DDR Assignment 01

Bilal Ahmed Khan

Roll Number: K200183

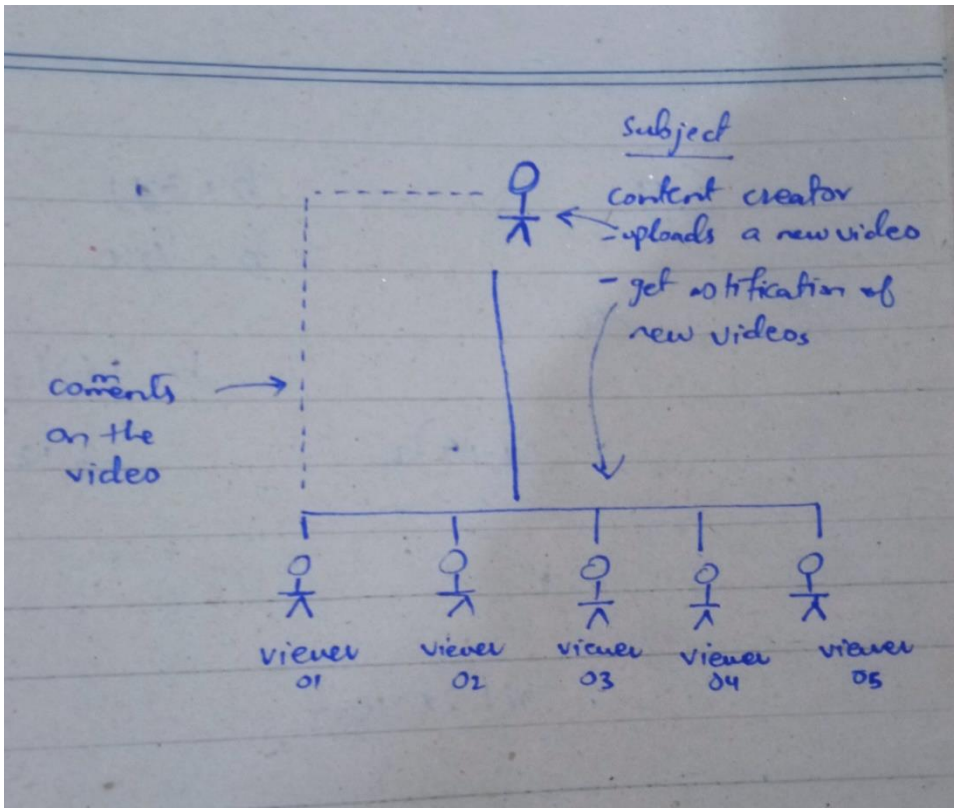
Question 01

Q1 – Part A)

Design Pattern: Command Pattern

Let us consider the scenario of a social media platform like YouTube, the Command Pattern integrates the interactions between video uploaders and viewers. Video uploaders, represented by the Video Uploader class, can use the platform through the user interface, performing actions such as uploading new videos, editing existing ones, or removing outdated content. Each of these actions translates into specific command objects, such as `*UploadVideoCommand*`, `*EditVideoCommand*`, and `*DeleteVideoCommand*`. These commands encapsulate the intended operations and are handed over to the *CommandInvoker*, an important component responsible for invoking the commands. The *CommandInvoker* operates as a mediator, efficiently triggering the execution of the corresponding commands, ensuring smooth integration and management of video content on the platform.

On the viewer's side, the Video Viewer class engages with the social media platform to watch the uploaded content. As the viewer interacts with the user interface to watch a video, a `*WatchVideoCommand*` is created and submitted to the *CommandInvoker*. The *CommandInvoker*, without the knowledge about the specific operations, invokes the `*WatchVideoCommand*`, leading to the execution of actions related to video viewing. This can include updating view counts, tracking user engagement (Comments etc.), and generating personalized video recommendations. Through the application of the Command Pattern, the social media platform achieves a modular and extensible architecture, allowing for the addition of new commands and alterations to existing functionalities without disrupting the overall system, enhancing the user experience for both uploaders and viewers.



Q1 – Part B)

Intent:

The Command Pattern, shared between the auction system and the YouTube, aligns with the primary aim of decoupling command senders and processors. In both scenarios, this separation ensures a modular design, and implements a one-to-many relationship while providing relevant functionalities.

Problem:

Inspired by the auction example where bidders trigger bid changes, the social media scenario mirrors this with video uploaders initiating actions, creating specific command objects (e.g., `UploadVideoCommand`). The CommandInvoker, similar to an auctioneer, invokes these commands without complete knowledge of underlying operations, implementing a separation of concerns and enabling reusability.

Discussion:

In both cases, using the command pattern allows dynamic handling of different commands while maintaining a clear, extensible program structure. In auctions, it accommodates bid-related actions, and in YouTube, it manages a variety of video-related operations. For Example: Whenever the video creator uploads a video, his followers get an notification. The followers can subscribe, unsubscribe and interact with the video. The implementation of command design pattern allows us to implement these functionalities in an elegant and extensible manner.

Q1 – Part C)

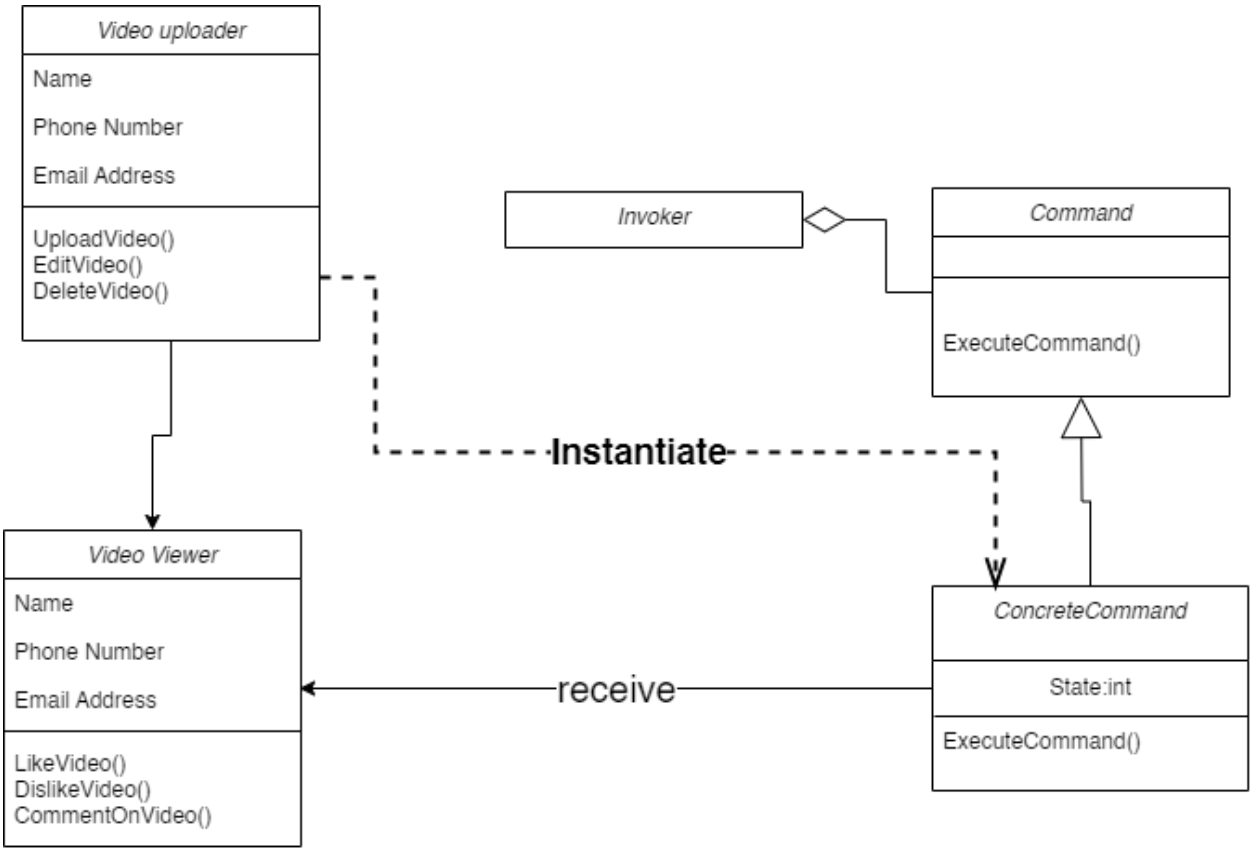


Figure 1UML Diagram

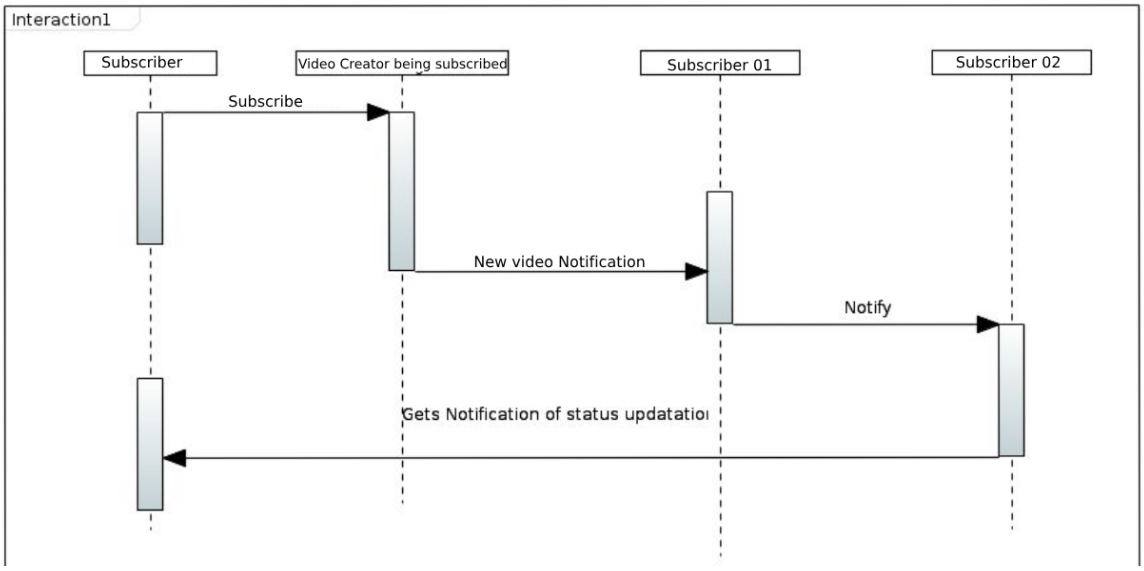


Figure 2 Sequence Diagram

Q1 – Part D)

In the context of the Command Pattern applied to the social media platform scenario, a potential disadvantage can arise in situations when frequent changes/updates in commands are needed. If there is a need to modify or extend the functionalities of existing commands, it could lead to increased complexity.

Since each command is encapsulated into its own class (e.g., `*UploadVideoCommand*`, `*EditVideoCommand*`), making changes may require altering multiple classes. This could result in maintaining a large number of command classes, leading to code bloat and making the system harder to manage and understand. While the Command Pattern offers flexibility, its downside lies in the potential for increased complexity when frequent changes to commands are necessary.

Question 02

Q2 – Part A)

For the given specifications a "**Factory design pattern**" will be a good fit for this use-case. We can define an interface for creating an object, but let subclasses alter the type of objects that will be created. The Factory Method Pattern defines an interface for creating an object but leaves the choice of its type to the subclasses, creation being deferred at the time of instantiation.

In our case, we can create a factory interface (or an abstract class) with a method, let's say createList(), and then have three concrete factories implementing this interface – one for our new list implementation, one for ArrayList, and one for LinkedList. Each concrete factory would provide an instance of the corresponding list type.

Q2 – Part B)

For the given specifications the "**Façade Pattern**" will be a good fit for this use-case. The design pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

In the context of a compiler with multiple steps like parsing, transformation, and code generation, the Facade Pattern can be applied to create a simplified interface that hides the complexities of the individual components. Users of the compiler can then interact with a single, high-level interface without needing to understand the details of each compilation step.