Faculty of Computers, Informatics and Microelectronics

Technical University of Moldova

IPP

Lab#2

*Author:*

Mereuta Alex

*Supervisor:*

Anastasia SERSUN

Chisinau 2017

1. Objective

Structural Patterns

Implementing 5 chosen Structural Patterns:

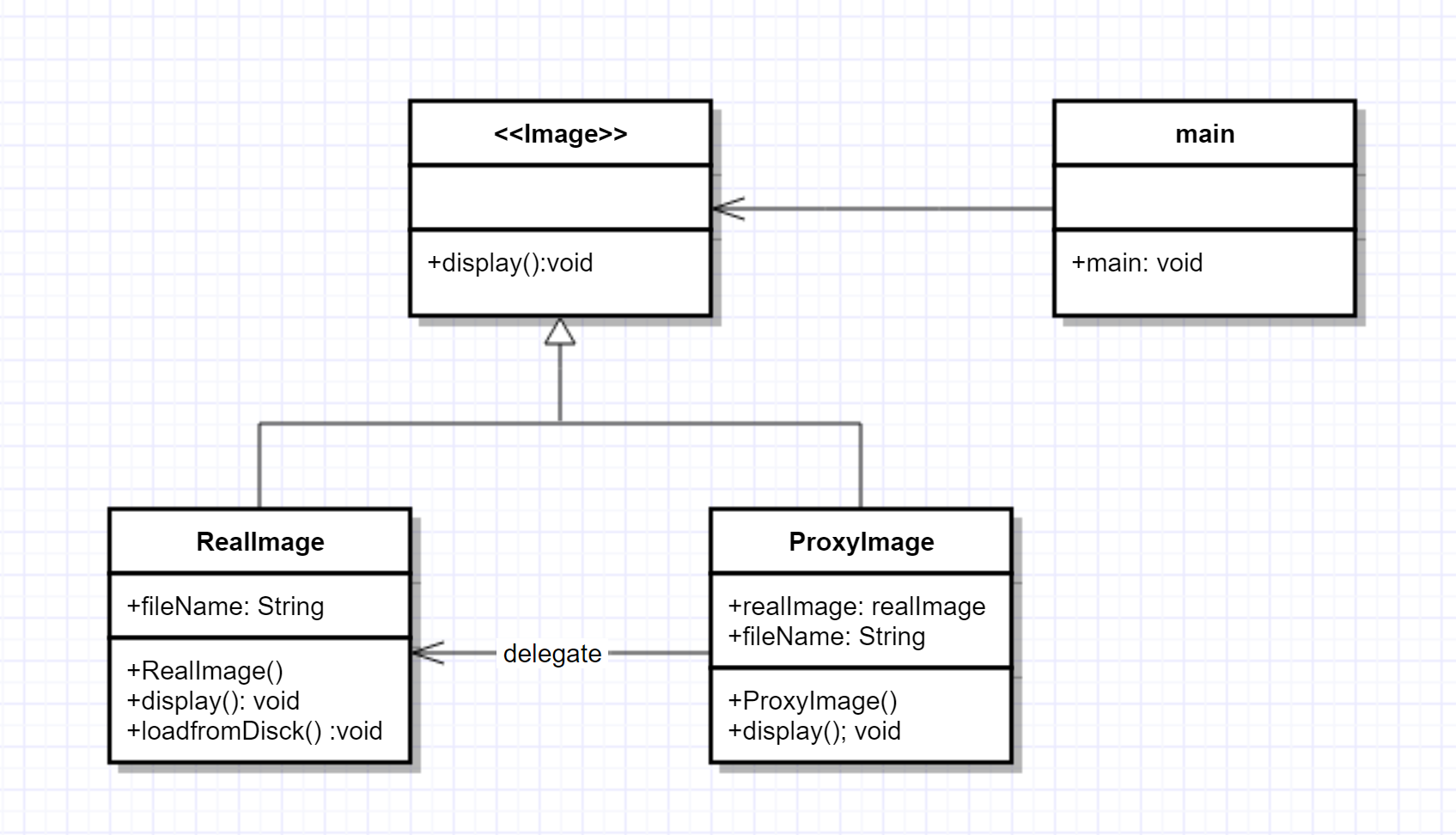
Proxy, Composite, Decorator, Flyweight, Adaptor

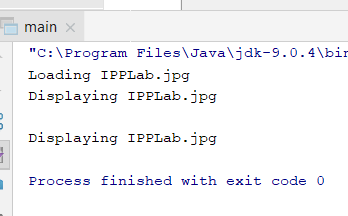
Proxy Pattern

* Provide a surrogate or placeholder for another object to control access to it.
* Use an extra level of indirection to support distributed, controlled, or intelligent access.
* Add a wrapper and delegation to protect the real component from undue complexity.

You need to support resource-hungry objects, and you do not want to instantiate such objects unless and until they are actually requested by the client.

Here is the Image interface and concrete classes implementing it, RealImage and ProxyImage. ProxyImage is a class to reduce time and memory of RealImage object loading. Main uses ProxyImage to get an Image object.



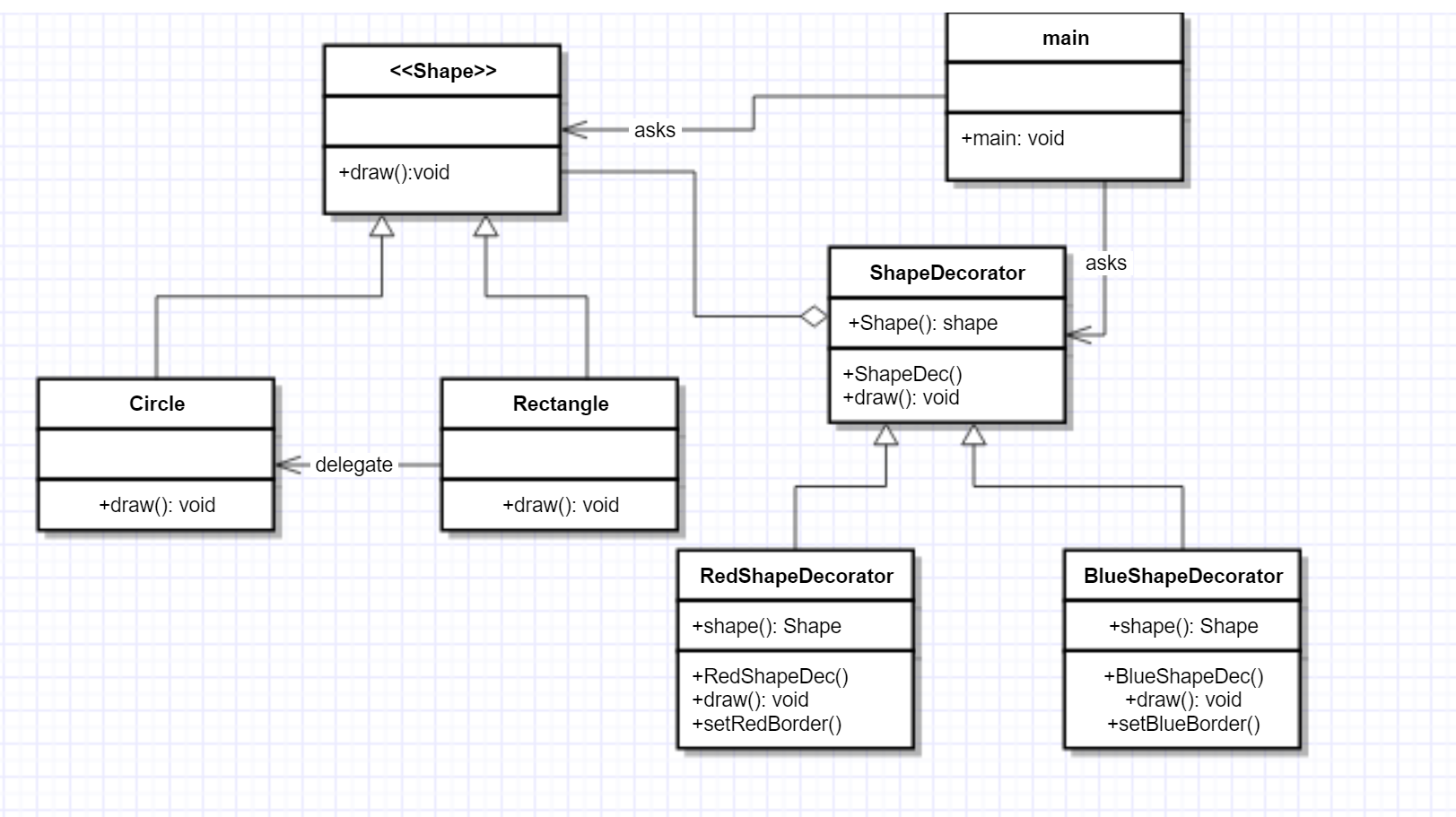


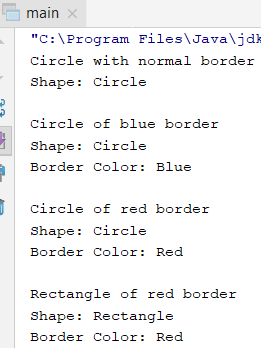
Decorator Pattern

* Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.
* Client-specified embellishment of a core object by recursively wrapping it.
* Wrapping a gift, putting it in a box, and wrapping the box.

You want to add behavior or state to individual objects at run-time. Inheritance is not feasible because it is static and applies to an entire class.

So lets say the main thing in which the client is interested is the Shape interface and he may or may be not interested in shapedecorator, but still this method is present and is not in any way changing the program. Because you can create simple circles and rectangles, but also you can ‘decorate’ them, giving them colors.



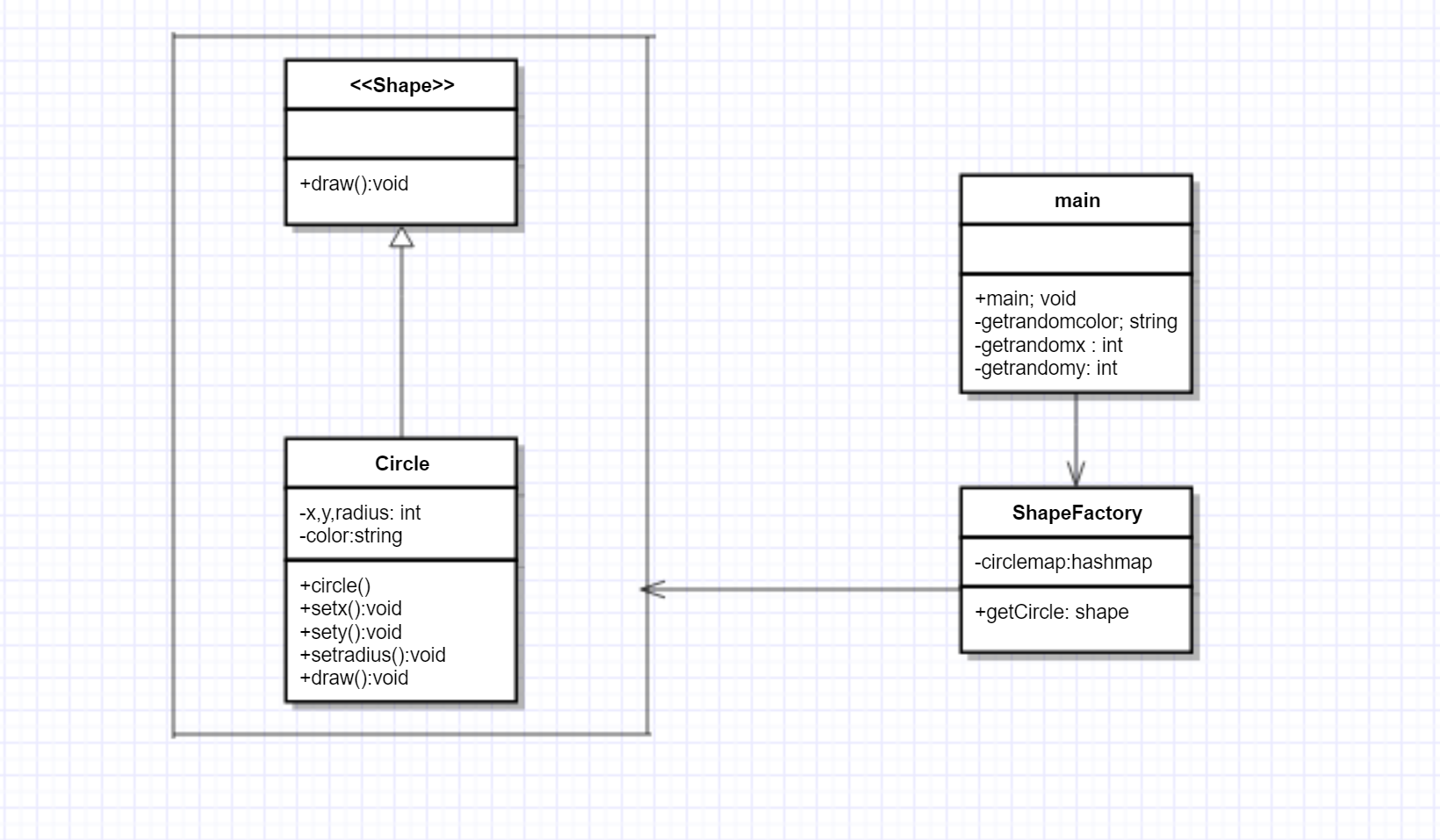


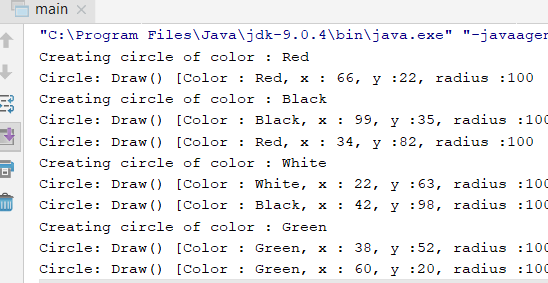
Flyweight Pattern

* Use sharing to support large numbers of fine-grained objects efficiently.
* The Motif GUI strategy of replacing heavy-weight widgets with light-weight gadgets.

Flyweight pattern is primarily used to reduce the number of objects created and to decrease memory footprint and increase performance. This type of design pattern comes under structural pattern as this pattern provides ways to decrease object count thus improving the object structure of application.

This looks like a simply factory, but here ShapeFactory has a hashmap which cashes objects, so it basically stores these objects. The client asks for a given object the ShapeFactory.





Composite Pattern

* Compose objects into tree structures to represent whole-part hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
* Recursive composition
* "Directories contain entries, each of which could be a directory."
* 1-to-many "has a" up the "is a" hierarchy

Composite design pattern allows you to have a tree structure and ask each node in the tree structure to perform a task. You can take real life example of a organization. It have general managers and under general managers, there can be managers and under managers there can be developers. Now you can set a tree structure and ask each node to perform common operation like getSalary().

Component

* declares interface for objects in composition.
* implements deafault behaviour for the interface common to all classes as appropriate.
* declares an interface for accessing and managing its child components.

Leaf

* represents leaf objects in the composition. A leaf has no children.
* defines behaviour for primitive objects in the composition.

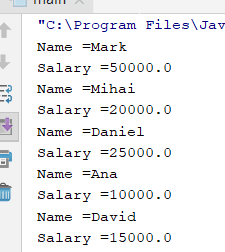
Composite

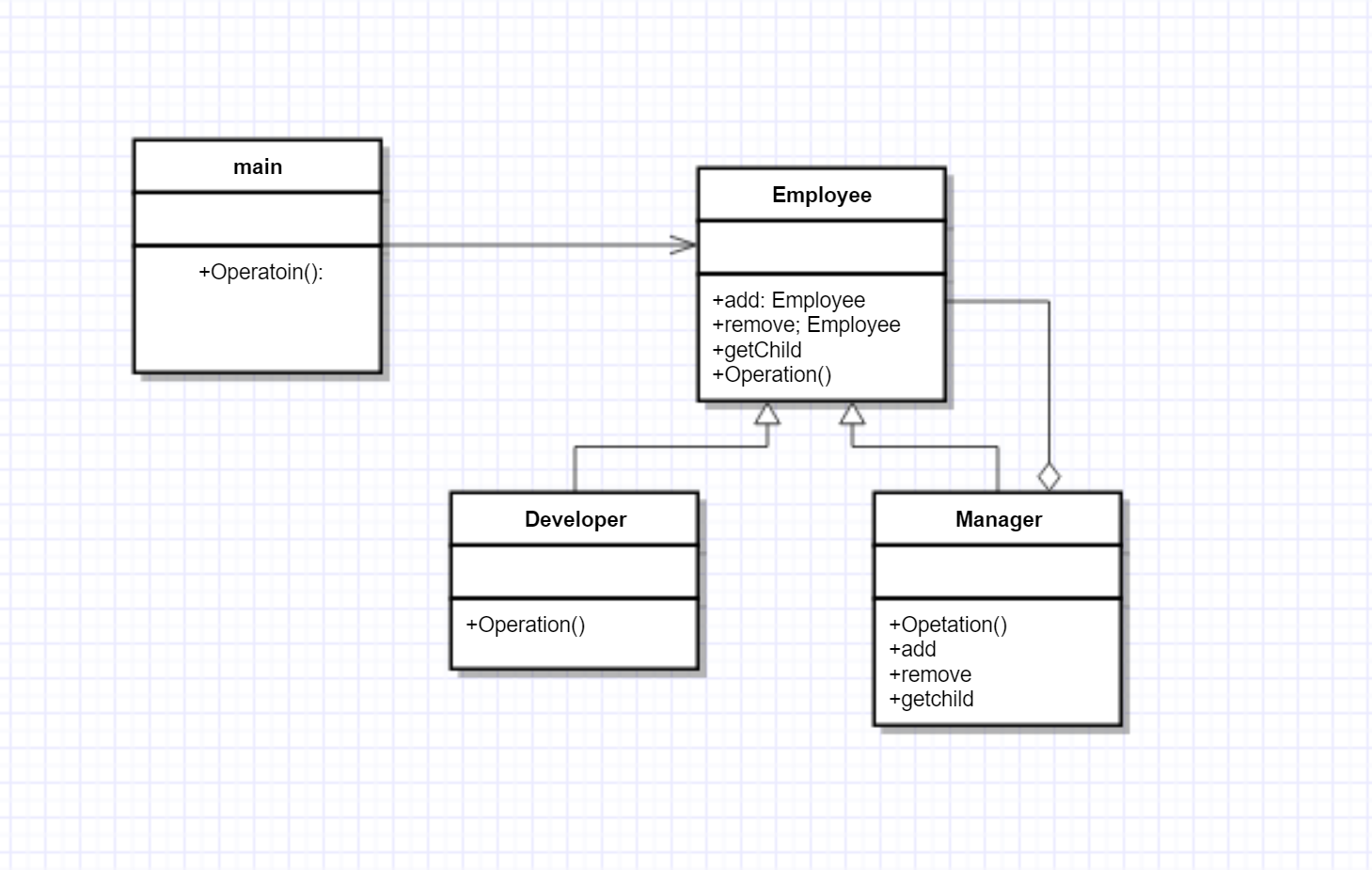
* defines behaviour for components having children.
* stores child components.
* implements child related operations in the component interface.

Client

* manipulates objects in the composition through the component interface.

Client use the component class interface to interact with objects in the composition structure.if recipient is a leaf then request is handled directly.If recipient is a composite,then it usually forwards request to its child components,possibly performing additional operations before and after forwarding





Adapter Pattern

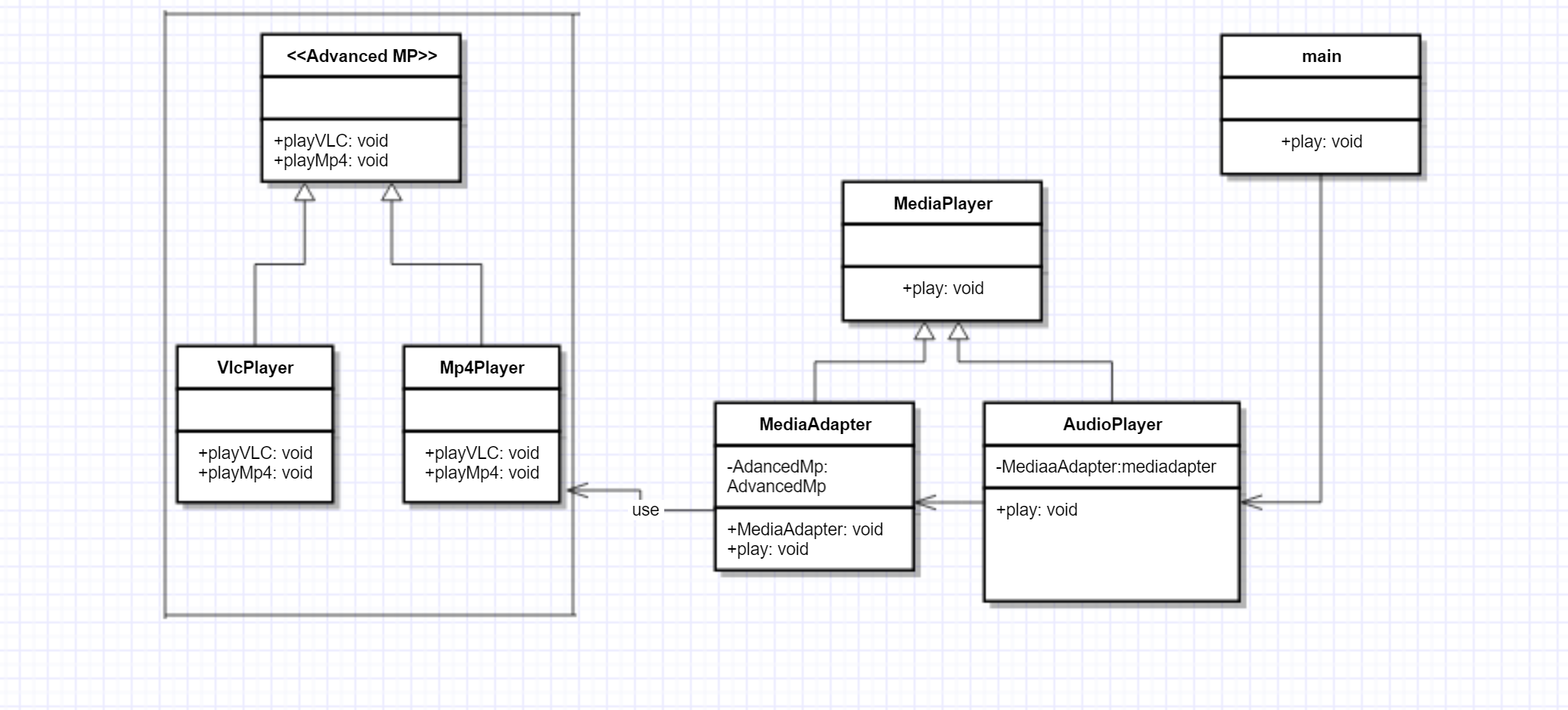
* Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
* Wrap an existing class with a new interface.
* Impedance match an old component to a new system

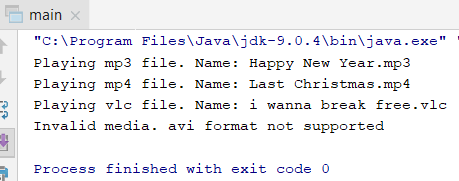
Adapter pattern works as a bridge between two incompatible interfaces. This type of design pattern comes under structural pattern as this pattern combines the capability of two independent interfaces.

This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plug in the memory card into card reader and card reader into the laptop so that memory card can be read via laptop.

Here I have a simple audio player who plays only mp3, but we also want other formats, so theres the need of an adapter between these formats.

In this case the adapter is the media adapter thing, and the adaptee is the advanced media player class.





Conclusion:

Elaborating there laboratories I learned about the structural patterns, which are kinda more difficult than the creational ones to understand. Got to see again that these patterns exist to improve the reusability and the easiness to write code. Also, doing the diagrams for each pattern, made me remember better their structure.