MedTech – Mediterranean Institute of Technology
Software Engineering

Chapter 4 – Design Patterns

Known Patterns and Design and Implementation Examples

Dr. Lilia SFAXI
www.liliasfaxi.wix.com/liliasfaxi

Design Patterns

DESIGN PATTERNS: DEFINITION AND UTILITY

Design Patterns

Design Patterns: Definition and Utility

- In software engineering, a design pattern is a general repeatable solution to a commonly occuring problem in software design

- It isn't a finished design that can be transformed directly into code, but a **description** or **template** for how to solve a problem that can be used in many different situations

Design Patterns: Usage

Design Patterns: Definition and Utility

- Design patterns:
  - Provide general solutions, documented in a format that doesn't require specifics tied to a particular problem
  - Can speed up the development process by providing tested, proven development paradigms
  - Help you benefit from the experience of fellow developers
  - Prevent subtle issues that can cause major problems
  - Improve code readability for coders and architects familiar with them

Design Patterns: Essential Elements

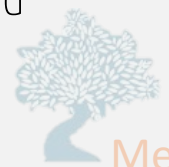
Design Patterns: Definition and Utility

- A pattern has four essential elements:
  - The pattern name that we use to describe a design problem
  - The problem that describes when to apply the pattern
  - The solution that describes the elements that make up the design
  - The consequences that are the results and trade-offs of applying the pattern

GoF Design Patterns

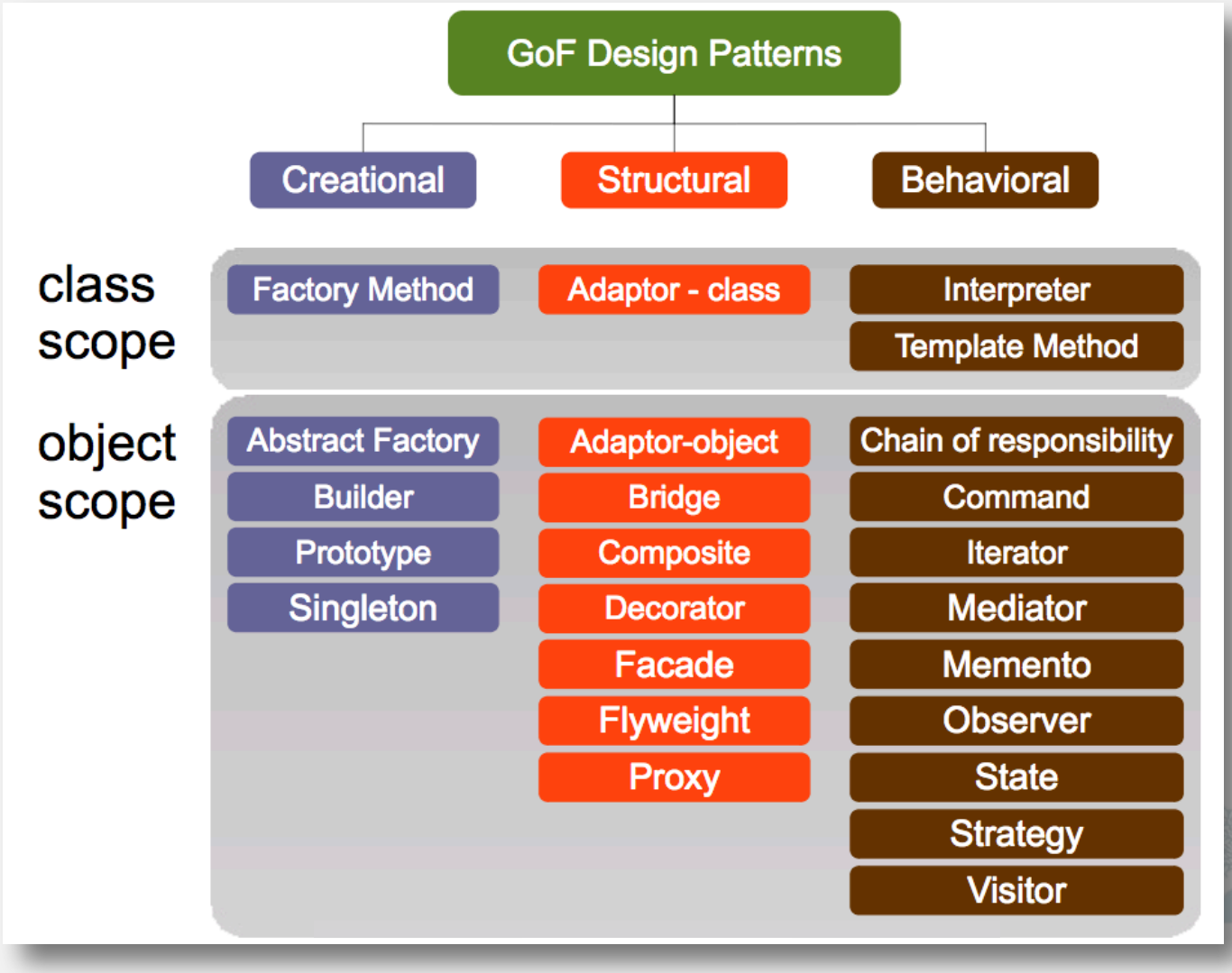
Design Patterns: Definition and Utility

- The **Gang of Four** are the four authors of the book « Design Patterns: Elements of Reusable Object-Oriented Software »
- Defined 23 design patterns for recurrent design issues, called GoF design patterns
- Classified by *purpose*:
  - Structural : Concerns the **composition** of classes and objects
  - Behavioral : Characterizes the **interaction and responsibility** of objects and classes
  - Creational : Concerns the **creation process** of objects and classes
- … and by *scope:*
  - Class scope: relationship between classes and subclasses, defined statically
  - Object scope: object relationships, dynamic



GoF Design Patterns

Design Patterns: Definition and Utility



Design Patterns

CREATIONAL PATTERNS

Creational Patterns: Definition

Creational Patterns

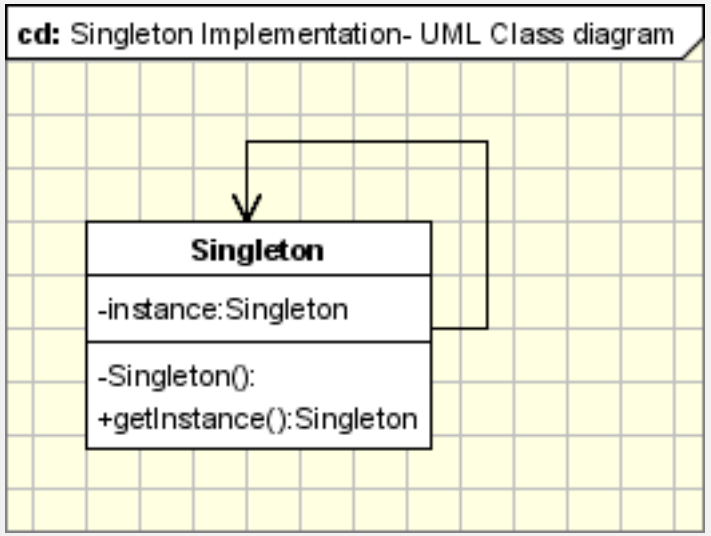
- Creational patterns abstract the instantiation process.
- They help to make a system independent of how its objects are created, composed, and represented
  - Creational patterns for classes use inheritance to vary the class that is instantiated.
  - Creational patterns for objects delegate instantiation to another object.

- Examples:
  - Factory
  - Singleton
  - Builder
  - Prototype

Singleton
Creational Patterns

- Ensure that only one instance of a class is created and provide a global access point to the object.

```
class Singleton
{
        private static Singleton instance;
        private Singleton()
        {
                ...
        }

        public static synchronized Singleton getInstance()
        {
                if (instance == null)
                        instance = new Singleton();

                return instance;
        }
        ...
        public void doSomething()
        {
                ...
        }
}
```



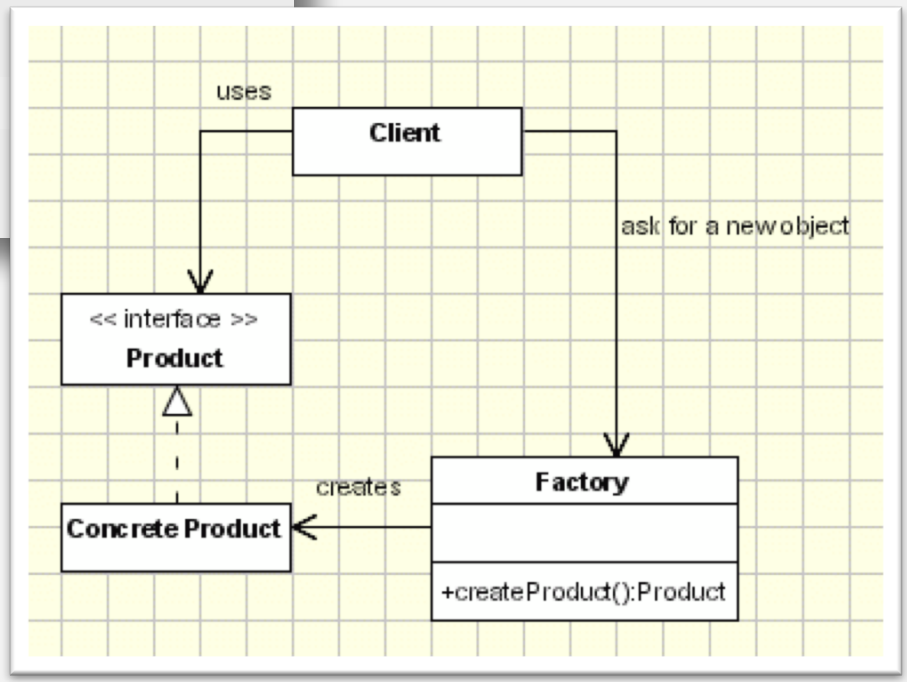
Singleton: Usage
Creational Patterns

- When to Use
  - When we must ensure that only one instance of a class is created
  - When the instance must be available through all the code
  - In multi-threading environments when multiple threads must access the same resources through the same singleton object.
- Common Usage
  - Logger Classes
  - Configuration Classes
  - Accessing resources in shared mode
  - Other design patterns implemented as Singletons:
    - Factories and Abstract Factories, Builder, Prototype

Factory
Creational Patterns

- Creates objects without exposing the instantiation logic to the client
- Refers to the newly created object through a common interface.

```
public class ProductFactory{
        public Product createProduct(String ProductID){
                if (id==ID1)
                        return new OneProduct();
                if (id==ID2) return
                        return new AnotherProduct();
                ... // so on for the other Ids

        return null;
    }
    ...
}
```



Factory: Usage
Creational Patterns

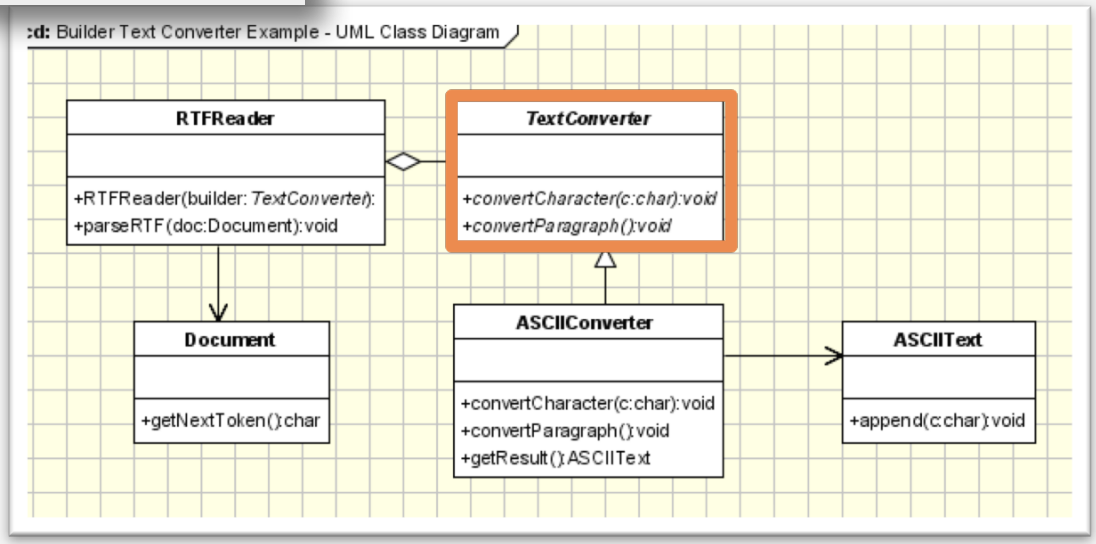
- When to use
  - When a framework delegates the creation of objects derived from a common superclass to the factory
  - When we need flexibility in adding new types of objects that must be created by the class

- Common Usage
  - factories providing an xml parser:
    - javax.xml.parsers.DocumentBuilderFactory
    - javax.xml.parsers.SAXParserFactory
  - java.net.URLConnection

Builder
Creational Patterns

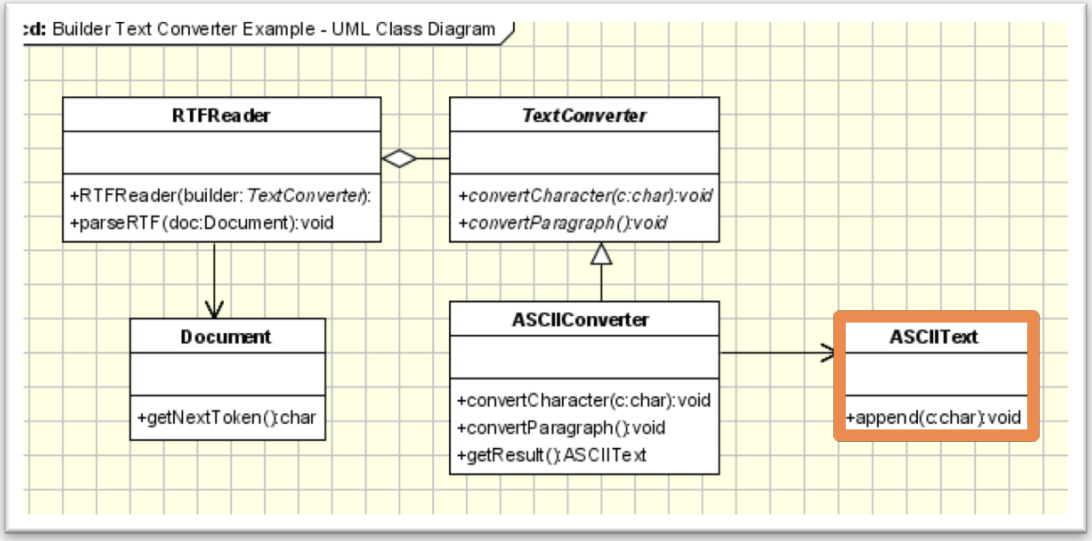
- Defines an instance for creating an object but letting subclasses decide which class to instanciate
- Allows finer control over the construction process

```
//Abstract Builder
class abstract class TextConverter{
        abstract void convertCharacter(char c);
        abstract void convertParagraph();
}
```



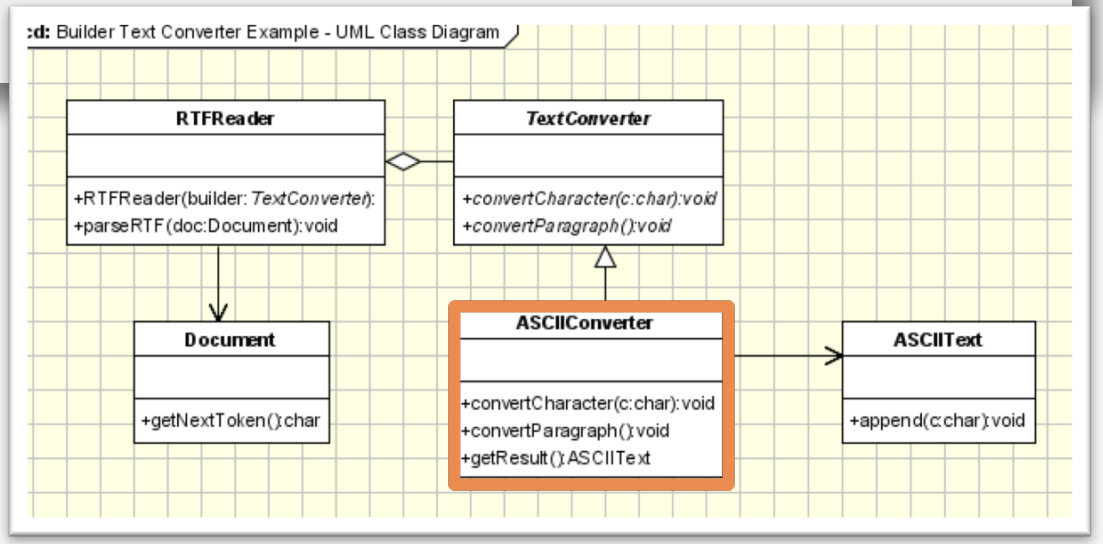
Builder
Creational Patterns

```
// Product
class ASCIIText{
        public void append(char c){ //Implement the code here }
}
```



Builder
Creational Patterns

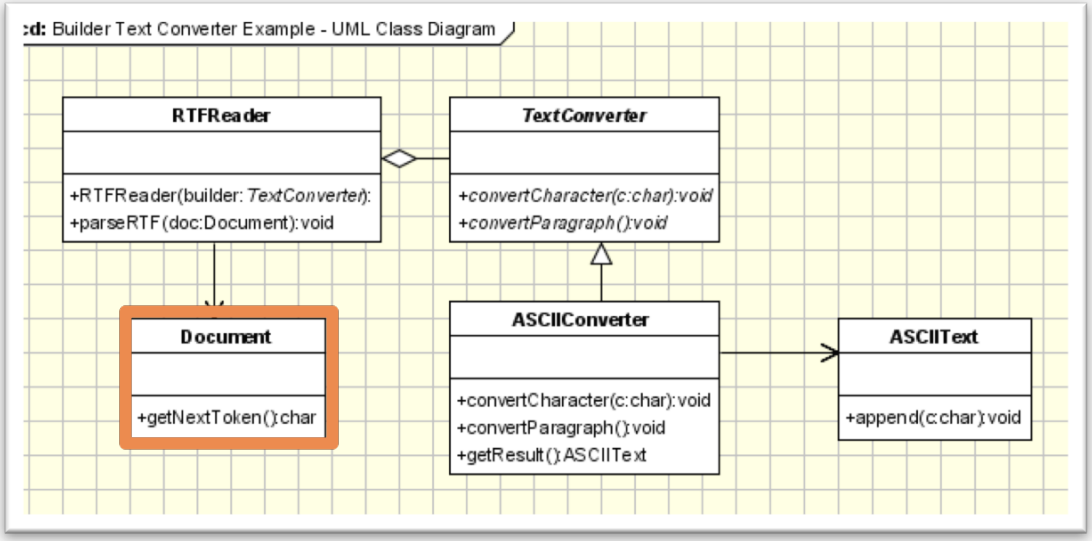
```
//Concrete Builder
class ASCIIConverter extends TextConverter{
        ASCIIText asciiTextObj;//resulting product

        /*converts a character to target representation and appends to the resulting*/
        object void convertCharacter(char c){
                char asciiChar = new Character(c).charValue();
                        //gets the ascii character
                asciiTextObj.append(asciiChar);
        }
        void convertParagraph(){}
        ASCIIText getResult(){
                return asciiTextObj;
        }
}
```



Builder

Creational Patterns

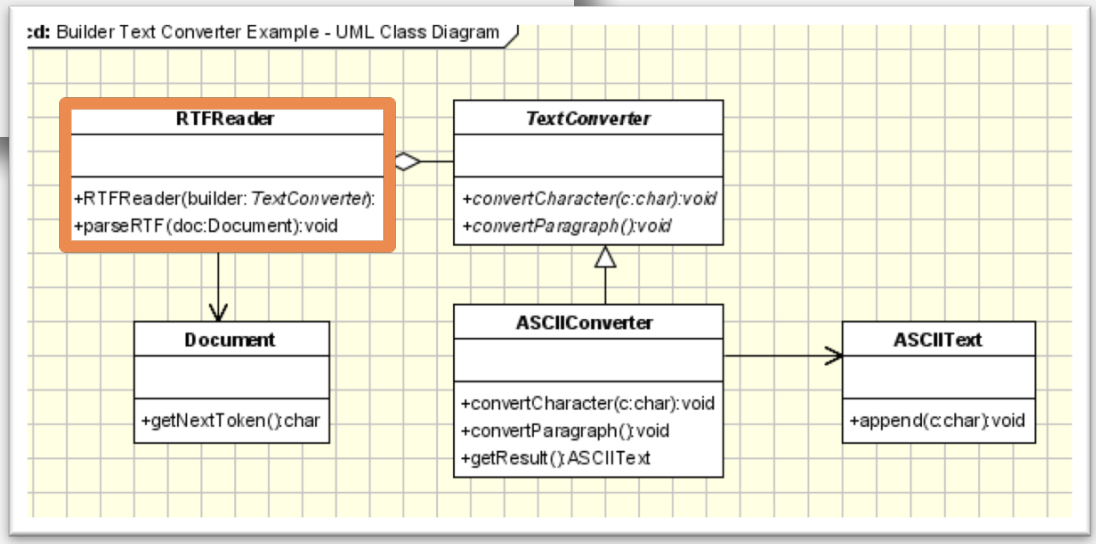
```
//This class abstracts the document object
class Document{
        static int value;
        char token;
        public char getNextToken(){
                //Get the next token
                return token;
        }
}
```



Builder

Creational Patterns

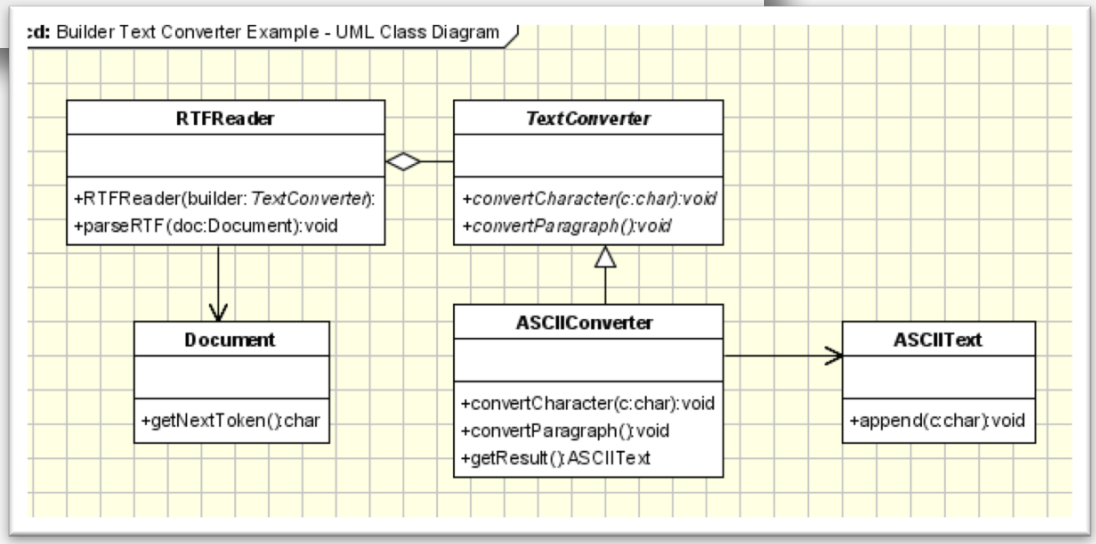
```
//Director
class RTFReader{
        private static final char EOF='0'; //Delimitor for End of File
        final char CHAR='c';
        final char PARA='p';
        char t;
        TextConverter builder;
        RTFReader(TextConverter obj){
                builder=obj;
        }
        void parseRTF(Document doc){
                while ((t=doc.getNextToken())!= EOF){
                        switch (t){
                                case CHAR: builder.convertCharacter(t);
                                case PARA: builder.convertParagraph();
                        }
                }
        }
}
```



Builder

Creational Patterns

```
//Client
public class Client{
        void createASCIIText(Document doc){
                ASCIIConverter asciiBuilder = new ASCIIConverter();
                RTFReader rtfReader = new RTFReader(asciiBuilder);
                rtfReader.parseRTF(doc);
                ASCIIText asciiText = asciiBuilder.getResult();
        }
        public static void main(String args[]){
                Client client=new Client();
                Document doc=new Document();
                client.createASCIIText(doc);
                system.out.println("This is an example of Builder Pattern");
        }
}
```



Builder: Usage
Creational Patterns

- When to use
  - When the creation algorithm of a complex object is independent from the parts that actually compose the object
  - When the system needs to allow different representations for the objects that are being built

- Builder and Factory
  - Very similar to the Factory pattern
  - **Factory**: the client uses the factory's methods to create its own objects
  - **Builder**: the Builder class is instructed on how to create the object and then it is asked for it, but the way that the class is put together is up to the Builder class

Design Patterns

STRUCTURAL PATTERNS

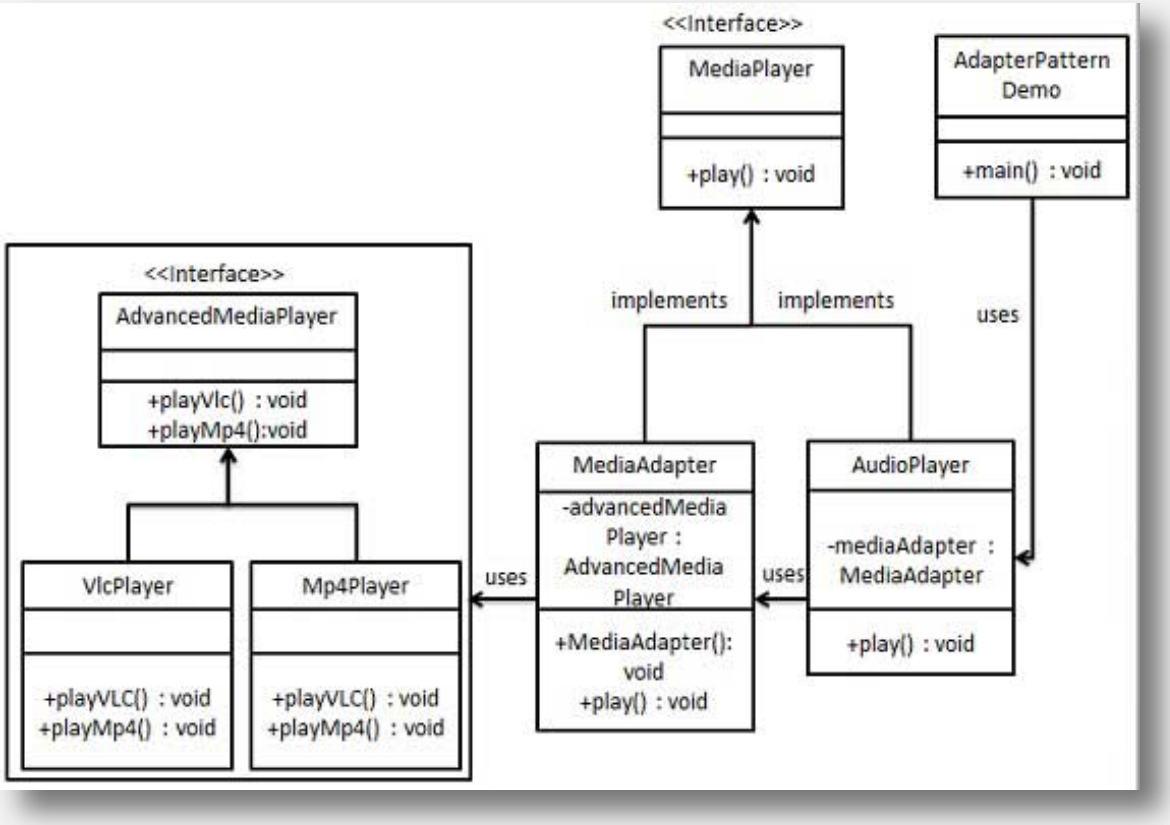
Definition
Structural Patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
  - Structural class patterns use inheritance to compose interfaces or implementations.
  - Structural object patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition.

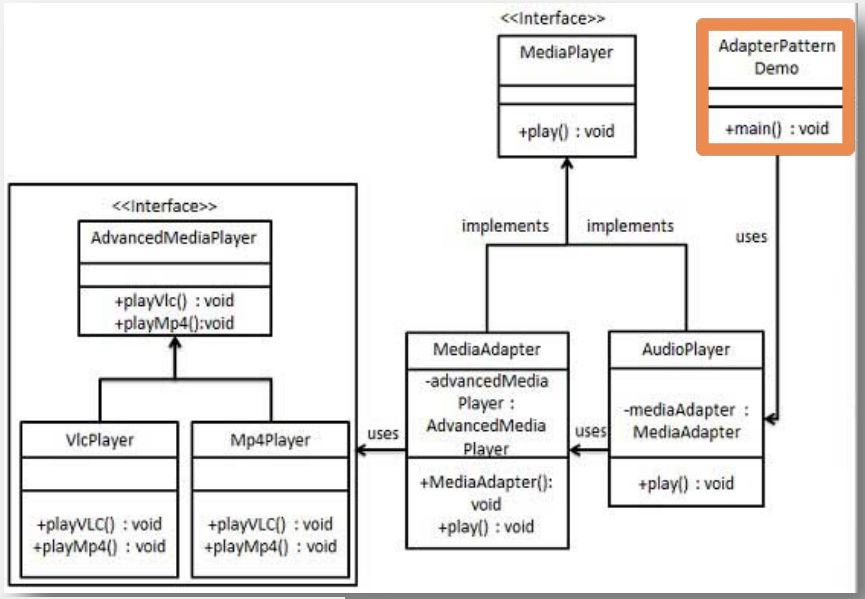
- Examples:
  - Adapter
  - Proxy
  - Bridge
  - Composite

Adapter
Structural Patterns

- Converts the interface of a class into another interface the clients expect
- Lets classes work together, that normally wouldn't, due to incompatible interfaces

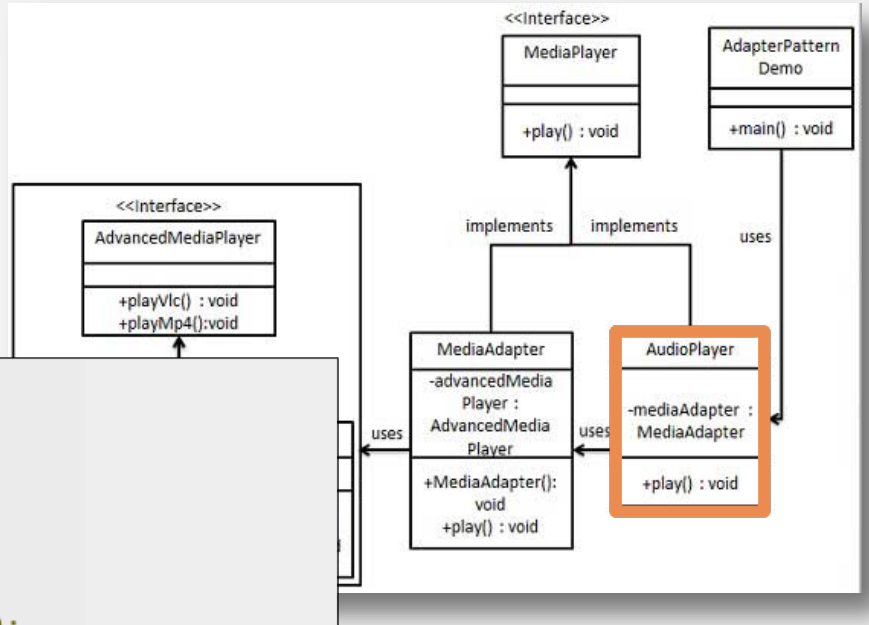


Adapter
Structural Patterns




```
public class AdapterPatternDemo {
   public static void main(String[] args) {
      AudioPlayer audioPlayer = new AudioPlayer();

      audioPlayer.play("mp3", "beyond the horizon.mp3");
      audioPlayer.play("mp4", "alone.mp4");
      audioPlayer.play("vlc", "far far away.vlc");
      audioPlayer.play("avi", "mind me.avi");
   }
}
```

Adapter
Structural Patterns

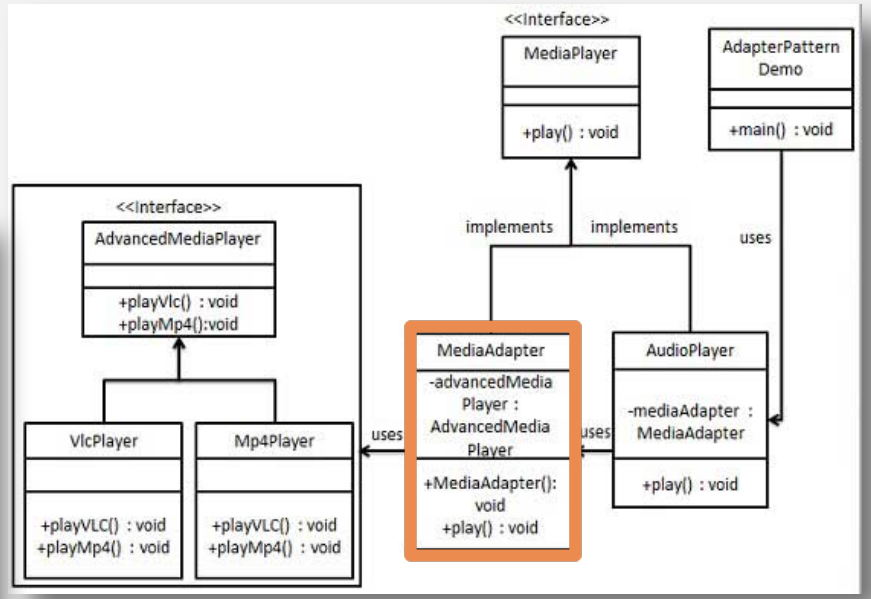



```java
public class AudioPlayer implements MediaPlayer {
   MediaAdapter mediaAdapter;

   @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");
      }
   }
}
```

MedTech

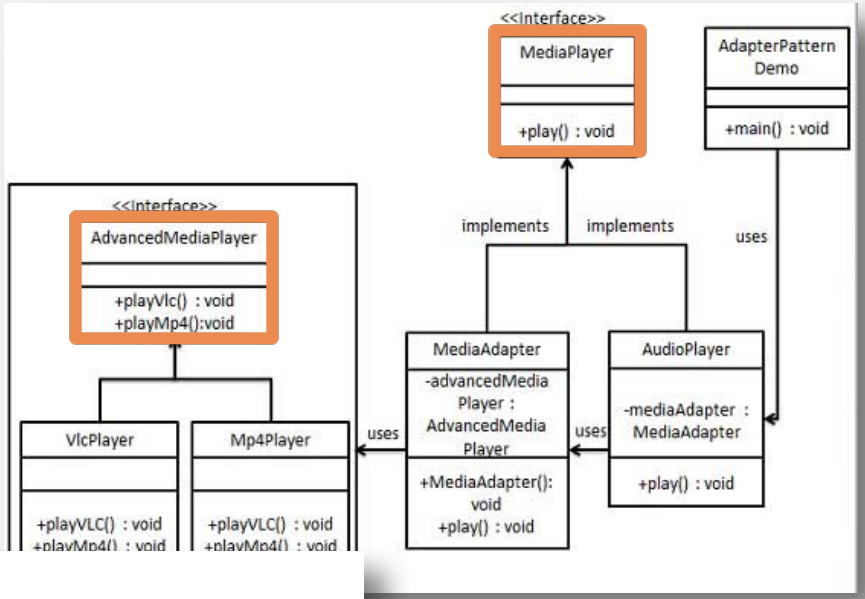
Adapter
Structural Patterns

```
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);
      }
   }
}
```



MedTech

Adapter
Structural Patterns



MediaPlayer.java

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

AdvancedMediaPlayer.java

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

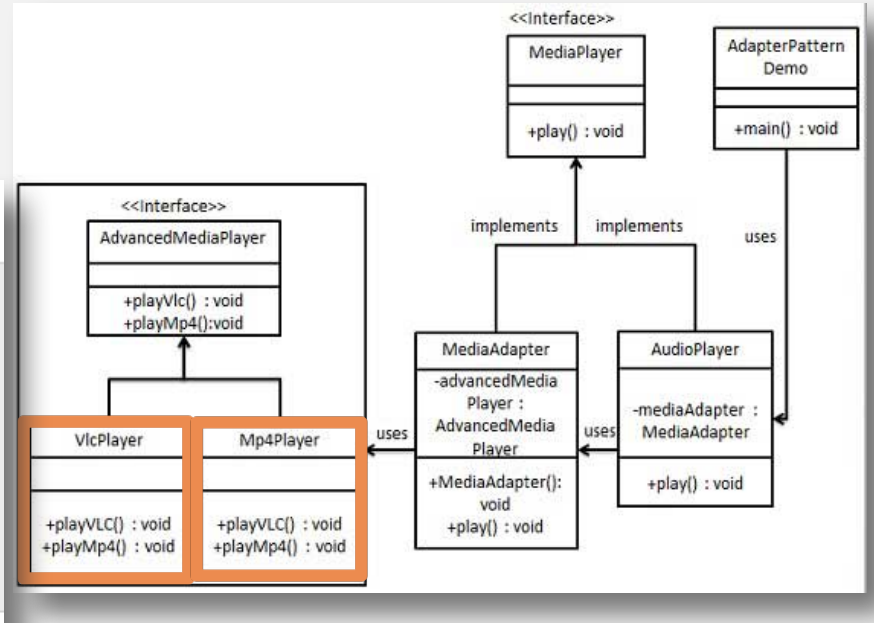
Adapter
Structural Patterns

VlcPlayer.java

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

   @Override
   public void playMp4(String fileName) {
      //do nothing
   }
}
```

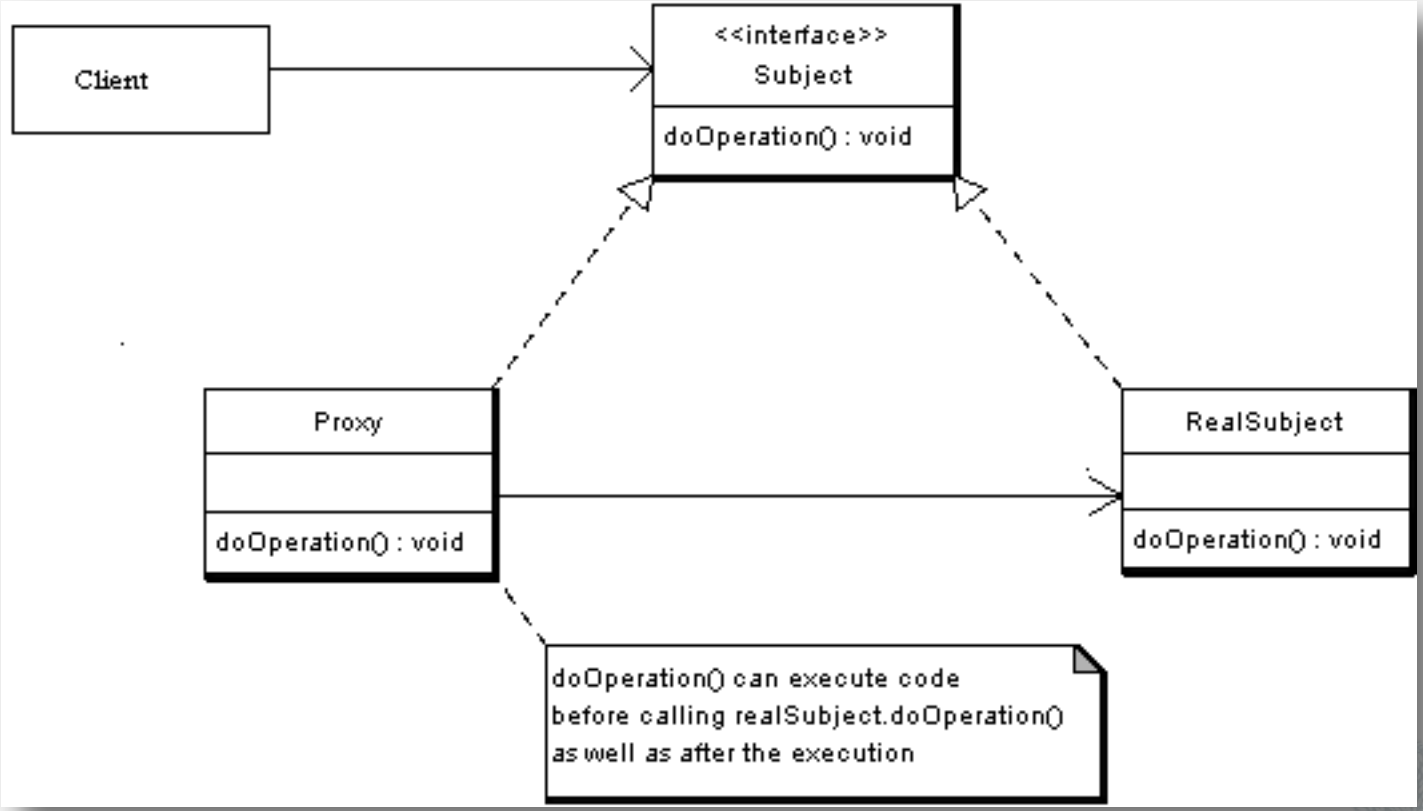
Mp4Player.java

```
public class Mp4Player implements AdvancedMediaPlayer{

   @Override
   public void playVlc(String fileName) {
      //do nothing
   }

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



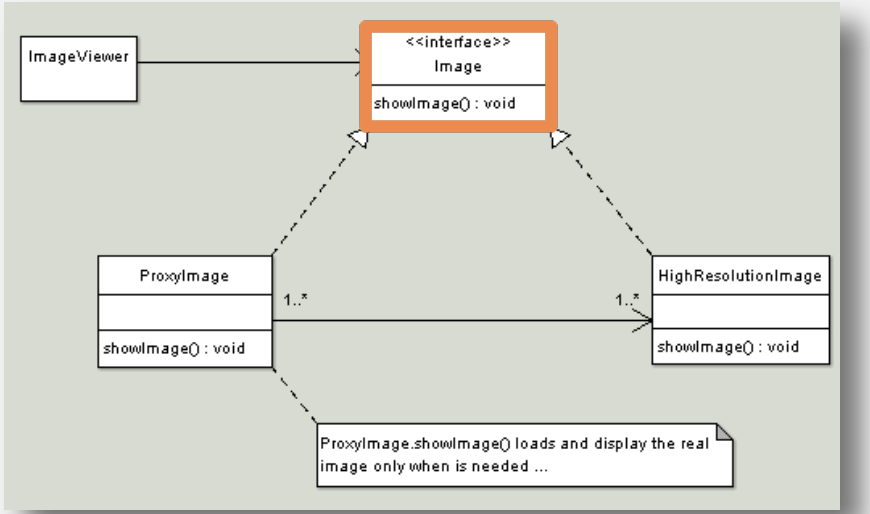
Proxy
Structural Patterns

- Provide a « Placeholder » for an object to control references to it



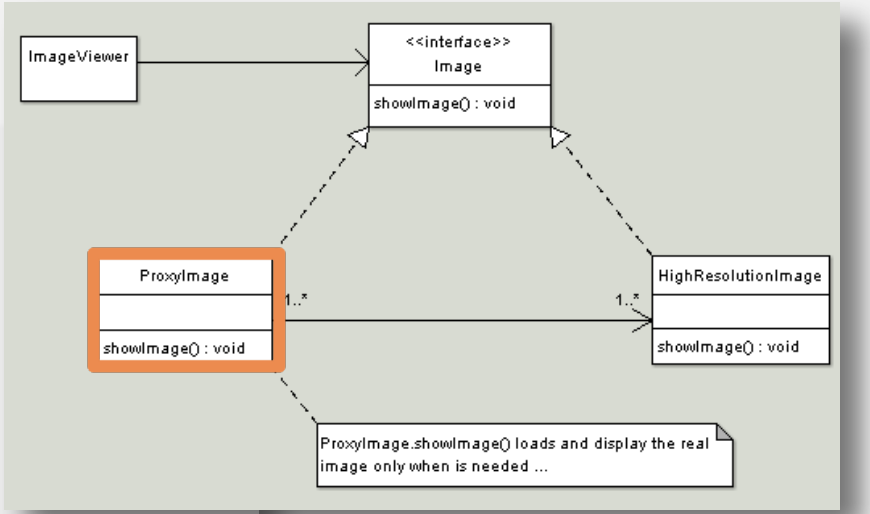
Proxy
Structural Patterns

```
package proxy;

/**
 * Subject Interface
 */
public interface Image {

        public void showImage();

}
```



MedTech

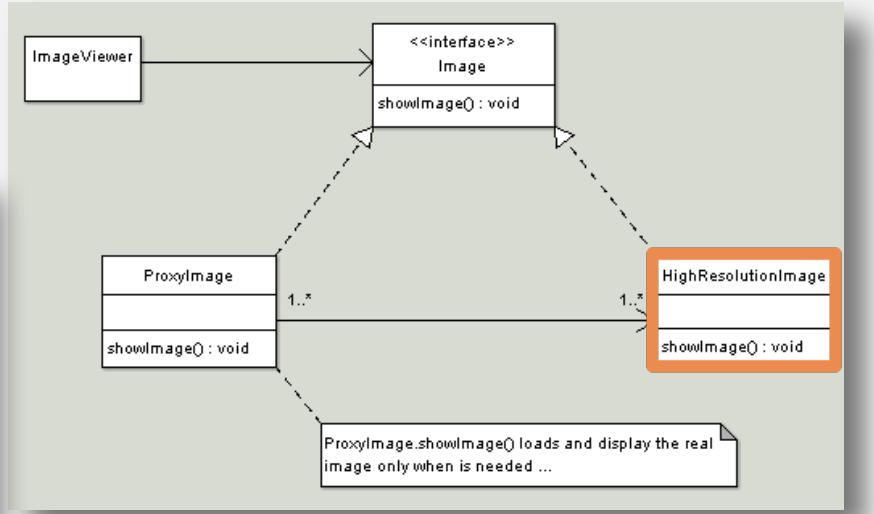
Proxy
Structural Patterns



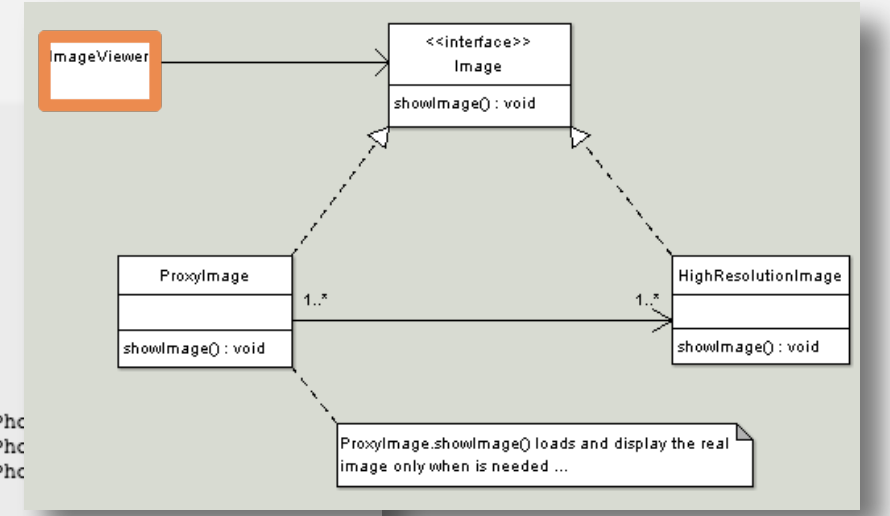

```
package proxy;

/**
 * Proxy
 */
public class ImageProxy implements Image {

        /**
         * Private Proxy data
         */
        private String imageFilePath;

        /**
         * Reference to RealSubject
         */
        private Image proxifiedImage;


        public ImageProxy(String imageFilePath) {
                this.imageFilePath= imageFilePath;
        }

        @Override
        public void showImage() {

                // create the Image Object only when the image is required to be shown

                proxifiedImage = new HighResolutionImage(imageFilePath);

                // now call showImage on realSubject
                proxifiedImage.showImage();

        }

}
```

Proxy
Structural Patterns

```
package proxy;

/**
 * RealSubject
 */
public class HighResolutionImage implements Image {

        public HighResolutionImage(String imageFilePath) {

                loadImage(imageFilePath);
        }

        private void loadImage(String imageFilePath) {

                // load Image from disk into memory
                // this is heavy and costly operation
        }

        @Override
        public void showImage() {

                // Actual Image rendering logic

        }

}
```



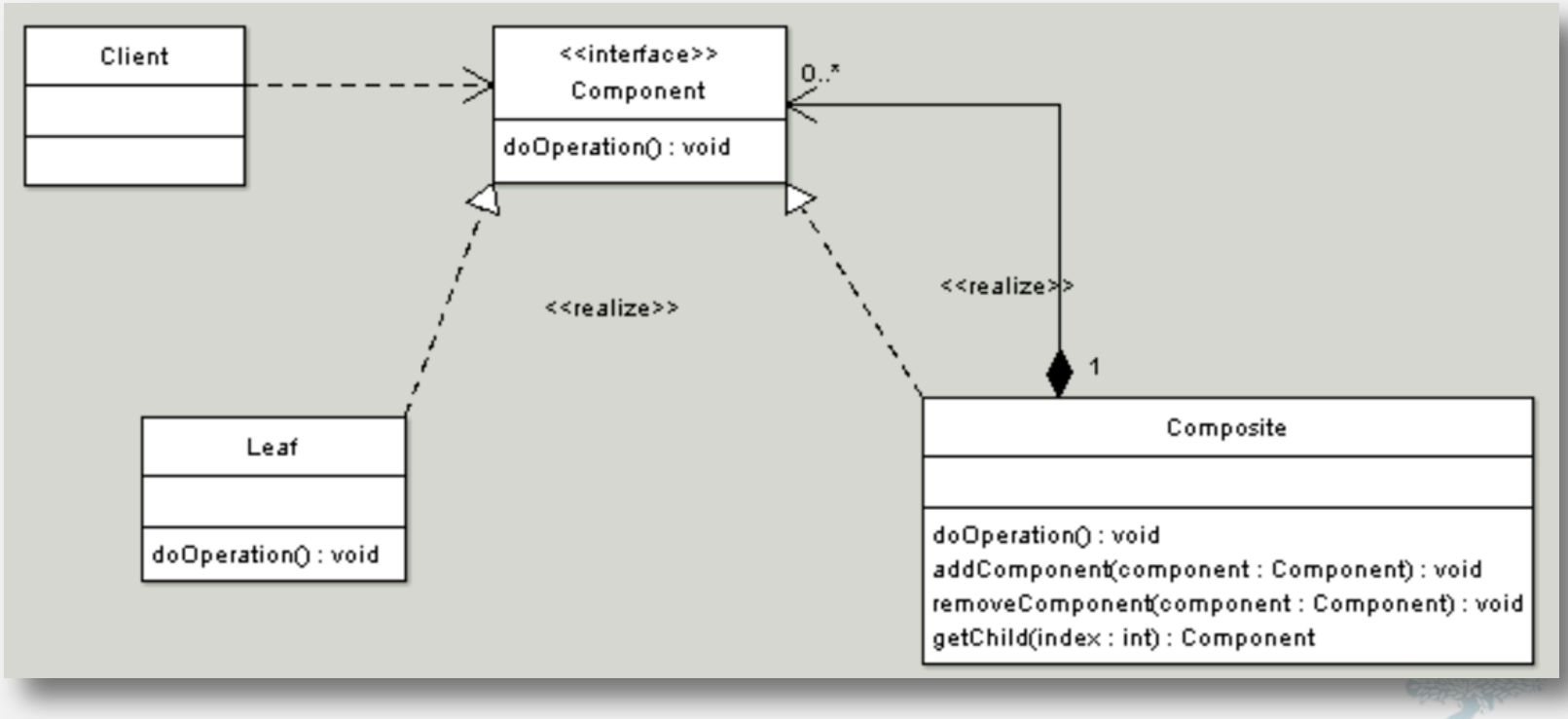
Proxy
Structural Patterns



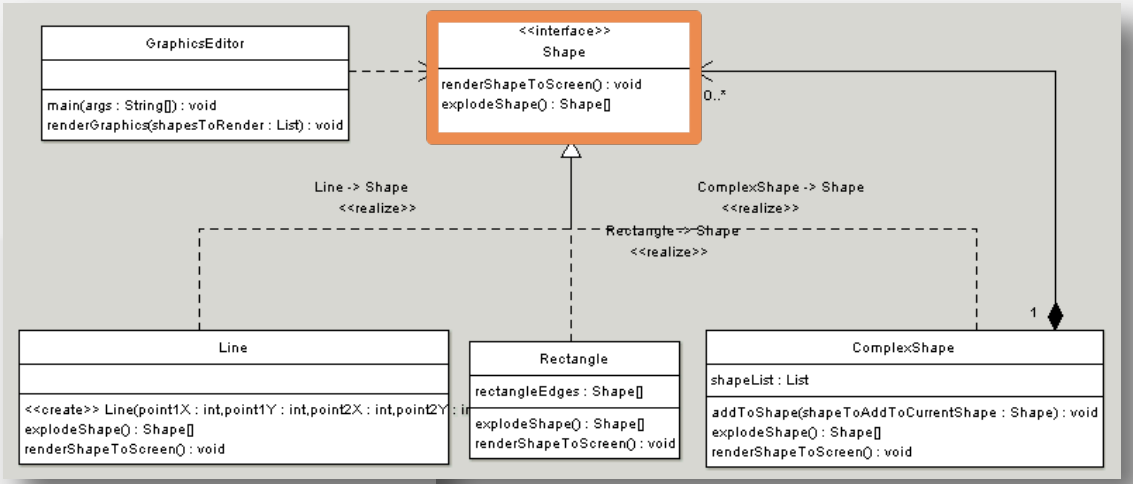

```
package proxy;

/**
 * Image Viewer program
 */
public class ImageViewer {


        public static void main(String[] args) {

        // assuming that the user selects a folder that has 3 images
        //create the 3 images
        Image highResolutionImage1 = new ImageProxy("sample/veryHighResPho
        Image highResolutionImage2 = new ImageProxy("sample/veryHighResPho
        Image highResolutionImage3 = new ImageProxy("sample/veryHighResPho

        // assume that the user clicks on Image one item in a list
        // this would cause the program to call showImage() for that image only
        // note that in this case only image one was loaded into memory
        highResolutionImage1.showImage();

        // consider using the high resolution image object directly
        Image highResolutionImageNoProxy1 = new HighResolutionImage("sample/veryHighResPhoto1.jpeg");
        Image highResolutionImageNoProxy2 = new HighResolutionImage("sample/veryHighResPhoto2.jpeg");
        Image highResolutionImageBoProxy3 = new HighResolutionImage("sample/veryHighResPhoto3.jpeg");


        // assume that the user selects image two item from images list
        highResolutionImageNoProxy2.showImage();

        // note that in this case all images have been loaded into memory
        // and not all have been actually displayed
        // this is a waste of memory resources

        }

}
```

Composite
Structural Patterns

- Compose objects into tree structures to represent part-whole hierarchies.
- Composite lets clients treat individual objects and compositions of objects uniformly.



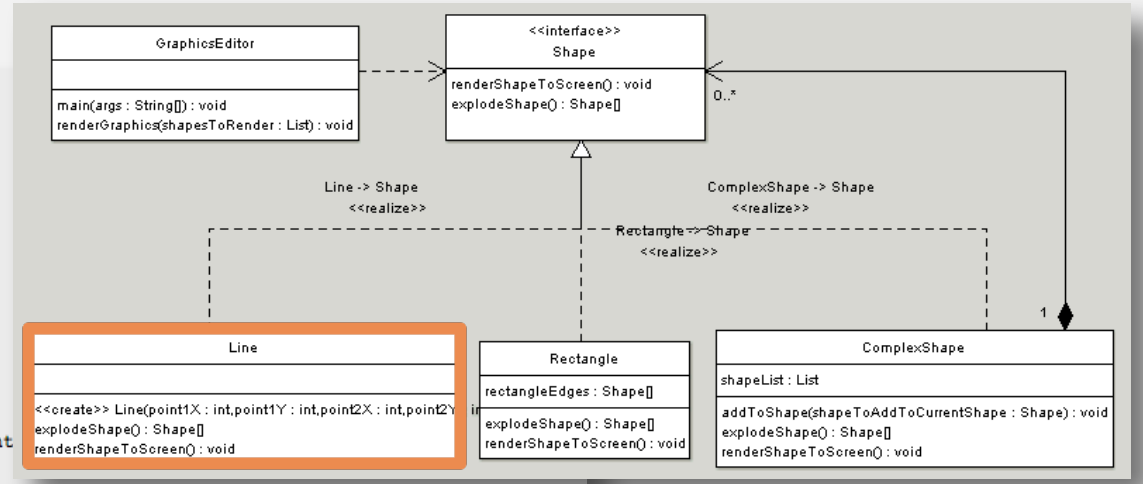
Composite
Structural Patterns




```
package composite;

/**
 *
 * Shape is the Component interface
 *
 */
public interface Shape {

	/**
	 * Draw shape on screen
	 *
	 * Method that must be implemented by Basic as well as
	 * complex shapes
	 */
	public void renderShapeToScreen();

	/**
	 * Making a complex shape explode results in getting a list of the
	 * shapes forming this shape
	 *
	 *  For example if a rectangle explodes it results in 4 line objects
	 *
	 * Making a simple shape explode results in returning the shape itself
	 */
	public Shape[] explodeShape();

}
```

Composite
Structural Patterns

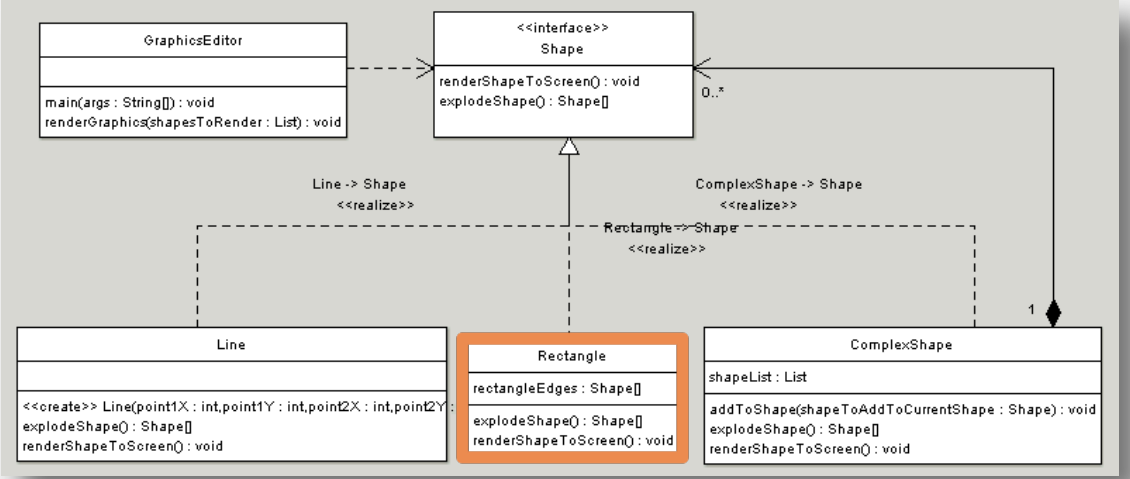
```
package composite;

/**
 *
 * Line is a basic shape that does not support adding shapes
 */
public class Line implements Shape {


	/**
	 * Create a line between point1 and point2
	 * @param point1X
	 * @param point1Y
	 * @param point2X
	 * @param point2Y
	 */
	public Line(int point1X, int point1Y, int point2X, int point

	}

	@Override
	public Shape[] explodeShape() {


		// making a simple shape explode would return only the shape itself, there are no parts of this shape

		Shape[] shapeParts = {this};

		return shapeParts;

	}

	/**
	 * this method must be implemented in this simple shape
	 */
	public void renderShapeToScreen() {


		// logic to render this shape to screen

	}

}

}
```



MedTech

Composite
Structural Patterns

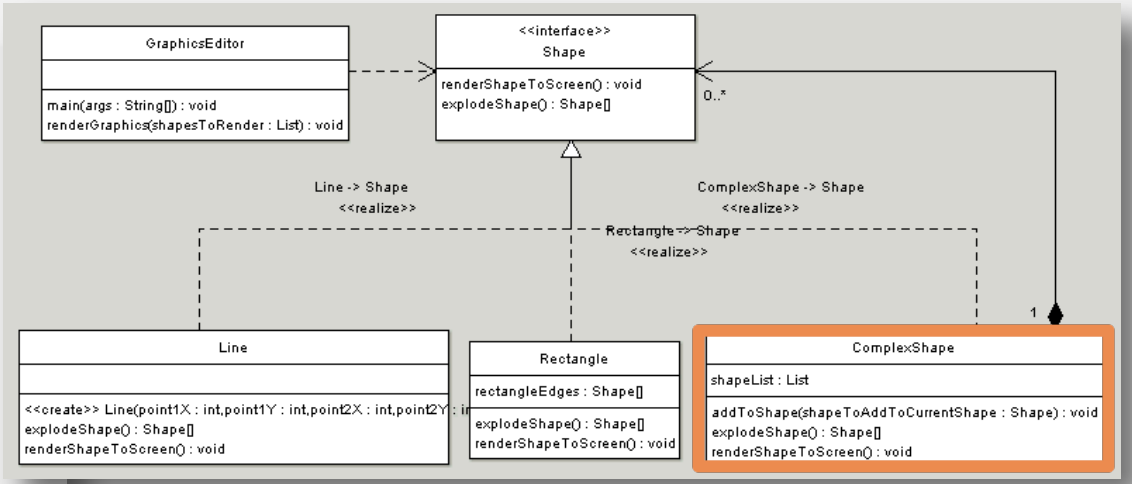
```
package composite;


/**
 * Rectangle is a composite
 *
 *Complex Shape
 */
public class Rectangle implements Shape{

        // List of shapes forming the rectangle
        // rectangle is centered around origin
        Shape[] rectangleEdges = {new Line(-1,-1,1,-1),
               new Line(-1,1,1,1),new Line(-1,-1,-1,1)
               ,new Line(1,-1,1,1)};

        @Override
        public Shape[] explodeShape() {

                return rectangleEdges;

        }
         public void renderShapeToScreen() {


                 for(Shape s : rectangleEdges){

                    // delegate to child objects
                         s.renderShapeToScreen();

                 }

         }

}
```



MedTech

Composite
Structural Patterns

```
package composite;

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

/**
 * Composite object supporting creation of more complex shapes
 *  Complex Shape
 */
public class ComplexShape implements Shape {


        /**
         * List of shapes
         */
        List shapeList = new ArrayList();

        /**
         *
         */
        public void addToShape(Shape shapeToAddToCurrentShape) {

                shapeList.add(shapeToAddToCurrentShape);

        }


        public Shape[] explodeShape() {

                return (Shape[]) shapeList.toArray();
        }

        /**
         * this method is implemented directly in basic shapes
         * in complex shapes this method is handled with delegation
         */
        public void renderShapeToScreen() {

                for(Shape s: shapeList){

                        // use delegation to handle this method
                        s.renderShapeToScreen();

                }
        }
}
```



Design Patterns

BEHAVIORAL PATTERNS

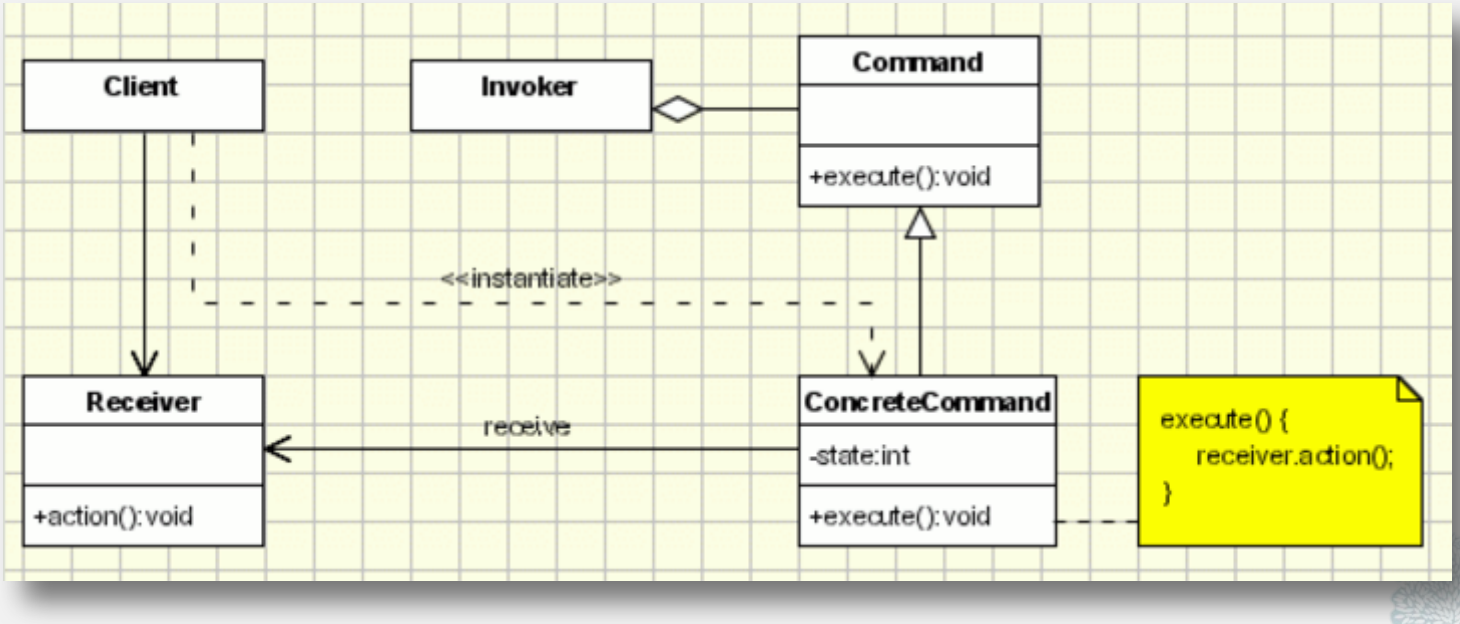
Definition
Behavioral Patterns

- Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects
  - Behavioral class patterns use inheritance to distribute behavior between classes.
  - Behavioral object patterns use composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.

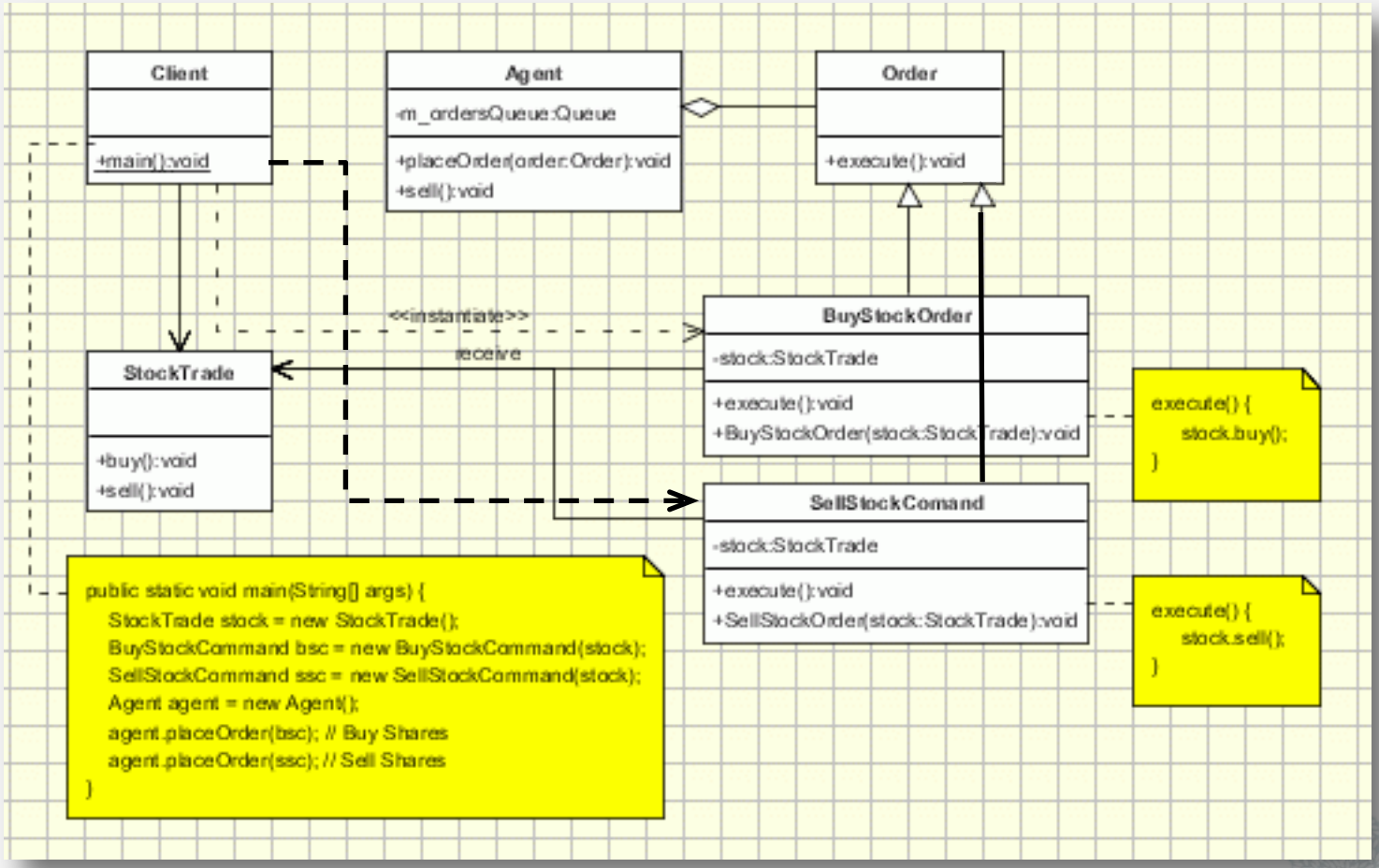
- Examples:
  - Command
  - Iterator
  - Observer
  - Strategy

Command
Behavioral Patterns

- Encapsulates a request in an object
- Allows the parameterization of clients with different requests
- Allows saving the request in a queue



Command
Behavioral Patterns



Command

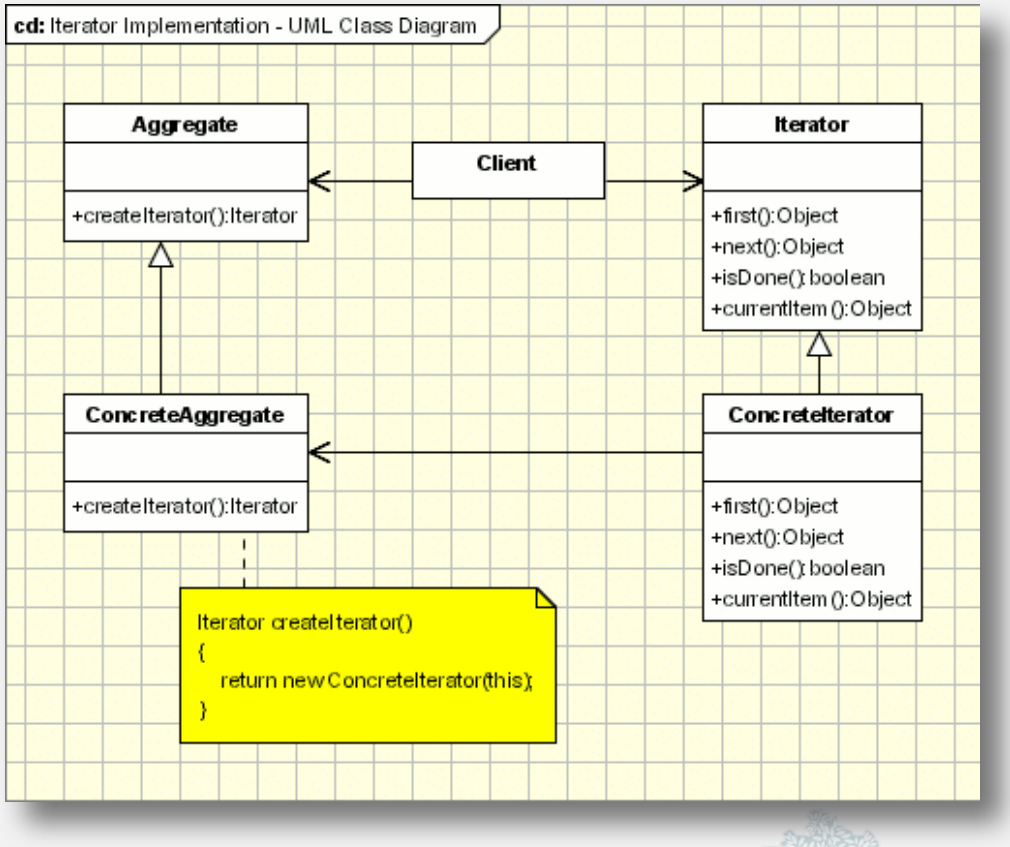
Behavioral Patterns

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

// Receiver class.
class StockTrade {
    public void buy() {
        System.out.println("You want to buy stocks");
    }
    public void sell() {
        System.out.println("You want to sell stocks ");
    }
}

// Invoker.
class Agent {
    private List m_ordersQueue = new ArrayList();

    public Agent() {
    }

    void placeOrder(Order order) {
        ordersQueue.addLast(order);
        order.execute(ordersQueue.getFirstAndRemove());
    }
}

//ConcreteCommand Class.
class BuyStockOrder implements Order {
    private StockTrade stock;
    public BuyStockOrder ( StockTrade st) {
        stock = st;
    }
    public void execute( ) {
        stock . buy( );
    }
}
```

```
//ConcreteCommand Class.
class SellStockOrder implements Order {
    private StockTrade stock;
    public SellStockOrder ( StockTrade st) {
        stock = st;
    }
    public void execute( ) {
        stock . sell( );
    }
}

// Client
public class Client {
    public static void main(String[] args) {
        StockTrade stock = new StockTrade();
        BuyStockOrder bsc = new BuyStockOrder (stock);
        SellStockOrder ssc = new SellStockOrder (stock);
        Agent agent = new Agent();

        agent.placeOrder(bsc); // Buy Shares
        agent.placeOrder(ssc); // Sell Shares
    }
}
```

Iterator
Behavioral Patterns

- The iterator pattern allows us to:
  - Access contents of a collection without exposing its internal structure.
  - Support multiple simultaneous traversals of a collection.
  - Provide a uniform interface for traversing different collections.



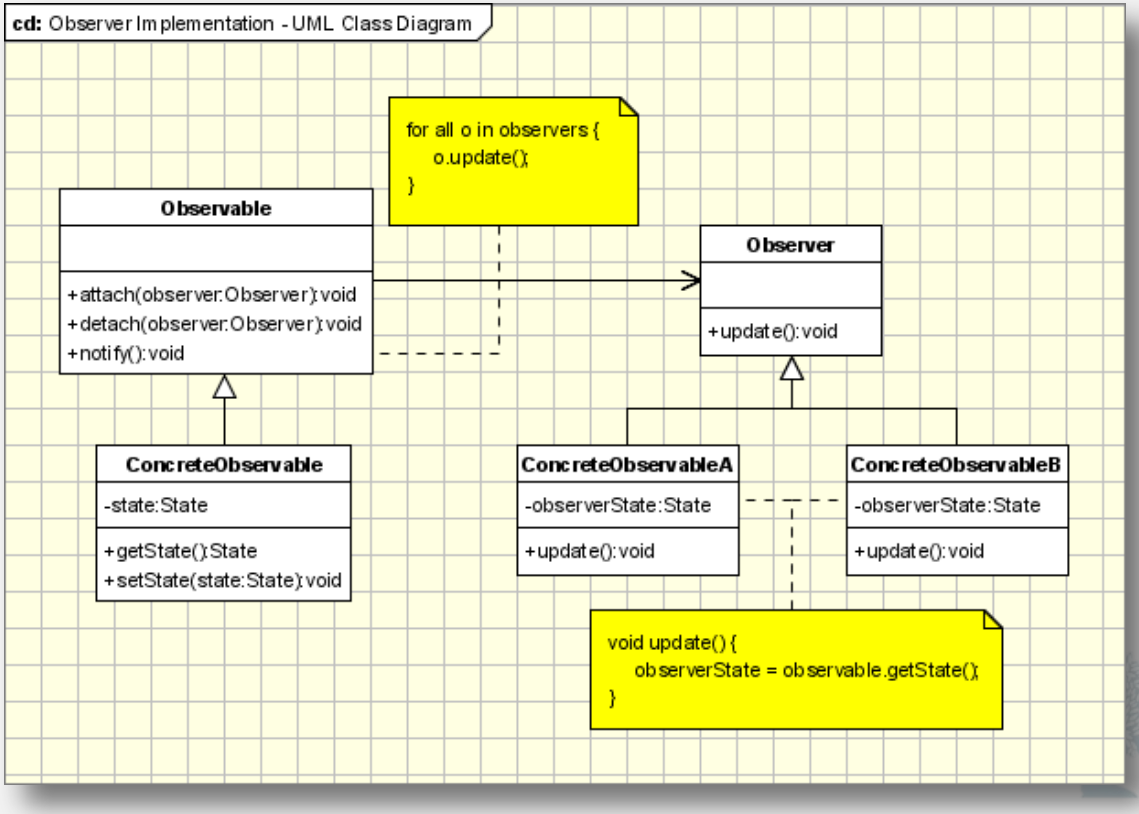
Iterator
Behavioral Patterns

```
class BooksCollection implements IContainer
{
        private String m_titles[] = {"Design Patterns","1","2","3","4"};

        public IIterator createIterator()
        {
                BookIterator result = new BookIterator();
                return result;
        }


        private class BookIterator implements IIterator
        {
                private int m_position;

                public boolean hasNext()
                {
                        if (m_position < m_titles.length)
                                return true;
                        else
                                return false;
                }
                public Object next()
                {
                        if (this.hasNext())
                                return m_titles[m_position++];
                        else
                                return null;
                }
        }
}
```

```
interface IIterator
{
        public boolean hasNext();
        public Object next();
}

interface IContainer
{
        public IIterator createIterator();
}
```

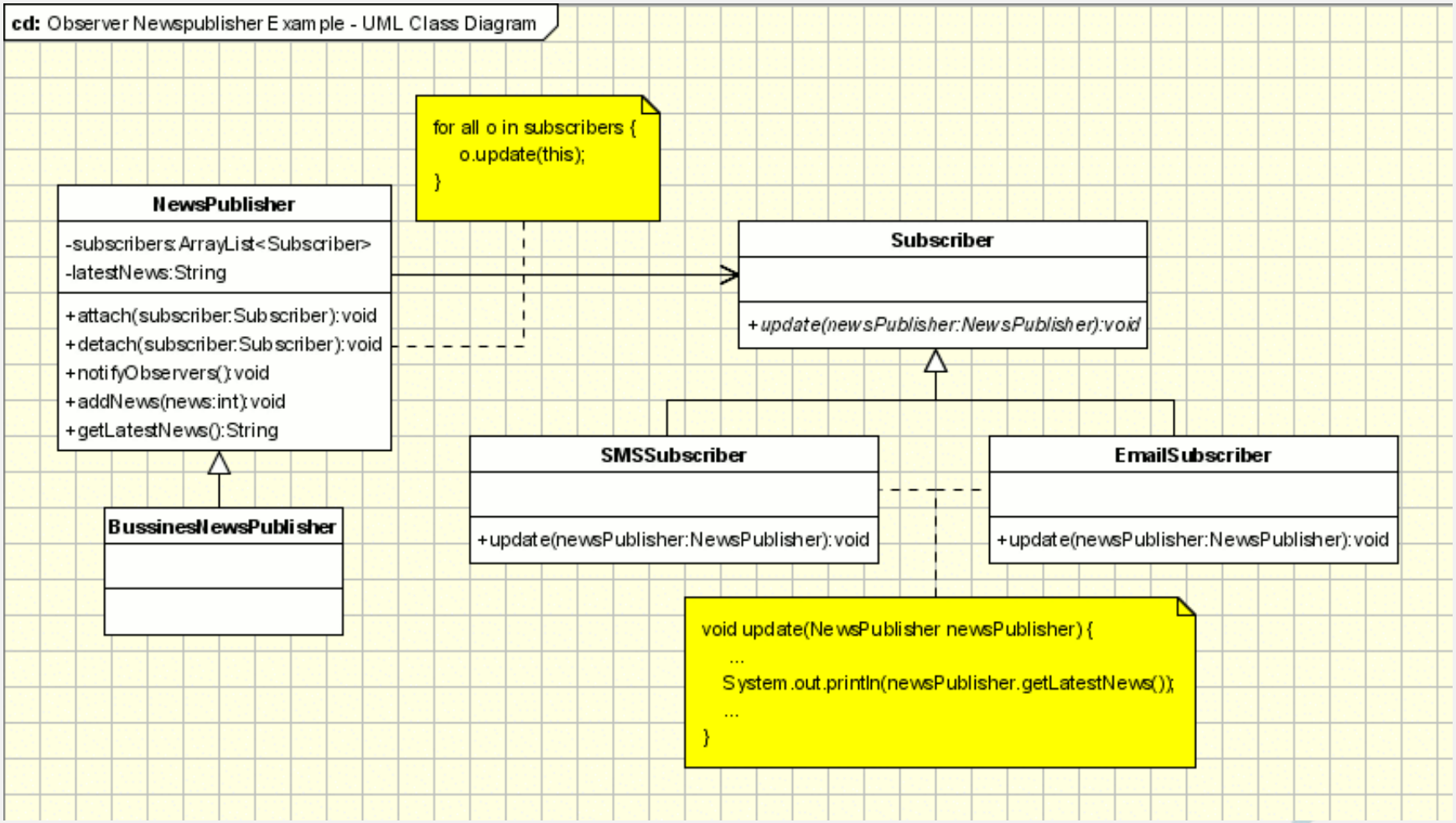
Observer
Behavioral Patterns

- Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



Observer

Behavioral Patterns



References

- Object Oriented Design, http://www.oodesign.com/, consulted november 2016
- *Gang of Four (GoF)OO Design Patterns*, (*Course*) WATERLOO CHERITON SCHOOL OF COMPUTER SCIENCE, 2011
- *Design Patterns*, (*Course*) Faculty of Science, Engineering and Technology, 2007

- Textbooks
  - E. Gamma & al. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994
  - B. Christiansson & al. *GoF Design Patterns -with examples using Java and UML2*, 2008