Object Oriented Analysis and Design with Java

UE19CS353

Dr. Sudeepa Roy Dey
Department of Computer Science and Engineering

Acknowledgements: Significant portions of the information in the slide sets presented through the course in the class, are extracted from the prescribed text books, information from the Internet. Since these are only intended for presentation for teaching within PESU, there was no explicit permission solicited. We would like to sincerely thank and acknowledge that the credit/rights remain with the original authors/creators only

UE19CS353: Object Oriented Analysis and Design with Java

OO Design Patterns & Anti-Patterns with Sample implementation in Java

Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

UE19CS353: Object Oriented Analysis and Design with Java

Unit-5

Structural Patterns – Adapter

Object Oriented Analysis and Design with Java

Agenda

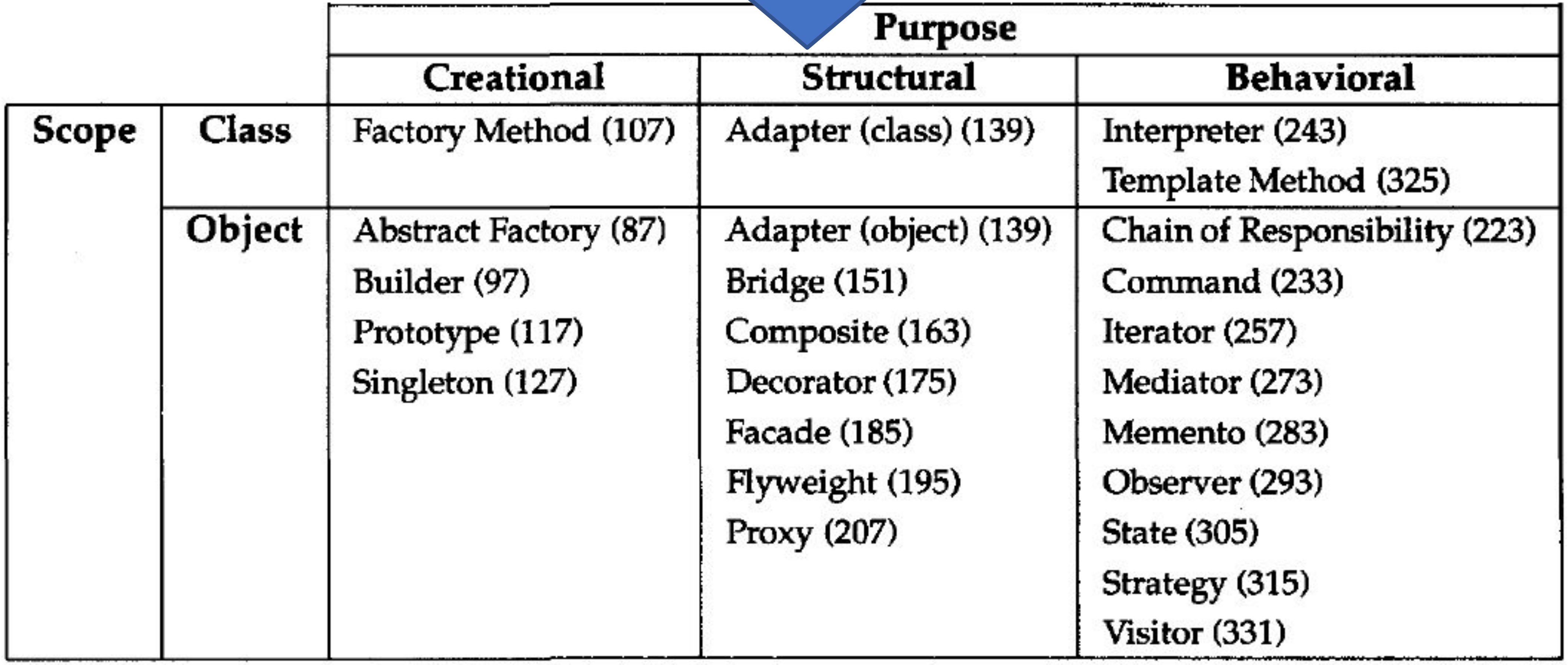
- ❑ Introduction to Structural design patterns
- ❑ Types of Structural Design Patterns.
- ❑ Adapter-definition
- ✔ Motivation
- ✔ Intent
- ✔ Implementation
- ✔ Applicability
- ✔ Structure-Consequence
- ✔ Issues

Introduction to Structural Design Pattern

- Structural design patterns are concerned with how classes and objects can be composed, to form larger structures.
- The structural design patterns simplifies the structure by identifying the relationships.
- These patterns focus on, how the classes inherit from each other and how they are composed from other classes.
- Structural *class* patterns use inheritance to compose interfaces or implementations.
- Structural design patterns explain how to assemble objects and classes into larger structures, while keeping these structures flexible and efficient.

Object Oriented Analysis and Design with Java

Types of Structural Design pattern: Scope



		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Adapter: Class, Object Structural

Motivation

- The adapter pattern is adapting between classes and objects.
- Like any adapter in the real world it is used to be an interface, a bridge between two objects. Ex: In real world we have adapters for power supplies, adapters for camera memory cards, and so on.
- Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

Advantage of Adapter Pattern

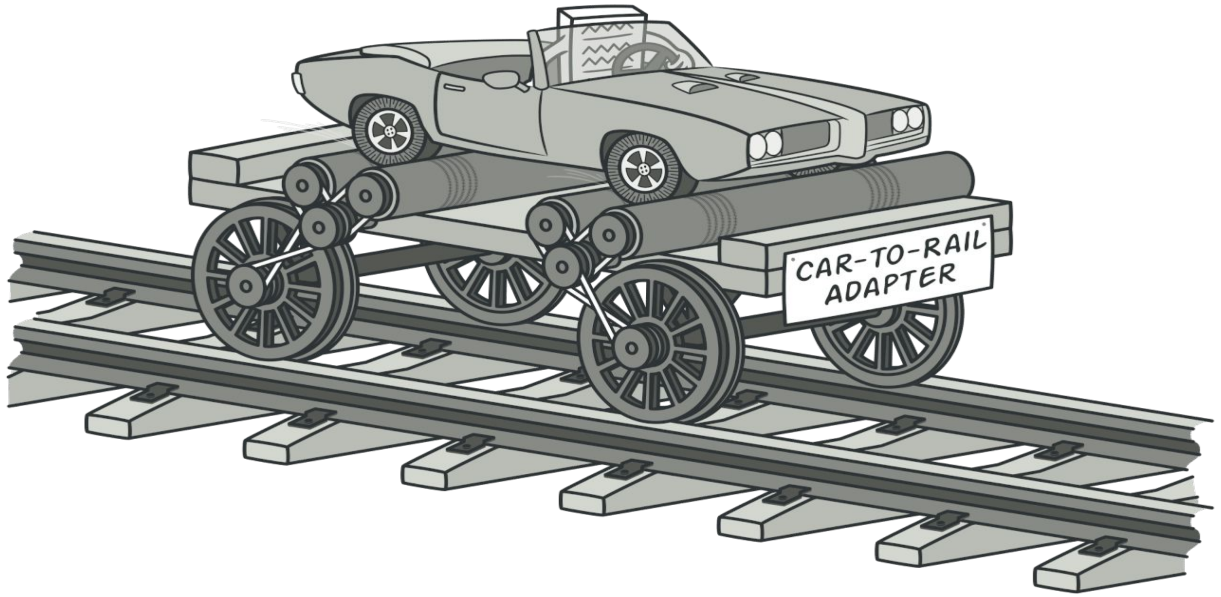
- It allows two or more previously incompatible objects to interact.
- It allows reusability of existing functionality.

Adapter: Class, Object Structural

Intent

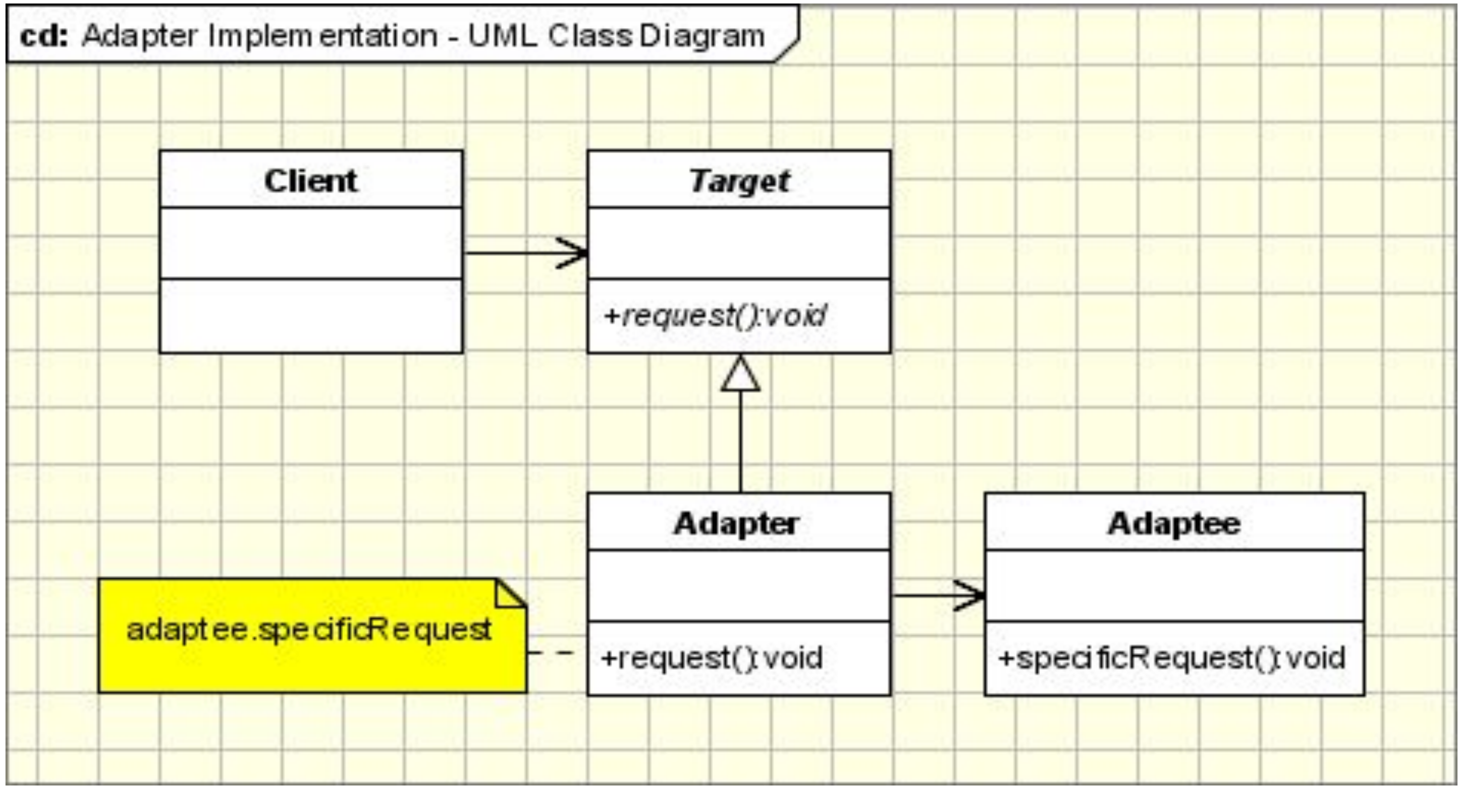
- Adapter lets classes work together that couldn' t otherwise because of incompatible interfaces.
- An Adapter Pattern says that "convert the interface of a class into another interface that a client wants". In other words, to provide the interface according to client requirement while using the services of a class with a different interface.
- The Adapter Pattern is also known as Wrapper.

Adapter is a structural design pattern that allows classes with incompatible interfaces to collaborate.



Adapter: Implementation

UML class diagram for the Adapter Pattern



The classes/objects participating in adapter pattern:

1. Target - defines the domain-specific interface that Client uses.
2. Adapter - adapts the interface Adaptee to the Target interface.
3. Adaptee - defines an existing interface that needs adapting.
4. Client - collaborates with objects conforming to the Target interface.

Adapter class: This class is a wrapper class which implements the desired target interface and modifies the specific request available from the Adaptee class.

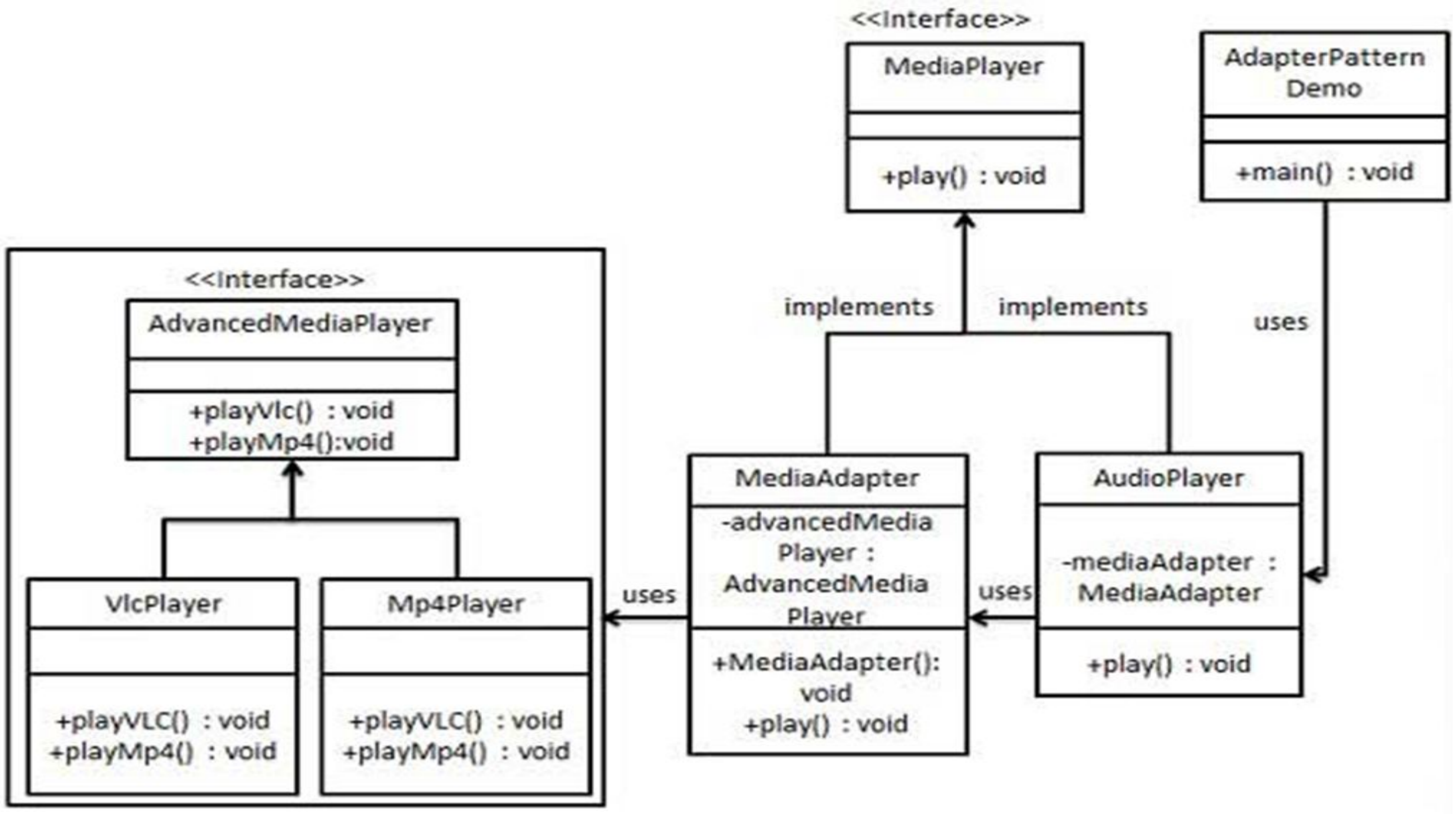
Adapter: Implementation example-1

Problem Statement: We have a MediaPlayer interface and a concrete class AudioPlayer implementing the MediaPlayer interface. AudioPlayer can play mp3 format audio files by default. We are having another interface AdvancedMediaPlayer and concrete classes implementing the AdvancedMediaPlayer interface. These classes can play vlc and mp4 format files. We want to make AudioPlayer to play other formats as well.

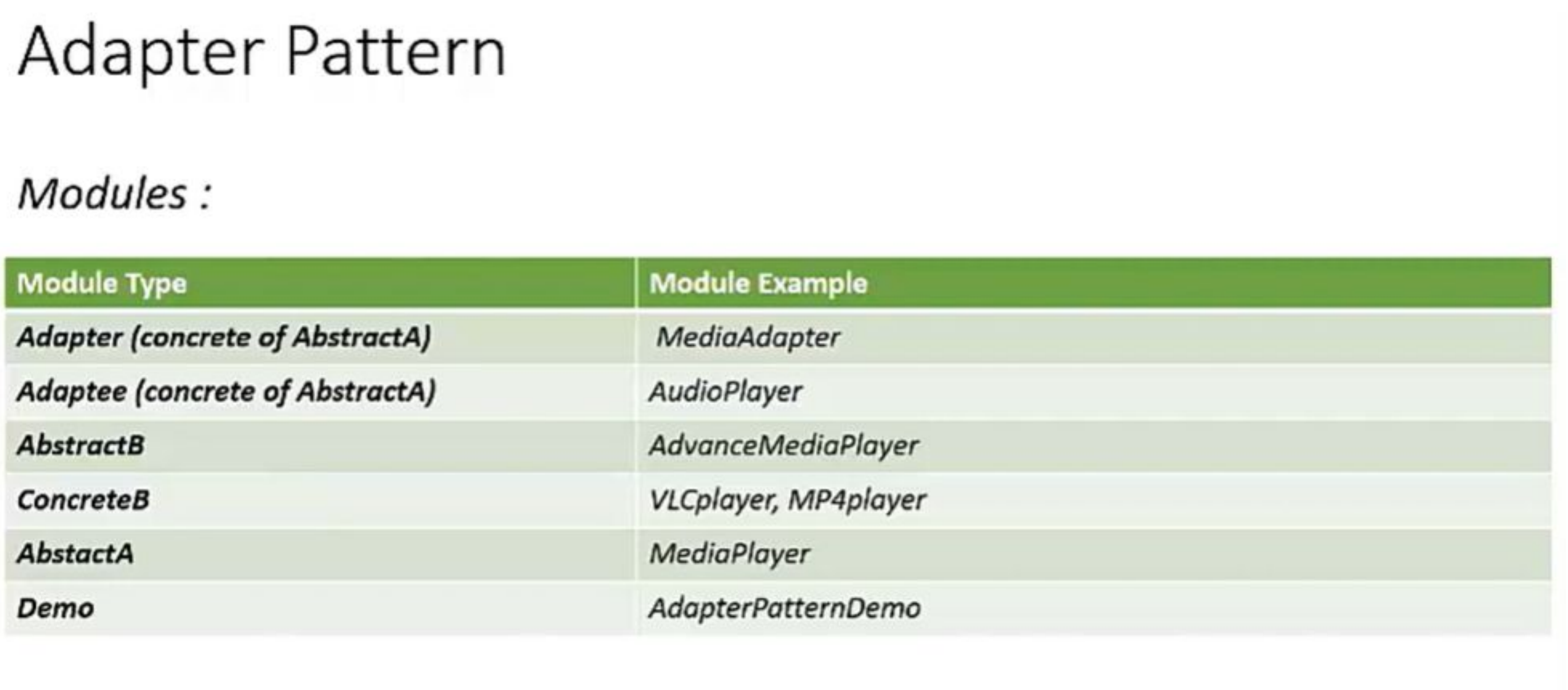
Solution: To attain this, we have created an adapter class MediaAdapter which implements the MediaPlayer interface and uses AdvancedMediaPlayer objects to play the required format. AudioPlayer uses the adapter class MediaAdapter passing it the desired audio type without knowing the actual class which can play the desired format. AdapterPatternDemo, our demo class will use AudioPlayer class to play various formats.

Demo….>refer to AdapterPatternDemo.java

Design Solution



Design Solution



Adapter Pattern

Modules :

Module Type	Module Example
Adapter (concrete of AbstractA)	*MediaAdapter*
Adaptee (concrete of AbstractA)	*AudioPlayer*
AbstractB	*AdvanceMediaPlayer*
ConcreteB	*VLCplayer, MP4player*
AbstactA	*MediaPlayer*
Demo	*AdapterPatternDemo*

Adapter: Scenario example-2

Suppose you have a Bird class with fly() and makeSound()methods. And also a ToyDuck class with squeak() method. Let's assume that you are short on ToyDuck objects and you would like to use Bird objects in their place. Birds have some similar functionality but implement a different interface, so we can't use them directly.

Solution: So, we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.(refer to demo main.java

The adapter pattern we have implemented above is called Object Adapter Pattern because the adapter holds an instance of adaptee.
What a Object adapter??? Next slides we will explain in detail

Adapter: Scenario example-2

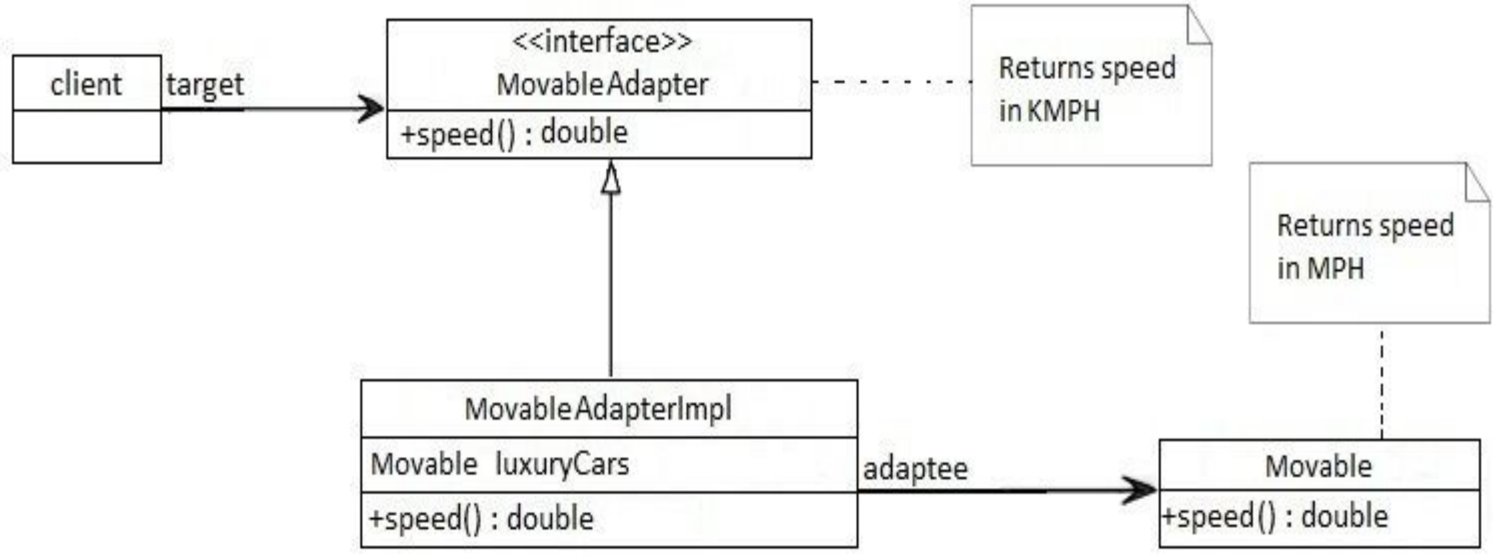
Explain: Here we have created a BirdAdapter class implementing inteface ToyDuck to convert Bird 's makeSound() method to makeSound() 's squeak() ToyDuck . This way, we can still call the squeak() of Bird objects through BirdAdapter

Suppose we have a bird that can makeSound(), and we have a plastic toy duck that can squeak(). Now suppose our client changes the requirement and he wants the toyDuck to makeSound then ?

Simple solution is that we will just change the implementation class to the new adapter class and tell the client to pass the instance of the bird(which wants to squeak()) to that class.
Before : ToyDuck toyDuck = new PlasticToyDuck();
After : ToyDuck toyDuck = new BirdAdapter(sparrow);
You can see that by changing just one line the toyDuck can now do Chirp Chirp !!

Adapter: Scenario example-3

Consider a scenario in which there is an app that's developed in the US which returns the top speed of luxury cars in miles per hour (MPH). Now we need to use the same app for our client in the UK that wants the same results but in kilometers per hour (km/h).
To deal with this problem, we'll create an adapter which will convert the values and give us the desired results:



Design solution

(Students can Try coding with the given design solution)

Applicability

Use the Adapter pattern when

- you want to use an existing class, and its interface does not match the one you need.

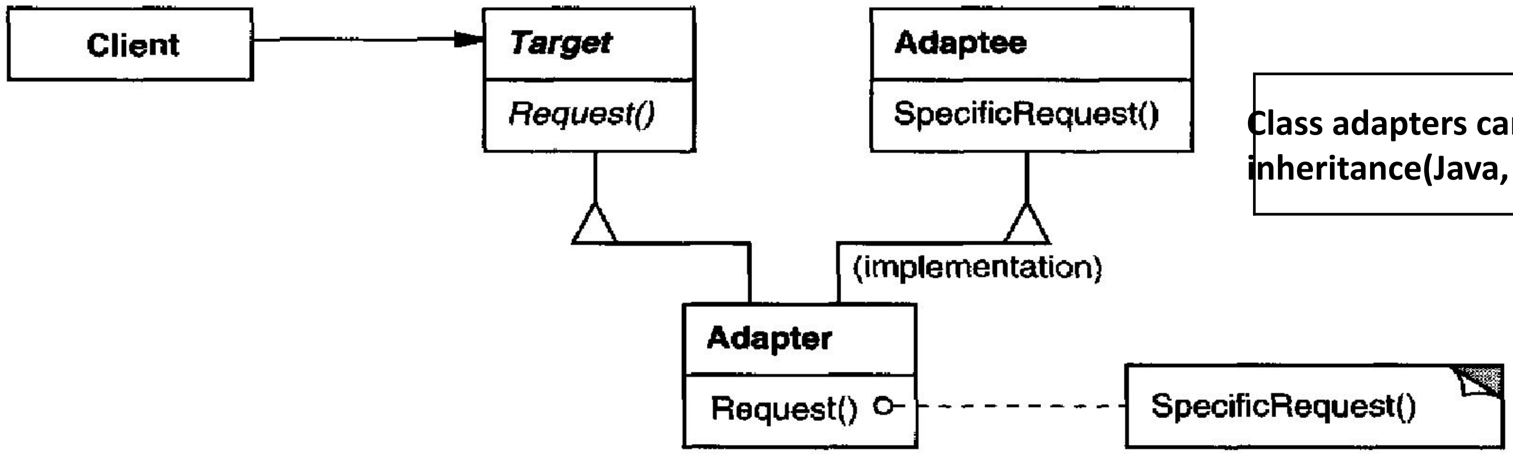
- you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.

- (object adapter only) you need to use several existing subclasses, but it's unpractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Software Examples of Adapter Patterns: Wrappers used to adopt 3rd parties libraries and frameworks - most of the applications using third party libraries use adapters as a middle layer between the application and the 3rd party library to decouple the application from the library.

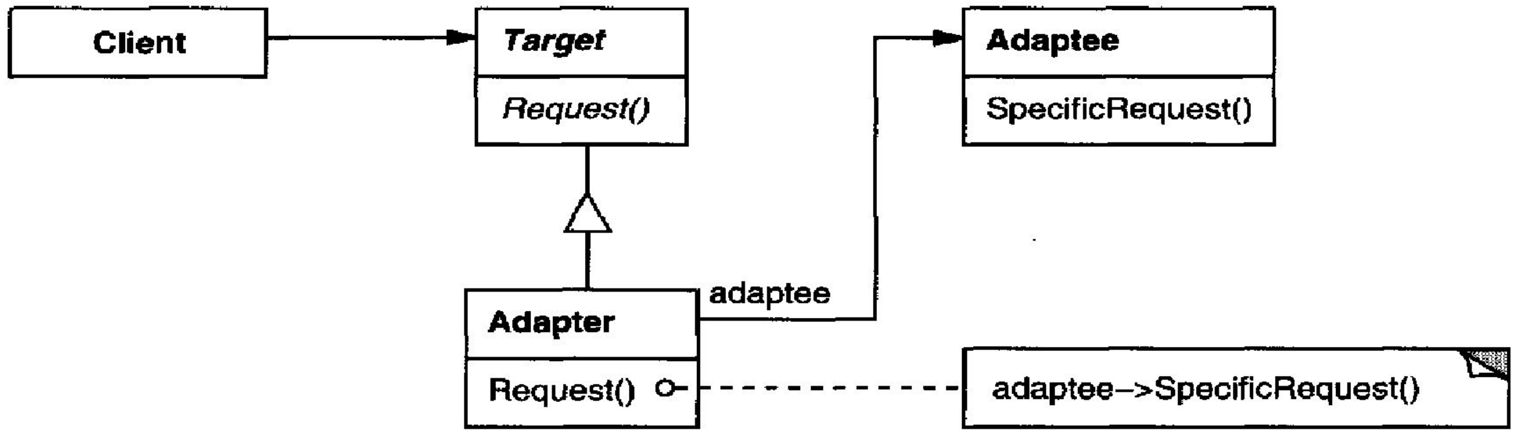
Structure

A class adapter uses multiple inheritance to adapt one interface to another:



Class adapters can be implemented in languages supporting multiple inheritance(Java, C# or PHP does not support multiple inheritance)

An object adapter relies on object composition: Based on delegation



Consequence

Class and object adapters have different trade-offs.

A class adapter

- ❑ adapts Adaptee to Target by committing to a concrete Adaptee class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
- ❑ let's Adapter override some of Adaptee’s behavior, since Adapter is a subclass of Adaptee.
- ❑ introduces only one object, and no additional pointer indirection is needed to get to the adaptee.

An object adapter

- ❑ **let's a single Adapter work with many Adaptees**—that is, the Adaptee itself and all of its subclasses (if any). The Adapter can also add functionality to all Adaptees at once.
- ❑ makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

Difference: Class and Object structure pattern

- Objects Adapters uses composition, the Adaptee delegates the calls to Adaptee (opossed to class adapters which extends the Adaptee).
- The main advantage is that the object Adapter adapts not only the Adaptee but all its subclasses. All it's subclasses with one "small" restriction: all the subclasses which don't add new methods, because the used mechanism is delegation. So for any new method the Adapter must be changed or extended to expose the new methods as well.
- The main disadvantage is that it requires to write all the code for delegating all the necessary requests to the Adaptee.
- Class adapter uses inheritance instead of composition. It means that instead of delegating the calls to the Adaptee, it subclasses it. In conclusion it must subclass both the Target and the Adaptee.
- There are advantages and disadvantages:
  It adapts the specific Adaptee class. The class it extends. If that one is subclassed it can not be adapted by the existing adapter.
- It doesn't require all the code required for delegation, which must be written for an Object Adapter.
- If the Target is represented by an interface instead of a class then we can talk about "class" adapters, because we can implement as many interfaces as we want.

Issues to consider when using the Adapter pattern

- *How Much the Adapter Should Do?*

It should do how much it has to in order to adapt. It's very simple, if the Target and Adaptee are similar then the adapter has just to delegate the requests from the Target to the Adaptee. If Target and Adaptee are not similar, then the adapter might have to convert the data structures between those and to implement the operations required by the Target but not implemented by the Adaptee.

- *Using two-way adapters to provide transparency.*

A potential problem with adapters is that they aren't transparent to all clients. An adapted object no longer conforms to the Adaptee interface, so it can't be used as is wherever an Adaptee object can. Two-way adapters can provide such transparency. Specifically, they're useful when two different clients need to view an object differently.

References

Text Reference
- Design Patterns: Elements of Reusable Object-Oriented Software, GOF

Web Reference
- https://www.javatpoint.com/adapter-pattern
- https://www.oodesign.com/adapter-pattern.html
- https://www.geeksforgeeks.org/adapter-pattern/
- https://www.tutorialspoint.com/design_pattern/adapter_pattern.htm
- https://itzone.com.vn/en/article/adapter-design-pattern-in-java

PES UNIVERSITY ONLINE

THANK YOU

Dr. Sudeepa Roy Dey

Department of Computer Science and Engineering

sudeepar@pes.edu