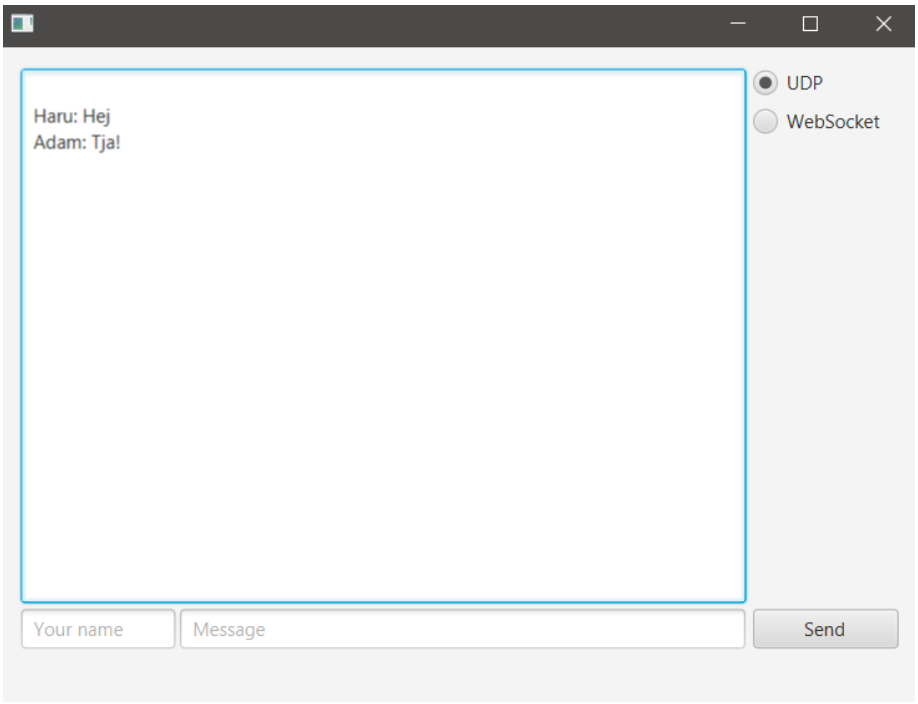
Arkitektur & Design Mönster -

Vilda mönster

Haru Tran & Adam El Soudi



Inledning:

Denna uppgift handlar om ett chattprogram som har i funktionen att kunna och ändra kommunicering alternativ, antingen UDP eller via Web Socket. Logiken finns dock inte och implementation av logiken dokumenteras här. Vi har implementerat två designmönster, både Abstract Factory och Observer Pattern (Push metodik), för att klargöra chattapplikationen. Med hjälp av Designmönstren har vi lyckats skapa en strukturerad och skalbar kod som är lätt att underhålla och förstå.

Design mönster som har använts:

För att lösa uppgiften så implementerades mönstret ”Abstract Factory” som handlar om att skapa objektet via abstraktion utan att specificera konkreta klasser. I denna uppgift så kan vi i framtiden skapa flera ”kommunikations” klasser som inte är bara UDP och WebSocket. Denna sorts abstraktion gör så att en klass inte har flera ansvar vilket följer SRP (Single Responsibility Principle) och sedan så skapar vi ett interface för kommunikationer så det blir en abstraktion vilket följer ISP (Interface Segregation Princple)

Vi använde också sedan Observer Pattern med en push metodik. Eftersom vi använde Observer Pattern så skapas en lösning som är löst kopplad och skapar en bättre separation mellan objekt som är beroende av varandra. Med denna implementation kan vi enkelt lägga till nya chatter (observers) utan att behöva ändra befintlig kod.

Observer Pattern separerar ansvar mellan objekt som gör att kopplingen blir mindre mellan klasser. Vi gör detta med att separera vårt Subject och alla observers som är beroende, vilket följer principerna Single Responsibility Principle (SRP) och Interface Segregation Principle (ISP). SRP ser till att varje objekt har bara ett ansvar, medan ISP gör att beroende objekt bara behöver implementera de metoder som den behöver.

Använda mönster:

Vi använder både Abstract Factory mönstret och Observer Pattern mönstret. Vi använder oss av Abstract Factory mönstret för att skapa en grupp av relaterade objekt utan att vi ska behövs specificera deras konkreta klasser. Mönstret används i klasserna WebSocketCommunicator och UDPChatCommunicator. Våran Observer Pattern mönster använder push metodik och används för att se till att om ett objekt ändrar tillstånd så ska alla objekt som är beroende av det objektet uppdateras automatiskt. I det här projektet används mönster för att uppdatera chattgränssnittet när nya meddelanden skrivs in av användaren.

Implementering av mönstren i applikationen:

Abstract Factory:

För att implementera Absract Factory har vi skapat ett interface *IFactory* som ska ta emot vilken typ av communicator som vi ska skapa. Vi har också skapat en konkret klass *CommunicatorFactory* som implementerar *IFactory* där en if–sats som först kollar om det är *UDP* eller *WebSocket* och skapar en instans av det.

```
public class CommunicatorFactory implements IFactory{
    2 usages
    public ICommunicator createComm(String type, MainWindowController CHAT) {
    if (type.equals("UDP")) {
        return new UDPChatCommunicator(CHAT);
    }
    else if(type.equals("WebSocket")) {
        return new WebSocketCommunicator(CHAT);
    }
        return null;
```

Observer Pattern (Push):

Vi har skapat två interfaces *ISubject* och *IObserver*. *MainWindowController* är vår subject. *WebSocketCommunicator* och *UDPChatCommunicator* är våra observers. När nytt meddelande skrivs så kallas *notifyObs* för att uppdatera alla registrerade observers.

```
private void sendMessage(String name, String message) {
    try {
        notifyObs(name, message);
```

Förhållande till SOLID-principerna:

Sättet vi följer Single Responsibility Principle (SRP) är genom att vi har separerat logiken för kommunikationen som sker i *WebSocket* och *UDP* från *MainWindowController*. Pågrund av det blir koden tydligt uppdelad i ansvarsområden och det blir även lättare att utveckla den vidare.

Med hjälp av användningen av Abstract Factory-mönstret följer vi Open/Closed Principle (OCP). Detta är för att vi enkelt kan lägga till nya communicators utan att vi behöver ändra om någonting i den kod som redan finns i *MainWindowController.* Det enda man faktiskt skulle behöva göra om man vill lägga till en ny communicator är att skapa en ny klass som implementerar *ICommunicator* och även uppdatera *CommunicatorFactory* koden så att den också kan hantera den nya communicatorn.

Genom att vi har använt oss av fokuserade gränsnitt som *IChat, ICommunicator, IObserver* och *ISubject,* följer vårat projekt Interface Segregation Principle (ISP) .

Eftersom vi använder abstraktioner och inte konkreta implementationer så följer vi Dependency Inversion Principle (DIP). Till exempel, *MainWindowController* är beroende av *ICommunicator* och *IObserver* interface istället för de konkreta klasserna *UDPChatCommunicator* och *WebSocketCommunicator*.

Vi är dock osäkra om vår *CommunicatorFactory* följer DIP då klassen ser ut att vara beroende av *UDPChatCommunicator* och *WebSocketCommunicator*. Vi är osäkra på en lösning för detta men skulle tro att skapa en sorts abstraktion för detta skulle göra att vi följer DIP.

Arbetsmetodik

Vi har arbetat i grupp och träffats i Systemet. Vi har använt parprogrammering där vi har turats om att programmera. Vårat schema har sett ut på så sätt att vii har jobbat en timme och sen tagit korta pauser på ca 15 minuter mellan omgångarna. Denna metodik har hjälpt oss att samarbeta effektivt och lösa problem tillsammans. Pauserna var också viktiga och hjäpte verkligen arbetet när vi fastnade med något.