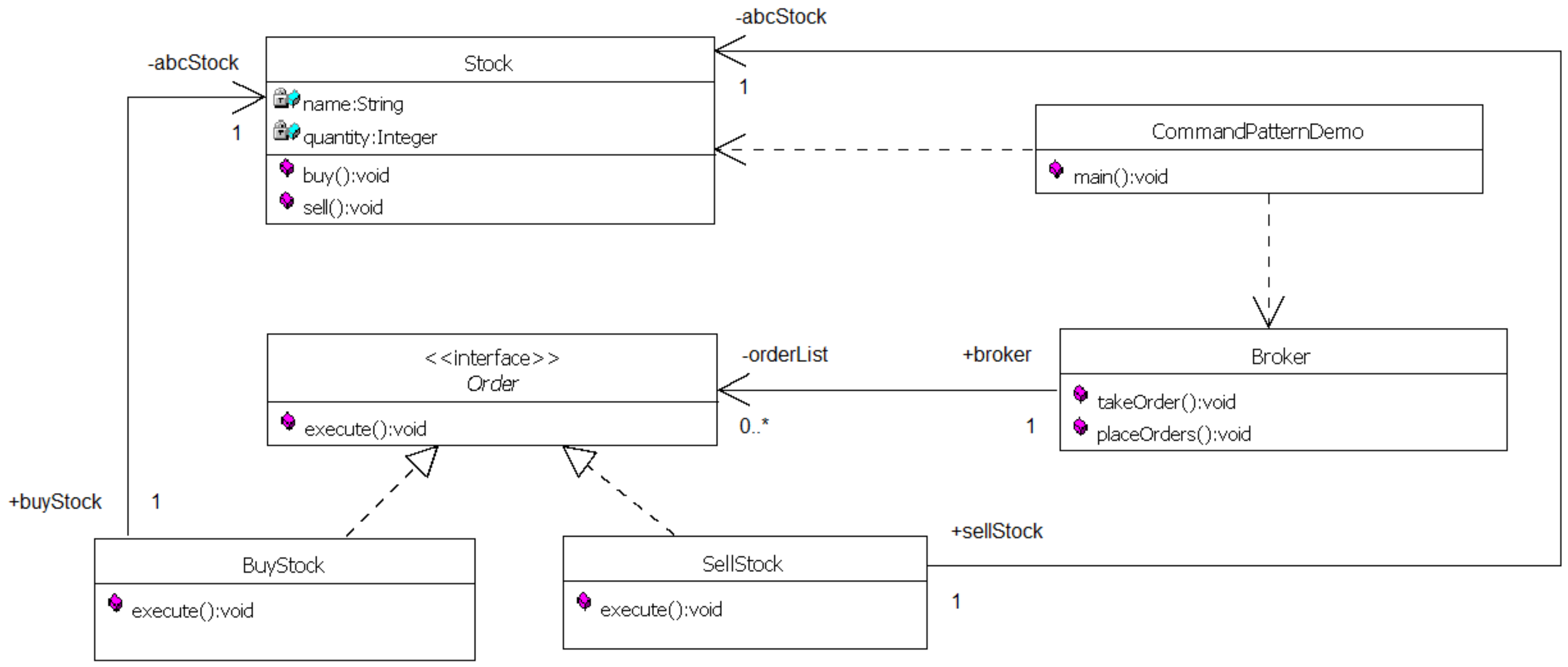


Figure 8-16 Applying the Composite design pattern to user interface widgets (UML class diagram). The Swing Component hierarchy is a `Composite` in which leaf widgets (e.g., `Checkbox`, `Button`, `Label`) specialize the `Component` interface, and aggregates (e.g., `Panel`, `Window`) specialize the `Composite` abstract class. Moving or resizing a `Composite` impacts all of its children.

The Command Pattern

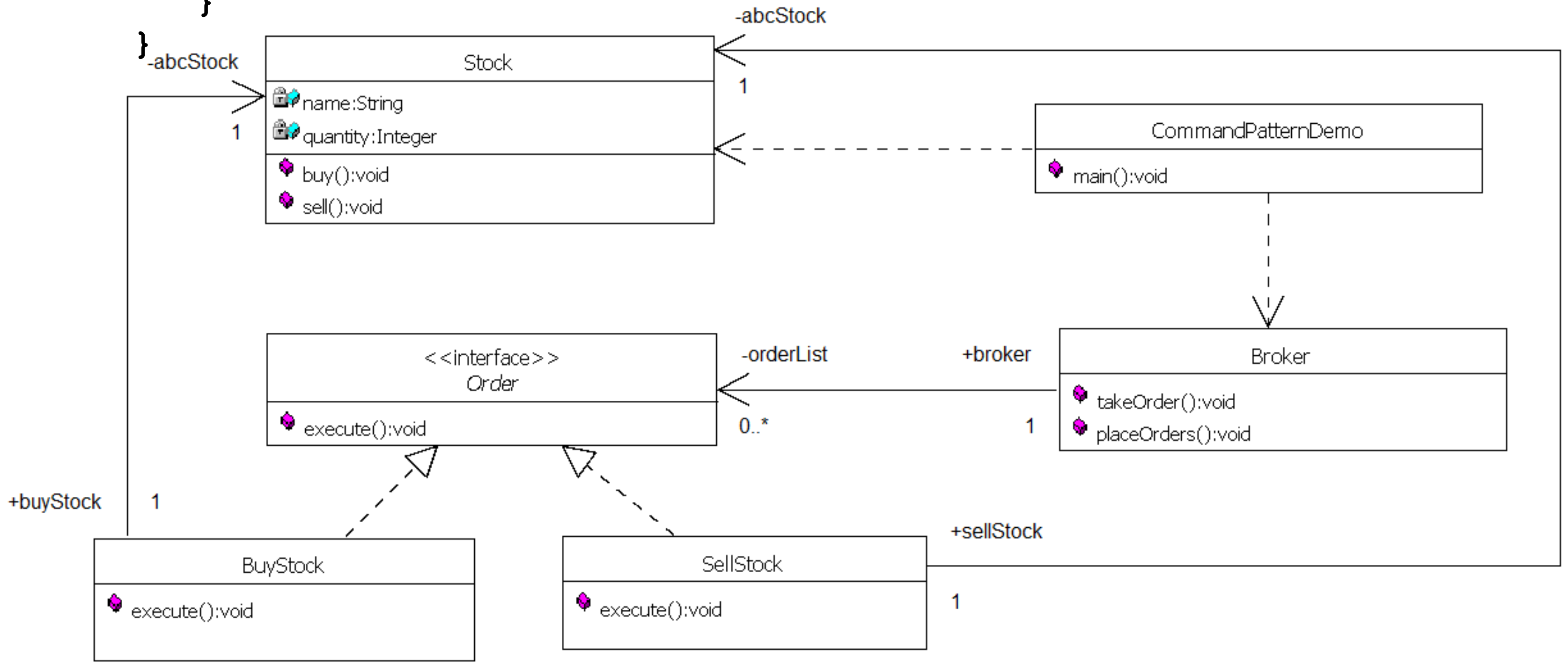


A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command.

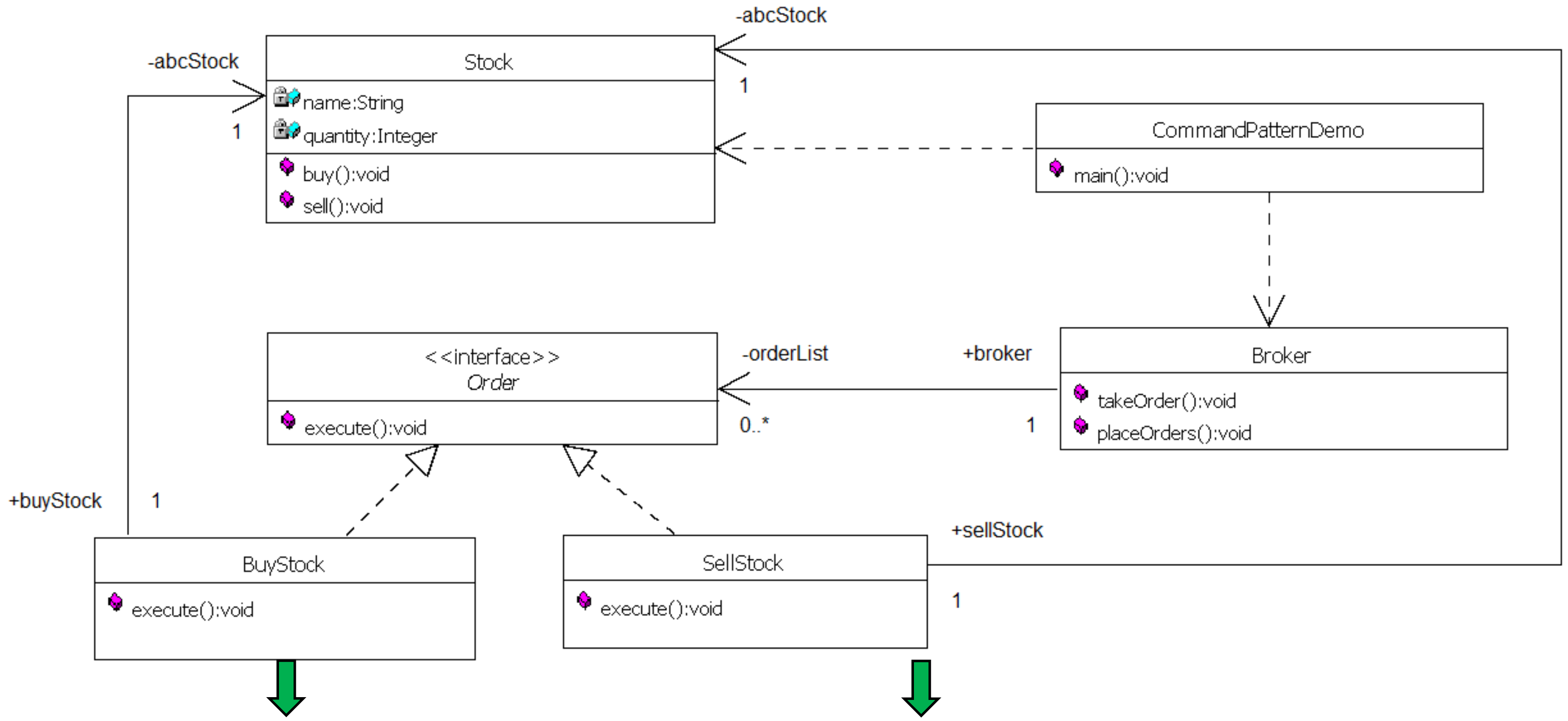
```
public interface Order {
 void execute();
}
```

The Command Pattern_2

```
public class Stock {
   private String name = "ABC";
   private int quantity = 10;
   public void buy(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] bought");
   }
   public void sell(){
      System.out.println("Stock [ Name: "+name+",
         Quantity: " + quantity +" ] sold");
   }
}
```



The Command Pattern_3

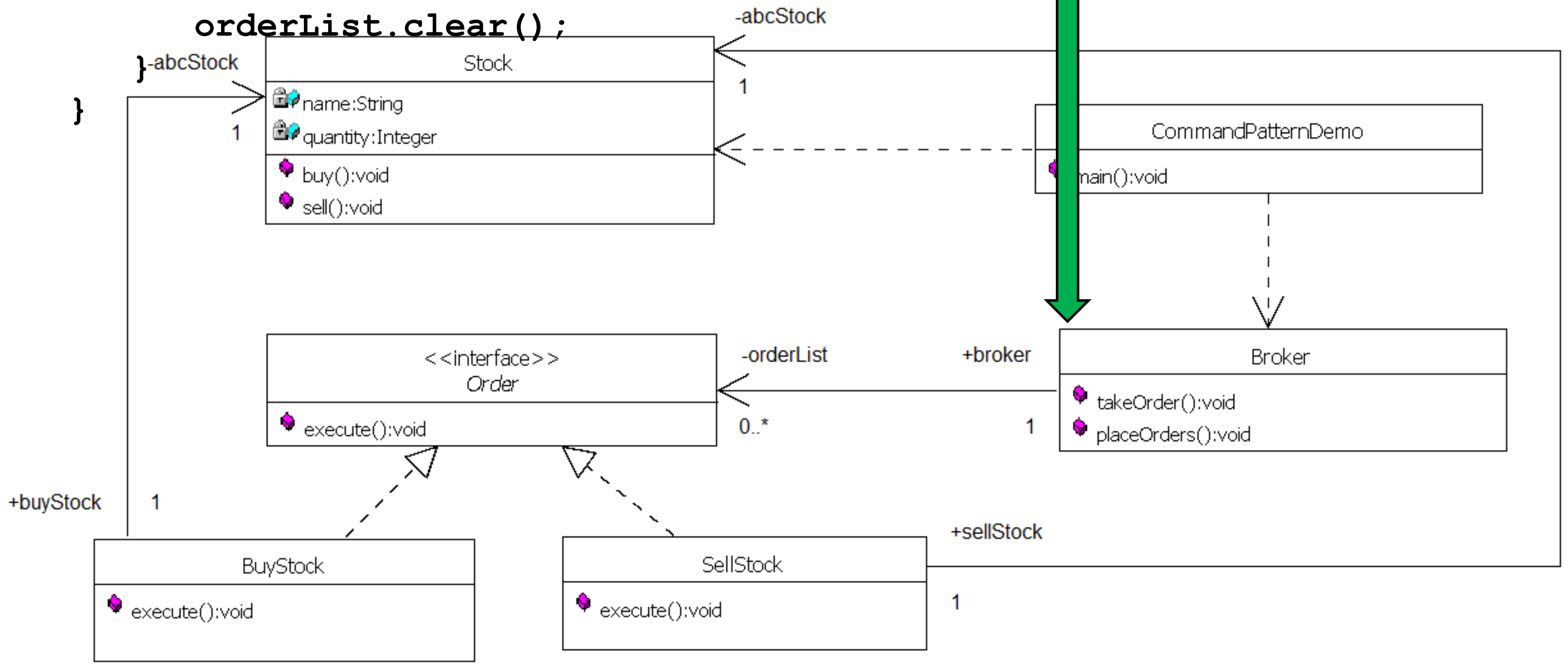


```
public class BuyStock implements Order {
   private Stock abcStock;
public BuyStock(Stock abcStock){
      this.abcStock = abcStock;
   }
   public void execute() {
      abcStock.buy();
   }
}
```

```
public class BuyStock implements Order {

public SellStock(Stock abcStock){
      this.abcStock = abcStock;
   }
   public void execute() {
      abcStock.sell();
   }
}
```

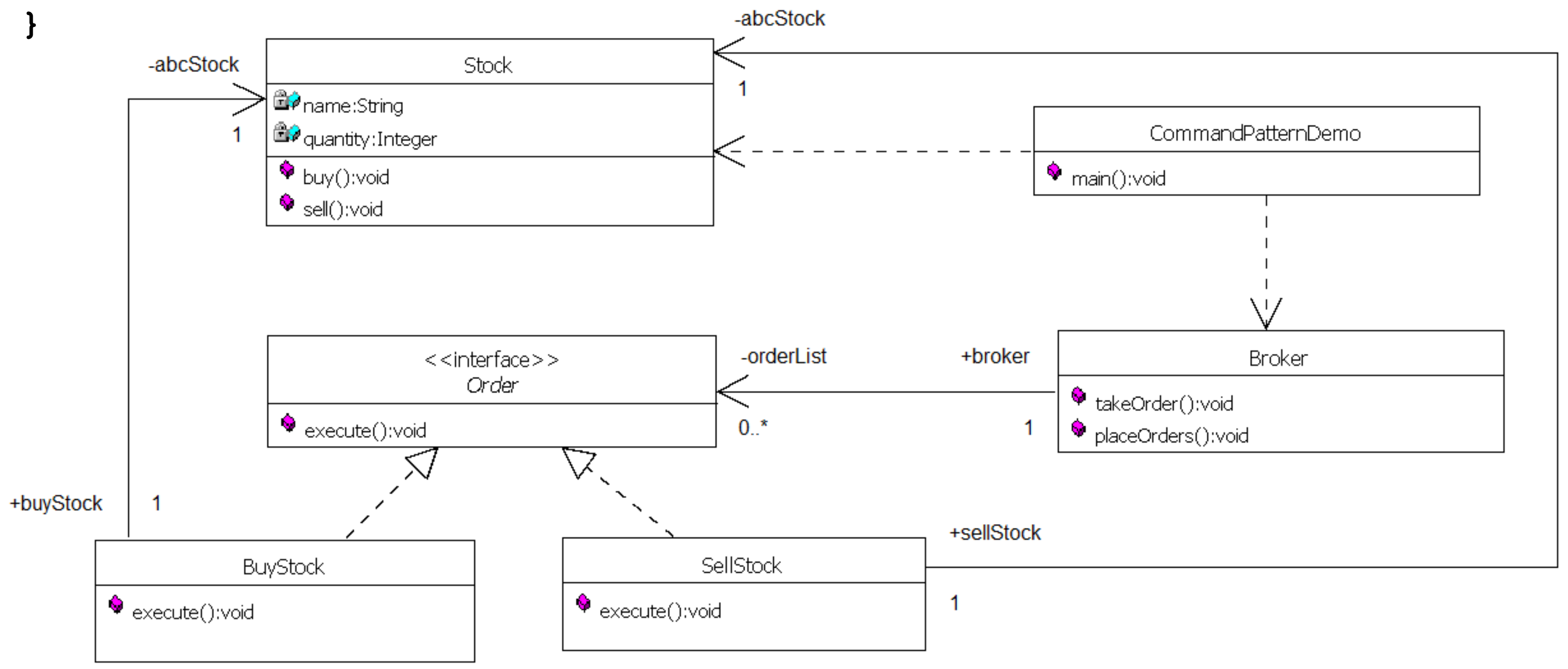
The Command Pattern_4

```
import java.util.ArrayList;
import java.util.List;
   public class Broker {
   private List<Order> orderList = new ArrayList<Order>();
   public void takeOrder(Order order){
      orderList.add(order);
   }
   public void placeOrders(){
      for (Order order : orderList) {
         order.execute();
      }
      orderList.clear();
   }
}
```



The Command Pattern_5

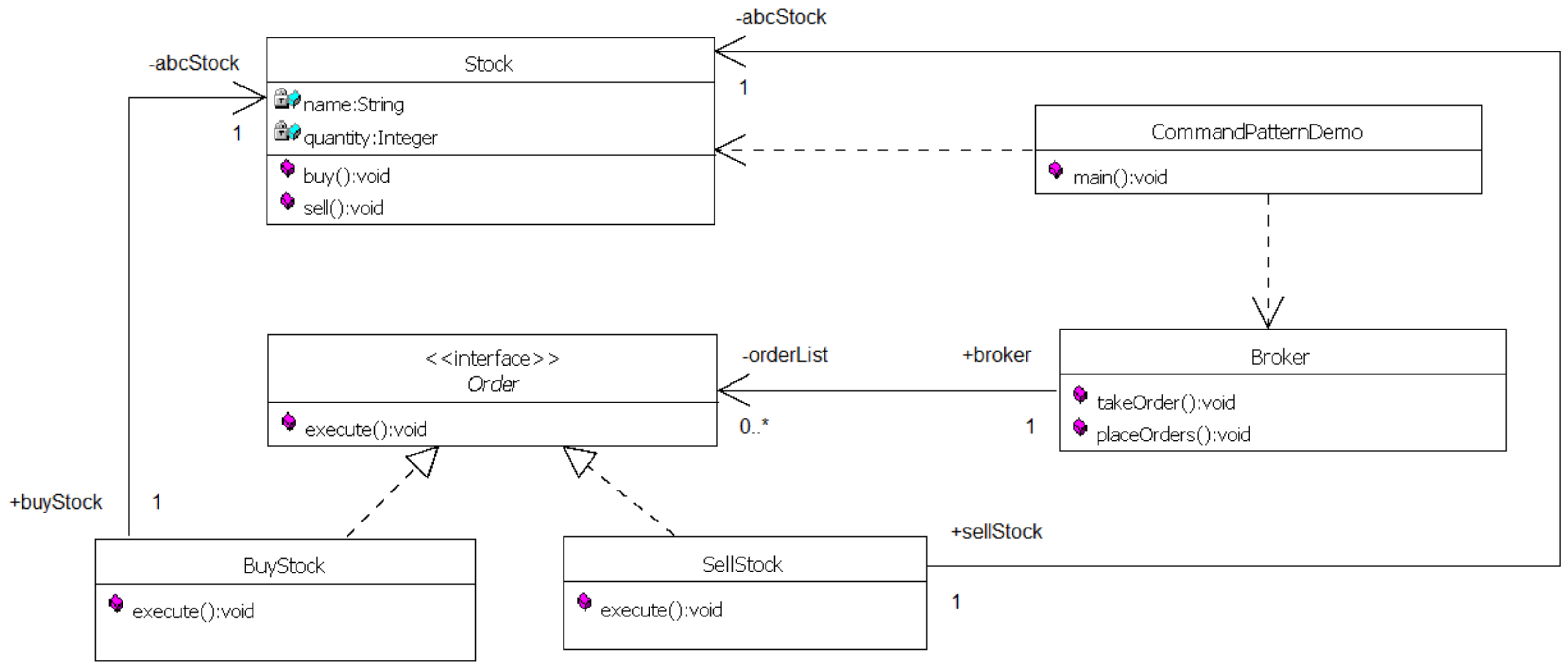
```
public class CommandPatternDemo {
   public static void main(String[] args) {
      Stock abcStock = new Stock();
      BuyStock buyStockOrder = new BuyStock(abcStock);
      SellStock sellStockOrder = new SellStock(abcStock);
      Broker broker = new Broker();
      broker.takeOrder(buyStockOrder);
      broker.takeOrder(sellStockOrder);
      broker.placeOrders();
   }
}
```



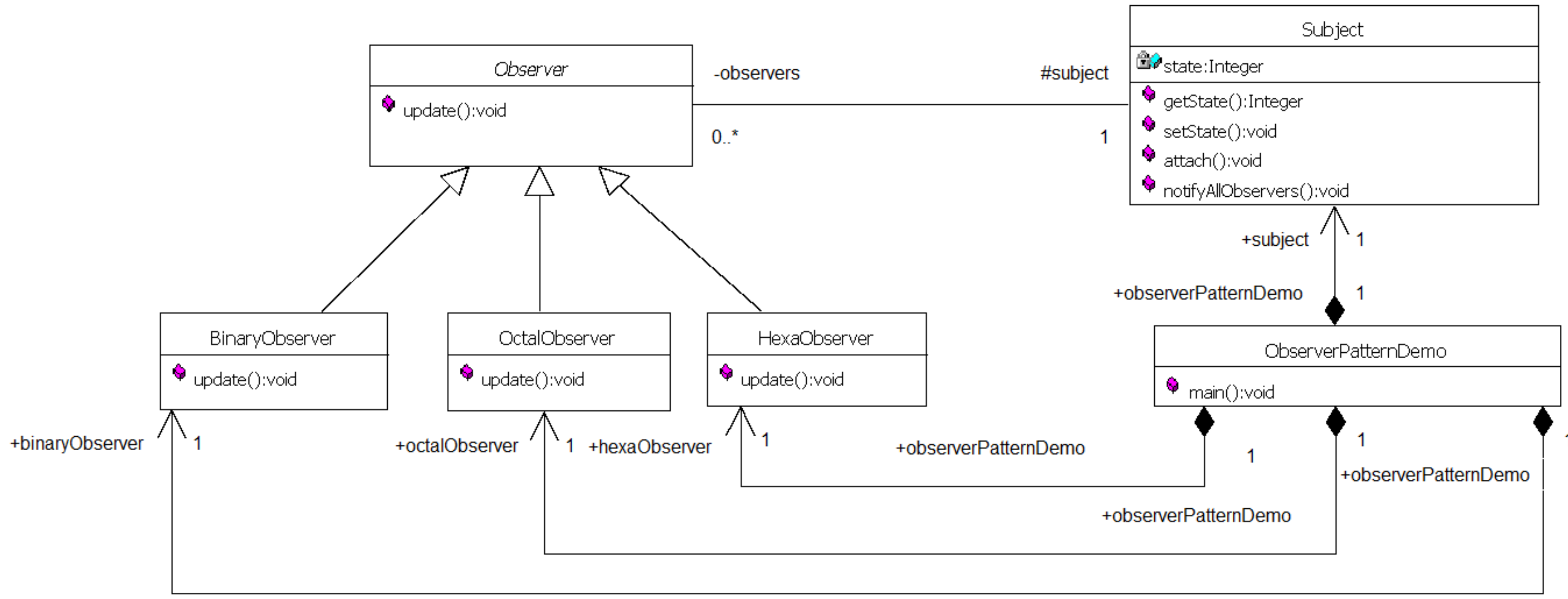
The Command Pattern_6

The output displayed after running CommandPatternDemo:

```
Stock [ Name: ABC, Quantity: 10 ] bought

Stock [ Name: ABC, Quantity: 10 ] sold
```



The Observer Pattern



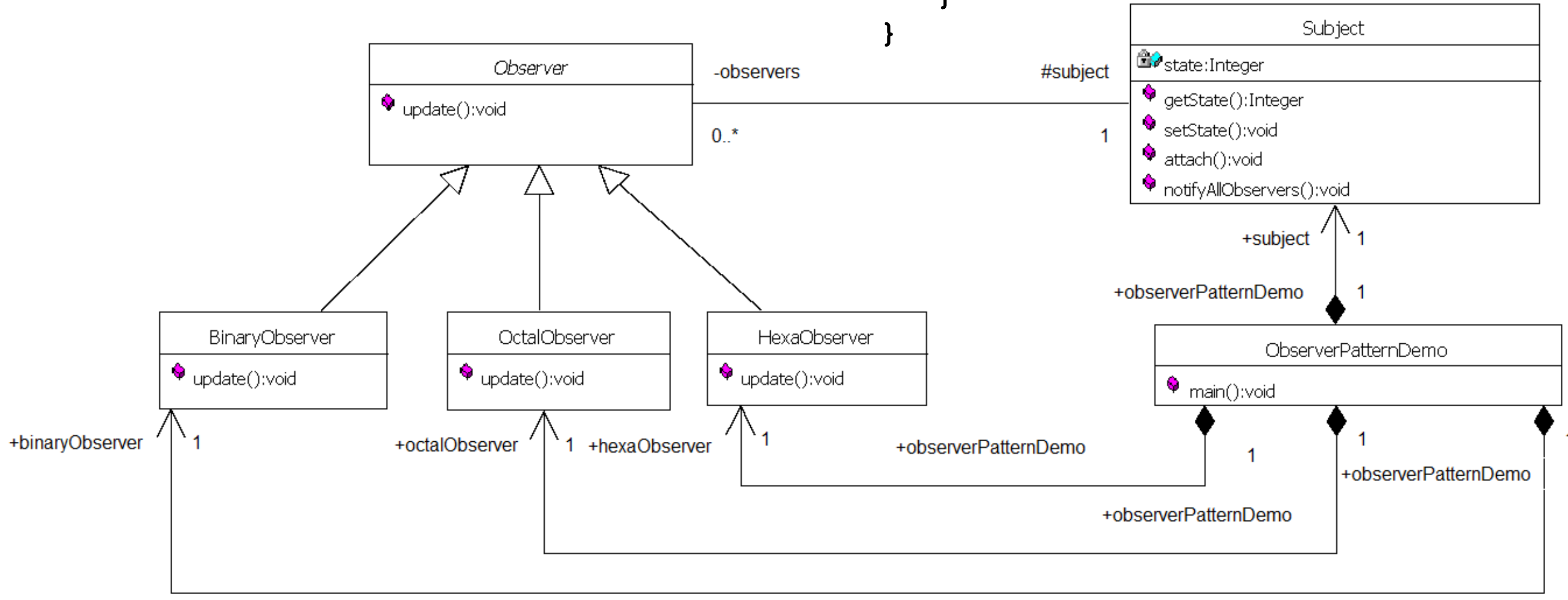
Observer pattern is used when there is one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. Observer pattern falls under behavioral pattern category.

Observer pattern uses three actor classes. Subject, Observer and Client. Subject is an object having methods to attach and detach observers to a client object.

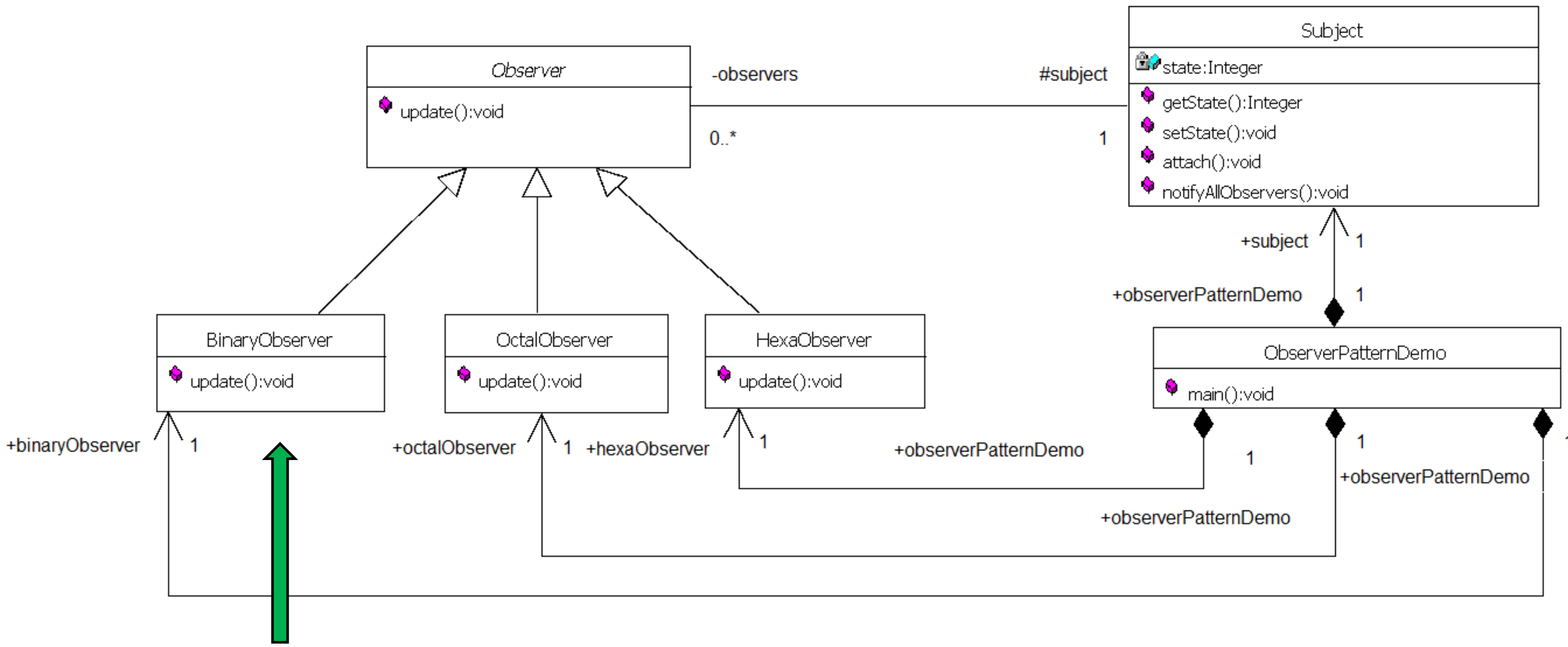
The Observer Pattern_2

```
import java.util.ArrayList;
import java.util.List;

public class Subject {

   private List<Observer> observers = new
ArrayList<Observer>();

   private int state;

public int getState() {
      return state;
   }
```

```
public void setState(int state) {
      this.state = state;
      notifyAllObservers();
   }

   public void attach(Observer observer){
      observers.add(observer);

   }
   public void notifyAllObservers(){
      for (Observer observer : observers)
{
         observer.update();
      }
   }
}
```

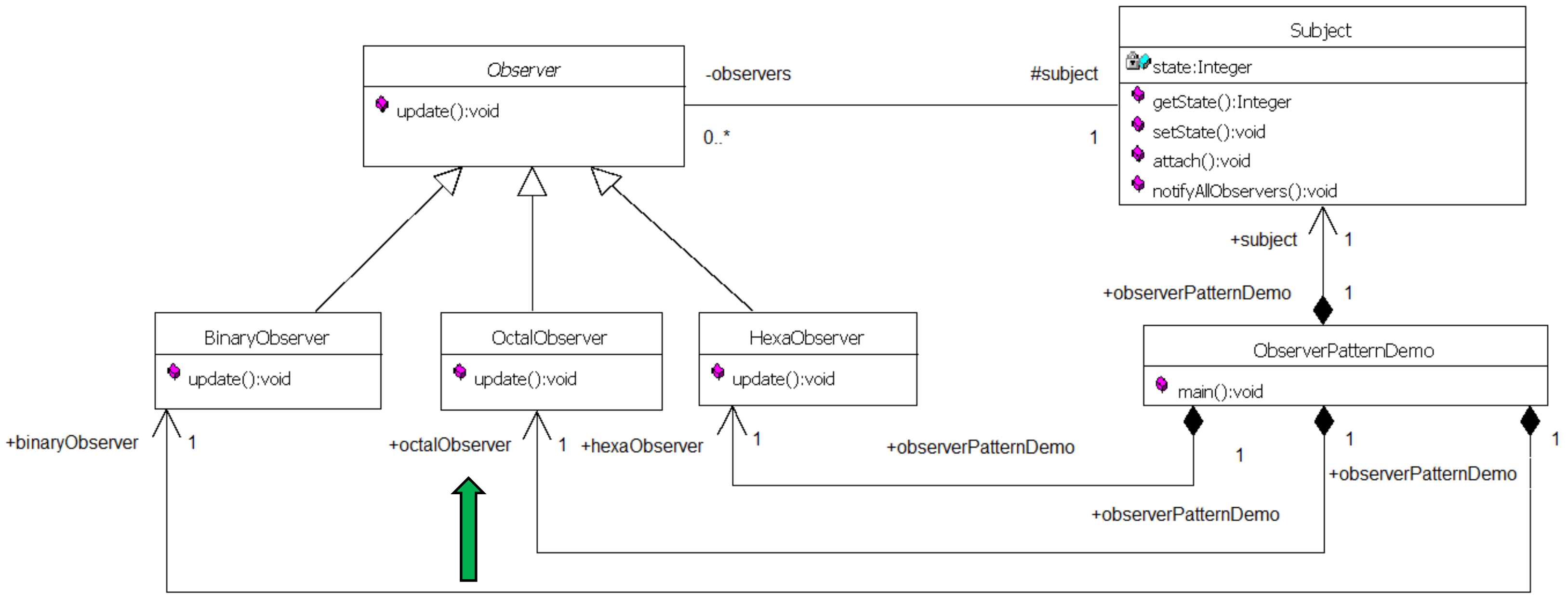


The Observer Pattern_3



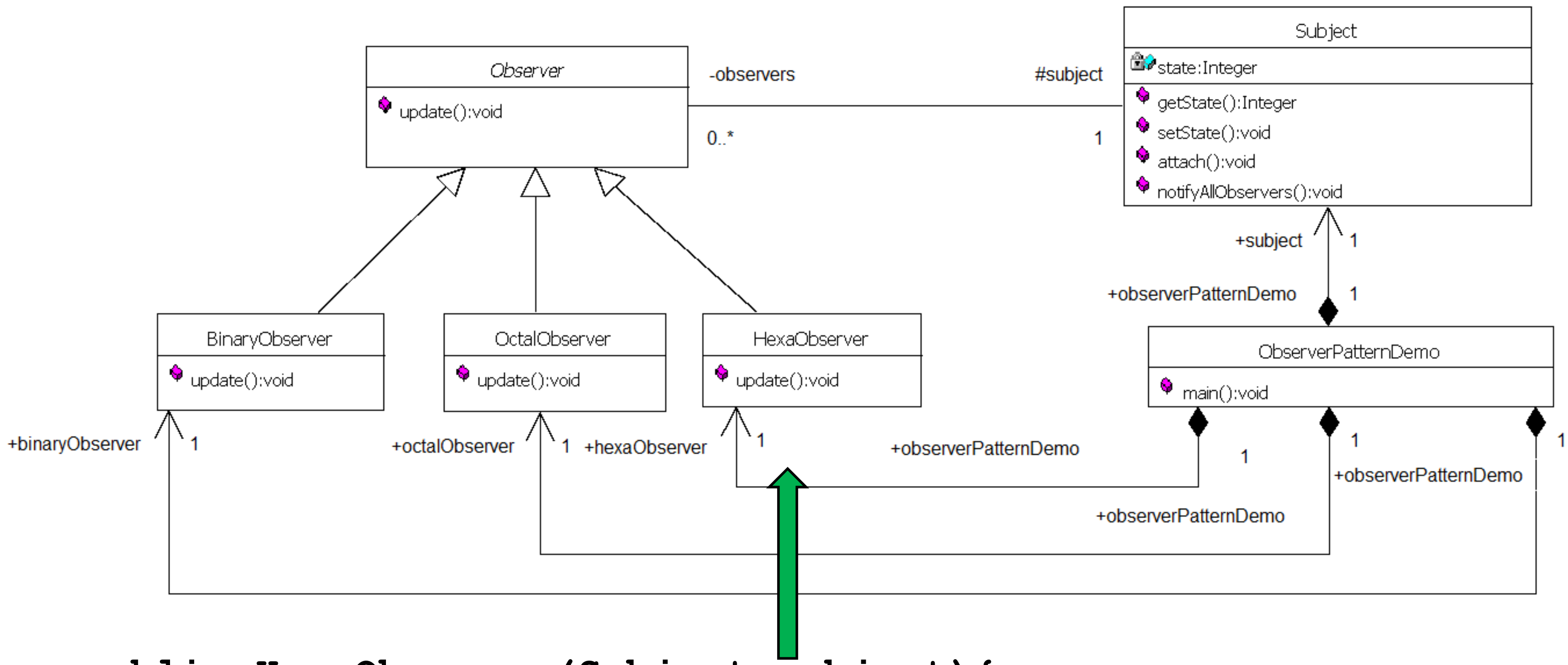
```
public BinaryObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }
   @Override
   public void update() {
      System.out.println( "Binary String: " +
Integer.toBinaryString( subject.getState() ) );
   }
```

The Observer Pattern_4



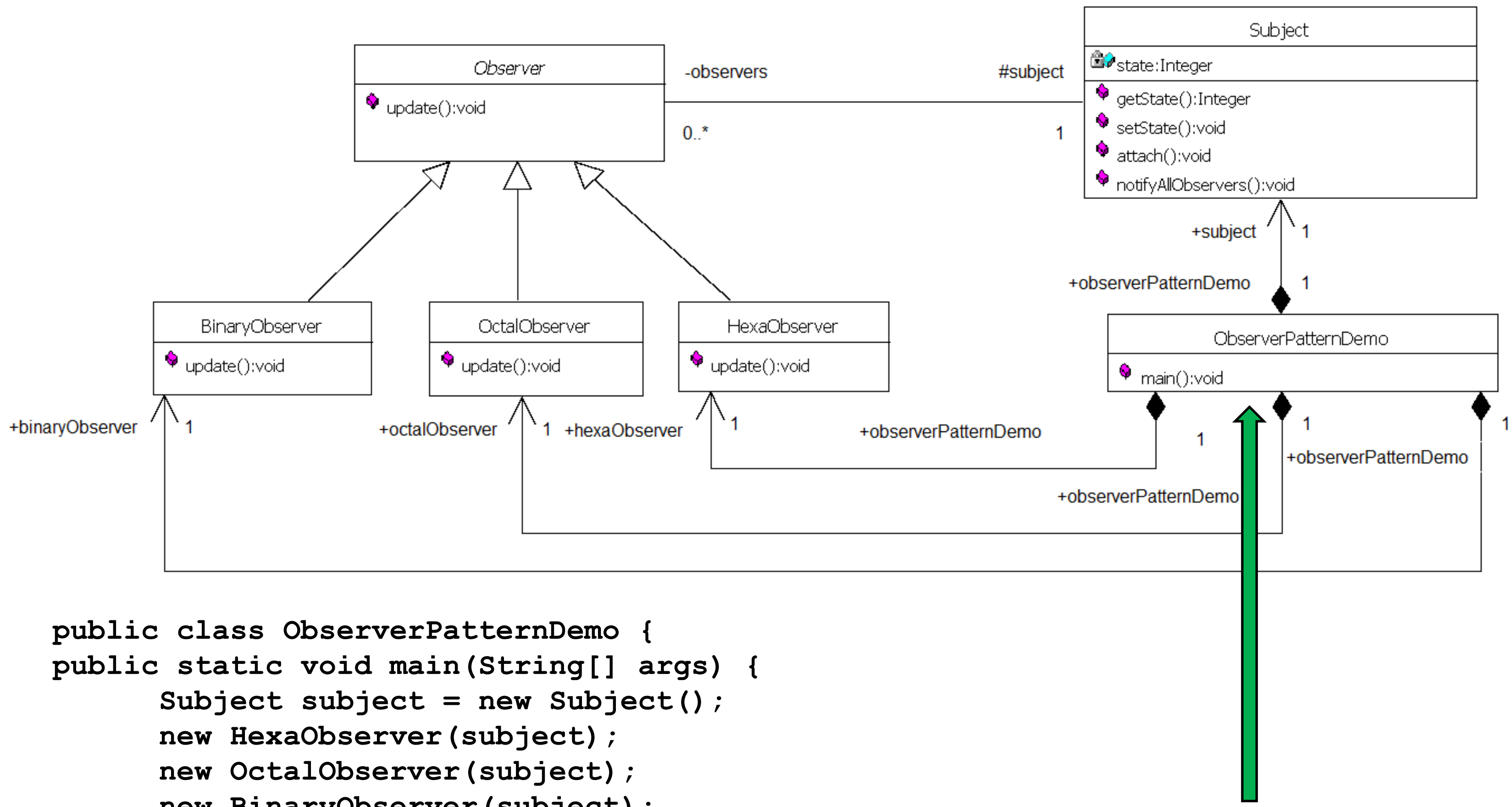
```
public OctalObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }
   @Override
   public void update() {
     System.out.println( "Octal String: " +
Integer.toOctalString( subject.getState() ) );
   }
```

The Observer Pattern_5



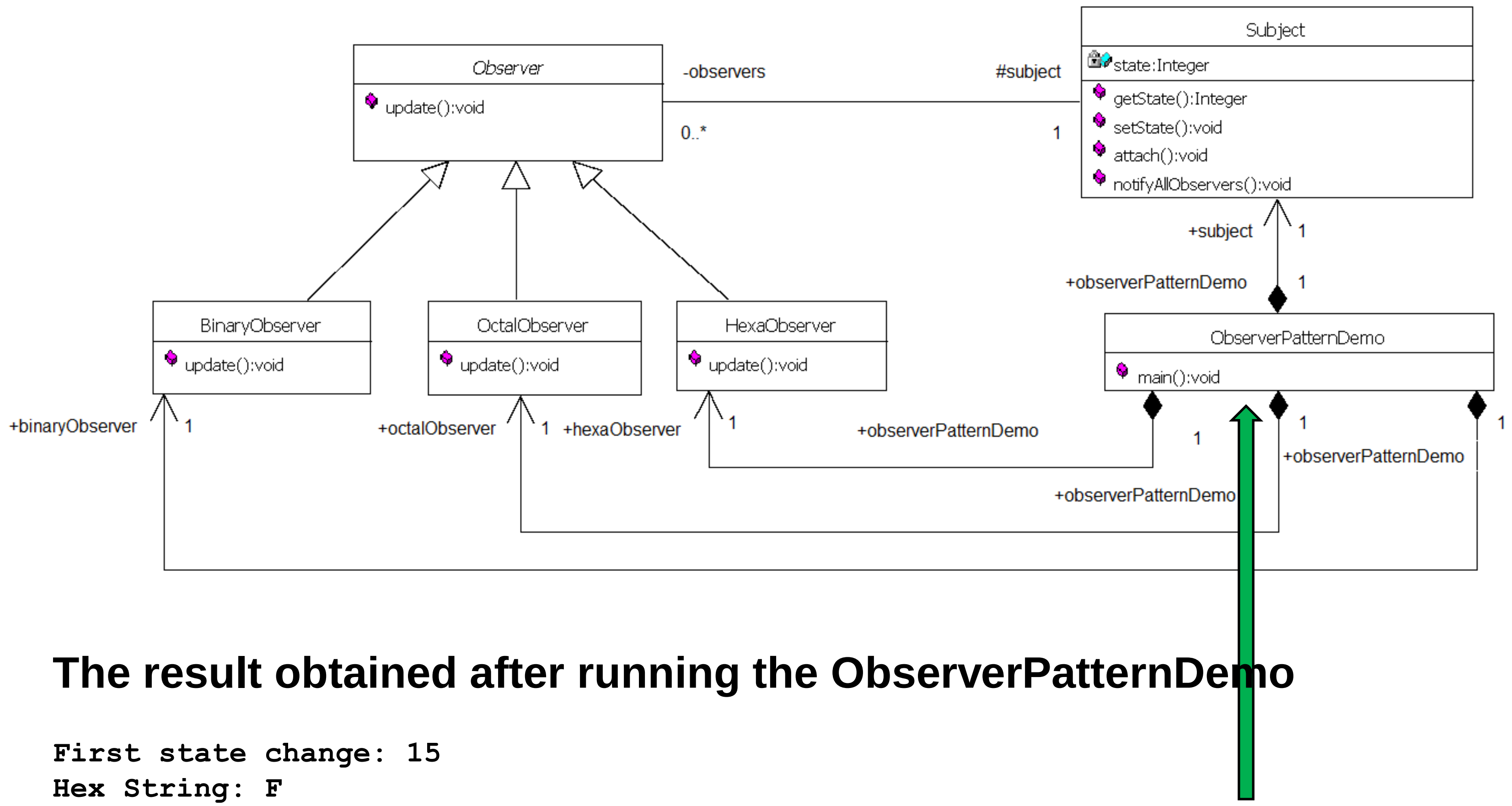
```
public HexaObserver(Subject subject){
      this.subject = subject;
      this.subject.attach(this);
   }

   @Override
   public void update() {
      System.out.println( "Hex String: " + Integer.toHexString(
subject.getState() ).toUpperCase() );
   }
```

The Observer Pattern_6



```
public class ObserverPatternDemo {
public static void main(String[] args) {
      Subject subject = new Subject();
      new HexaObserver(subject);
      new OctalObserver(subject);
      new BinaryObserver(subject);
      System.out.println("First state change: 15");
      subject.setState(15);
      System.out.println("Second state change: 10");
      subject.setState(10);
   }
}
```

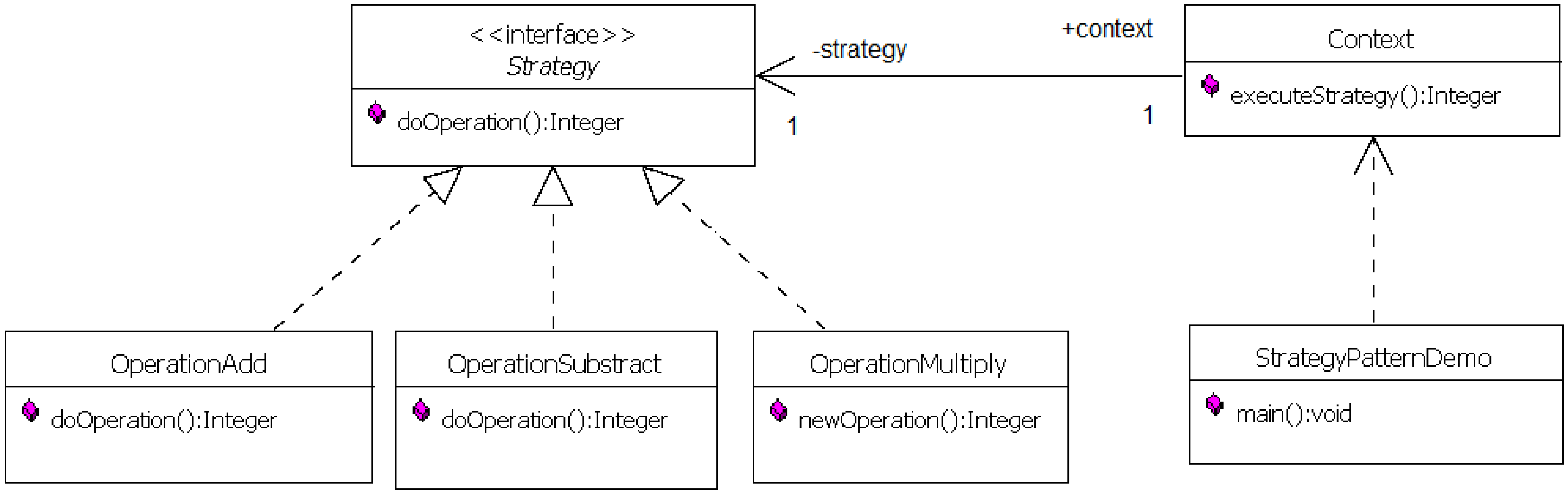
The Observer Pattern_6



The result obtained after running the ObserverPatternDemo

```
First state change: 15
Hex String: F
Octal String: 17
Binary String: 1111
Second state change: 10
Hex String: A
Octal String: 12
Binary String: 1010
```

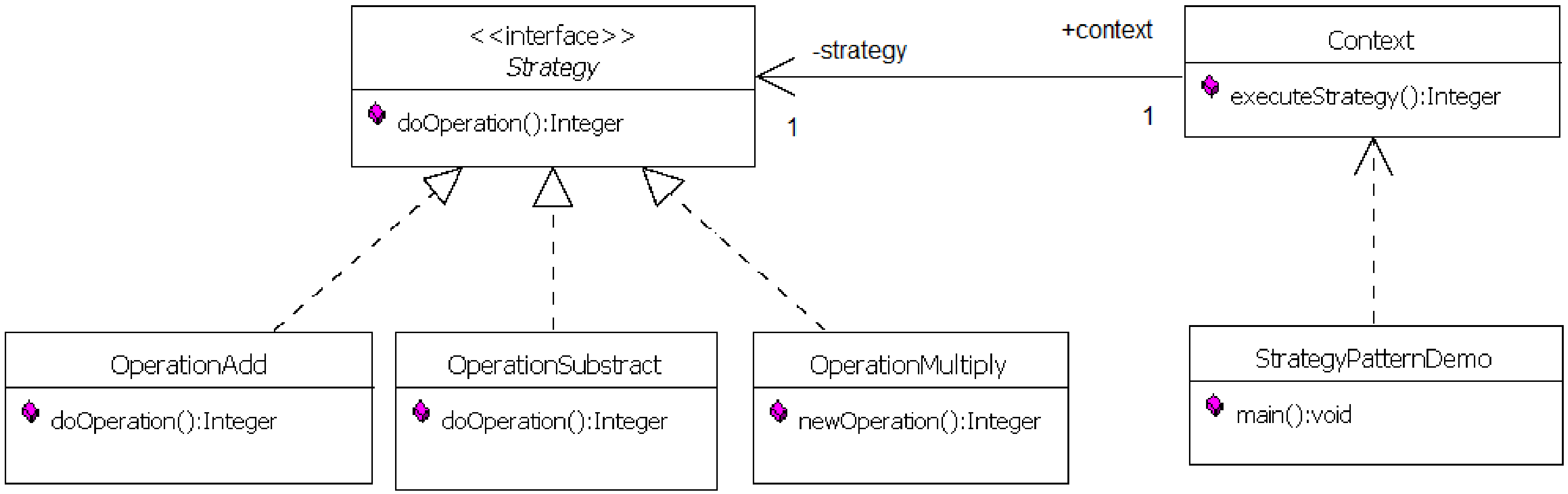
The Strategy Pattern



In Strategy pattern, a class behavior or its algorithm can be changed at run time. This type of design pattern comes under behavior pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

The Strategy Pattern_2

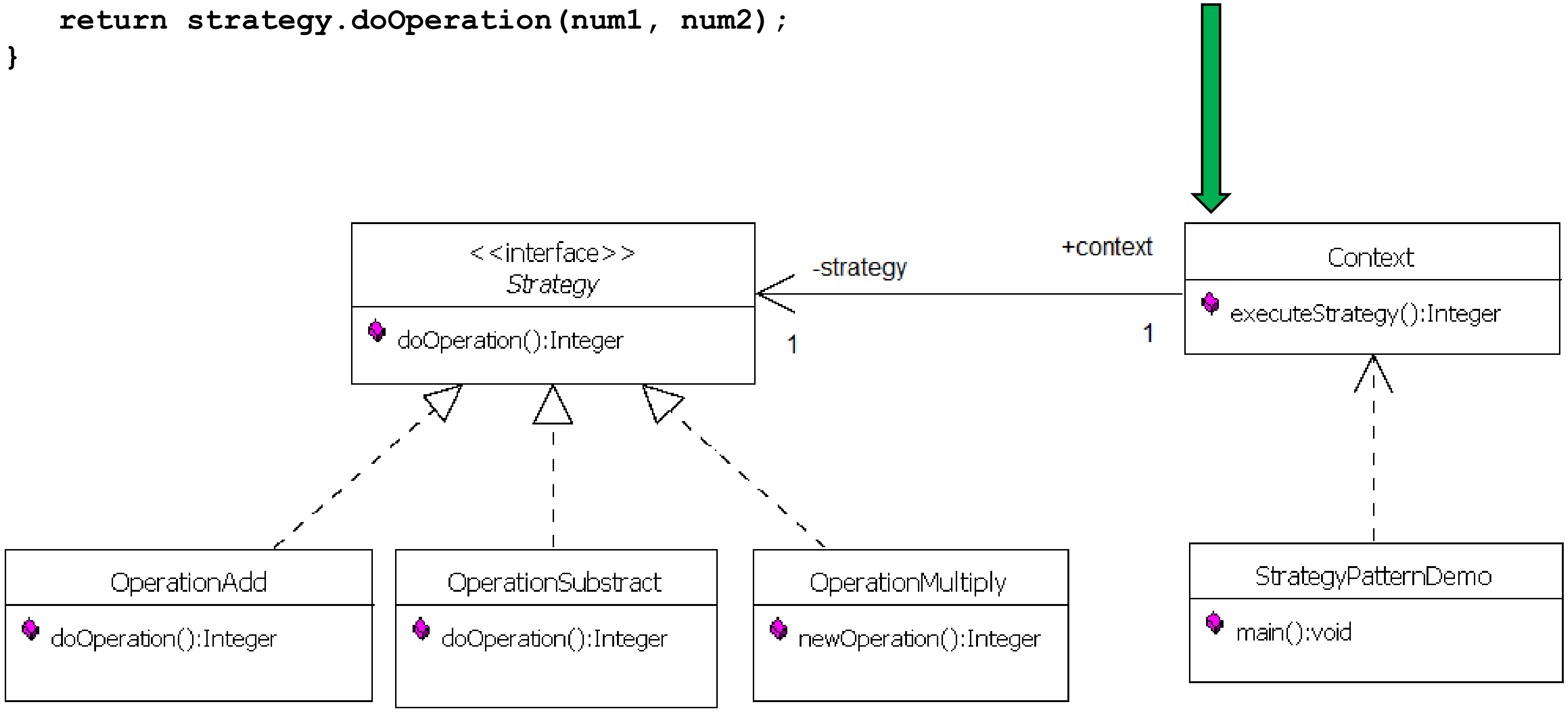


```
public interface Strategy {
   public int doOperation(int num1, int num2);
}

public class OperationAdd implements Strategy{
   @Override
   public int doOperation(int num1, int num2)
{
      return num1 + num2;
   }
}
```

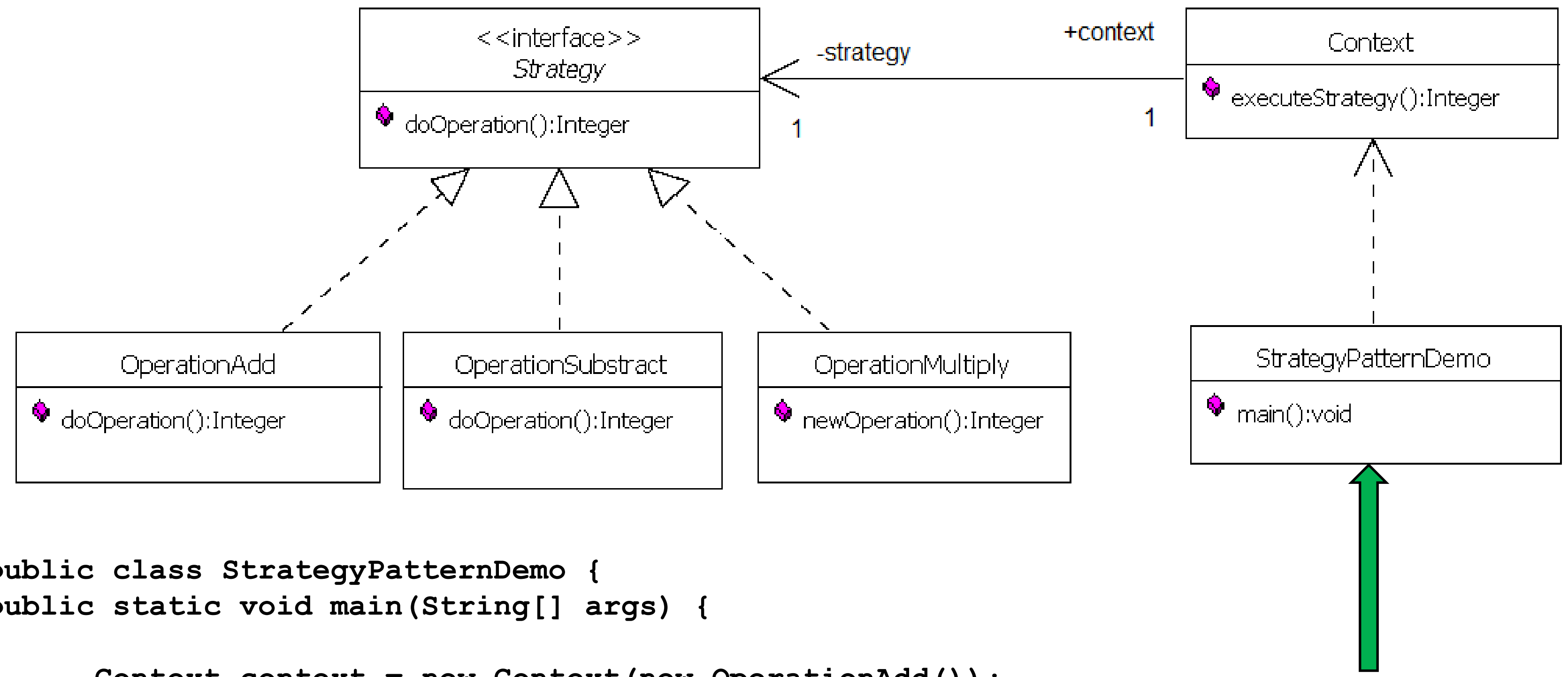
```
public class OperationSubstract implements
Strategy{
   @Override
   public int doOperation(int num1, int num2)
{
      return num1 - num2;
   }
}

public class OperationMultiply implements
Strategy{
   @Override
   public int doOperation(int num1, int num2)
{
      return num1 * num2;
   }
}
```

The Strategy Pattern_3

```
public class Context {
   private Strategy strategy;
public Context(Strategy strategy){
      this.strategy = strategy;
   }

   public int executeStrategy(int num1, int num2){
      return strategy.doOperation(num1, num2);
   }
}
```

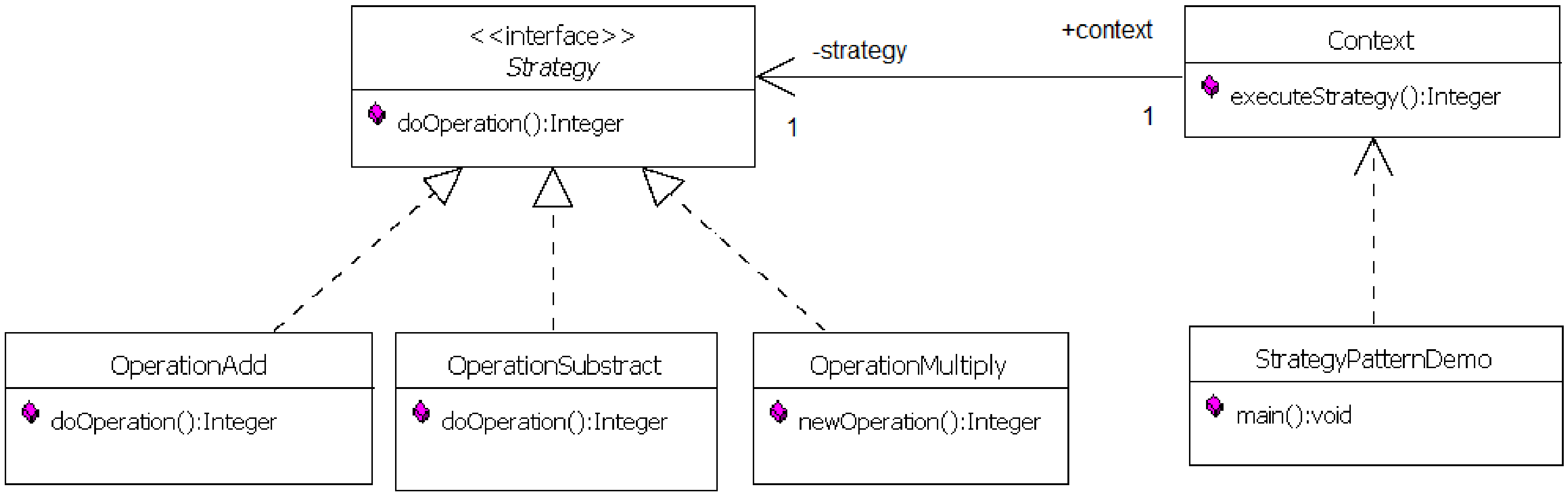


The Strategy Pattern_4



```
public class StrategyPatternDemo {
public static void main(String[] args) {

      Context context = new Context(new OperationAdd());
      System.out.println("10 + 5 = " + context.executeStrategy(10, 5));

 context = new Context(new OperationSubstract());
      System.out.println("10 - 5 = " + context.executeStrategy(10, 5));

      context = new Context(new OperationMultiply());
      System.out.println("10 * 5 = " + context.executeStrategy(10, 5));
   }
}         .
```

The Strategy Pattern_5



The result displayed after running StrategyPatternDemo

```
10 + 5 = 15
10 - 5 = 5
10 * 5 = 50
```

Object Design Activities

1. Reuse: Identification of existing solutions
   - Use of inheritance
   - Off-the-shelf components and additional solution objects
   - Design patterns
2. Interface specification
   - Describes precisely each class interface
3. Object model restructuring
   - Transforms the object design model to improve its understandability and extensibility
4. Object model optimization
   - Transforms the object design model to address performance criteria such as response time or memory utilization.

Object Design

Mapping Models to Code

Modeling of the Real World

- Modeling of the real world leads to a system that reflects today's realities but not necessarily tomorrows.
- There is a need for *reusable* and flexible designs
- Design knowledge such as the adapter pattern complements application domain knowledge and solution domain knowledge.

References for Patterns examples

- Design Patterns in Java Tutorial
- https://www.tutorialspoint.com/design_pattern/index.htm