Projektbeskrivning

IRC chatt-klient

2019-03-18

Projektmedlemmar:
Joseph Hughes <joshu135@student.liu.se>

Handledare:
Mikael Nilsson <mikni07@ida.liu.se>

Table of Contents

Planering

1. Introduktion till projektet

Detta projekt utgår från inspirationsprojektet ”IRC chatt-klient”.

Internet Relay Chat (IRC) är ett protokoll för chattmeddelanden. Protokollet använder sig av en klient-server-arkitektur. Det vill säga det finns en server som flera klienter kommunicerar med istället för att kommunicera mellan varandra. Protokollet tillåter att man skriver till kanaler, öppna meddelanden för alla på kanalen, skriver direktmeddelanden till andra användare och mycket mer.

2. Ytterligare bakgrundsinformation

Wikipedia-artikel ”Internet Relay Chat”: https://en.wikipedia.org/wiki/Internet_Relay_Chat.

RFC-2812 - IRC protokoll för klienter: https://tools.ietf.org/html/rfc2812

IRC-protokollet använder sig av det underliggande nätverksprotokollet TCP för att överföra information mellan klienten och servern. För att komma igång med en grundläggande klient behövs dock inte särskilt mycket kunskap om TCP förutom hur man öppnar och läser från en TCP-socket.

3. Milstolpar

#	Beskrivning
1	Bekantskap med protokollet. Läs igenom RFC-2812 för att förstå hur interaktionen mellan klient och server fungerar. Kolla på fungerande IRC-klienter och hur dem ser ut och används.
2	Anslut till server: Anslut till en hårdkodad IRC-server och registrera en användare via ett CLI.
3	Gå med i en kanal: Efter att ha anslutit till en server är nästa steg att gå med i en kanal på servern. Denna bit kan tills vidare vara hårdkodad.
4	Grundläggande användargränssnitt. Gå över från att testa i CLI till att skapa ett chattfönster med Swing och MiGLayout. Ännu tolkas inte svaren från servern utan alla visas ofiltrerat i chattfönstret.
5	Implementera händelser: Det är nu dags att tolka och formatera svaren som kommit från servern och specificera handlingar baserat på det. När någon skriver till användaren eller kanalen den är med i ska det visas som ett formaterat meddelande i chattrutan. Om någon ansluter sig eller kopplar ifrån kanalen skall även det framgå. Detta innefattar även servergenererade händelser.
6	Val av server: Det här är en av de första milstolparna där ni låter användarna ange vart de vill ansluta sig. Tänk på att användarna kan vilja byta vilken server de är ansluten till.
7	Skicka meddelanden: När man har anslutit till en server och en kanal är det bra att

	kunna skicka meddelanden till alla som är anslutna till kanalen.
8	Användbar klient: Klienten bör nu vara någorlunda användbar. Stanna upp på den här milstolpen och testa programmet för att se till att det fungerar korrekt. Fixa därmed eventuella problem.
9	Lista och gå med i kanaler: Ett första steg för att låta användaren välja vilken kanal hen ska gå med i på en server är att lista vilka som finns tillgängliga. Syftet med den här milstolpen är att lista alla kanaler som finns på den nuvarande servern. Tänk på att vilka kanaler som finns kan ändras med tiden varpå listan måste uppdateras.
10	Lista användare: Många kommandon som finns tillgängliga blir betydligt kraftfullare om man har möjligheten att lista alla användare. Det finns två alternativ för att lista användare. Den ena är att lista alla användare som finns på kanalen och den andra är att lista alla användare på servern. Båda kan vara önskvärda så den här milstolpen inbegriper båda.
11	Privata meddelanden: En vanlig funktion i chatt-klienter i allmänhet är att kunna skriva privata meddelande direkt till andra användare. Det går att göra i IRC också så därför ska klienten i den här målstolpen utökas med den funktionaliteten.
12	Away-status: IRC har stöd för att man ska kunna sätta sin status till ”Away” för att visa att man inte läser meddelanden av någon anledning (som kan anges). Det är något som kan vara av intresse i en chatt-klient. Den här milstolpen syftar till att låta användaren ställa in sin status till Away samt vilket meddelande som ska visas.
	Då jag jobbar ensam drar jag gränsen för vad jag tror att jag hinner med här. Klienten går att använda för att chatta i kanaler och enskilda användare. Viss önskbar funktionalitet går dock att implementera i mån om tid.
13	Bokmarkera servrar: Möjlighet att spara servrar för att snabbt kunna byta emellan dem.
14	Överföra filer mellan enskilda användare
15	Bjud in: IRC protokollet har möjligheten att bjuda in användare till kanaler (och därmed skapa nya ifall kanalen inte redan finns). Syftet med den här milstolpen är att ge användaren möjlighet att bjuda in en eller fler användare till en kanal.
16	Kasta ut: Steget efter att man kan bjuda in personer till en kanal är naturligtvis att kunna kasta ut dem igen. Det här bör även kunna göras på ett smidigt sätt vilket även är målet med den här milstolpen.

4. Övriga implementationsförberedelser

Projektets GUI kommer att implementeras med hjälp Javas standardbibiliotek Swing. Som layout manager använder jag enligt rekommendation MiG Layout.

IRC-protokollet är i sin natur asynkront. För att då kunna upprätthålla kommunikationen med servern och ständigt skicka/ta emot meddelanden samtidigt som jag tillåter användaren interagera med programmet kommer jag behöva använda mig av multi-threading. Min initiala

tanke är att jag har en huvudtråd som sköter själva interaktionen mellan användargränssnittet och 2 nätverkstrådar som skickar meddelanden och tar emot svar från servern.

5. Utveckling och samarbete

Jag arbetar ensam och har därför inte mycket att tillägga under den här rubriken.

Slutinlämning

6. Implementationsbeskrivning

6.1. Milstolpar

I slutinlämningen hann jag med att implementera alla milstolpar till och med 12.

1. Bekantskap med protokollet

Genomfört. Detta hände också till viss del över projektets gång. Jag kollade även på den fungerande IRC-klienten AdiIIRC för att bekanta mig ytterligare.

2. Anslut till server

Genomfört.

3. Gå med i en kanal

Genomfört.

4. Grundläggande användargränssnitt

Genomfört.

5. Implementera händelser

Genomfört.

6. Val av server

Genomfört.

7. Skicka meddelanden

Genomfört.

8. Användbar klient

Genomfört.

9. Lista och gå med i kanaler

Genomfört. Det går att byta kanal men programmet stödjer i nuläget bara en kanal åt gången.

10. Lista användare

Genomfört för både kanal och server. Det finns dock vissa IRC-servrar som inte returnerar något alls när man frågar om alla namn på servern. Det går t.ex. att lista alla på irc.freenode.net men inte irc.mibbit.net.

11. Privata meddelanden

Genomfört. Det går dock inte att stänga en påbörjad konversation.

12. Away-status

Genomfört.

13. Bokmarkera servrar

Ej genomfört.

14. Överföra filer

Ej genomfört.

15. Bjud in

Ej genomfört.

16. Kasta ut

Ej genomfört.

6.2. Dokumentation för programstruktur, med UML-diagram

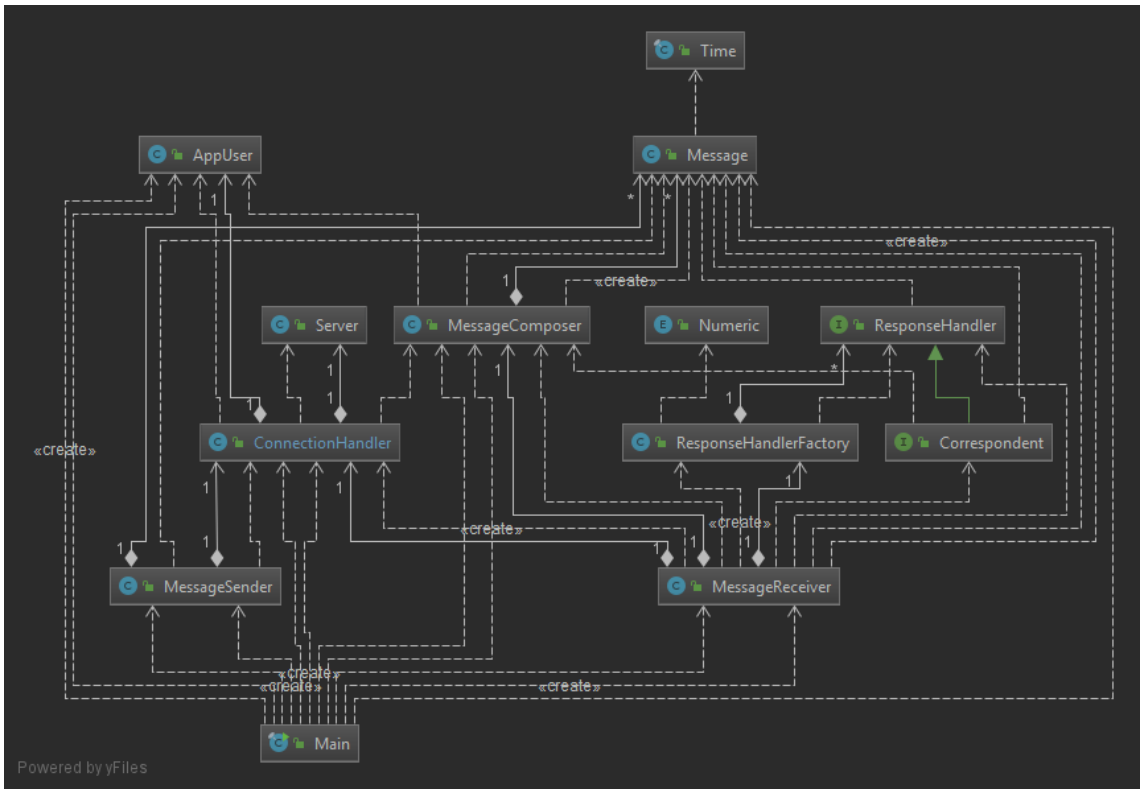
6.2.1 Övergripande programstruktur

Programmet startas av en statisk Main-klass som sätter igång några av applikationens grundstenar. Däribland ChatViewer som bygger användargränssnittet, ConnectionHandler som sköter nätverkssocketen och MessageReceiver/MessageSender som tar emot/skickar servermeddelanden på separata trådar.

Projektet delas upp i paketen backend, frontend, models och util.

- **Backend** kommunicerar med IRC-servern och initialiserar handlingar utifrån det. Handlingarna placeras i underpaketet **response**.
- **Frontend** står för det grafiska användargränssnittet som användaren interagerar med.
- **Models** nyttjas av båda de föregående paketen och definierar modeller för saker som meddelanden, användare och kanaler, bland annat.
- **Util** innehåller hjälpklasser för loggning, tidbehandling och tolkning av serverkommandon.

6.2.2 Översikt backend



Backend-paketet har två huvudsakliga funktioner. Dels att skicka information till servern och dels att ta emot svar från servern. Dessa funktioner delegeras till Runnable-klasserna MessageReceiver och MessageSender som initialiseras när programmet startas och körs asynkront tills programmet stängs av.

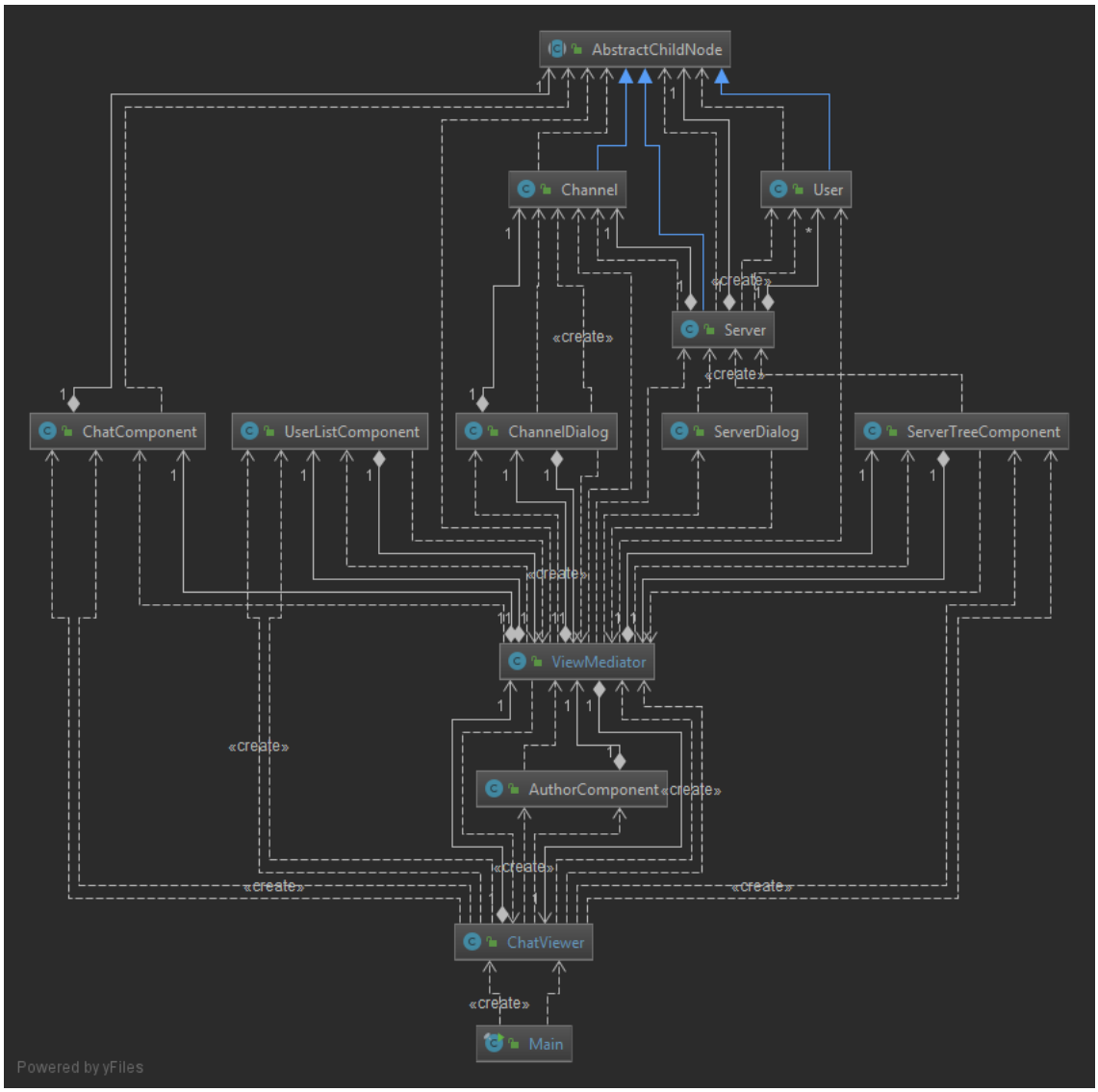
Den gemensamma nämnaren mellan MessageReceiver och MessageSender är aggregat-förhållandet till ConnectionHandler, vars uppgift är att läsa/skriva meddelanden till en TCP-socket samt hålla koll på vilken server socketen är kopplad till och vilken användare (AppUser) som registrerats där. Server-objektet innehåller även en del information om underliggande kanaler och privata konversationer men det har mer att göra med front-end och diskuteras i 6.2.3 och 6.2.4.

Message-klassen fungerar som ändpunkt för många associationer i backend-paketet. Ett meddelande har en strikt definition i RFC-2812-protokollet (https://tools.ietf.org/html/rfc2812#section-2.3). När ett meddelande således tas emot tolkas det med hjälp av reguljära uttryck och delas upp i lämpliga fält som lätt kan kommas åt via objektets publika metoder. När ett meddelande skickas iväg av klienten används hjälpklassen MessageComposer för att korrekt formatera det önskade meddelandet och köa det för att skickas till servern.

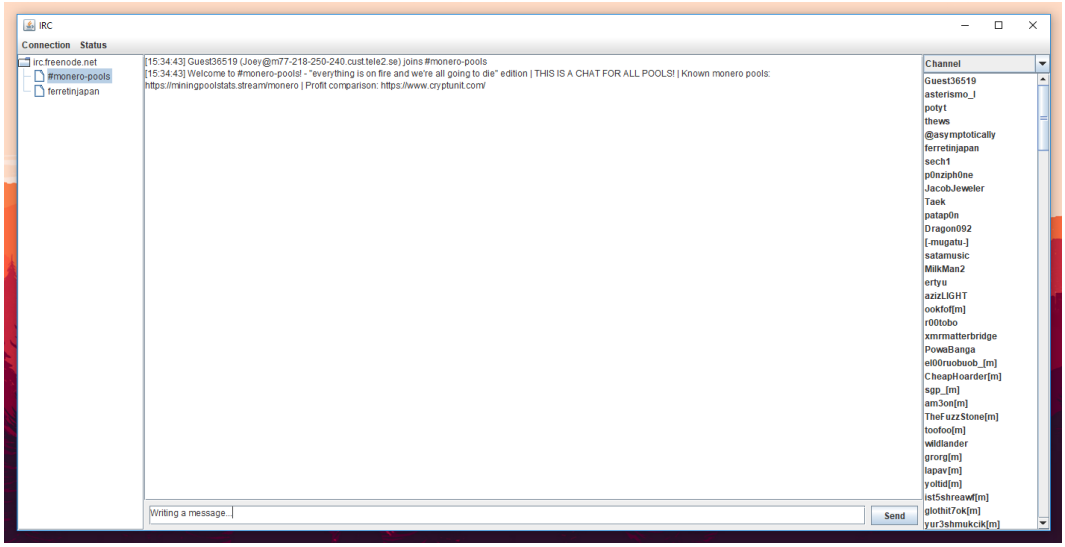
För att kunna köa meddelanden från MessageComposer till MessageSender, som körs på olika trådar, initialiseras båda objekten med en LinkedTransferQueue som aggregerar instanser av Message. Meddelanden läggs till i kön genom MessageComposers ”queueMessage”-metod och behandlas sedan av MessageSender i ordningen FIFO (first-in-first-out).

Hur MessageReceiver behandlar inkommande meddelanden förklaras i detalj under 6.2.5 och 6.2.6. I korthet finns det en HashMap mellan meddelandets RFC-kommando och en lämplig åtgärd som ska vidtas som följd av det.

6.2.3 Översikt frontend



OBS: LÄNK TILL STÖRRE BILD: https://i.imgur.com/N81U8qh.png



Frontend-paketet står applikationens grafiska gränssnitt som skapats med hjälp av Swing och MiGLayout. Det huvudsakliga fönstret innehåller en JFrame bestående av 4 egendefinierade Swing-komponenter och en JMenuBar. Därtill finns det två dialoger som används för att byta server såväl som kanal.

ChatViewer är ansvarig för att skapa hela huvudfönstret, innehållande dess komponenter och menyn, och placera dem i en JFrame. ChatViewer tar därefter endast ansvar för menyn och överlåter kontroll över gränsnittet och skapandet av dialoger till ViewMediator.

Komponenter:

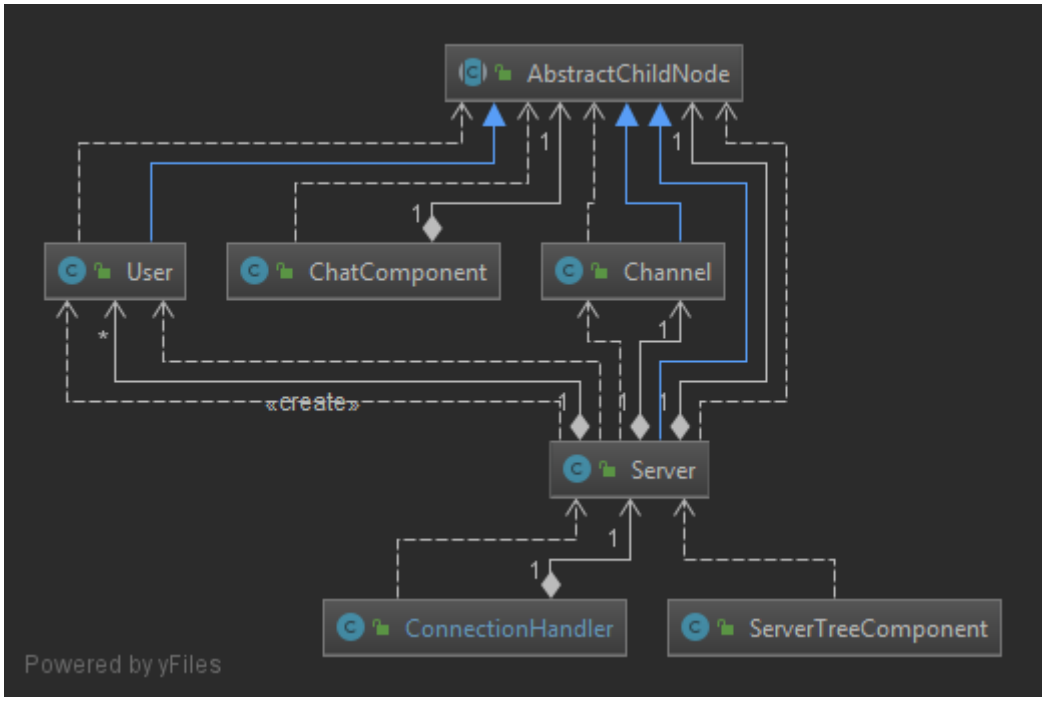
- **AuthorComponent** är ansvarig för den nedre delen av skärmen där användaren kan författa nya meddelanden och validerar att meddelandena innan de skickas till servern.
- **ChatComponent** innehåller en skrollbar yta med texten som skrivits i den valda servern, kanalen eller privata konversationen.
- **UserListComponent** innehåller en skrollbar lista med användare. Det finns även en ComboBox där användaren kan välja om den ska lista alla på servern eller bara de som är i den nuvarande kanalen/konversationen.
- **ServerTreeComponent** är ansvarig för att visa trädhierarkin, vars noder består av servern och kanalerna/konversationerna som tillhör den. Dessa ärver alla av typen AbstractChildNode. När man klickar på de olika noderna uppdateras ChatComponent och UserListComponent för att reflektera ens val. Mer om serverträdet i 6.2.4.

Dialogerna öppnas genom menyn under ”Connection”.

- I **ServerDialog** anger användaren önskad IRC-server och registrerar sin identitet.
- **ChannelDialog** listar alla valbara kanaler på servern. Då denna behöver invänta listan av alla kanaler från servern implementerar den Runnable för att kunna köras på en separat tråd.

För att minska beroenden för frontend-klasser sinsemellan abstraheras interaktioner mellan dem till ViewMediator. De objekt som behöver interagera med frontend-klasser tilldelas en referens till ViewMediator som i sin tur interagerar med dem direkt. Till exempel kallar ChannelDialog på en metod i ViewMediator när användaren vill byta kanal. ViewMediator ser då till att meddela servern att byta kanal, och uppdaterar ServerTreeComponent, ChatComponent och UserListComponent för att reflektera ändringen.

6.2.4 Översikt serverträd



Vi tittar lite närmre på serverträdet dess noder. ServerTreeComponent innehåller ett JTree vars TreeModel är en hierarki av DefaultMutableTreeNodes. Server, Channel och User ärver alla från AbstractChildNode som har ett fält av typen DefaultMutableTreeNode, ett namn och en chatthistorik som fält.

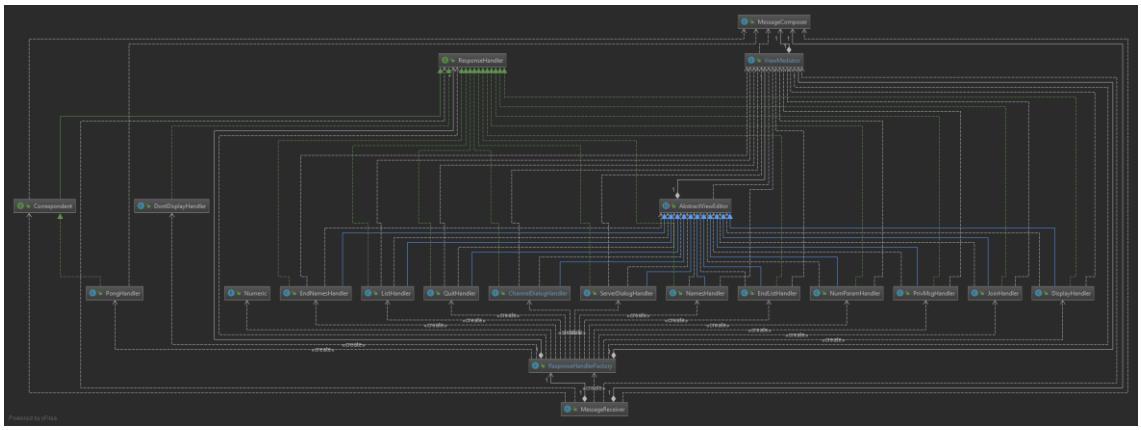
Server fungerar som ingångspunkten för både UI-komponenten ServerTreeComponent och nätverkshanteraren ConnectionHandler då den innehåller serverns URL och port samtidigt som den ärver från AbstractChildNode. Utöver funktionaliteten den får från AbstractChildNode hanterar den sina barn som kan bestå av en kanal och flera privata konversationer (instanser av User).

ServerTreeComponent har själv inget ansvar över vilka kanaler eller privatkonversationer som den innehåller, utan delegerar den uppgiften till Server.

När man väljer en AbstractChildNode i ServerTreeComponent registreras det som den aktiva vyn i ChatComponent (för att visa chatthistoriken) och Server (för att ens meddelanden ska skickas till rätt destination).

6.2.5 Översikt ResponseFactory

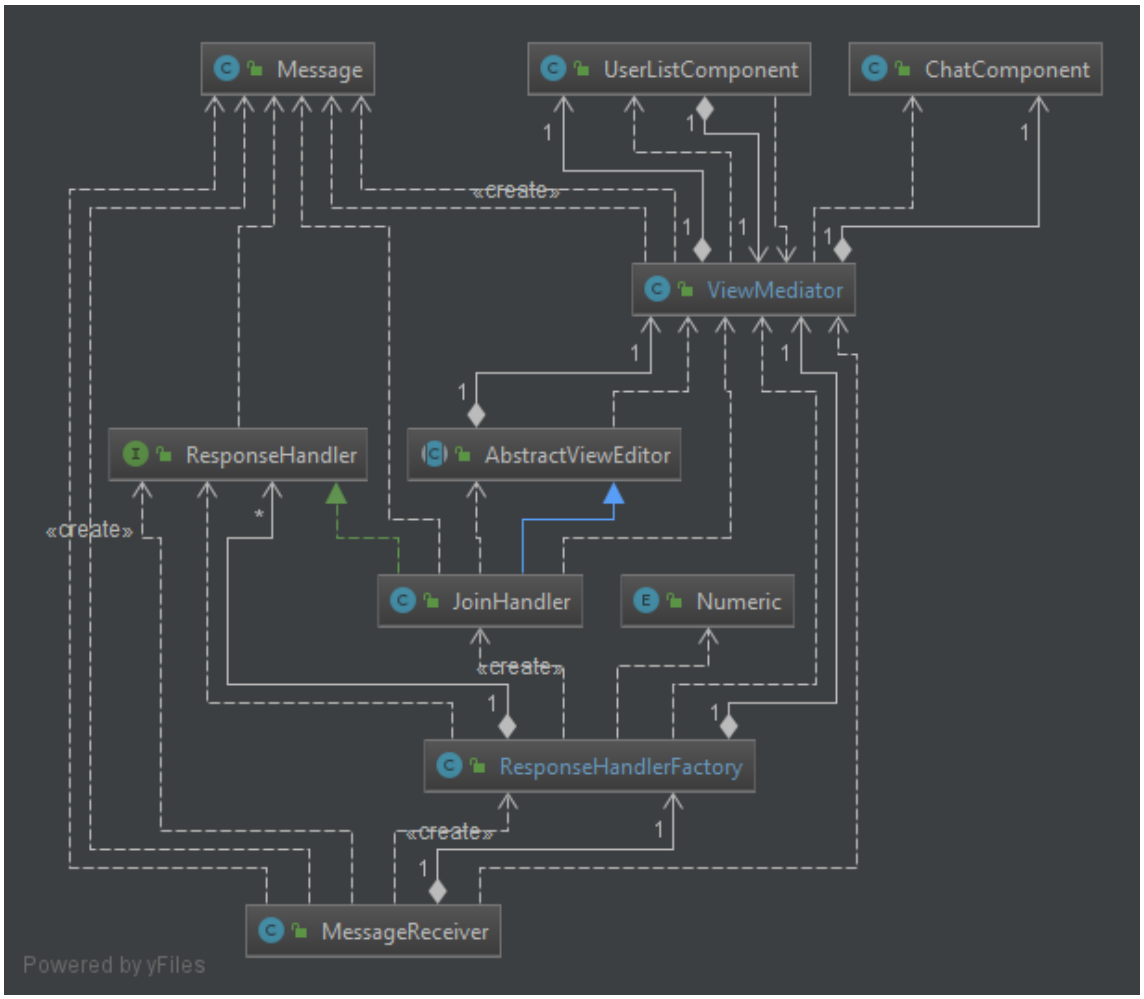
OBS: LÄNK TILL STÖRRE BILD: https://i.imgur.com/CrcR3vu.png



För att på ett flexibelt sätt kunna hantera en mängd med kända men varierande serversvar använde jag mig av Factory-mönstret. Enligt specifikation innehåller varje meddelande en ”command”-del som kan vara ett ord eller en tresiffrig kod (i programmet definierade i enumen Numeric). I ResponseHandlerFactory definieras en HashMap mellan kommandot och en instans av en passande ResponseHandler. Om kommandot inte är definierat i mappen är default-handlingen en DisplayHandler som visar meddelandet i ChatComponent.

MessageReceiver instantierar en ResponseHandlerFactory som beroende på det inkommande svaret returnerar en instans av den önskvärda hanteraren. Beroende på vad för åtgärd som behöver vidtas på svaret behöver objektet olika nivåer av privilegier. De klasser som behöver kunna göra ändringar i gränssnittet får tillgång till ViewMediator genom att ärva från AbstractViewEditor. Om en klass behöver kunna svara tillbaka till servern implementerar den Corresponder, som i sin tur implementerar ResponseHandler. Klasser som är Corresponders får tillgång till MessageComposer som har rätt att köa meddelanden till MessageSender.

6.2.6 ResponseHandler exempel – ChannelDialog



Här är ett exempel på hur ett inkommande kommando påverkar programmet från att det tas emot till att det påverkar gränssnittet.

En användare går med i den nuvarande kanalen. MessageReceiver tar emot en rad text från servern. Den strängen konverteras till ett Message objekt. Meddelandets kommando (”JOIN”) ges som argument till ResponseHandlerFactory som i sin return returnerar en JoinHandler. JoinHandler tolkar meddelandet och extraherar användarens namn. JoinHandler kallar på AbstractViewEditors referens till ViewMediator (som är protected) för att lägga till användaren i UserListComponent och skriva i ChatComponent att användaren har gått med i kanalen.

6.3. Användning av fritt material

- Alla klasser som ingår i **Java 11**.
- **Miglayout** (version Swing 5.2)

  https://mvnrepository.com/artifact/com.miglayout/miglayout-swing/5.2
- **Apache Commons Lang** (version 3.8.1) – specifikt StringUtils och ExceptionUtils
- https://mvnrepository.com/artifact/org.apache.commons/commons-lang3/3.8.1
- Jag har **inte** använt mig av Gson i projektet.

6.3.1. Kodsnuttar från nätet. Mer info finns i Javadocs

- **util.borrowedcode.Numeric**

  Baserad på https://stackoverflow.com/a/8490188/4400799

Anpassad och konverterad från C# till Java av mig själv.

- **util.borrowedcode.BriefLogFormatter**

  Kopierad från https://www.javalobby.org//java/forums/t18515.html

- **frontend.ChatComponent.scrollSetupborrowedcode**

  Kopierad från https://stackoverflow.com/a/39410581

6.4. Motiverade designbeslut med alternativ

6.4.1 Factory-mönster

Till skillnad från många andra projekt som jag har genomfört är en IRC-klient i stora drag baserad på en eventorienterad programstruktur. När klienten skickar en önskad handling till servern finns den ett stort potentiellt tidsdelta mellan begäran och dess svar. Samtidigt är det inte heller garanterat att servern bevarar den egna kronologin i sina svar.

I min första implementation gjorde jag så att klienten skickade iväg ett meddelande och sedan väntade på ett förväntat svar innan programmet gick vidare. Detta visade sig inte fungera mycket längre än registreringen av en statisk användare. Jag konsulterade därefter Jonas Kvarnström som tipsade om att skapa en response factory som mappar kommandon till arvtagare till en abstrakt klass.

Detta tycker jag var en bra lösning då det går att göra asynkront. Klienten kan alltså skicka iväg ett meddelande och låta en annan del av programmet ta hand om utfallet. Fabriken är dessutom lätt att utöka när man implementerar ytterligare funktionalitet.

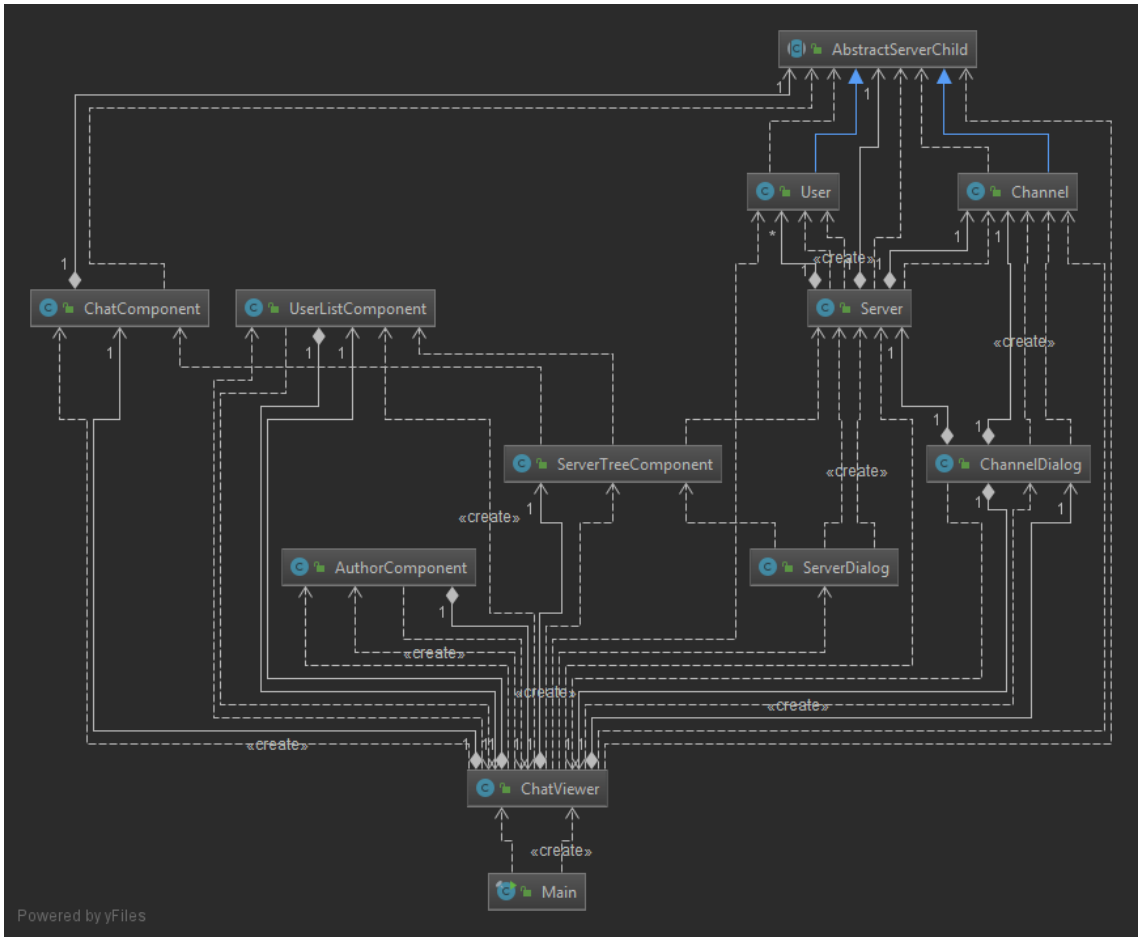
När jag först implementerade fabriken tilldelade jag maximal access till alla ResponseHandlers. För att minska tillgången till klasser i programmet delade jag upp det i 3 nivåer. ResponseHandlers som bara har tillgång till det inkommande meddelandet, Corresponders som dessutom kan svara på meddelandet, och AbstractViewEditors som har rätt att ändra i användargränssnittet genom ViewMediator.

6.4.2 GUI-komponenter

Tetris-projektet var en simplare applikation där i pincip hela användargränssnittet rimligt kunde begränsas till en komponent. Då chattprogrammets gränssnitt har flera naturligt diskreta delar har jag försökt dela upp det i logiskt definierade komponenter för att det inte ska konsolideras i en stor klass. Skapandet av huvudfönstret delegeras till ChatViewer men därefter har varje diskret del sin egen komponent. De kommunicerar därefter med varandra genom ViewMediator.

6.4.3 Mediator-mönster

Det var ganska långt in i projektet jag valde att använda mig av Mediator-mönstret för mina frontend-klasser. Under projektet hade jag försökt att undvika att centralisera logiken i programmet alltför mycket. Detta komplicerade strukturen av GUI-komponenterna och resulterade i att de innehöll referenser till de andra komponenter som dem påverkade. Se nedan UML-diagram. Därmed sjönk moduläriteten och jag behövde ofta anpassa klassernas konstruktorer för att ändra på funktionaliteten. Idén med ViewMediator är att förenkla flödet mellan komponenterna samt minska dependencies och repetition av kod. Nu sköter varje komponent uppgifter relaterade till sig själv och om de behöver komma åt andra komponenter gör dem det genom ViewMediator. Samma sak gäller för backend-klassernas åtkomst till komponenterna.



6.4.4 Caching av kanaler & användare

Stora IRC-servrar kan ha tusentals kanaler och användare. När man frågar efter alla kanaler/användare kan det därför ta väldigt lång tid då dessa delas upp i flera meddelanden då protokollet specificerar 510 karaktärer per meddelande. Detta gör det önskvärt att lagra undan dessa listor istället för att fråga servern igen när användaren t.ex. öppnar kanalbläddraren.

Del av kraven på mina milstolpar var dock att dessa skulle uppdateras då kanalerna såväl som användarlistan kunde komma att uppdateras. Rimligtvis skulle man kunna lägga in en uppdaterings-knapp för dessa två listor men det tvingades jag försumma på grund av tidsbrist.

Vad jag istället gjorde var att för användarlistan undvika att uppdatera den om dess kontext förändras. Om användaren exempelvis tittar på en konversation och har valt att visa hela servern i listan – och sedan byter konversation – förstår programmet att det inte är av någon nytta att uppdatera listan.

Kanalbläddraren tar betydligt längre tid att fylla och kan inte ta nytta av kontext-logiken. För att användaren inte ska behöva vänta så länge på att den fylls valde jag att uppdatera den så fort kanalnamnen tas emot av klienten istället för att vänta på att alla anlänt innan dem visas.

6.4.5 Skillnad på AppUser och User

I programmet finns två klasser som representerar användare – AppUser och User. Dessa hade semantiskt kunnat vara samma klass men jag valde att separera på dem då de har helt skilda syften. AppUser är håller helt enkelt koll på vilken användare som applikationen har registrerat på servern. Detta innefatter dess nickname, real name och username samt mode som exempelvis indikerar att användaren är osynlig. User är en annan användare på servern som man kan kommunicera med. Det är inte viktigt vad dess real name och username är då endast nickname visas i som författare av ett meddelande. Samtidigt behöver User vara en

AbstractChildNode för att visas i ServerTreeComponent och ChatComponent.

6.4.6 Upplägg serverträd

Det gick in en del tankekraft i att bestämma hur jag bäst skulle strukturera serverträdsstrukturen. Detta huvudsakligen då den består av två diskreta delar. Dels finns det en logisk struktur i att man kopplar upp sig till en server och tar del av olika diskussioner på den. Både servern och diskussionerna måste ha sin egen vy för att kunna visa en chatt. Därtill ska hierarkin också representeras grafiskt i serverträdet till vänster i användargränssnittet. Det hade säkert gått att strukturera det på ett sätt som separerar logiken från den grafiska representationen men då projektbedömningssidan (sektion 11. Korrekta designmönster) avråder från MVC-mönstret bestämde jag för att konsolidera det. Jag funderade också på om det var nödvändigt för Server att ärva från AbstractChildNode men bestämde mig för att göra det för att kunna bevara historik som kommer från servern, men inte någon specifik konversation (ex. Message of the Day).

6.4.7 MessageComposer icke-statisk

Från början var MessageComposer en statisk hjälpklass. Det hade den kunnat fortsätta vara men jag bestämde mig för att det skulle vara MessageComposers jobb att köa meddelanden till MessageSender. Därmed behövde de dela en instans av en LinkedTransferQueue. Det kändes onödigt att skicka med den instansen till alla klasser som behövde köa meddelanden genom MessageComposer och därför kapslade jag in den i en instans av MessageComposer som skickas runt istället. En bättre lösning som jag inte kollade utforskade speciellt djupt var att ha en eventlistener som MessageSender prenumerar på istället för att ha den körandes asynkront för evigt.

6.4.8 Inkapsling av ViewMediator i AbstractViewEditor

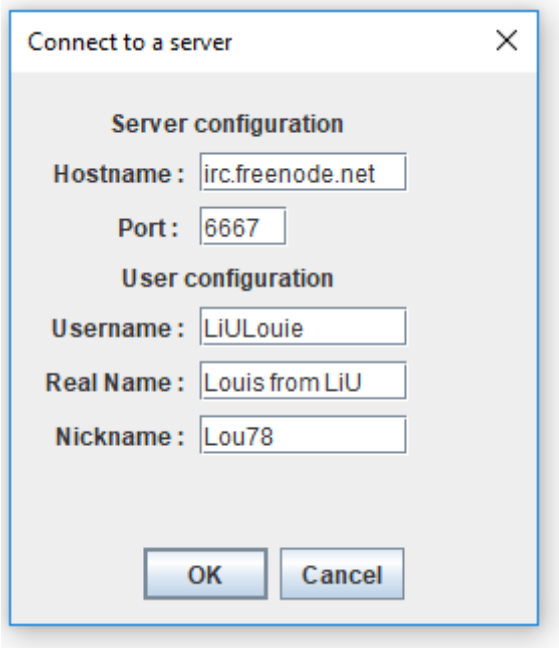
Tidigare gjorde jag så att AbstractViewEditors fält som innehöll ViewMediator var privat istället för protected. Istället för att använda ViewMediators metoder direkt fick arvtagare av AbstractViewEditor komma åt ViewMediator genom protected metoder i AbstractViewEditor. Detta gjorde däremot att jag fick skriva extra kod som definierade metoder som helt enkelt bara körde andra metoder. Jag bestämde mig för att göra bort med detta och göra ViewMediator protected för att undvika repetitiv kod. De enda metoderna som definieras i AbstractViewEditor nu är sådana som flera ResponseHandlers använder sig av. Exempelvis finns det 11 stycken ResponseHandlers som behöver skriva meddelanden till en kanal och då kan de använda sig av metoden displayChannelInfoMessage.

6.4.9 Numeric Enum

I min ResponseHandlerFactory finns det en metod setMap() som definierar vilket ResponseHandler-objekt som ska mappas till vilket kommando. Vissa av kommandona är ord som ”PING” eller ”PRIVMSG” men andra är 3-siffriga strängar. Till exempel indikerar ”375” ”RPL_MOTDSTART” vilket är början på serverns Message of the Day. Jag hade kunnat skriva dessa siffror direkt i metoden men valde istället för att skapa ett enumen Numeric där namnet hade varit ”RPL_MOTDSTART” med ett fält ”375”. Javadoc:en innehåller dessutom formatet på meddelandet från servern. Denna lösning gör det mycket lättare att se vilken ResponseHandler alla sifferkoder skapar.

7. Användarmanual

7.1 Anslutning till server

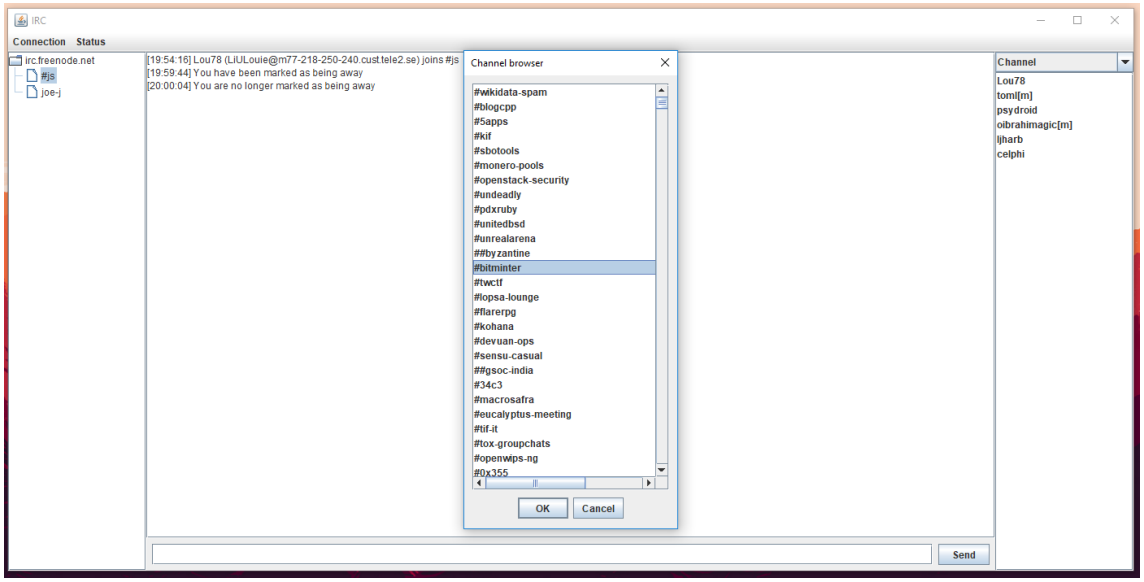


När du först öppnar upp programmet bemöts du av en dialog där du väljer vilken server du vill ansluta till. Du måste även ange 3 namn som används för att representera dig på servern. Du kan när som helst byta server igen genom att trycka Connection -> Server configuration i menyn.

Ditt Nickname är vad användarna på servern kommer att associera dig med när du skriver. Username och Real Name används för det mesta i bakgrunden men krävs för att registrera dig på servern.

7.2 Välja kanal

Större bild: https://i.imgur.com/uOXJC1z.png

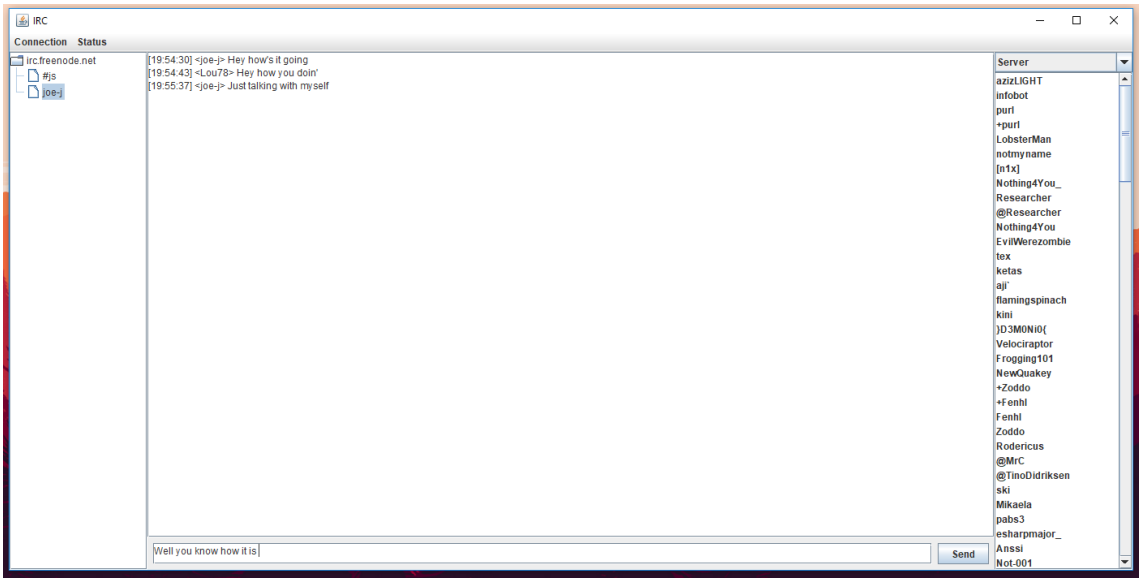


När du anslutit dig till en server är det dags att välja kanal. Du kan bara välja en kanal åt gången. Alla och alla tillgängliga kanaler finns att välja i kanalutforskaren som går att hitta i menyn under Connection -> Channel browser.

Väl ansluten till en kanal går det att se alla användare i kanalen i listan till höger. Du kan även välja att visa alla användare på servern. Observera att inte alla servrar har den funktionaliteten.

7.3 Privata meddelanden

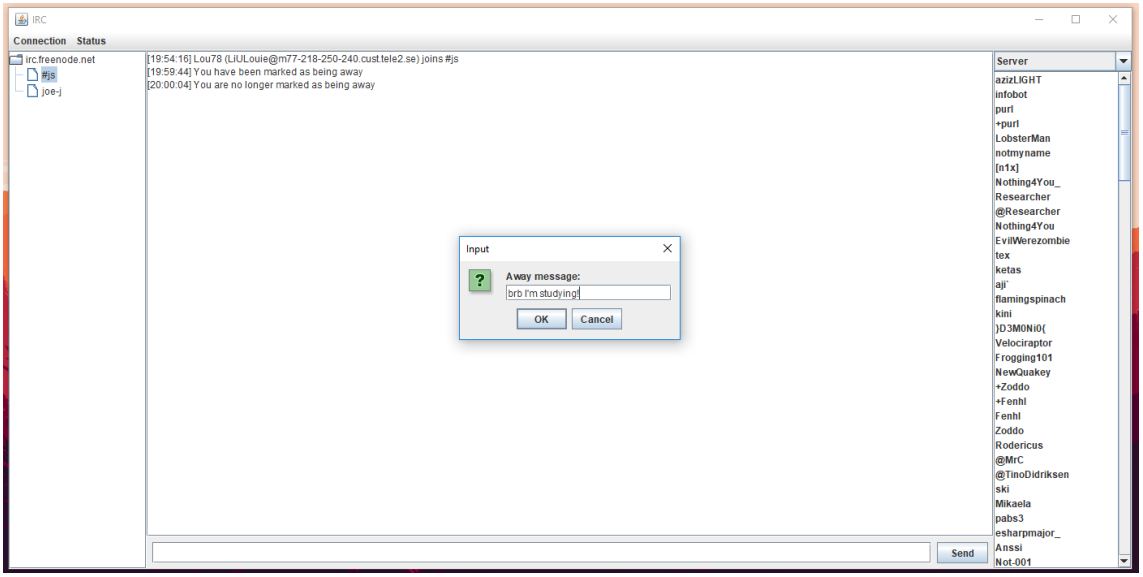
Större bild: https://i.imgur.com/VnSr77i.png



Genom att dubbelklicka på en användare i användarlistan kan du starta en privat konversation med denne. Du kan starta hur många privata konversationer som helst och byta mellan dem genom trädvyn på vänster.

”Away”-meddelande

Större bild: https://i.imgur.com/1x3a3GN.png



Behöver du gå iväg från datorn en stund finns möjligheten att sätta ett ”Away”-meddelande. Om någon försöker nå dig får de automatiskt meddelandet du angett.

8. Utvärdering och erfarenheter

- *Vad gick bra? Mindre bra?*
  - Jag tycker att jag i slutändan uppnått en bra struktur i mitt program. Det tog lite tid och trial-and-error att förstå sig på protokollet och hur jag skulle strukturera vissa saker på ett logiskt sätt.
  - Det har gått bra att jobba ensam tycker jag. Vi har haft så många andra projektkurser under denna period att det är skönt att ha något man kan göra på sin egen tid. Jag gissar att jag lagt mer än 80h på projektet men så får det

vara.
    - Jag känner att det inte alltid gick så bra att hitta motiveringar till alla kodinspektioner. Det känns som jag i vissa delar av projektet skrivit sämre kod för att undvika en specifik inspektion.
- *Vilket material och vilken hjälp har ni använt er av? Har ni gått på föreläsningar? Läst boken? Letat på nätet? Gått på handledda labbar? Ställt många frågor? Vad har ”hjälpt” bäst? Vi vill gärna veta för att kunna vidareutveckla kurs och kursmaterial åt rätt håll!*
    - *Jag gick inte på de flesta föreläsningarna då jag programmerat en del i Java och C# innan. Däremot tycker jag att det varit intressant med extraföreläsningarna.*
    - *Jag har inte heller behövt ställe speciellt mycket frågor på labbpassen men när jag haft dem har jag fått det svaren jag behöver.*
    - *Den finns väldigt många bra Java-resurser online tycker jag. Jag har använt allt från StackOverflow, TutorialPoint och Oracles egen dokumentation (speciellt denna har varit bra för Swing http://web.mit.edu/6.005/www/sp14/psets/ps4/java-6-tutorial/components.html)*
- *Har ni haft någon nytta av projektbeskrivningen? Vad har varit mest användbart med den? Minst?*
    - Det har varit skönt att ha en förbestämd lista med milstolpar för att veta vad man ska göra näst. Eftersom IRC-chatten var ett fördefinierat projekt har jag kunnat vara trygg i vad som är en bra omfattning för projektet.
- *Vad har varit mest problematiskt, om man utesluter den programmeringstekniska delen? Alltså saker runt omkring, som att hitta ledig tid eller plats att vara på.*
    - Tidshorisonten för projektet. Jag tycker att projektet bör ha suttit igång tidigare in i perioden. Den här kursen har varit väldigt baktung för min del.
- *Vilka tips skulle ni vilja ge till studenter i nästa års kurs?*
    - Börja tidigt och ha en plan! Kolla igenom projektbedömningssidan innan ni börjar för att få en bra idé för vad som krävs för önskat betyg.
- *Har ni saknat något i kursen som hade underlättat projektet?*
    - Fler exempelprojekt som inte är spel. Nu kunde jag i princip välja mellan Chattprogrammet och Kalenderprogrammet.