Software Architectures
Lab. 2. Homework

Learn by your own the following design patterns: Adapter, Strategy, Chain of Responsibility, and Composite.

Generalities about design patterns

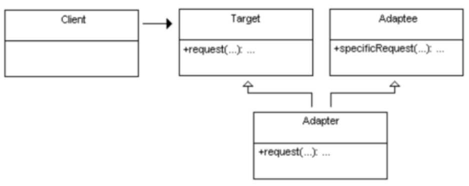
Def1: Reusable solutions to commonly occurring problems.
Def2: OO design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
Each pattern has:

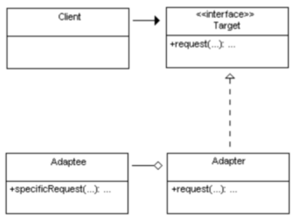
- name
- problem – when to apply the problem
- solution – elements that make up the design, relationships, responsibilities and collaborations
- consequences – results and trade-offs of applying the pattern

Reference:
The course lectures: http://www.serbanati.com/poli/Lecture_Notes/ARCH/ARCH_Chap1.pdf
https://www.oodesign.com/
https://medium.com/@ronnieschaniel/object-oriented-design-patterns-explained-using-practical-examples-84807445b092
https://medium.com/@adeshsrivastava0/decorator-design-patterns-3d6ed6d5aba9
Please, read descriptions for *Adapter, Strategy, and Chain of Responsibility* from references.

Adapter

Adapter pattern is a structural pattern used to adapt interfaces to clients using different interfaces, sometimes also transforming the data into appropriate forms. The pattern is adapting between classes and objects. An adapter allows two incompatible interfaces to work together. Like any adapter in the real world it is used to be an interface, a bridge between the two objects.



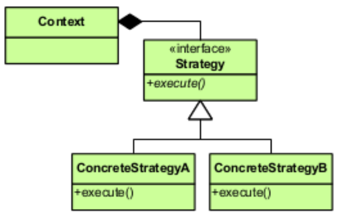


This pattern involves a single class which is responsible to join functionalities of independent or incompatible interfaces. A real life example could be a case of card reader which acts as an adapter between memory card and a laptop. You plugin the memory card into card reader and card reader into the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

- How can a class be reused that does not have an interface that a client requires?
- How can classes that have incompatible interfaces work together?
- How can an alternative interface be provided for a class?

Strategy

The Strategy pattern is a behavioral pattern that enables us to select an algorithm at runtime: a class behavior or its algorithm can be changed at run time. Instead of implementing a single algorithm directly, code receives run-time instructions as to which in a family of algorithms to use. 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 **strategies** include a range of algorithms which are distinct from the actual program and are autonomous (i.e. exchangeable).



In Java framework **Strategy** pattern is used in Collection framework for sorting. Eg. *sort* method has a default Comparator which can be replaced by own implementing classes.

Advantages:

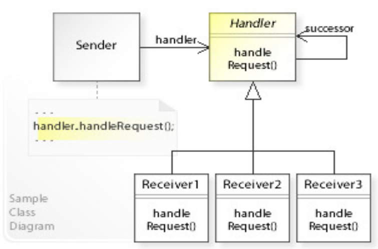
- Helps achieve reusability and flexibility.
- Client class is not complicated by having to use a different interface and can use polymorphism to swap between different implementations of adapters.

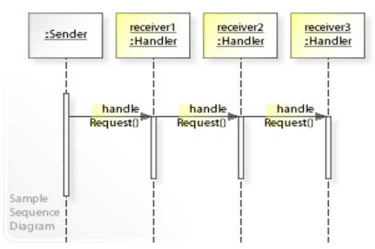
Disadvantages:

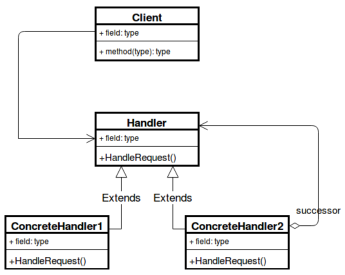
- All requests are forwarded, so there is a slight increase in the overhead.
- Sometimes many adaptations are required along an adapter chain to reach the type which is required.

Chain of Responsibility

Chain of responsibility is a behavioral design pattern that is used to achieve loose coupling in software design where a request from the client is passed to a chain of objects to process them. Later, the object in the chain will decide themselves who will be processing the request and whether the request is required to be sent to the next object in the chain or not. It consists of a *source of command objects* and a series of *processing objects*. The Client/Sender in need of a request to be handled sends it to the chain of handlers which are classes that extend the Handler class. Each of the handlers in the chain takes its turn at trying to handle the request it receives from the client. Each processing object contains logic that defines the types of command objects that it can handle; the rest are passed to the next processing object in the chain. A mechanism for adding new processing objects to the end of this chain exists.







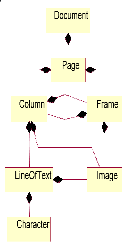
In a variation of the standard chain-of-responsibility model, some handlers may act as dispatchers, capable of sending commands out in a variety of directions, forming a tree of responsibility.

Composite

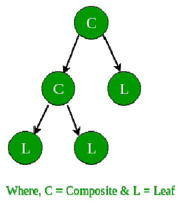
1. Sometimes we are dealing with data structures that aggregate in a manner similar to a whole-part hierarchy (tree-structured) but with many exceptions.
2. When dealing with tree-structured data, programmers often have to discriminate between a leaf-node and a branch.

This makes code more complex, and therefore, error prone. The solution is an interface that allows treating complex and primitive objects uniformly -> Composite pattern.

1)



2)

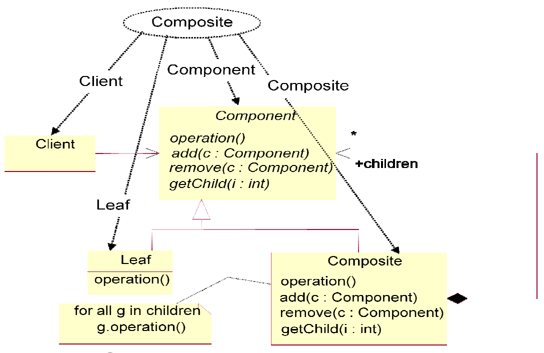


Composite pattern is a partitioning design pattern and describes a group of objects that is treated the same way as a single instance of the same type of object. The intent of a composite is to “compose” objects into tree structures to represent part-whole hierarchies.

Composite lets client treat individual objects and compositions of objects uniformly”. It allows you to have a tree structure and ask each node in the tree structure to perform a task.

In object-oriented programming, a composite is an object designed as a composition of one-or-more similar objects, all exhibiting similar functionality. This is known as a “*has-a*” relationship between objects.

The key concept is that you can manipulate a single instance of the object just as you would manipulate a group of them. The operations you can perform on all the composite objects often have a least common denominator relationship.



When not to use Composite Design Pattern?

1. Composite Design Pattern makes it harder to restrict the type of components of a composite. So it should not be used when you don't want to represent a full or partial hierarchy of objects.
2. Composite Design Pattern can make the design overly general. It makes harder to restrict the components of a composite. Sometimes you want a composite to have only certain components. With Composite, you can't rely on the type system to enforce those constraints for you. Instead you'll have to use run-time checks.

Exercises

Adapter
Exercises 1
1a) Suppose you have a Bird class with fly() , and makeSound()methods. And also a ToyDuck class with screech() 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. So we will use adapter pattern. Here our client would be ToyDuck and adaptee would be Bird.
Draw the design class diagram, following Adapter design pattern structure.

1b) An audio player device can play *mp3* files only and wants to use an advanced audio player capable of playing *vlc* and *mp4* files.
Draw the design class diagram, following Adapter design pattern structure.
End of exercise 1

Strategy
Exercise 2
A navigation app should calculate a route based on normal modes of transport. The user can choose between three options: Pedestrian, Car, and Public transport. The graphical user interface (GUI) of the navigation app has buttons for calculating routes. Once the user makes a selection and taps on a button, a concrete route is calculated corresponding to the selection. The Navigator class has the task of calculating and presenting a range of control points on the map. The navigator class has a method for switching the active routing strategy. This means it is possible to switch between modes of transport via the client buttons. For example, if the user triggers a command with the pedestrian button of the client, the service "Calculate the pedestrian route" is requested.

Draw the design class diagram, and the interaction diagram of the scenario that calculates the route using the three options.

End of exercise 2

Chain of Responsibility
Exercise 3
Use Chain of responsibility pattern to allow a Request to access using the endpoint of a protected Resource, by creating a naive security filter chain that will check the following responsibilities in order:

1. If any endpoint matches the request
   - *False* - return *Not Found*
2. If the endpoint has any security rules
   - *False* - return *Resource*
3. If the request contains any authorization header
   - *False* - return *Unauthorized request*

4. If the authorization is approved
   - *False* - return *Invalid authorization*
5. return *Resource*

Draw the design class diagram, and the interaction diagram of the scenario that sets up the chain of handlers when a request for accessing the Resource arrives.
Hint
You may create a Helper class to handle some conditional logic.
End of exercise 3

Composite
Exercise 4
In a software company, we have several types of employees. So, we have general managers and under general managers there can be simple managers and under any manager there can be developers.
The Human Resources Division wants to be able to have a uniform access to the information of each employee, for example her/his salary, regardless of the employee's position in the company.
Draw the design class diagram, following Composite design pattern structure.
End of exercise 4