Software Architectures
Lab. 3. Homework

Learn by your own the following design patterns: Object Pool, Prototype, Iterator, and Observer.

Generalities about design patterns

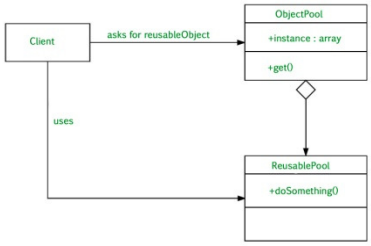
Def1: Reusable solutions to commonly occurring problems.
Def2: OO design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
Each pattern has:

- 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.

Object Pool.

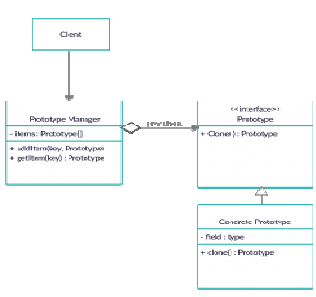
Object pool pattern is a software creational design pattern which is used in situations where the cost of initializing a class instance is very high. Object Pool Pattern says us "to reuse the objects that are expensive to create".
Basically, an Object pool is a container which contains a specified amount of objects. Objects in the pool have a lifecycle: creation, validation and destroy. When an object is taken from the pool, it is not available in the pool until it is put back. A pool helps us to manage available resources in a better way. It is also useful when there are several clients who need the same resource at different times. There are many using examples: especially in application servers where there are data source pools, thread pools, component pools, etc.

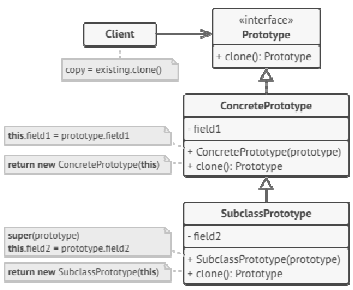


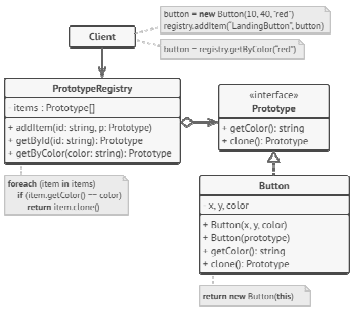
Prototype

Prototype pattern refers to creating duplicate object while keeping performance in mind. This type of design pattern comes under creational pattern and provides one of the best ways to create an object.

This pattern involves implementing a prototype interface which tells to create a clone of the current object. This pattern is used when creation of object directly is costly. For example, an object is to be created after a costly database operation. We can cache the object, returns its clone on next request and update the database as and when needed thus reducing database calls.



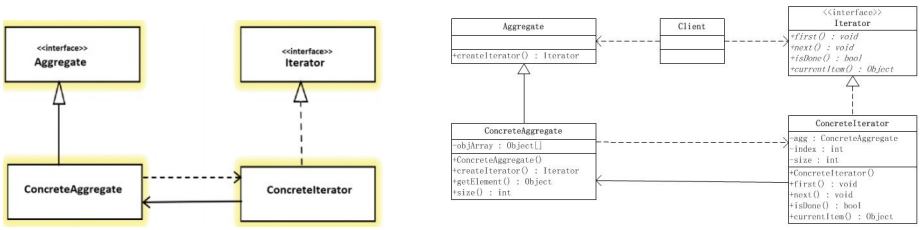




The **Prototype Registry** provides an easy way to access frequently-used prototypes. It stores a set of pre-built objects that are ready to be copied.
Advantages: By using the prototype pattern we reduce the subclasses, have the ability to create unlimited number of objects copies with different styles, less resources and without needing any dependency in the client code.
Disadvantages: Cloning objects can be difficult when their internals include objects that don't support copying or having circular references
Some notes for implementation:
– Each prototype class must explicitly implement the clone operation.
– The clone method returns a new prototypical version of the constructor
– If the client needs to initialize some objects with different values, you can define an Initialize operation that takes initialization parameters as arguments and sets the clone's internal state accordingly.

Iterator

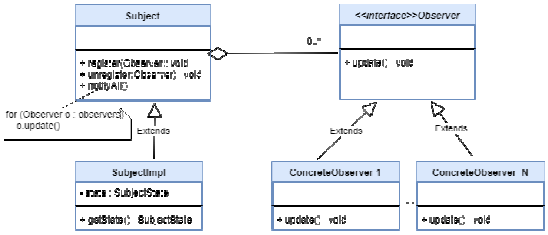
This behavioral design pattern provides a way to access the elements of an aggregate/container object sequentially without exposing its underlying representation. An aggregate/container can be a collection of objects like list, set, multiset, tree, graph, and by underlying representation we mean the way the container has been composed with its methods and properties.



Participants
Iterator defines an interface for accessing and traversing elements.
ConcreteIterator implements the Iterator interface and keeps track of the current position in the traversal of the aggregate.
Aggregate defines an interface for creating an Iterator object.
ConcreteAggregate implements the Iterator creation interface to return an instance of the proper ConcreteIterator
The Client is the ItemDisplayer.
The pattern Iterator is used in java.util.Iterator, java.util.Scanner and java.util.Enumeration.

Observer

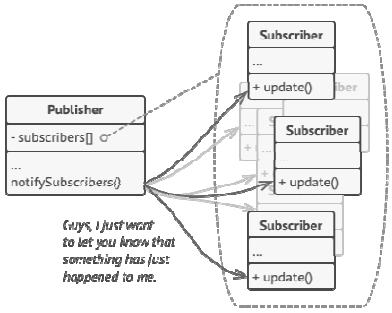
Observer pattern is a behavioral pattern that is used to form relationships between objects at runtime, for instance a one-to-many relationship between objects such as if one object is modified, its dependent objects are to be notified automatically. So, the Observer pattern is the gold standard in decoupling - the separation of objects that depend on each other.



Participants:

- **the Subject:** maintains a list of observers, provides methods to register/unregister observers. Also, has a *notifyAll()* method to notify all registered observers of any state change
- **SubjectImpl:** the class extending the functionality of the *Subject* class, it holds a state object representing its current state. Note that **t's a good idea to have an immutable state object to prevent any unintentional updates by the observer**
- **Observer:** it's an interface with an *update()* method which is invoked by the *Subject* to notify the observer of any changes in its current state
- **ConcreteObserver:** these are the classes implementing the *Observer* interface, the observer objects register themselves to listen to a *Subject*

The idea behind the pattern is simple - one of more **Observer**s are interested in the state of a **Subject** and register their interest with the Subject by **attach/register**ing themselves. When something changes in our Subject that the Observer may be interested in, a **notify** message is sent, which calls the **update** method in each Observer. When the Observer is no longer interested in the Subject's state, they can simply **detach/unregister** themselves.



The benefits here are quite clear. To pass data onto the observers, our subject doesn't need to know who needs to know. Instead, everything is done through a common interface, and the notify method just calls all the objects out there that have registered their interest. This is a very powerful decoupling - meaning that any object can simply implement the Observer interface and get updates from the Subject.
In general, you want to use this pattern to reduce coupling. If you have an object that needs to share it's state with others, without knowing who those objects are, the Observer is exactly what you need.
The pattern Observer is used in the Java package Swing to implement event listeners (ActionListener).

Exercises

Exercise 1
Object Pool
Let's take the example of the database connections. Accessing a database involves the creation of *database connection objects*. The creation of the connection object is an expensive work to do as it involves loading drivers, validating the statements and several other things. It's obviously that opening too many connections might affect the performance for several reasons:

- Creating a connection is an expensive operation.
- When there are too many connections opened it takes longer to create a new one and the database server will become overloaded.

So, as a solution to the above problem, we will be creating a pool of connection objects. This pool will hold expensive objects which can be distributed to clients as per the requirement. The object pool manages the connections and provides a way to reuse and share them. It can also limit the maximum number of objects that can be created.

Design the model of the object pool of database connections. Pool creation and initialization with a configurable number of new connections should be included in the model. Use for this the Factory Method design pattern. We will also be creating an interface to get the reusable objects from that pool and return them to the pool.
End of exercise 1

Exercise 2
Prototype
In this exercise, the **Prototype** pattern lets you produce exact copies of geometric objects, without coupling the code to their classes. We're going to create an abstract class Shape and concrete classes extending the Shape class representing circles, rectangles and squares. A class ShapeCache is defined which stores shape objects in a Hashtable and returns their clone when requested. PrototypPatternDemo, our demo class will use ShapeCache class to get a Shape object.
End of exercise 2

Exercise 3
Iterator
Consider an application for a supermarket with various departments like for instance *BeveragesDepartment* and *VegetablesDepartment*. Each department holds/encompasses a collection of items. Consider also a *Department* interface which models the only means to access the information of these departments: a unique method *displayAllItems* that gets the items from any department and prints them. Each department uses a different data structure (list, map, tree, etc.) for holding its collection. For instance, the *BeveragesDepartment* uses a Map whereas the *VegetableDepartment* uses a Set to hold the items. Let call *aggregate* such a collection of items.
There is a problem in this implementation that should be solved:

For the client (*ItemDisplayer*), the actual type used by a Department to store the Items should be transparent because when we add a new department, the client code has not to change to enable iterating over the particular collection of items present within the new department. This is why we should use the design principle of *encapsulating what varies* to hide the particularity of collection implementations.
Analyze how the Iterator pattern can solve this problem and draw the application model that uses it.
End of exercise 3

Exercise 4
Observer
An e-commerce shop publishes its products in web and also articles about the benefits of its product lines.
If the product you want to purchase is out of stock, you may ask to be informed when the product becomes available again. You provide your email and subscribe to the customer list. As soon as the product becomes available, all customers are informed.
You may also visit the e-commerce shop website. If you read an article and like it, you may want to be informed whenever a new article is posted on the website. You provide your email and subscribe to the reader list. As soon as there is a new article posted on the website, all the readers are notified.
Model the notification process of the e-commerce shop using the Observer pattern.
End of exercise 4