Utvecklingsprojekt (UP) - Meddelandesystem

1 Inledning

Det här utvecklingsprojektet ska bidra till en grundläggande förståelse för:

- Användning av trådar och strömmar i samband med nätverkskommunikation
- Nätverkskommunikation och Client/Server-modellen
- Objektorienterad analys och design

1.1 Genomförande

Vi rekommenderar att utföra projektet i grupp (max 6 medlemmar per grupp). Ni anmäler era grupper på Canvas.

Det finns en **frågestund** inplanerad den **7/3 kl 13.15**.

1.2 Redovisning

Inlämning av uppgiften ska ske **senast torsdagen 10/3 kl 14.00** på Canvas. Redovisning äger rum måndag, onsdag och torsdag under vecka 11 (**14-17/3**). Var noga med att alla gruppmedlemmar är aktiva vid redovisningen. Redovisningsschema och granskande grupp publiceras under eftermiddagen den **11/3**.

Inlämningen sker genom att lämna in filen **DA343A_UP_aa.zip** på Canvas där **aa** ersätts med gruppens nummer. Filen ska innehålla följande:

1. Dokumentet ***DA343A_UP_aa.pdf*** vilket ska följa mallen som finns i filen *DA343A_UP_Rapportmall_VT22.docx*. Denna mall har utförligare instruktioner än nedan som ska följas. Dokumentet i stort:
   a. Innehåller en lista på gruppens medlemmar
   b. Uttrycker vad gruppens olika medlemmar har bidragit med i lösningen.
   c. En instruktion så att lärare och andra grupper kan kompilera och köra applikationen.
   d. Dokumentation i form av klassdiagram och sekvensdiagram ritade med något verktyg.
   e. Alla källkodsfiler som ingår i projektet. Källkodsfilerna ska vara kommenterade så att det rimligt lätt går att förstå olika delar av koden.
2. Katalogen ***src*** vilken ska innehålla alla källkodsfiler som ingår i projektet.
3. Övriga filer som behövs för att köra programmet (bildfiler, ljudfiler, textfiler,…)

Redovisningen sker gruppvis och under redovisningen deltar normalt ytterligare ett par grupper som åhörare och granskare. En grupp kan räkna med ca 10-15 minuter för att presentera sitt arbete. Alla gruppmedlemmar måste vara beredda på att svara på frågor om systemets design och implementation. Efter presentationen framför den granskande gruppen sina resultat av granskningen.

2 Beskrivning av uppgiften

Uppgiften går ut på att designa och implementera ett **chatt-system** där användare kan utbyta meddelanden. Ett meddelande består av text och bild. Naturligtvis ska det även gå bra med meddelanden med enbart text eller bild.

2.1 Funktioner som ska implementeras i chatt-systemet

- En klientapplikation utgör programvaran som en person (användare) använder i meddelandesystemet.
- En användare består av ett användarnamn och en användarbild (liten). Du kan utgå från att alla användarnamn är unika i systemet. En användare har alltid samma användarnamn och användarbild.
- Klientapplikationen består av ett grafiskt användargränssnitt och av funktionalitet för att kommunicera med en server.
- Servern exekveras på en dator och klientapplikationerna på andra datorer. All kommunikation mellan klienter sker via servern.
- Tre olika typer av kommunikation sker mellan klient och servern:
  - Vid uppkoppling skickas en användare (namn + bild) från klient till server
  - All övrig kommunikation består av meddelande-objekt eller subklasser till meddelande-objekt.

  Ett meddelande från en användare består av text och bild. Ett meddelande behöver dock inte alltid innehålla både text och bild. Varje meddelande från en klient ska innehålla:

  - Avsändare – en användare
  - Mottagarlista – en lista av användare
  - Text
  - Bild
  - Tidpunkt då meddelandet togs emot av servern. Tidpunkt ska lagras med uppgift om år, månad, dag, timme och minut. Denna information läggs till av servern.
  - Tidpunkt då meddelandet levererades till mottagaren. Se punkten ovan.

  En uppdatering av användare från servern innehåller alltid

  - Användare – den användare som just kopplat upp sig
  - Uppkoppladlista – en lista av uppkopplade användare

Servern ska klara av att:

- Ett stort antal användare (klienter) ansluter sig till meddelandesystemet. Vid anslutning identifierar sig användaren med användarnamn. En användarbild ska också överföras vid anslutningen.
- En användare (klient) avslutar sin anslutning till systemet.
- Uppdatera anslutna användare med en lista vilken innehåller samtliga anslutna användare. Uppdatering ska ske varje gång någon användare ansluter sig / avslutar sin anslutning.
- En person skickar ett meddelande till ett antal användare, de som är i mottagarlistan i ett meddelande.
- Lagra ett meddelande som inte kan levereras p.g.a. att mottagaren inte är ansluten till systemet. Då mottagaren ansluter sig till systemet ska lagrade meddelanden levereras.
- Logga all trafik i systemet till fil/filer på hårddisken. Via ett UI på serversidan ska man kunna se all trafik mellan två valda tidpunkter.

För klienten gäller att:

- En klient ska kunna ansluta sig till systemet, avsluta anslutning till systemet, skicka meddelande, ta emot och visa meddelande och visa anslutna användare.
- Ett användarnamn och en användarbild ska kunna anges av användaren före uppkoppling.
- En klient ska kunna lägga till en uppkopplad användare till *Kontakter*. Kontakter ska sparas på hårddisken då klienten avslutas och läsas in och visas i applikationen då den startats. Då användaren förbereder att skicka ett meddelande ska man kunna välja mottagare dels från kontakter och dels från uppkopplade användare.
- En person ska kunna skriva in text och välja bild på ett smidigt sätt (t.ex. med hjälp av JFileChooser). Lämpliga bildformat är jpg eller png. Texten kan vara oformaterad text.

2.2 Krav på implementationen av systemet

Kommunikationen mellan klient och server ska ske via **TCP**. Eftersom en bild ingår i användare och i meddelande kan det vara lämpligt använda ImageIcon-objekt. Kommunikationen ska ske via objektströmmar: **ObjectInputStream** och **ObjectOutputStream**.

Servern måste kunna kontakta klienter vid behov (då användarlistan ska uppdateras, då meddelande ska levereras…). Det innebär att servern måste hålla reda på alla anslutna klienter. En lämplig Client-klass kommer att vara nödvändig. Vilken data som lagras i ett Client-objekt beror på design av systemet.

I servern ska Client-instanserna lagras i en datastruktur (objektsamling) och operationer mot datastrukturen ska vara synkroniserade. Detta gäller även för lagringen av ej skickade meddelanden. I föreläsning 6 i Datakommunikation ges exempel på hur man kan hantera detta.

I bilaga 1 visas

- vad som krävs för att använda en User-klass som nyckel i en HashMap
- delar av ett par klasser med synkroniserade metoder och som har en HashMap som attribut.

Designen av systemet ska följa tanken om boundary, control och entity-klasser.

- Boundary-klasser har ingen eller mycket lite av logiken i systemet. Boundary-klasser känner endast till andra boundary-klasser och/eller control-klasser.
- Control-klasser ansvarar för den övergripande logiken i systemet och behöver känna till både boundary- och entity-klasser. Control-klasser behöver dock inte känna till alla boundary- och entity-klasser.
- Entity-klasser lagrar data i systemet och ansvarar för beräkningar och transformationer av denna data. Entity-klasser känner endast till andra entityklasser och eventuellt control-klasser.

2.3 Krav på designdokumentation av systemet

Systemets design ska dokumenteras med klassdiagram och sekvensdiagram.

Det ska finnas två klassdiagram – ett som visar klasserna i klienten och ett som visar klasserna i servern. Vissa klasser förekommer i bägge klassdiagrammen, t.ex. User och Message. För klassdiagrammen gäller att:

- Klassdiagrammen gemensamt visar alla klasser i systemet som ni själva skrivit och hur dessa är associerade till varandra. Är där standardklasser som ni anser förtydligar diagrammet så bör även dessa visas, detta kan exempelvis vara klasser som hanterar datastrukturer.
- Associationer mellan klasser ska visas med så lämplig associationstyp som möjligt. Multiplicitet ska visas för alla associationer där det kan tillämpas.
- Alla klasser ska vara angivna som boundary, control eller entity-klass på något sätt (kan visas på olika sätt med UML – välj ett av dessa).
- Klassdiagrammet ska endast visa attribut och operationer som ni anser behövs av tydlighetsskäl.
- Klassdiagrammet behöver inte visa alla klasser som används för att bygga upp GUIt. Det räcker om den klass som representerar själva fönstret visas i klassdiagrammet. Används flera fönster i GUIt ska alla fönsterklasser synas i diagrammet.

Gruppen ska också skapa ett visst antal sekvensdiagram som visar interaktionen mellan objekt vid exekvering. Varje person i gruppen är ansvarig för och ritar minst ett sekvensdiagram. För sekvensdiagrammen så gäller att:

- Sekvensdiagrammen behöver endast visa de objekt för GUIt som finns i klassdiagrammet.
- Sekvensdiagrammen ska visa alla meddelanden som skickas till något objekt av de klasser som finns i klassdiagrammet.

- Alla selektioner och iterationer i kod ni själva skrivit ska visas i sekvensdiagrammen.
- De anrop eller strukturer som sker internt i hjälpklasser för exempelvis datastrukturer, objektsamlingar eller kommunikation behöver inte visas. Anrop från något objekt till ett objekt av en hjälpklass måste visas men det behöver inte visas vad som sker inuti den metod som anropas i hjälpklassen.
- Om en eller flera trådar startas i användningsfallet kommer lösningen bestå av flera diagram, med ett separat sekvensdiagram för varje tråd

Varje person i gruppen ska vara ansvarig för ett sekvensdiagram. Oavsett gruppensstorlek så ska alltid sekvensdiagram 1-4 nedan göras. Om gruppen innehåller fler personer så görs ytterligare sekvensdiaram motsvarande antal personer i gruppen. Sekvensdiagrammen ska visa kodens struktur för följande fall:

1. Vad som sker på serversidan när en klient ansluter till systemet. Detta fall ska inkludera utskick av ej levererade meddelanden.
2. Vad som sker på klientsidan när en användare startar klienten. Detta fall ska inkludera inläsning av Kontakter.
3. Vad som sker på servern när en klient skickar ett meddelande. Detta fall ska inkludera utskick av meddelande till uppkopplade mottagare och lagring av meddelande till ej uppkopplade mottagare.
4. Vad som sker i klienten då servern skickar uppdatering av anslutna användare.
5. Vad som sker i servern då loggning mellan två tidpunkter ska visas i server UI.
6. Vad som sker på klientsidan när en klient avslutar sin anslutning. Detta fall ska inkludera att skriva Kontakter till hårddisken.

3 Granskning

Varje grupp som redovisar ska också granska en annan grupp. Vilken grupp som ska granskas och tillhörande filer publiceras på Canvas 11/3. Vid redovisning så har den granskande gruppen 5 minuter att muntligt presentera en sammanfattning av sin granskning efter den granskade gruppens presentation.

Vid granskningen bör följande frågeställningar undersökas:

- Kunde koden köras enligt instruktionerna i dokumentationsmaterialet?
- Fungerar systemet som avsett? Detta innebär att ni ska kunna utföra den enligt uppgiften specificerade funktionaliteten.
- Hur fungerar koden vid undantag? Testa att mata in felaktiga värden, trycka på knappar i fel logisk ordning och liknande.
- Är koden bra strukturerad? Är exempelvis indelningen i klasser logisk, är metoder och variabler tydligt namngivna, med mera.
- Innehåller dokumentationen det den ska enligt specifikationen ovan?
- Stämmer klassdiagram och kod överens?
- Stämmer klassdiagram och sekvensdiagram överens?
- Stämmer interaktionen i sekvensdiagrammen överens med koden?

Bilaga 1

För att använda *User*-objekt som nyckel i en *Map* är det lämpligt att lägga till nedanstående överskuggning av *hashCode* och *equals*. Om man gör det är det användarnamnet som är nyckel.

```
public class User implements Serializable { // Även användas i strömmar
    private String username;
    private ImageIcon image;

    // konstruktor, get-metoder, ...

    public int hashCode() {
        return username.hashCode();
    }

    public boolean equals(Object obj) {
        if(obj!=null && obj instanceof User)
            return username.equals(((User)obj).getUserName());
        return false;
    }
}
```

// Lagra klienter, i exemplet Client-instanser

```
public class Clients {
    private HashMap<User,Client> clients = ...

    // egna tillägg

    public synchronized put(User user,Client client) {
        clients.put(user,client);
    }

    public synchronized Client get(User user) {
        return get(user);
    }

    // fler synchronized-metoder som behövs
}
```

// Lagra listor med osända meddelanden, i exemplet Message-instanser

```
public class UnsendMessages {
    private HashMap<User,ArrayList<Message>> unsend = ...

    // egna tillägg

    public synchronized put(User user,Message message) {
        // hämta ArrayList – om null skapa en och placera i unsend
        // lägga till Message i ArrayList
    }

    public synchronized ArrayList<Message> get(User user) {
        //
    }

    // fler synchronized-metoder som behövs
}
```