Software Engineering

Unit 2 Questions

Team 29

Sudipta Halder (2021202011)

Anjaneyulu Bairi (2021202008)

Sowmya Lahari Vajrala (2021202010)

Josh Elias Joy (2021204009)

Q1. Consider the interface java.util.Collection, which is implemented by the java collection classes. One of the methods defined in the interface is iterator(). Referring to the intention for the pattern and its structure why is iterator() a Factory Method?

Ans:

Factory method: In real time as a developer, you should not provide class constructors to let users of your class instantiate it. It is better to hide the direct ways to instantiate a class. So, using factory method we can define an interface to create an object and hiding the details of ways to instantiate the class objects.

A factory method hides the operation that creates an object and isolates its user/client from knowing which class to instantiate.

The JAVA JDK 1.2 version introduced iterator () operation in Collection interface. Using iterator method, you can create an object of a class which implements iterator interface and you do not know exactly about class, but you know only about methods that are defined in Iterator and you can call them. This iterator object returns a sequence of elements of a collection, this object encapsulates traversal details, like current position and how many elements are left till to traverse, Iterators can independently go through the same collection at the same time, so iterators are best examples of factory method. Iterator object isolates its caller from knowing which class to instantiate.

Q2. Defend or refute the statement “A factory method is an example of a very simple template method.”

Ans:

Both methods template and factory methods are having similar design, but the purpose is different.
Factory method is creational pattern and creating object is the responsibility of child class. An interface will be created for creating superclass objects, but subclass can alter the object type.

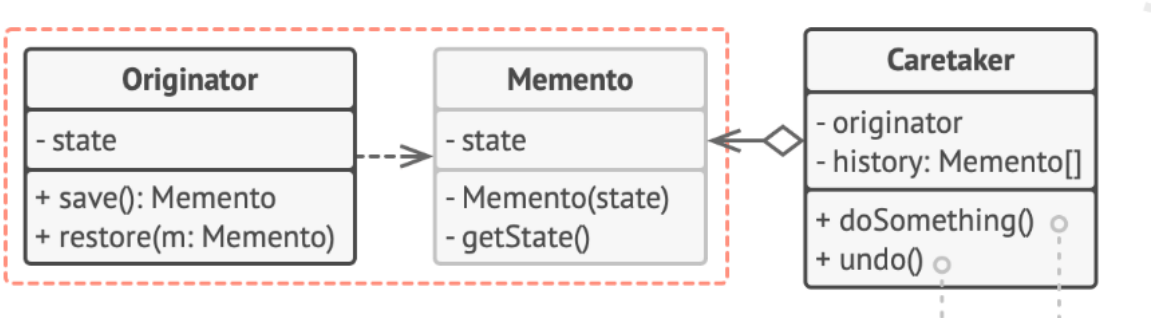
Template method is a Behavorial pattern where subclass takes the responsibility of behaviour. An Abstract method is defined for behaviour in a class and another method which is going to invoke this abstract method to have the behaviour executed. This behaviour is implemented by subclass or child class. As a result, the parent class calls the implementation of the child class without being explicitly compile-time dependent on it. This applies to factory method also, but as said both are differ in purpose. We can say factory method creates object and it is like template method pattern executes the behaviour.

Q3. What does it mean to say a memento object is opaque to its client? What implications does this have on the client's use of a memento?

Ans:

Memento pattern falls under the category of behavioural design pattern. In this design pattern, it allows the user to save and restore an object's prior state without exposing its implementation specification. So, this becomes really beneficial while performing redo and undo operation. For e.g., undo and redo operation in text editor.

Below is the UML diagram of a memento design pattern.



[1]

From the above diagram we can see 3 entities in the Memento design pattern.

- **Originator** (constraint solver): originator generates a memento that contains a snapshot of the system's present state. Later it uses the memento to restore the internal state.[2]
- **Memento** (SolverState): The internal state of the Originator object is saved in Memento.[2]
- **Caretaker** (undo mechanism): The memento's preservation is the responsibility of the Caretaker.[2]

i) When the client needs to save the source object's state, it asks a Memento from the source object. What the source object basically does is, it initializes the memento with a description of its current state. So, the client acts as the "care-tacker" of the Memento wherein the source object can store and retrieve data from the Memento. In this way, the Memento is "opaque" to its client and also to all other objects since the capability of storing and retrieving any information from memento object is confined to originator or source object only.

ii) We can claim that because of this design pattern, the Client has a limited perspective of the system's overall architecture and underlying structure. The only things which are available to the client is a list of saved states and a reference of the current or present state. The client has the capability of going back and forth between states with the help of undo and redo techniques. So, we can reach to the conclusion that client works as viewer rather than editor in this case. So, we can definitely conclude that the client has restricted access to the entire system in case of this Memento design pattern.

Q4. What issues must be considered if State objects are to be shared by multiple instances of the same Context class (hint: consider mutability)?

Ans: If state objects are to be shared multiple instances of the same context class, there will be always the possibility of information persistence and integrity. Basically, if one Instance of the state object changes something which was not desired for another Instance of the same object, in that case it will lead to a conflict.

We can propose two solutions to tackle this:

- We can declare the State object immutable. This phenomenon can be achieved by sending copies of object instead of object reference from getter methods. In that case, multiple instances of the object won't be able to update the state simultaneously. Hence, more time will be consumed and it will be less user friendly and less-flexible.
- Another we can way is to use database. In that case, basically we can store the changes done by different instances and track them properly. But it would obviously increase the cost of storage.

Q5. For Flyweight to be effective, only the Flyweight factory can be allowed to construct new Flyweight objects. How can this be accomplished in Java (hint: consider making the factory a static method in the Flyweight class itself!).

Ans:

Flyweight pattern is a structural design pattern which reduces space usage by separating the common attributes of objects and creating Flyweight objects out of them. These common attributes are called intrinsic attributes/data. The remaining attributes by which the objects differ are called extrinsic attributes. A set of objects will share common intrinsic data.

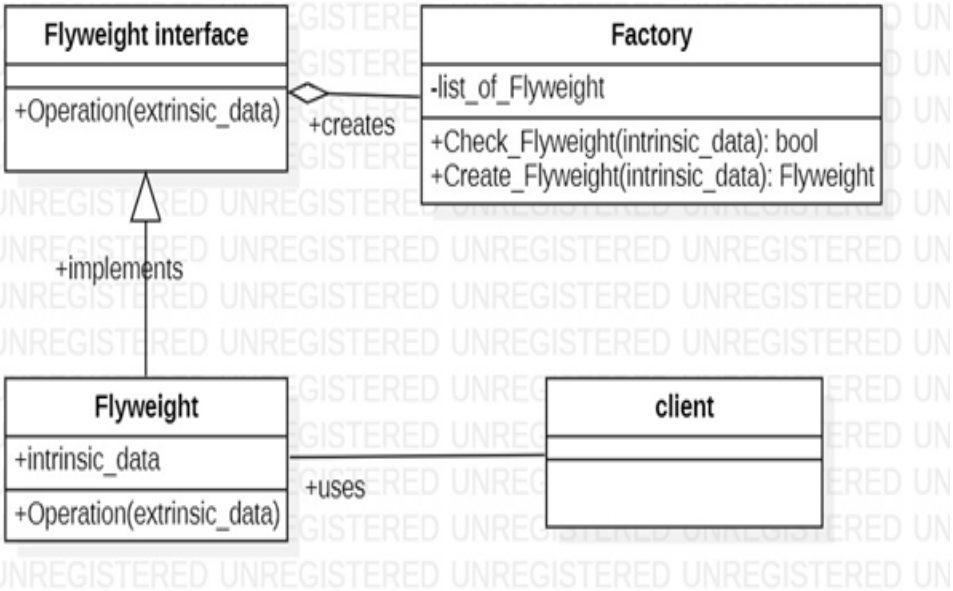
To make sure that the Flyweight objects are not duplicated, the creation responsibility is given to a separate class called Factory. The Factory object will store all the created Flyweight objects in a dictionary. Whenever the need to create a new Flyweight object arises, the Factory object will check if a Flyweight object with the same intrinsic data exists or not. If no such object exists, then Factory object will create a new Flyweight object and add it to the dictionary.

In terms of Java, there will be a common interface with a method to handle extrinsic data of Flyweight object. This interface will be implemented by different Flyweight classes each of them having a different intrinsic state. The Factory class will maintain a list or dictionary of Flyweight objects. It will have a static method which will first check whether the Flyweight object exists or not and create Flyweight object.

Having a static method will allow to access it without having to explicitly creating an object to access it. By creating a static method, we are providing a utility for all clients to create

Flyweight objects without having to create a Factory. There will be only one instance of factory that will control the creation of all Flyweight objects.

The class diagram of a generic Flyweight pattern is shown below.



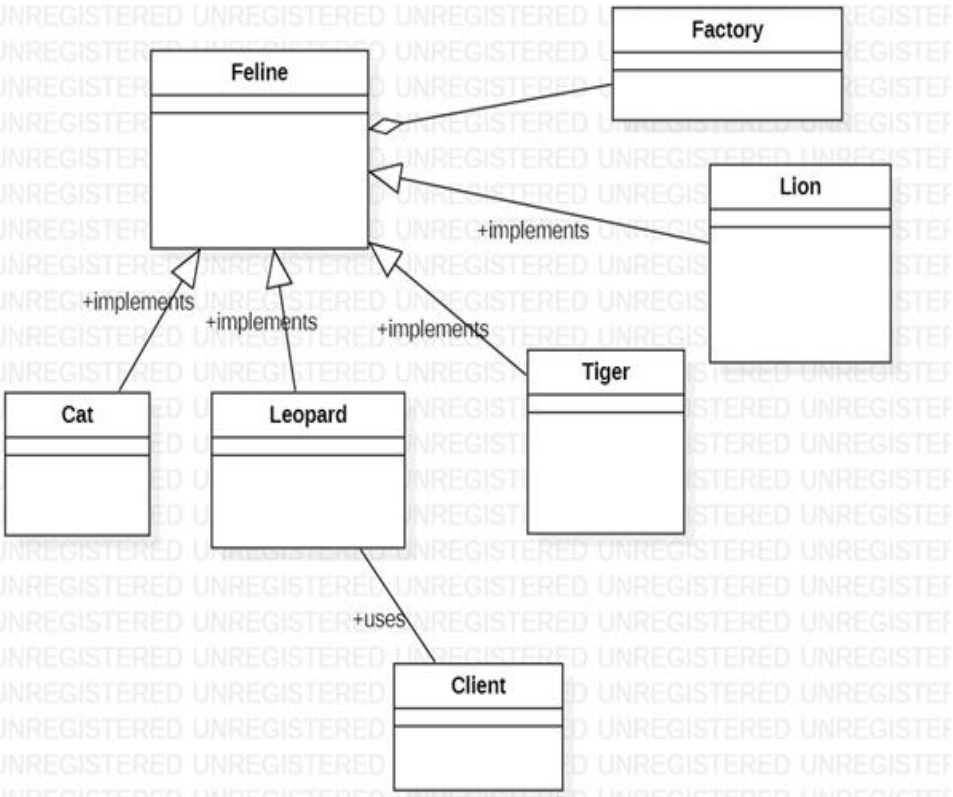
The Check_Flyweight and Create_Flyweight (can be combined into a single method) methods are static methods that checks for a flyweight object and create a flyweight object, respectively.

Q6. Defend or refute the statement "Flyweight objects must be immutable (i.e., unchangeable) once they are created."

Ans:

Flyweight pattern is used to reduce the creation of objects thus saving the space. This is applicable when objects have some common static data among them. This common data can be used to create common objects and these objects can be shared among the clients and are reused. The common data shared among the objects is called **intrinsic data** and the data due to which the objects differ is called **extrinsic data.**

The Flyweight objects are created by a separate object called **Factory object**. Factory object maintains a dictionary of already existing Flyweight objects. Whenever a new flyweight object must be created, the factory will check if a flyweight object already exists in the dictionary with the given intrinsic data. Below is a class diagram of flyweight pattern.



Immutability:

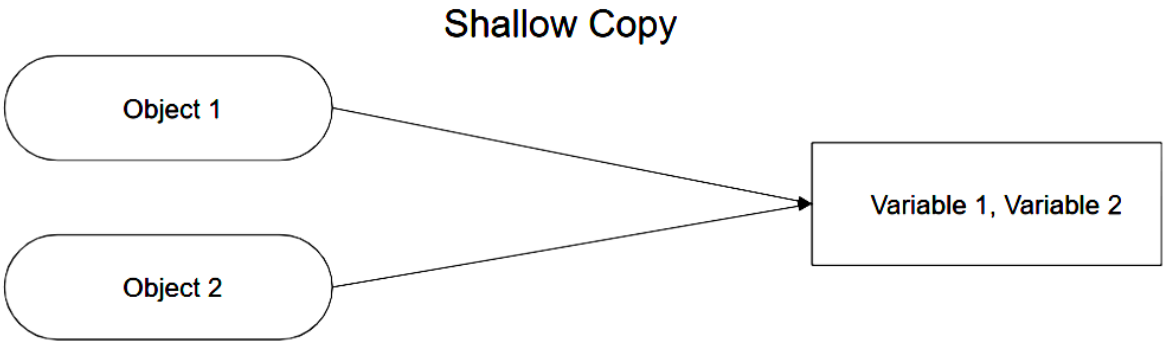
The flyweight objects are immutable which means they cannot be changed once they are created. If the Flyweight is mutable then the factory object will not be able to check if a Flyweight object already exists or not. The immutability helps the factory object to easily recognize whether a Flyweight object with given intrinsic data exits or not. Mutating the Flyweight object is nothing but changing the intrinsic data of the Flyweight object. If it is changed then the intrinsic data of all the related objects will be changed. It is better to create another Flyweight object with the new intrinsic data.

If the data to be stored in objects is mutable even, then we can use Flyweight pattern. This can be done by separating the immutable part of the objects and creating the Flyweight objects using it.

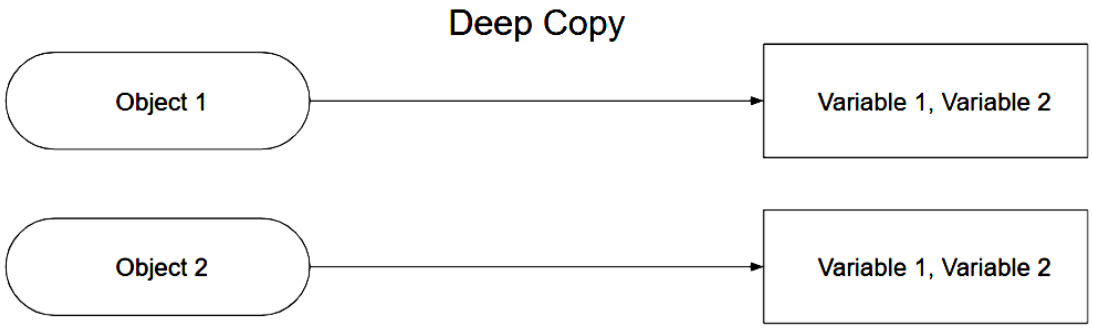
Q7. Discuss the issues involved in deciding whether cloning a prototype is done with shallow or deep copy.

Copying involves assigning a new object value same as that of an existing object. There are two types of copying: Shallow Copy and Deep Copy.

In Shallow Copy both the existing and new objects reference the same object variables. Thus, any change in one object will also reflect as a change in the other object.



In Deep Copy both the existing and new objects point to two different object variables. Thus, each object is independent of the other, and changes in attributes of object 1 will not be reflected in object 2.



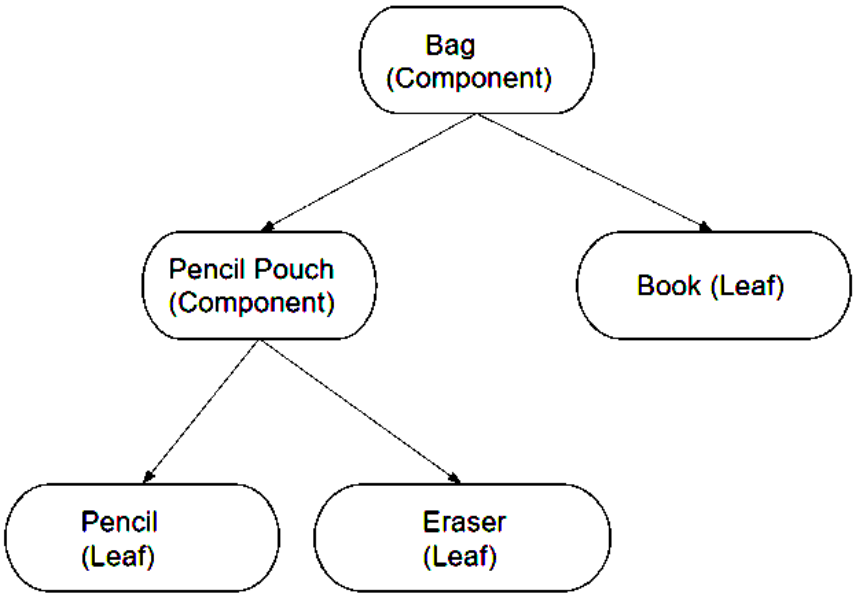
When we use the prototype pattern there is possibility to use shallow copy OR deep copy.

1. In the case where object variables are constants and do not change it would be better to use shallow copy.
2. If the object variables are mutable (can change) then it is possible to use both shallow and deep copy.
    a. If the referenced variables are being modified at some point and if this affects the behavior of the program, then proceed to use deep copy.
    b. If the referenced variables are not modified, then it would be ideal to use shallow copy.

Q8 Explain how the structure of the Interpreter pattern makes use of another pattern you have studied. Show the associations between the participants in the two patterns.

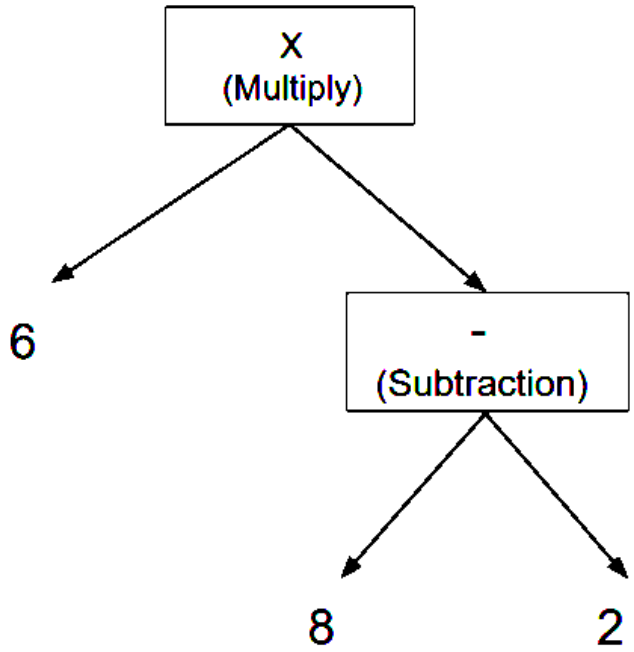
The interpreter design pattern is a behavioural pattern. The purpose of the interpreter pattern is to represent different language constructs/grammar and interpret them so that we can perform a required operation.

The interpreter design pattern makes use of the Composite design pattern structure, it uses the composite design pattern to represent a sentence or expression in a tree structure.



The above image shows an example of a component tree, in the Composite design pattern the component and leaf can be treated in the same way. The leaf is a primitive object, and the component creates an interface to access and manipulate the leaf objects.

The interpreter pattern uses an abstract syntax tree to process input expressions. An example of an abstract syntax tree:



In the above tree there are terminal and nonterminal expressions.

1) Terminal: They are the leaf nodes of the tree. In our numerical expression the operands 6,8 and 2 are terminal.

2) Nonterminal: They are non leaf nodes (component) of the tree. In our numerical expression the operator's 'x' and '-' are nonterminal. They require terminal to function.

The Composite design pattern is used to define the structure; hence it is a structural design pattern. Whereas the Interpreter design pattern defines behavior, as it interprets language.

Q9 Decorator claims to provide a pay-as-you-go approach to adding features. Do you agree that you can not get this same effect using class inheritance? Why do you agree or disagree?

I do agree with the statement as Decorator patterns allow dynamic extension of new functionality during runtime. The decorator pattern is much more flexible than compared to regular inheritance, as the decorator does not form a class hierarchy. Often times adding several features with inheritance creates an exploding class hierarchy. By using decorators, new features can be added and removed at run time easily. Another advantage of the decorator pattern is that it provides for code testability, which means that each new feature can be tested in isolation to the other features, which is not possible in inheritance.

The decorator pattern does offer a pay-as-you-go approach to adding features over inheritance. Instead of trying to implement a complex class which can possibly implement all the necessary features, we can implement a simple class and add features one by one. It is also easy to define new decorators that are independent of class object of which they extend. The process of extending a complex class usually forces the inherited class to have unrelated attributes visible to the new feature. Another benefit of using decorator pattern is that it encourages developers to develop software which follows the SOLID design principles.

Q10 Argue against the statement "Coordinating behavior should be defined in the objects being coordinated". How does the Mediator pattern fit into this argument?

We can better understand the statement by considering a real-world analogy. Consider the situation where multiple flights must land at an airport, this is done by each flight communicating with the air traffic controller rather than each flight contacting each other. Here the air traffic controller mediates with all the incoming flights so that they can provide the information that the runway is clear to land. All the air traffic controller must do is put constraints such that two flights do not land at the same time. Thus, the flights which are being coordinated define the coordinating behaviour of the air traffic controller. If the air traffic controller defined the coordinating behaviour then this could create a situation where some flights would have to wait in line until the runway is clear. This can lead to inefficiency and time wastage.

The Mediator pattern is a behavioural design pattern that prevents direct communication with objects to reduce chaotic dependences and only allows objects to communicate through a mediator object. The mediator pattern helps reduce coupling. The mediator patter fits into the argument as generally when we have several objects it would be necessary to use a mediator pattern. The mediator pattern is the most widely used design pattern. Thus, it is only possible for coordinated objects to define coordinating behaviour only when mediator pattern is used.

References:

[1] https://refactoring.guru/design-patterns/memento

[2] http://www.cs.unc.edu/~stotts/GOF/hires/pat5ffso.htm