# Annoteringsverktyg för Korpusarbete - användarapplikation

Helena Ihrfors

## Sammanfattning

För utveckling inom datorstödd språkgranskning av naturliga språk såsom svenska krävs att man har stora mängder text. Till texten vill man ofta ha tilläggsinformation såsom meningsgränser, stycken och ordklass för varje ord. Att lägga till information kallas för att annotera texten och resultatet blir en korpus – en stor annoterad textmängd. Denna korpus kan sedan användas inom språkforskningen för att erhålla effektivare språkverktyg.

Syftet med detta examensarbete är att ta fram ett annoteringsverktyg som minimerar arbetet vid rättning och underhåll av korpus. Min uppgift gick ut på att utforma och implementera en användarapplikation till annoteringsverktyget innehållande ett grafiskt användargränssnitt, ett API som döljer nätverkskommunkationen för externa användare och en server som hanterar kommandon från gränssnittet.

I rapporten undersöker jag olika lämpliga utvecklingsmiljöer för verktyget samt förklarar varför vissa lösningar och metoder slutligen valdes. Jag tittar också på hur ett grafiskt användargränssnitt bör konstrueras för att det skall bli användarvänligt.



## Abstract

*Construction of an annotating tool, the user application part*

During the development of computer grammar checking for natural languages like Swedish, large quantities of text are collected. To each word in the text additional information is added like wordclass, sentence delimiter and part-of-speech. To add information is called to annotate the text and the result becomes a corpus – a big annotated text. This corpus is often used in language technology research to obtain effective grammar checking programs.

The purpose of this master's project is to construct an annotation tool that can handle correction and maintenance of corpus. My task was to construct and implement a user application for the annotation tool that contains a graphical user interface, an API that hides the network communication for external users and a server that manages commands from the interface.

In this report different development environments are examined for the tool and we explain why different solutions finally were selected. The end result is also described in this report. The report also discusses how a graphical user interface can be constructed in order to obtain good usability characteristics.

# Förord

Denna uppsats utgör mitt examensarbete i datalogi vid institutionen för Numerisk analys och Datalogi (Nada) vid Stockholms universitet. Denna magisteruppsats är en del av ett större examensarbete som delats upp mellan mig och Andreas Aarflot. Arbetet har utförts på Nada med Johnny Bigert som handledare och professor Stefan Arnborg som examinator. Förutom dem vill jag också framföra ett tack till Björn Eiderbäck, Jonas Sjöbergh och Viggo Kann.

# Innehållsförteckning


1 Bakgrund och mål ..... 9

1.1 Granskasystemet ..... 9

1.2 Korpus ..... 10

1.3 Uppgiften ..... 11

1.4 Tillvägagångssätt ..... 12

2 Utvecklingen ..... 14

2.1 Användar- och uppgiftsanalys ..... 14

2.2 Utvecklingsmiljö ..... 15

2.2.1 Val av exekveringsmiljö ..... 16

2.2.2 Applikationsarkitektur ..... 18

2.2.3 Kommunikationsprotokoll ..... 20

3 Systemprinciper ..... 25

3.1 Designriktlinjer för grafiskt gränssnitt ..... 25

3.2 Designmönster ..... 26

4 Lösningen ..... 28

4.1 Lösningstaktik ..... 28

4.2 Systemmodell ..... 29

4.3 API ..... 31

4.4 Nätverkskommunikation ..... 32

4.5 Det grafiska gränssnittet ..... 34

4.5.1 Inloggningsfönster ..... 34

4.5.2 Huvudfönster ..... 35

4.5.3 Editera korpus ..... 36

4.5.4 Skapa korpus ..... 37

4.5.5 Välja korpus att köra mot ..... 38

4.5.6 Regelfönster ..... 39

4.5.7 Visa sparade regler ..... 40

4.5.8 Svarsfönster ..... 41

4.5.9 Loggfil ..... 42

4.5.10 Modal meddelandebox ..... 44

4.6 Server ..... 45

5 Körning av prototypen ..... 47

6 Slutsats ..... 49

7 Litteraturförteckning ..... 50

Bilagor.....52
Bilaga A: Användarfall.....52
Bilaga B: API.....59
Bilaga C: DTD.....65
Bilaga D: Klassdiagram.....68

# 1 Bakgrund och mål

Ordbehandlingsprogram är idag en av de mest använda datortillämpningarna. Datorn som skrivverktyg används inom många olika områden. Detta har lett till ett ökat behov av språklig datorstödd granskning. För att få fram nya språkgranskningsprogram finns vid Nada ett forskningsprojekt som heter Grammatikgranskningsprojektet. Inom detta projekt utvecklas bland annat ett språkgranskningsprogram kallat Granska. Granska är ett program utvecklat för datorstödd språkgranskning av svensk text. Uppgiften med mitt examensarbete var att ta fram ett nytt experimentellt verktyg för utveckling inom detta projekt. För att förstå uppgiften krävs en beskrivning av Granskasystemet och dess omgivning. Jag börjar därför med en kort beskrivning av dessa för att ge en tillräcklig bakgrundsbild.

## 1.1 Granskasystemet

Granska är ett experimentellt program utvecklat för datorstödd språkgranskning utvecklat på Nada. Granska arbetar idag på texter och försöker att hitta olika typer av fel i dessa. Exempel på typer av fel som programmet hittar är t.ex. stavfel, felaktigt skrivna tecken, stilavvikelser och grammatiska fel. För att hitta fel i texten använder sig Granska av en uppsättning regler. En regel är en typ av grammatisk konstruktion. Exempel på typer av konstruktioner som reglerna detekterar är t.ex. ett kongruensfel som *ett katt*, *ett stol* etc. I regeln ingår också ett rättningsförslag, som i det här fallet skulle vara *en katt, en stol*.

Granskningen är uppbyggd enligt bild 1. Först görs en tokenisering av alla textens ord. Sedan förser taggaren varje ord med en ordklassinformation, en s.k. tagg. Denna tagg innehåller bl.a. information om ordklass. Taggaren slår upp ordet i ett lexikon och på statistisk grund räknar ut vilken av flera möjliga ordklasstilldelningar ett ord får i sitt sammanhang. Granska använder sig av en andra ordningens markovmodell för att räkna ut ordklasstilldelningen, vilket ger en lyckad taggning till ca 95 procent. När taggaren i Granska stöter på nya ord så taggas ordet genom att kombinera informationen om vilka taggsekvenser som är vanliga med en morfologisk analys av ordet. På detta sätt kan rätt tagg gissas med stor precision. Nya ord taggas med sådana taggar som är sannolika för andra ord som slutar på samma bokstäver. Den taggade texten skickas sedan till en regelmatchare som mening för mening försöker matcha de grammatiska konstruktionerna som definierats i granskningsreglerna. Granskningsfunktionen i Granska

markerar de problem den hittar i texten och presenterar detta för användaren med de rättningsförslag som specificerats i regeln. Granska rättar idag inte de felaktigheter den hittar i texten, utan presenterar bara rättningsförslagen för användaren.

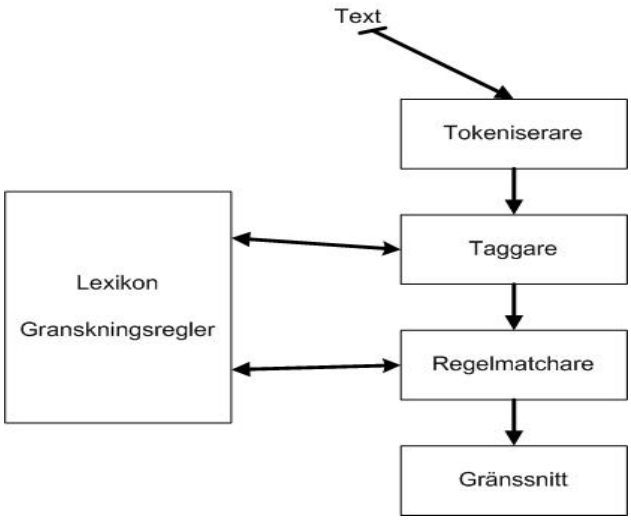



*Bild 1. Granskasystemets uppbyggnad.*

## 1.2 Korpus

För utveckling inom datorstödd språkgranskning för naturligt språk såsom svenska krävs att man har stora mängder text. Till texten vill man ofta ha tilläggsinformation såsom meningsgränser, stycken och ordklassinformation för varje ord. Att lägga till information kallas för att annotera texten och resultatet blir en korpus – en stor annoterad textmängd.

En svensk korpus är *Stockholm-Umeå Corpus*, SUC [14], som innehåller drygt en miljon ord fördelat på ca 500 texter med 2000 ord vardera. SUC innehåller texter av olika typer från skönlitteratur, facklitteratur, tidningstext, myndighetstexter m.m., samtliga tryckta på 1990-talet. *Parole* är ett internationellt projekt där korpusar byggs för många av de europeiska språken. En engelsk korpus är BNC, *The British National Corpus,* som innehåller 100 miljoner ord hämtade från en mängd olika källor, från både tal- och skriftspråk.

På Nada bygger man en stor svensk korpus. Hittills har man samlat ihop över 60 miljoner ord.

En korpus innehåller alltså ett mycket stort antal ord fördelade på en mängd texter. I det här examensarbetet arbetar vi med korpus på SUC-format,

där texterna är kodade på SGML-format.

Att bygga en korpus kan innebära mycket manuellt arbete. Om man använder ett dataprogram för att utföra uppgiften har man svårigheten med att konstruera bra matchningsregler, samt att vissa feltyper är svåra att beskriva med regler och lexikon. Dessutom har ett automattagande program alltid en liten felprocent vid taggningen. Handrättade SUC, som Granska använder som statistisk referens, innehåller ca 5000 felaktiga taggar, en felfrekvens på 0.5 procent.

Med ett annoteringsverktyg, som detta examensarbete går ut på att skapa, har en språkforskare möjlighet att bearbeta en korpus interaktivt, och på så sätt minimera felfrekvensen. Denna korpus kan sedan användas inom språkforskningen för att erhålla effektivare grammatikgranskningsprogram.

## 1.3 Uppgiften

För att minimera det manuella arbetet vid rättning och underhåll av korpus behöver man kraftfulla verktyg. Det här examensarbetet, som är ett större arbete uppdelat på två personer, går ut på att utforma och delvis implementera ett sådant annoteringsverktyg. Verktyget skall bestå av en logisk del och en användarapplikation. Den första delen har Andreas Aarflot haft hand om och den andra delen har jag utfört.

Den logiska delen innebär att modifiera granskningsfunktionen i Granska enligt bild 2. Istället för att arbeta mot en text skall verktyget arbeta mot korpus. Verktyget skall också arbeta mot en regel i taget istället för granskas regeluppsättning.

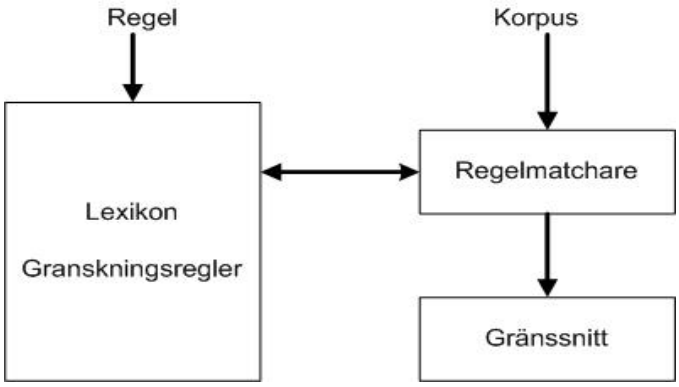



*Bild 2. Det nya granskningssystemets uppbyggnad.*

Den logiska delen kommer också att innehålla en optimeringsfunktion för snabbare sök- och ersättningsfunktionalitet. Det är här själva logiken för att köra en sökfråga (köra en regel på en given korpus) kommer att ligga.

Den andra delen består i att utforma och delvis implementera en användarapplikation till verktyget innehållande ett grafiskt användargränssnitt, ett API som döljer nätverkskommunkationen för externa användare och en server som hanterar kommandon från gränssnittet.

Målet med mitt examensarbete var därför att:

- Utforma ett grafiskt användargränssnitt gentemot användaren.
- Utforma och implementera en server som hanterar kommandon från användargränssnittet. Servern skall också innehålla användarkataloger.
- Hantera nätverkskommunikationen mellan klient och server.
- Utforma och implementera ett API som döljer nätverkskommunikationen för externa användare.
- Kommunikationen mellan server och logikdel.

Verktygets logikdel kommer att skrivas i C++ (Granska är implementerat i C++). Valet av programmeringsspråk för användarapplikationen är valfritt. Jag har tittat på olika tekniker för kommunikationen mellan komponenterna samt olika sätt att organisera komponenterna i en arkitektur.

## 1.4 Tillvägagångssätt

Jag har under utvecklingen av detta projekt använt mig av valda delar ur UML. Utvecklingsprocessen delades in i olika faser enligt följande. En *analys- och utvecklingsfas* där tonvikten lades på att undersöka de problem och de krav som fanns på verktyget. Jag började med att bestämma mig för utvecklingsmiljö. Jag tittade på olika alternativ som skulle kunna passa samt vägde för- och nackdelar mot varandra utifrån en analys av verktygets tänkta funktionalitet. Under denna fas ville jag också få fram mer detaljerad information om de användarkrav som fanns på verktyget. För detta tog jag fram användarfall för hela verktyget. Under denna fas träffade jag min handledare kontinuerligt för att stämma av om jag var på rätt väg. Efter detta kom en *designfas* där jag tog fram klassdiagram och för att få fram den statiska strukturen i systemet (se appendix D). Efter att detta var klart började jag med *implementationsfasen.* Under denna fas implementerades

systemet i en itererande process där jag började med de viktigaste användarfallen först.

# 2 Utvecklingen

Målet med examensarbetet är som sagts ovan, att utveckla och implementerta ett klient-serversystem, innehållande ett grafiskt användargränssnitt och server med tillhörande logik med hjälp av rätt vald utvecklingsmiljö anpassad till användarkraven.

## 2.1 Användar- och uppgiftsanalys

Grunden för att kunna utveckla ett användbart och funktionellt verktyg är att förstå och identifiera slutanvändarna och att förstå den miljö i vilken dessa arbetar.

Det här verktyget är inte tänkt att användas av privatpersoner utan språkforskare på universitet och högskolor. Vidare skall verktyget tas fram i syfte att stödja utvecklingen runt språkforskningen. Därför har jag fokuserat på funktionaliteten snarare än användarvänligheten. Datorvanan bland dessa personer får anses hög. Eftersom verktyget bara är tänkt att användas av forskare kommer antalet samtidiga användare ej vara särskilt stort. Däremot kan dataflödet, på nätverket, kanske bli stort då verktyget kommer att arbeta mot stora mängder text.

Det är också viktigt att förstå vilka de uppgifter är, som användarna måste kunna utföra med systemet, för att få fram den rätta funktionaliteten. Genom tidiga diskussioner med min handledare kom jag fram till att följande uppgifter skall verktyget klara av:

**Kataloghanteringsfunktion**
En användare skall ha möjlighet att spara de korpus, regler etc. denne har skapat och vill arbeta mot. På serversidan skall därför logik implementeras för att hantera användarkatalogerna.

**Möjlighet att skapa en egen korpus**
Alla korpusar kommer fysiskt ligga sparade nära logikdelen, där varje korpus är uppdelad på ett stort antal texter. En användare skall genom det grafiska gränssnittet ha möjlighet att skapa egna korpus att arbeta mot. När en användare vill skapa en egen korpus kan denne välja bland alla texter som finns i systemet. Den skapade korpusen kan sedan sparas under egen katalog i användarkatalogerna för att användas vid senare tillfällen.

**Gruppering**
En sökning kan ge upphov till ett mycket stort antal träffar. Användaren har då möjlighet att välja gruppering av de matchningar som programmet hittar för att få en tydligare struktur på resultatet.

**Sökning**
Användaren skall kunna specificera grammatiska regler för att söka efter en speciell typ av konstruktion i korpusen. Programmet skall leta upp alla förekomster som matchar regeln och presentera dessa för användaren på det grafiska gränssnittet.

**Ersättning**
De matchningar som hittats presenteras för användaren med ersättningsförslag. Användaren skall på något sätt kunna välja vilka av matchningarna han/hon vill rätta.

**Införa ändringar**
De valda ersättningsförslagen skall kunna sparas på en loggfil för vidare automaträttning av korpusen.

**Spara regler**
En användare skall kunna spara egna konstruerade regler under en egen katalog.

**Skapa egen profil**
En användare kan sätta om det grafiska gränssnittet efter eget tycke, t.ex. ändra storlek, ändra färg etc. När användaren avslutar programmet, sparas profilen under användarkatalogen och det grafiska gränssnittet startas om som det såg ut vid senaste avslutningen.

**Flera samtidiga användare**
Flera användare skall samtidigt kunna använda verktyget.

## 2.2 Utvecklingsmiljö

En viktig del av examensarbetet bestod i att välja en lämplig utvecklingsmiljö för verktyget. Innan jag bestämde mig tittade jag på olika lämpliga alternativ som skulle passa till uppgiften. Jag fann tre viktiga punkter att titta på:

- Val av exekveringsmiljö.
- Applikationsarkitektur.
- Nätverkskommunikation.

### 2.2.1 Val av exekveringsmiljö

I examensarbetet ingår det att utforma och delvis implmentera ett grafiskt användargränssnitt för användaren, ett s.k. GUI (Graphical User Interface). Denna applikation måste laddas till klienten på något sätt och exekveras där. Jag har valt att titta på och jämföra tre olika alternativ: fristående applikation, applet och java webstart.

#### 2.2.1.1 Fristående applikation

Applikationen laddas hem via nätet till klientmaskinen och exekveras där. En nackdel med denna metod är att när man skall hämta frigående applikationer från nätet krävs en krånglig installationsprocess där först en installerare måste hittas, laddas hem, installeras och köras. En annan nackdel är att applikationen kommer att kunna exekveras hos klienten utan restriktioner.

#### 2.2.1.2 Program som exekveras i webläsaren - applets

En applet är ett vanligt javaprogram som kan exekveras inuti en webläsare. Det fungerar så att den redan färdig kompilerade appleten hämtas till klienten av webläsaren som binär data, webläsaren innehåller en JVM (Java Virtuall Machine) som exekverar appleten. För säkerhets skull har appleten flera restriktioner. En är att appleten har begränsad tillgång till datorns filsystem. En annan är att appleten ej kan göra nätverksuppkopplingar annat än till webservern den laddades ifrån.

En fördel med applettekniken är att ingen installationsprocess krävs av användaren, eftersom applikationen körs och hämtas automatiskt av webläsaren. Webläsaren har automatiserat hela processen. En nackdel med applettekniken är att programmet ej kan göra nätverksuppkopplingar. En annan nackdel är att webläsaren inte kan utnyttja alla javas GUI-komponenter fullt ut för att göra komplexa gränssnitt [11].

#### 2.2.1.3 Java webstart

Java webstart är en teknologi utvecklad för javaapplikationer där man kan sätta igång java-baserade applikationer direkt från webläsaren. Java webstart är en kompromiss mellan en fristående applikation med Javas alla egenskaper och en applet utan de komplicerade uppgraderings- och installationsprocesserna. Kompromissen med java webstart är att man måste ladda hem java webstart-programvaran första gången, om man inte redan har den. Det blir alltså en större aktiveringstid då man hämtar hem applikationen första gången än med en web-baserad applikation. I gengäld får man en ”rikare” applikation med alla javas egenskaper och som ej körs i webläsaren.

Kravet för att kunna få allt att fungera är att klienten har java webstart-mjukvaran installerad på sin maskin. Detta är en engångsinstallation. Java webstart sätts igång och exekveras av webläsaren som sedan laddar, sparar och exekverar den givna applikationen. Java webstart cachar alla nerladdade filer lokalt på maskinen. Dessa filer körs sedan i en begränsad miljö (”sandbox”) med inskränkt tillgång till maskinens filsystem och nätverk. Java webstart ser alltid till att du kör den senaste versionen. Vid varje igångsättning av Java webstart kommer mjukvaran att kontrollera om applikationen inte redan finns nerladdad hos klienten och om senaste versionen finns där. Annars laddas applikationen automatiskt av java webstart. Java webstart kommer alltid att kommunicera tillbaka till servern för att se om inte en senare version av applikationen finns.

Java webstart kan startas på ett antal olika sätt, det vanligaste är dock att använda en webläsare, vilket jag har valt för detta examensarbete.

#### 2.2.1.4 Applet kontra Java webstart

Skillnaden mellan applet och Java webstart teknologi är att appleten körs i webläsaren medan Java webstart applikationer ej är bundna till webläsaren. För när applikationen väl körs är förbindelsen till webläsaren avbruten. En annan skillnad är att både appleten och java webstart körs i en begränsad miljö (”sandbox”) med inskränkt tillgång till maskinens filsystem och nätverk. Men java webstart kan tillåta att öppna upp vissa restriktioner och tillåta nätverksuppkopplingar.

Jag har sammanfattningsvis valt att köra java webstartteknologi på klientsidan med webläsare som laddare av följande skäl:

- Fristående applikation.

- Möjlighet till alla Javas egenskaper för applikationen vilket ger mig möjlighet att skapa ett ”rikt” grafiskt användargränssnitt.
- Säkerheten.
- Inga stora komplicerade installations- och nerladdningsprocesser för användaren.
- Kan göra nätverksuppkopplingar.

### 2.2.2 Applikationsarkitektur

En arkitektur är en strukturering och organisering av huvudelementen i ett mjukvarusystem. Innan en arkitektur väljs, måste en undersökning göras av de funktionella och icke-funktionella krav som har en stor påverkan på systemet. En undersökning av de icke-funktionella kraven kan inkludera att titta på tillförlitlighet, kostnad, prestanda, utbyggbarhet, underhåll etc. Med andra ord innebär undersökningen en kravanalys med fokus på de krav som har ett stort inflytande på arkitekturen.

I mitt examensarbete, som är en av två delar av en större uppgift, kan jag finna ett antal huvudelement som måste organiseras i en lämplig arkitektur:

- En klient med grafiskt användargränssnitt och API.
- En server som hanterar klienter och innehåller logik för hantering av användarkatalogerna.
- En logikdel gentemot Granska. (Denna del ingår ej i mitt exjobb men är en del av verktyget).

I mitt val har jag också vägt in de icke-funktionella krav som utbyggbarhet, underhåll, prestanda, utvecklingskostnad, återanvändbarhet och plattformsoberoende. När jag tittade på de funktionella kraven i mitt system kunde jag konstatera att många meddelanden kommer att skickas mellan klient och server, eftersom servern innehåller användarkatalogerna. Storleken på data kan bli stort, då en virtuell korpus kan ha i storleksordningen 100 000 ingående texter samt att en körning av sökfråga kan resultera i många svar. Det är också lätt att tänka sig att det kan bli en utbyggnad av verktyget vid ett senare tillfälle, då är det också viktigt att designa arkitekturen med avseende på utbyggbarhet.

Eftersom den logiska delen kommer att vara en integrerad del av Granska kommer denna att skrivas i C++.

För struktureringen av systemet fann jag två naturliga lösningar att välja på: tvådelad och tredelad arkitektur.

#### 2.2.2.1 Tvådelad arkitektur

I denna arkitektur utvidgas logikdelen med server-funktionalitet för att hantera kommandon från klienten. På logikdelen kommer också användarkatalogerna att ligga. Detta resulterar i en tvådelad arkitektur med en klient huvudsakligen ansvarig för presentationen och en server skriven i C++ som en integrerad del av logikdelen.

Fördelar:

- Enklare kommunikation mellan server och logikdel.
- Enkel arkitektur.
- Mindre kommunikation via nätverk.

Nackdelar

- Logikdel får flera roller.
- Risk för tyngre klient.
- Stark koppling mellan server och logikdel.

#### 2.2.2.2 Tredelad arkitektur

I denna arkitektur introduceras en mellanserver som kan hantera den logik som inte är direkt kopplad till den logiska delens funktionalitet. Denna servers uppgift blir att: hantera kommandon från klienter, hantera användarkatalogerna och innehålla logik för klienterna. Logikdelen kommer nu endast att hantera vissa kommandon som berör själva sökfunktionen. Dessa kommandon kommer ej från klienterna direkt, utan går via mellanservern. Klienten blir tunnare, då mer logik kan läggas på mellanservern och klienten blir endast ansvarig för själva presentationen. Detta resulterar i en tredelad arkitektur med tre logiska komponenter, alla med ett väldefinierat gränssnitt och en klar avgränsning mellan de olika delarnas funktionalitet. Mellan-servern kan implementeras som en platformsoberoende komponent utan direkt koppling till logikdelen. Kommunikationen mellan server och logikdel sker också via nätverk.

Fördelar

- Tydlig avgränsning av de olika delarnas funktionalitet.
- Tunnare klient.
- Färre beroenden mellan delarna.
- Platformsoberoende mellanserver som kan användas, om så vill, av andra servrar liggandes överallt i världen.
- Hög flexibilitet på produkten vad gäller förändring.

Nackdelar

- Introduktion av ytterligare komponent ger ökad komplexitet och längre implementationstider.
- Ökad nätverkskommunikation.

#### 2.2.2.3 Sammanfattning

Vi valde att implementera verktyget som en tredelad arkitektur. Anledningen att vi valde denna arkitektur var att modellen passade väl in på detta verktyg med så klart avgränsande funktioner (klient, server, sök-/ersättning).

### 2.2.3 Kommunikationsprotokoll

Två kommunikationsprotokoll måste väljas, dels ett mellan klient och server, dels ett mellan server och logikdel.

Jag har valt att titta på och jämföra ett antal populära nätverksmekanismer: RMI, CORBA och socket med XML.

#### 2.2.3.1 RMI

Huvudidén bakom RMI (Remote Method Invocation) är att objekt enkelt skall kunna delas mellan olika maskiner i ett distribuerat system, d.v.s. ett objekt liggandes på en maskin transparent kan anropas av andra objekt tillhörande andra maskiner.

På servern ligger objektet fysiskt sparat, klienten får en stubbe av servern. När klienten anropar objektet ser det ut som om objektet ligger på klientsidan, men anropar istället stubben som agerar som ställföreträdare för objektet. Stubben ansvarar för att vidarebefordra meddelandet från klienten till det riktiga objektet på servern.

RMI använder sig av ett Java-språkberoende gränssnitt för kommunikationen och tillåter alltså endast kommunikation mellan applikationer skrivna i Java.

Fördelar:

- Är inte ett applikationsberoende protokoll, vilket gör det enklare att uppdatera.
- RMI tillhandahåller en mängd bra resurser, t.ex. möjligheten att anropa metoder på andra maskiner utan att själv behöva tänka ut ett protokoll.
- RMI abstraherar bort detaljerna om kommunikationen på låg nivå och möjliggör för programmeraren att fokusera på annat.

Nackdelar:

- Vid sändning/hämtning av stora mängder data kan RMI vara betydligt långsammare än en traditionell socketlösning [15].
- RMI fungerar bara på Java.

#### 2.2.3.2 CORBA

I huvuddrag fungerar CORBA väldigt likt RMI. Klienten får en stubbe av servern som agerar ställföreträdare för objektet. På så vis ser det ut som om objektet anropas lokalt. Den största skillnaden är att CORBA använder sig av ett språkoberoende gränssnitt för kommunikationen och tillåter kommunikation mellan applikationer skrivna i olika språk. För kommunikationen använder CORBA sig av s.k. ORB (Object Request Broker). Det är ORB:en som sköter kommunikationen mot underliggande transportmedium och förmedlar meddelanden mellan klient och server.

Fördelar:

- Är inte ett applikationsberoende protokoll, vilket gör det enklare att uppdatera.
- CORBA, liksom RMI, tillhandahåller en mängd bra resurser, t.ex. möjligheten att anropa metoder på andra maskiner utan att själv behöva tänka ut ett protokoll.
- Kan användas mellan applikationer skrivna i olika språk.
- CORBA, liksom RMI, abstraherar bort detaljerna om kommunikationen på låg nivå och möjliggör för programmeraren att fokusera på annat.

Nackdelar:

- Vid kommunikation av stora mängder data kan CORBA liksom RMI vara betydligt långsammare än en traditionell socketlösning [15].
- klienten måste använda sig av ORB:n, vilket leder till en involvering och distribution av en stor JAR-fil till klienten.

### 2.2.3.3 Socket och XML

**Socket**
Socket är en nätverksabstraktion som möjliggör för programmeraren att enkelt utveckla program som sänder och tar emot data över nätverk. Socketen representerar en förbindelse mellan en klient och en server. RMI använder sig underliggande av socket.

Fördelar:

- Möjliggör på ett smidigt och uniformt sätt kommunikation mellan maskiner av olika typer.
- Snabb överföring för stora datamängder.
- Möjliggör programspråksoberoende kommunikation. Enkelt att få program som är skrivna i olika programspråk att kommunicera med varandra.

Nackdelar:

- Socket är en lågnivåabstraktion som kräver mer av programmeraren för att skicka och ta emot data.
- Kräver att man själv utformar ett protokoll för kommunikationen.

Om man väljer socket som nätverksmekanism måste ett protokoll utformas för meddelandekommunikationen mellan klient och server. Hur skall nu data överföras? En dataöverföringsmekanism måste väljas. XML är en kandidat jag tittat på för datarepresentationen.

**XML**
XML är ett textbaserat språk som används för att beskriva data som är hierarkiskt ordnat. XML innehåller taggad information där varje del beskrivs av taggen. Taggarna bryter upp informationen i XML-data i delar. Eftersom de olika delarna av informationen är identifierat av taggarna kan de användas av olika applikationer på olika sätt.

XML gör att man får en platformsoberoende datarepresentation för nätverkscentrerade program som skickar och tar emot data [8]. Java har ett väl inbyggt stöd för XML som gör att det är lätt att använda i Javaprogram.

Här följer ett kort exempel på hur XML-data kan vara strukturerat:

```
<message>
     <to>helena.ihrfors@telia.com</to>
     <from>su99-hih@nada.kth.se<(from>
     <text>Hej du glade</text>
</message>
```

Fördelar:

- En fördel med XML är att det blir enklare att skicka och ta emot objekt mellan olika applikationer skrivna i olika språk [8].
- Eftersom XML-data är strukturerat språk där varje del börjar och avslutas med en tagg, blir det väldigt enkelt att skriva program som processar XML-data, en s.k. XML-parser. Det finns ett antal redan färdigskrivna att välja mellan.
- Standardiserat format.

Nackdelar

- En nackdel med XML är att det blir mer data att skicka och ta emot över närverket (varje tagg/sluttagg måste ju också skickas med).

För att kunna använda en XML-parser måste den veta hur XML-data ser ut, hur den förväntar sig att data ser ut och att den agerar efter det. En så kallad DTD (Document Type definition) måste kopplas till XML-data. En DTD är en specifikation över de olika taggar som kan förekomma i XML-data. DTD:n talar om för parserprogrammet vilka taggar som är giltiga och var i hierarkin de förväntas vara. Den talar också om var data förväntas förekomma. DTD:n innehåller alltså en grammatik som specificerar vilka datastrukturer som kan förekomma och i vilken ordning.

Ex. på DTD.

```
<!ELEMENT message (to,from,text)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT text (#PCDATA)>
```

Detta exempel berättar för parsern att `message` är ett element som innehåller andra element. Elementen `to`, `from` och `text` innehåller data som kan processas (PCDATA).

Det som återstår är nu att välja en parser-modell som passar till detta projekt. Ett antal kandidater finns att välja på för Java. Jag valde ut två och tittade på dessa i detalj:

**JAXP (Java API for XML Processning)**
JAXP är ett enkelt API som kan användas tillsammans med en SAX-parser eller en DOM-parser.

En SAX-parser (Simple API for XML) är en parser som söker igenom indata linjärt och anropar en händelsehanterare varje gång den stöter på en tagg.

En DOM-parser (Document Object Model) använder sig av en helt annan strategi än SAX för att processa indata. Istället för att läsa igenom rad för rad, läser DOM-parsern igenom hela indata och skapar ett träd som reflekterar den hierarkiska strukturen i indata där all information sparas. Trädet innehåller noder, där varje nod representerar ett element, ett attribut eller data. Man kan sedan modifiera, radera, eller lägga till information i trädet om man så vill.

Skillnaden mellan dessa två är att SAX är en snabb sekventiell parser, där data besöks under exekveringen, men sparas inte. DOM däremot, besöker data i det sparade trädet efter exekveringen. Sammanfattningsvis kan sägas att SAX är snabb och minnessparande, men mindre lämpad för applikationer som vill hämta eller manipulera data i efterhand. DOM å andra sidan är minneskrävande, har längre exekveringstid och lämpar sig för applikationen som behöver manipulera data i efterhand. En nackdel med både DOM och SAX är att programmeraren måste skriva komplicerade algoritmer för att läsa informationen.

**JAXB (Java Architecture for XML Binding)**
JAXB erbjuder ett enkelt sätt att skapa en tvåvägsmappning mellan XMLdata och Javaobjekt. Givet en DTD, som specificerar strukturen i XML:n, genererar JAXB ett antal javaklasser. Dessa klasser innehåller metoder för att, utifrån XML-data skapa objekt av klassen (unmarshalling) och utifrån skapade objekt av klassen generera XML-data (marshalling). Objektens attribut representerar indata eller utdata.

Fördelen med JAXB är att den har SAX snabbhet och kan samtidigt spara innehållet i en datastruktur i minnet som en DOM. Men till skillnad från DOM-parsern innehåller den inte en massa extra funktionalitet för trädmanipulationen och blir därför inte lika minneskrävande.

**Sammanfattning**
Jag valde att använda mig av JAXB-modellen för att processa data av följande skäl:

- Är både snabbare än SAX och inte lika minneskrävande som DOM.
- Är lätt att använda för att processa XML och manipulera data.

# 3 Systemprinciper

Detta kapitel beskriver att antal lösningsförslag som jag använt mig av vid implemenationen av verktyget.

## 3.1 Designriktlinjer för grafiskt gränssnitt

Vid gränssnittsdesign finns ett antal riktlinjer som kan användas vid framtagning av gränssnitt i syfte att få en god användbarhet [12], [4]. Jag har plockat ut några och följt dessa då jag implementerade det grafiska användargränssnittet.

**Konsistens:**
Funktionerna i gränssnittet måste vara konsistenta, d.v.s. följa samma mönster och producera samma resultat varje gång. Om användaren vet att samma kommando eller handling alltid har samma effekt kommer gränssnittet att upplevas som enklare att använda.

**Återkoppling:**
Det är viktigt för användaren att känna kontroll. Om man använder en funktion och inte förstår vad som händer blir man frustrerad. Visa alltid för användaren att något sker genom t.ex. en statuspanel eller timglas.

**Möjlighet att ångra sig:**
Användaren bör ha möjlighet att ångra sig och ta sig ur situationer som inte är önskvärda. I samtliga dialogrutor bör det finnas en knapp som rullar tillbaks till det föregående tillståndet.

**Layout:**
Strukturera layouten på gränssnittet så att komponenter som funktionellt hör ihop grupperas på något sätt, t.ex. i boxar eller inramade. Detta hjälper användaren att förstå strukturen i gränssnittet och underlättar användandet. Låt viktiga funktioner placeras extra synligt och mindre viktiga döljas under menyer.

**Otillåtna operationer**:
Omöjliggör för användaren att utföra otillåtna operationer, t.ex. gråmarkera de funktioner som för tillfället inte är aktiva.

**Dialogrutor:**
Använd dialogrutor för felmeddelanden. Skapa dialogrutor med meningsfulla, förståliga och precisa felmeddelanden så att användaren förstår felkällan. Förutom felmeddelanden bör systemet också ge positiv återkoppling t.ex. ”regel sparad”.

**Färgsättning:**
Begränsa antalet färger, för många färger distraherar användaren. Använd starka färger endast för att tydligt markera.

## 3.2 Designmönster

Designmönster (eng. Design Patterns) är ett formaliserat sätt att dokumentera ett bevisat bra lösningsförslag på ett frekvent återkommande problem. Det är ett sätt att dokumentera bra lösningar till vissa givna problem. Inom objektorientering är designmönster en namngiven beskrivning på ett problem och dess lösningsförslag [10]. Meningen med dessa regler är inte att hitta nya idéer utan att beskriva redan kända problem med lösningsförslag. I ett objektorienterat system är det otroligt viktigt att fördela ansvarsområden under design- och programmeringsfaser. Följande designmönster [10] har jag använt mig av under design och implementation av både klient och server:

**Information expert**
Ett system kan bestå av många olika klasser som tillsammans skall implementera många olika uppgifter. Under designfasen måste vi tilldela dessa uppgifter till olika klasser. Om detta utförs bra blir systemen lätta att uppdatera, utvidga och förstå.

Problem: Finns det någon generell princip för hur ansvar fördelas till objekt?

Lösning: Tilldela en uppgift till det objekt som har nödvändig information att fullfölja ansvaret. Objekt skall tilldelas uppgifter utifrån den information de innehåller.

**Creator**
Problem: Vem skall få uppgiften att skapa nya klassinstanser?

Lösning: Låt en klass få uppgiften att skapa en annan klassinstans om klassen innehåller instansen eller nära använder instansen.

**Low coupling**
Kopplingsgrad är ett mått på hur starkt beroende ett element har till andra element. En klass med låg kopplingsgrad är inte beroende av så många andra klasser för att fullfölja sin uppgift. Sådana klasser gör att inverkan blir mindre om ändringar måste göras till systemet och detta resulterar i bättre återanvändbarhet.

Problem: Hur får man låg beroendegrad mellan klasser, låg påverkan om ändringar måste göras, samt hög återanvändbarhet?

Lösning: Tilldela klasserna uppgifter så att kopplingen dem emellan blir låg.

**High cohesion**
Sammanhang är ett mått på hur starkt relaterade en klass uppgifter och ansvarsområden är. En klass med starkt relaterade uppgifter och som inte gör väldigt mycket arbete sägs ha starkt sammanhang. En klass med lågt sammanhang gör många orelaterade uppgifter. Sådana klasser kan bli svåra att återanvända, uppdatera eller hantera.

Problem: Hur skall man göra komplexa system mer lätthanterliga?

Lösning: Skapa klasser med klart avgränsade ansvarsområden.

**Controller**
Många system tar emot externa händelser, i synnerhet GUI-baserade system. Någon klass måste hantera dessa. Detta kontrollermönster ger riktlinjer för hur detta val kan göras på ett passande sätt. Kontrollern skall vara något slags fasad mellan gränssnittet och det övriga systemet. Det kan vara bra att ha samma kontrollerklass för alla systemhändelser inom samma användarfall. Kontrollern skall delegera arbetet till andra objekt. Den koordinerar bara all aktivitet, men innehåller ingen logik själv.

Problem: Vem skall vara ansvarig för att agera på en händelse?

Lösning: Tilldela ansvaret att ta emot och hantera en systemhändelse till samma kontrollerklass inom samma användarfall. Alternativt använd en kontrollerklass till hela systemet.

# 4 Lösningen

## 4.1 Lösningstaktik

För att kunna implementera verktyget på ett effektivt sätt är det bra att ha alla funktionella och icke funktionella krav på systemet klart för sig på ett tidigt stadium. Ett bra sätt att försöka få ut den dynamiska strukturen i systemet är att skapa detaljerade amvändarfall (bilaga A). Ur dessa användarfall kan sedan systemets detaljerade funktionalitet hittas. På det sättet fick jag snabbt fram vad systemet skulle klara av samt vilka meddelanden som skulle skickas över nätverket. Utifrån användarfallen gjordes en skiss på det API som skulle skapas mellan klient och server (bilaga B). Skissen gjorde det sedan enkelt att se hur XML-data skulle struktureras och hur tillhörande DTD skulle se ut (bilaga C).

Jag gjorde sedan en prototyp av användargränssnittet. Jag upptäckte att det är bra att ta fram en prototyp på ett tidigt stadium i utvärderingsssyfte. Det är lättare att visa en konkret prototyp än en massa skisser. Missförstånd om vad som skall ingå i produkten och hur den skall fungera minskas om man har ett konkret exempel att visa. Prototypen visades för min handledare som kom med förslag på förbättringar vad gäller funktionalitet och utseende. Innan jag utvecklade prototypen ytterligare bestämde jag mig för hur användargränssnittet skulle fungera i mitt system. Jag började med att titta på de system som fanns t.ex. Microsofts produkter. På så vis försökte jag efterlikna Microsoft Word så långt som möjligt, så att användaren skulle känna igen sig.

Nästa steg i projektet var att hitta den statiska strukturen i programmet. För att få en överblick över systemet och bemästra komplexiteten skapade jag två domändiagram: ett för klienten och ett för servern. Målet med designen var att skapa ett objektorienterat system efter de designregler jag specificerat i avsnitt 3.2. Utifrån domänmodellerna kunde jag sedan börja definiera de klasser och metoder som systemet skulle bestå av. Utifrån detta skapades sedan klassdiagrammen: ett för klienten (bilaga C) och ett för servern (avsnitt 4.6).

För implementationen valdes en iterativ process där varje användarfall implementerades var och en för sig. Jag började med de viktigaste användarfallen först.

## 4.2 Systemmodell

Arkitekturen med nätverk visas i bild 3. Flera klienter kan ansluta sig mot servern som har parallella egenskaper. Klienten består av två huvudelement: GUI (Grafiskt användargränssnitt) och API som döljer nätverkskommunikationen. Nätverkskommunikationen sker med XML i hela systemet. Genom det grafiska gränssnittet kan användaren utnyttja användarkatalogen på servern, initiera en sökfråga (köra en regel på en given korpus) samt införa de önskade korrektionerna. Logikdelen tar emot kommandon från servern och utför själva körningen. Alla behöriga användare har en egen katalog i användarkatalogen på serversidan. Där kan de spara regler, korpus, profiler etc. Varje korpus som ligger i användarkatalogen är virtuella korpus, d.v.s. de består endast av de ingående texternas namn, som valts ut från korpus-/indexkatalog. I korpus-/indexkatalogen ligger alla korpus med underliggande texter fysiskt sparade.

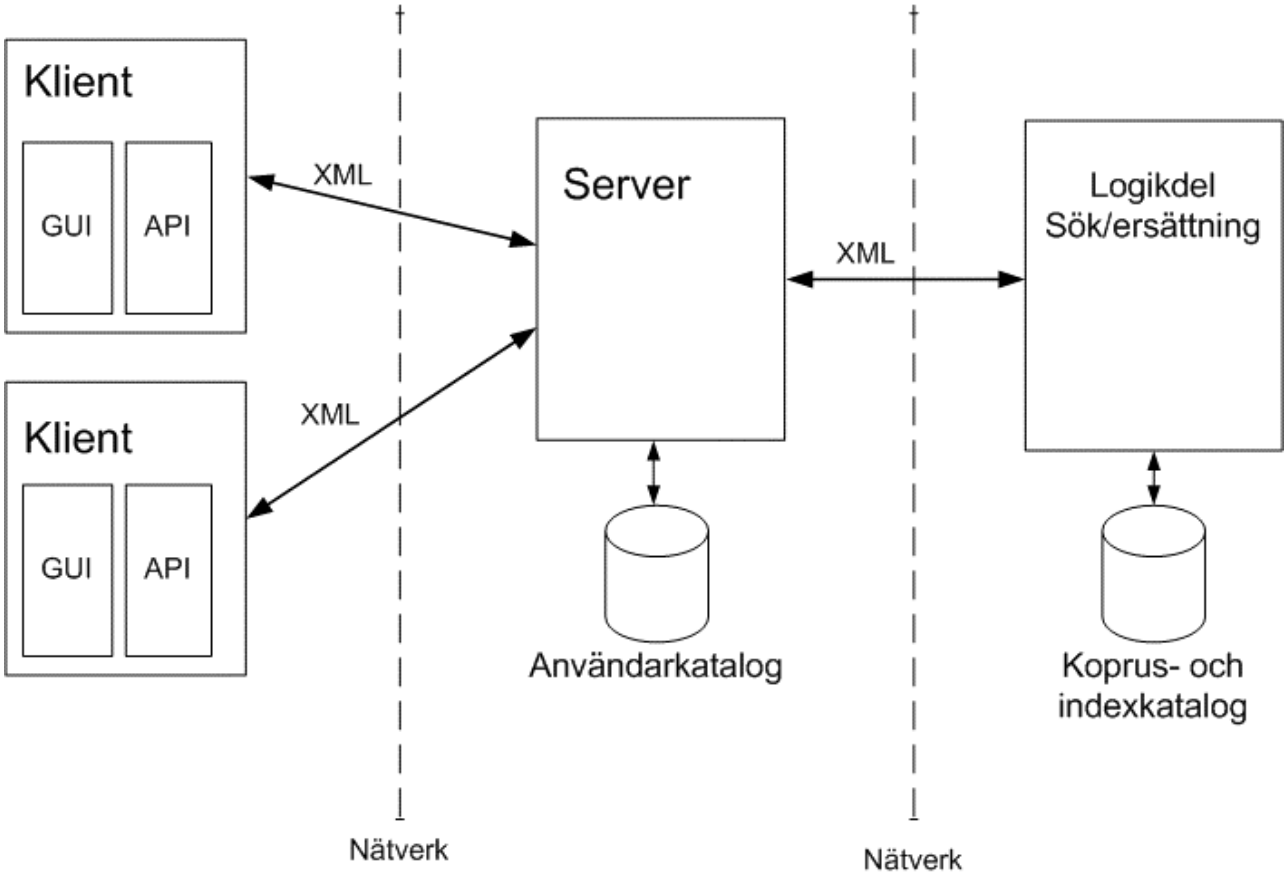



*Bild 3. Modell över hela systemet*

**Korpus- och indexkatalog**
Korpus- och indexkatalogen innehåller alla tillgängliga korpus i systemet med ingående texter. Varje korpus är indelad i flera s.k. texter. Det är

således här alla korpus fysiskt ligger sparade. Valet att låta alla korpus fysiskt ligga nära logikdelen är gjort för att undvika att stora datamängder måste skickas över nätverket. På så vis får man en bättre prestanda med snabbare söktider.

**Användarkatalog**

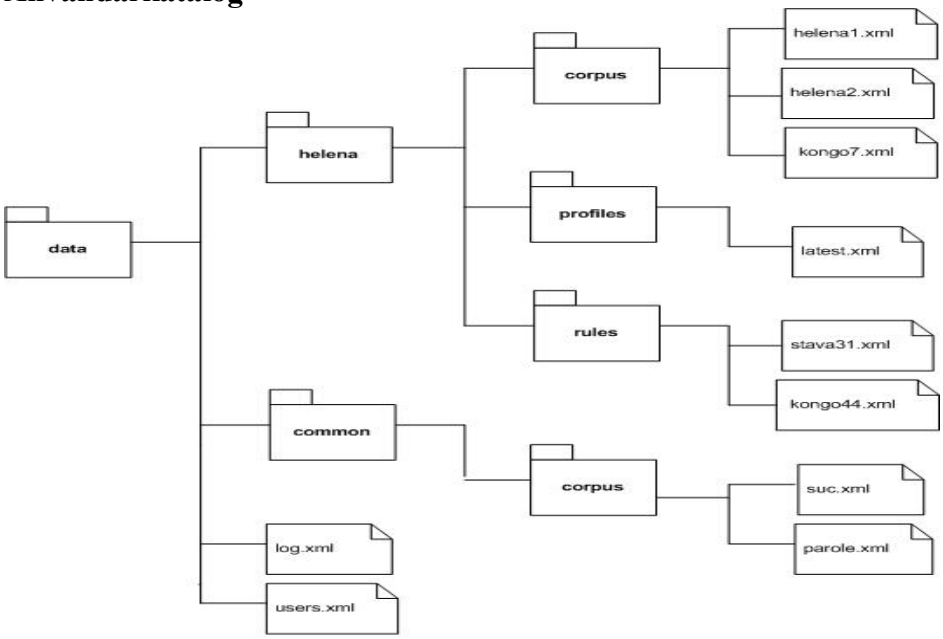



*Bild 4 visar användarkatalogen.*

Alla behöriga användare har en egen katalog under användarkatalogen (se bild 4). Den innehåller regler, korpus och profiler för varje användare samt en gemensam katalog med allmänna korpus, tillgängliga för alla, och en loggfil. Alla korpusar är s.k. virtuella korpusar, de innehåller endast namnen på de ingående texterna.

**Reproduktion**

När en sökfråga körs måste logikdelen veta över vilka texter en sökning skall göras. Ett sätt för den att få reda på detta är att skicka med data om korpus- och textinnehåll varje gång sökfrågan körs. Ett annat är att data skickas över med ett speciellt kommando. Eftersom en användare troligen kör många gånger i rad mot samma korpus och eftersom att körning av en sökfråga (som ändras ofta) ska bli så snabb som möjligt, har vi valt det senare alternativet med ett kommando. Detta kommando initieras då användaren ”laddar korpusen” från GUI:t. Nu sparas alltså användarens virtuella korpus med ingående texter på logikdelen vid laddning av korpus.

Vid körning av sökfrågan skickas användarnamnet med som identifikation och logikdelen kan hämta motsvarande data på korpus-/indexkatalogen.

Jag har också valt att använda reproduktion vid initiering av GUI:t.. Alla regler och korpus en användare har tillgängliga hämtas från användarkatalogen vid initieringen. Dessa sparas lokalt på GUI:t för att undvika nätverksuppkopplingar vid användning av dessa.

Om ett korpusinnehåll ändras under korpuskatalogerna på logiksidan kan en virtuell korpus få inaktuellt innehåll. Vi nöjer oss med att detta upptäcks vid körning av sökfråga. Om en text tagits bort returneras felmeddelande om saknad text från logikdelen. Om text lagts till löper allt normalt. Den nya texten syns först då användaren gör ”edit copus” eller ”new corpus” i GUI:t. Den nya texten syns då som valbart alternativ.

## 4.3 API

Detta API (Application Programming Interface) har till uppgift att dölja nätverkskommunikation på låg nivå och exponera ett antal metoder på högre nivå som olika gränssnittskonstruktörer kan använda sig av. Detta är alltså det gränssnitt som är synligt för ett GUI gentemot systemet. Mitt mål var att skapa ett lättanvändligt API med generella metoder och självförklarande namn. Inparametrarna lät jag vara av enkla typer och returvärdena av enkla typer eller interface-typer.

**Metoder**
Se bilaga B för detaljerad beskrivning av ingående metoder i API.
Sammanfattning av metoder:

---

public ResultObject **saveRule**(String user, String rulename, String ruletext)

public ResultObject **loadCorpus**(String user, String corpusName)

public String[] **getCorpusTexts**(String user, String corpusName)

public ResultObject **saveCorpus**(String user, String CorpusName,
String[] texts)

public ResultObject **saveCorrections**(List corrections)

public ResultObject **saveProfile**(String user, String name, String height,
String width, String ruleText, String corpusName)

public ResultObject **removeRule**(String user, String ruleName)

public ResultObject **removeCorpus**(String user, String name)

public CorpusObject[] **getCorpus**(String user)

public ProfileObject **getProfile**(String user, String name)

public RuleObject[] **getRules**(String user)

public String[] **getTexts**()

public GramerrorObject[] **run**(String user, String corpusName,
String ruleText, String random, String size)

public ResultObject **login**(String user, String password)

## 4.4 Nätverkskommunikation

Kommunikationen mellan klient–server och server–logikserver använder sig, som tidigare sagts, av en JAXB-kompilator och tillhörande DTD för att processa det XML-data som skickas/sänds över nätverket. Utifrån DTD:n genererar JAXB-kompilatorn Java-klasser för varje specificerat element. Dessa genererade Java-klasser innehåller metoder att skicka/ta emot XML över en socket (ström). Instanserna av dessa klasser kan sedan representera det XML-data som skall skickas/tas emot.

Eftersom varje anrop till servern kräver ett svar (synkron överföring), kommer alltid ett svarsobjekt att returneras. Om fel uppstått i processen, returneras ett felobjekt med felmeddelande som presenteras i en meddelandebox på GUI:t. Systemet ger också positiv återkoppling i form av ett resultatobjekt som innehåller ett meddelande för anrop som ”rule saved”, ”corpus saved ” etc. Transaktionsmeddelandet presenteras för användaren på huvudfönstret i dess statusbar.

Nedan följer en schematisk bild på de olika objekt (kommandon) som kan skickas över nätverken.

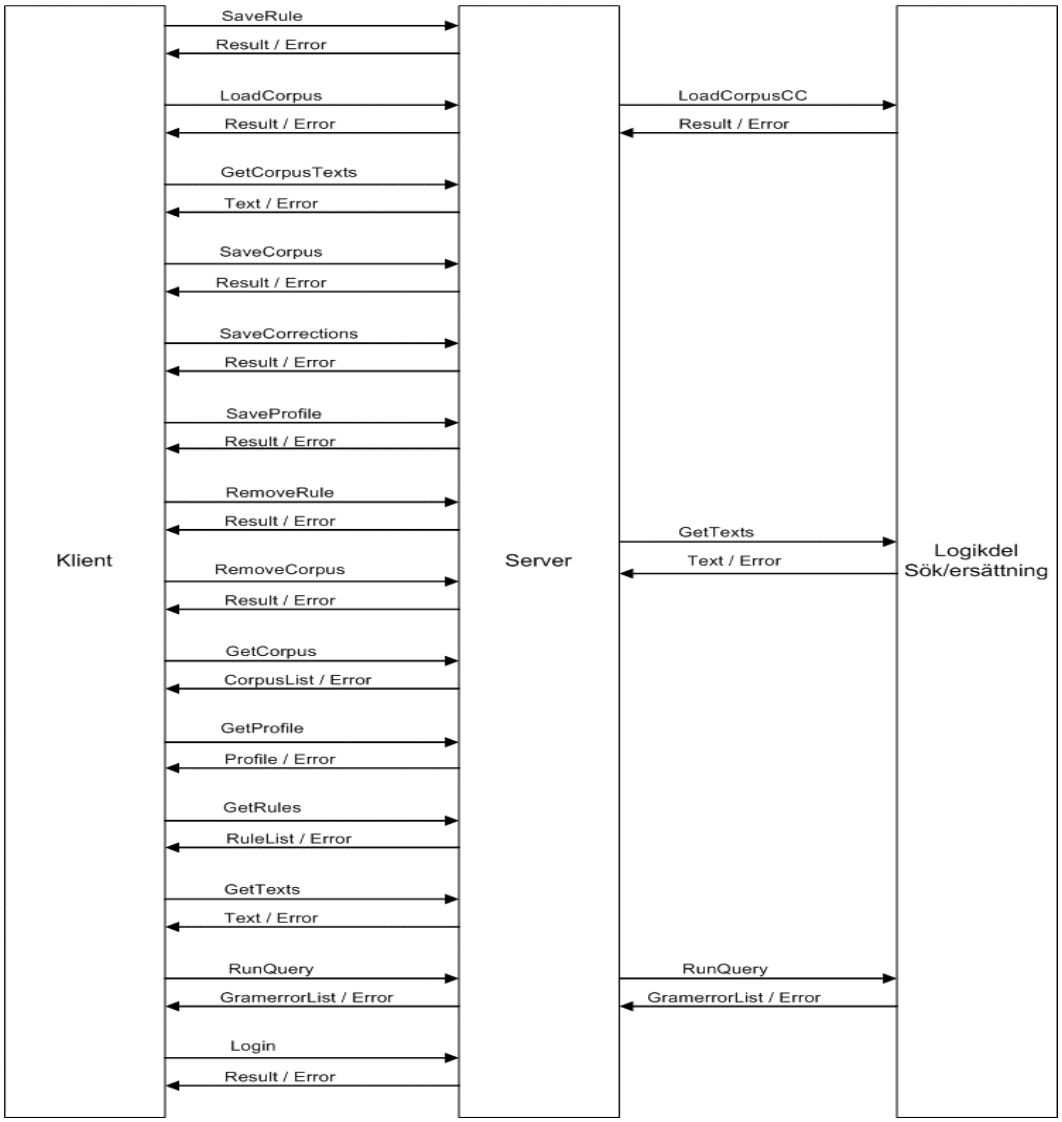



*Bild 5 visar nätverkskommunikationen för hela verktyget.*

För mer detaljerad information om meddelandeprocessen i systemet se bilaga A: användarfall.

**Synkron överföring**

Alla anropen till servern sker synkront. Vid varje anrop kommer klienten att vänta på servern, som processar anropet och skickar ett svar tillbaks. Varje anrop resulterar i således i ett svar. Detta gäller även för anropen mellan server och logikdel. I en utbyggd version av verktyget skulle asynkrona anrop kunna användas. På så vis slipper användaren vänta på svar. Ett annat alternativ är att använda trådar.

## 4.5 Det grafiska gränssnittet

Det grafiska gränssnittet är implementerat i Java med hjälp av swingbiblioteken. Javaswing tillhandahåller många användbara komponenter som jag utnyttjat i implementationen (t.ex. tabeller). Komplexiteten under implementationen av gränssnittet var till största del att lära sig och förstå hur swingpaketet är uppbyggt. Java swing är den senaste generationen av klasser för att bygga gränssnitt för javaapplikationer. Detta paket är väldigt kraftfullt och flexibelt men har också en hög inlärningströskel och är svårt att bemästra till fullo. Vid implementationen har jag försökt att följa de designregler som angivits i avsnitt 3.2 för att skapa ett återanvändbart, utbyggbart och skalbart system. Vid implementationen av användargränssnittet har jag också följt riktlinjerna, specificerade i avsnitt 3.1, för att uppnå god användbarhet. Se appendix D som visar ett övergripande klassdiagram med de viktigaste klasserna.

### 4.5.1 Inloggningsfönster

Från detta fönster kan de behöriga användarna logga in. På servern finns en fil med de behöriga användarna inlagda. Om användaren är behörig, kommer denne till det grafiska användargränssnittet (GUI).

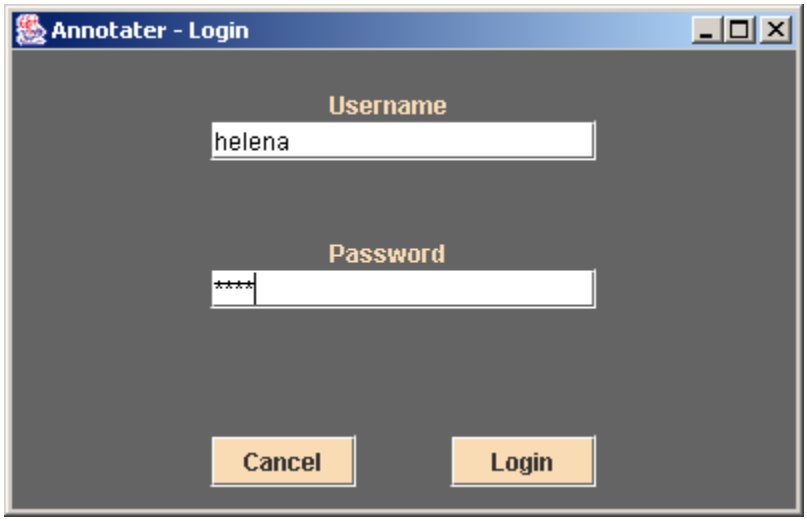



*Bild 6 visar inloggningsfönstret.*

## 4.5.2 Huvudfönster

Från huvudfönstret når man alla andra fönster. Användaren går in och väljer under menyerna för att kopplas till underfönster. Huvudfönstret innehåller ett synligt regelfönster där en regel kan skrivas eller hämtas in. Det finns också ett svarsfönster där resultatet av en körning presenteras. En viktig riktlinje är att viktiga funktioner skall placeras extra synliga [10]. Detta har jag försökt att följa. För att exekvera en körning, måste följande kriterier vara uppfyllda: användaren har matat in en regel, en korpus har valts och laddats.

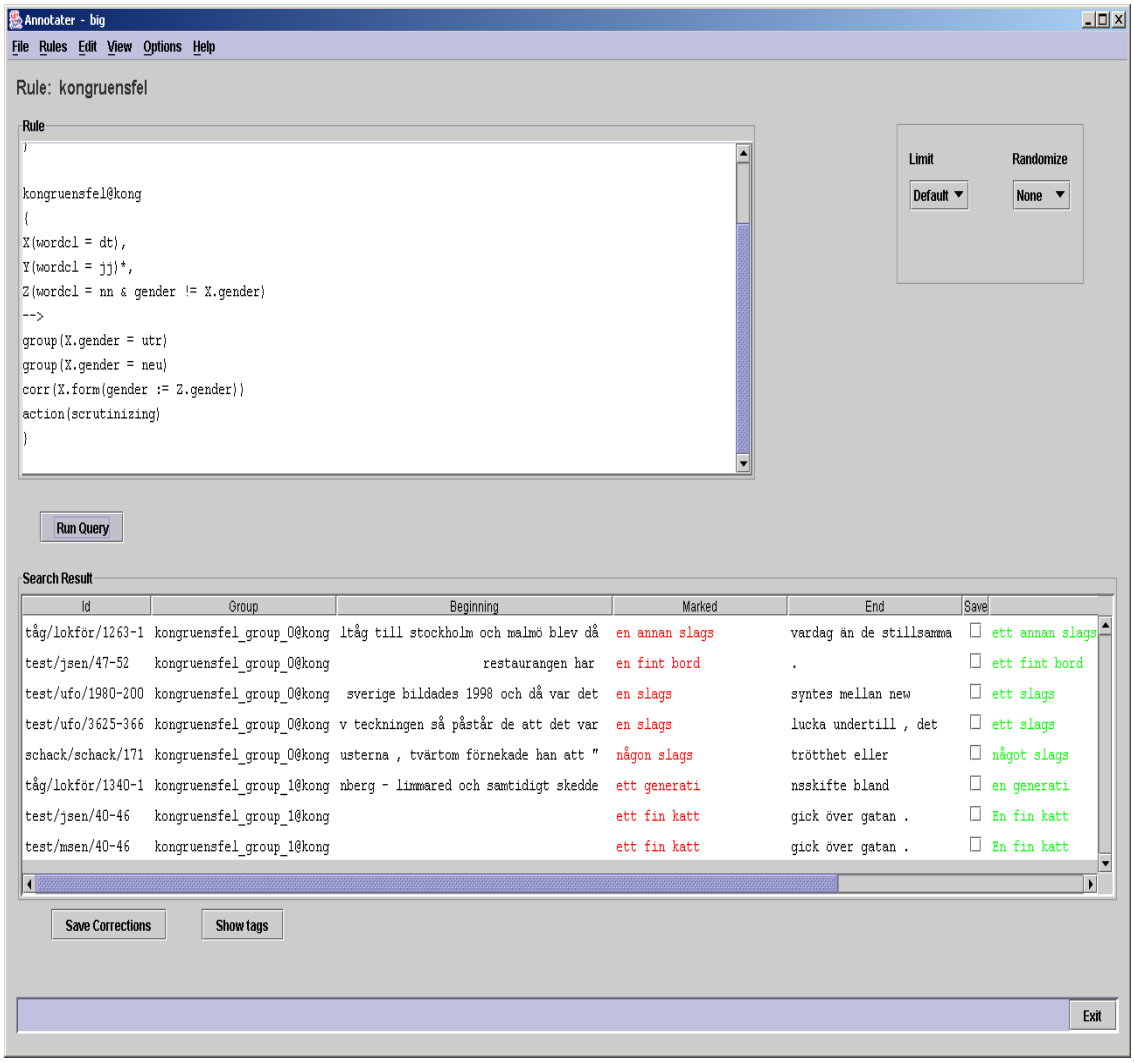



*Bild 7 visar huvudfönstret. En användare har kört en sökfråga.*

### 4.5.3 Editera korpus

Dialogrutan i bild 8 får man genom att under menyn ”file” välja ”edit corpus”. Dialogrutan visar en lista med alla tillgängliga texter från korpus-/indexkatalogen., där de texter som tillhör korpusen är markerade. Listan används för att användaren skall ha möjlighet att lägga till/ta bort ingående texter till korpusen. (Se bilaga A användarfall: *Ändra korpus)*.

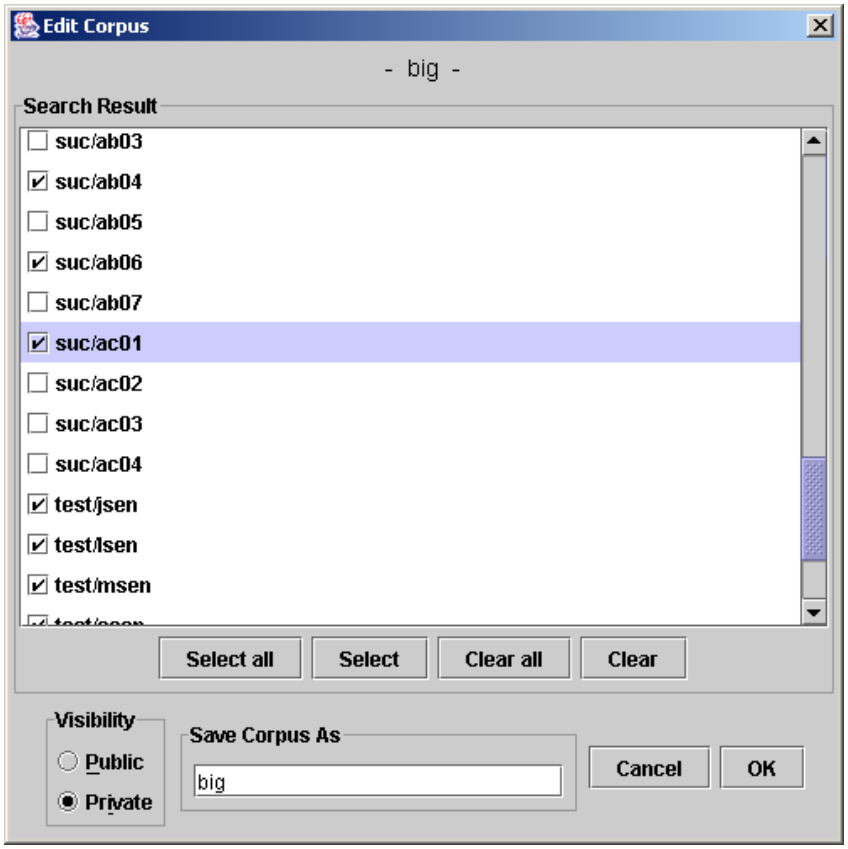



*Bild 8 visar de ingående texterna till korpusen ”big”. ”Visibility” talar om att korpusen ligger i användarens egen katalog. De till korpusen redan tillhörande texter är markerade de övriga är icke markerade. Användaren kan markera/avmarkera nya texter.*

### 4.5.4 Skapa korpus

Genom att i menyn ”file” välja ”new corpus” erhålls en dialogruta som visar en lista med alla tillgängliga texter. Dessa texter fås från korpus-/indexkatalogen på logiksidan. I listan kan sedan de texter som skall ingå i korpusen markeras. Användaren kan sedan spara korpusen under valfritt namn.

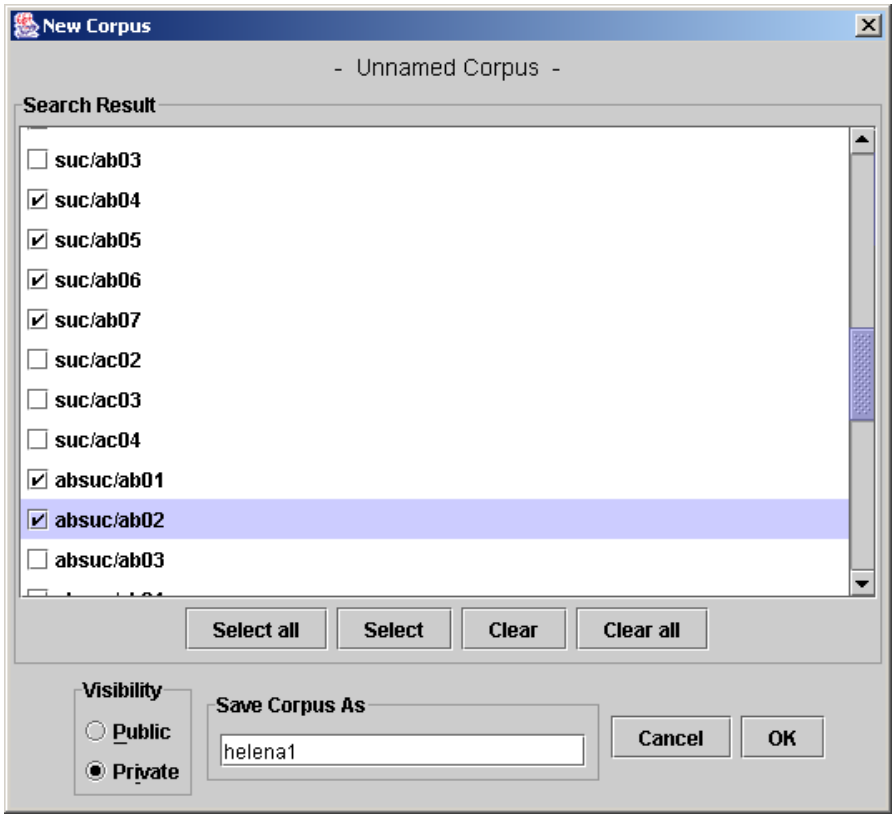



*Bild 9 visar alla texter som överhuvudtaget finns i systemet. Användaren har markerat några av texterna och valt ett namn. När användaren trycker ”ok” sparas korpusen ”helena1” i användarens katalog som en fil med namnen på de markerade texterna.*

### 4.5.5 Välja korpus att köra mot

Genom att i menyn ”file” välja ”load korpus” erhålls en dialogruta med alla för användaren tillgängliga korpusar. En tillgänglig korpus är antingen de korpusar en användare har sparat i sin katalog eller de allmänna korpusar som finns tillgängliga för alla användare (se bild 4). Dialogrutan visar en lista med alla korpusar. Genom att markera en korpus och trycka ”load” händer två saker: korpusen visas i titelfältet och korpusen laddas på logikdelen med alla ingående texter. För mer detaljerad beskrivning av flödet se bilaga A, användarfall: *Load Coprus*.

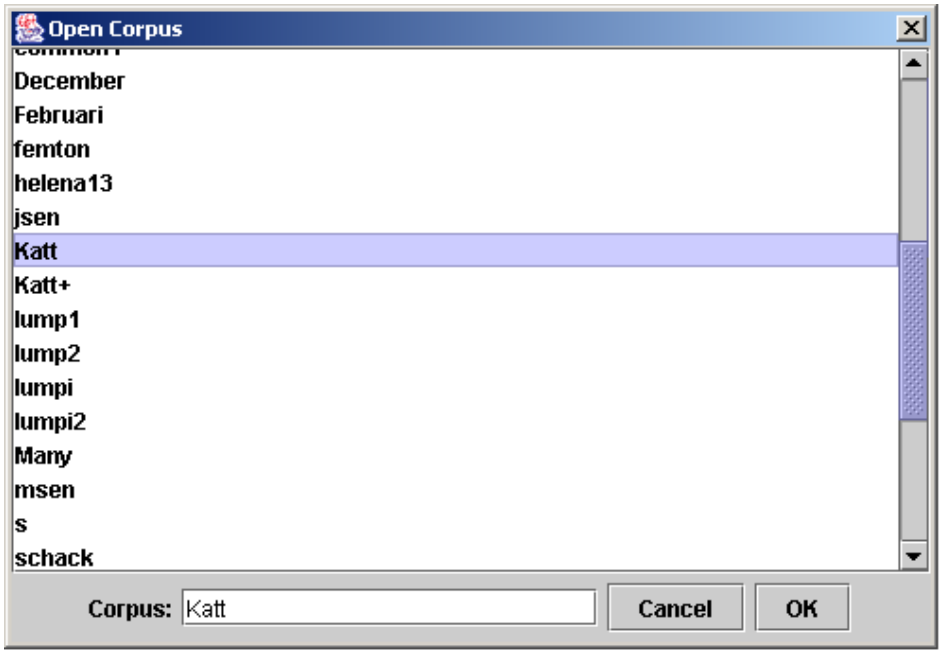



*Bild 10 visar alla virtuella korpusar som är tillgängliga för användaren. En tillgänglig korpus är antingen en korpus sparad i användarens katalog i användarkatalogen eller en allmän korpus, tillgänglig för alla behöriga användare.*

### 4.5.6 Regelfönster

I regelfönstret skrivs granskningsregeln användaren vill köra. Endast en granskningsregel åt gången kan exekveras. I en utbyggd version av verktyget skulle flera granskningsregler åt gången kunna köras.

Jag ger en kort beskrivning nedan av hur en regel måste se ut utan att gå in på för mycket detaljer. För mer information om matchningsregler se [9].

Granska har särskilda granskningsregler som genomsöker resultatet från taggningen på jakt efter grammatiska felaktigheter. En granskningsregel i Granska för att finna kongruensfelet *en hus* ser ut på följande sätt:

```
kong22@inkongruens
{
X(wordcl=dt),
Y(wordcl=jj)*,
Z(wordcl=nn & (gender!=X.gender | num!=X.num |
spec!=X.spec))
-->
mark(X Y Z)
corr(X.get_form(gender:=Z.gender, num:=Z.num,
spec:=Z.spec) Y Z)
info("Artikeln" X.text "stämmer inte överens med
substantivet" Z.text)
action(granskning)
}
```

Regeln har två led som avskiljs med en pil. I det vänstra ledet anges vad som skall matchas i indata. I det högra ledet anges vad som skall göras med det matchade i första ledet. Det högra ledet består av olika fält. I dessa fält anges vad man vill göra med de matchningsvariabler, som finns i det vänstra ledet. Alla fält utom `action` är frivilliga. De regler en användare vill köra följer Granskas syntax ovan. Dock finns en extra utvidgning i det nya verktyget: group. Group används om användaren vill gruppera de svar som presenteras på GUI:t med avseende på t.ex.ordklass.

### 4.5.7 Visa sparade regler

Dialogrutan i bild 11 fås genom att under menyn ”rule” välja ”open rule”. Fönstret visar en lista med alla tillgängliga regler. Dessa regler får man från användarkatalogen. Genom att markera en regel i listan och trycka ”ok” visas regeln i regelfönstret.

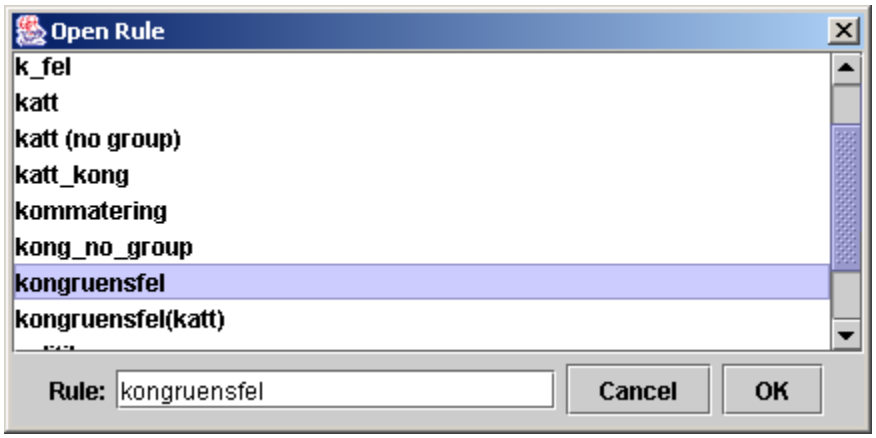



*Bild 11 visar alla regler en användare tidigare sparat under sin katalog i användarkatalogen på servern.*

### 4.5.8 Svarsfönster

Svarsfönstret är implementerat i form av en tabell. Varje rad i tabellen representerar ett svar från körningen, d.v.s. varje rad visar en funnen matchning i korpusen. Tabellen innehåller 7 kolumner enl. följande: Kolumn 1 visar vilken text och var i texten matchningen hittats. Kolumn 2 visar grupptillhörighet. Kolumn 3–5 visar den matchade meningen. Uppdelningen av meningen i 3 kolumner har jag gjort för att den markerade delen av meningen, den del av meningen där matchningen hittats, skall centreras i mitten (kolumn 4) så att användaren får en bättre överblick. På så vis behöver inte användaren skrolla i tabellen om meningen skulle vara lång. Kolumn 6 visar en checkbox, där användaren kan markera de svar denne vill spara. Kolumn 7 visar rättningsförslaget till den markerade delen i meningen. Svarsfönstret innehåller en knapp *show tags*. Användaren kan genom att trycka på knappen få tagginformation till varje ord i meningen. Knappen ”save corrections” används för att spara de i tabellen markerade svaren på en loggfil. Tabellen kan också sorteras på de olika kolumnerna.

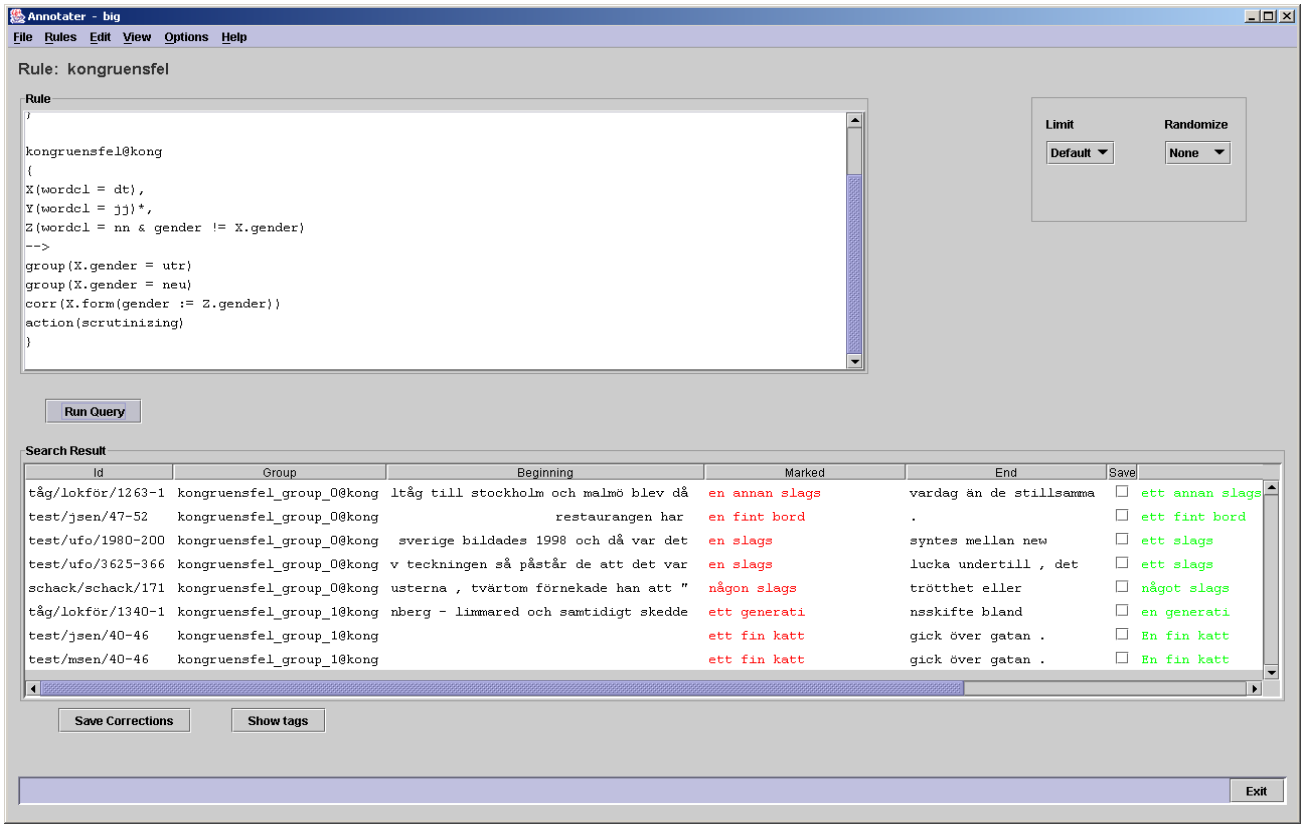



*Bild 12 visar resultatet av en körning av korpusen ”big”. Den felaktiga delen av meningen är centrerad i mitten med röd text. Rättningsförslaget längst till höger är markerad med grön text.*

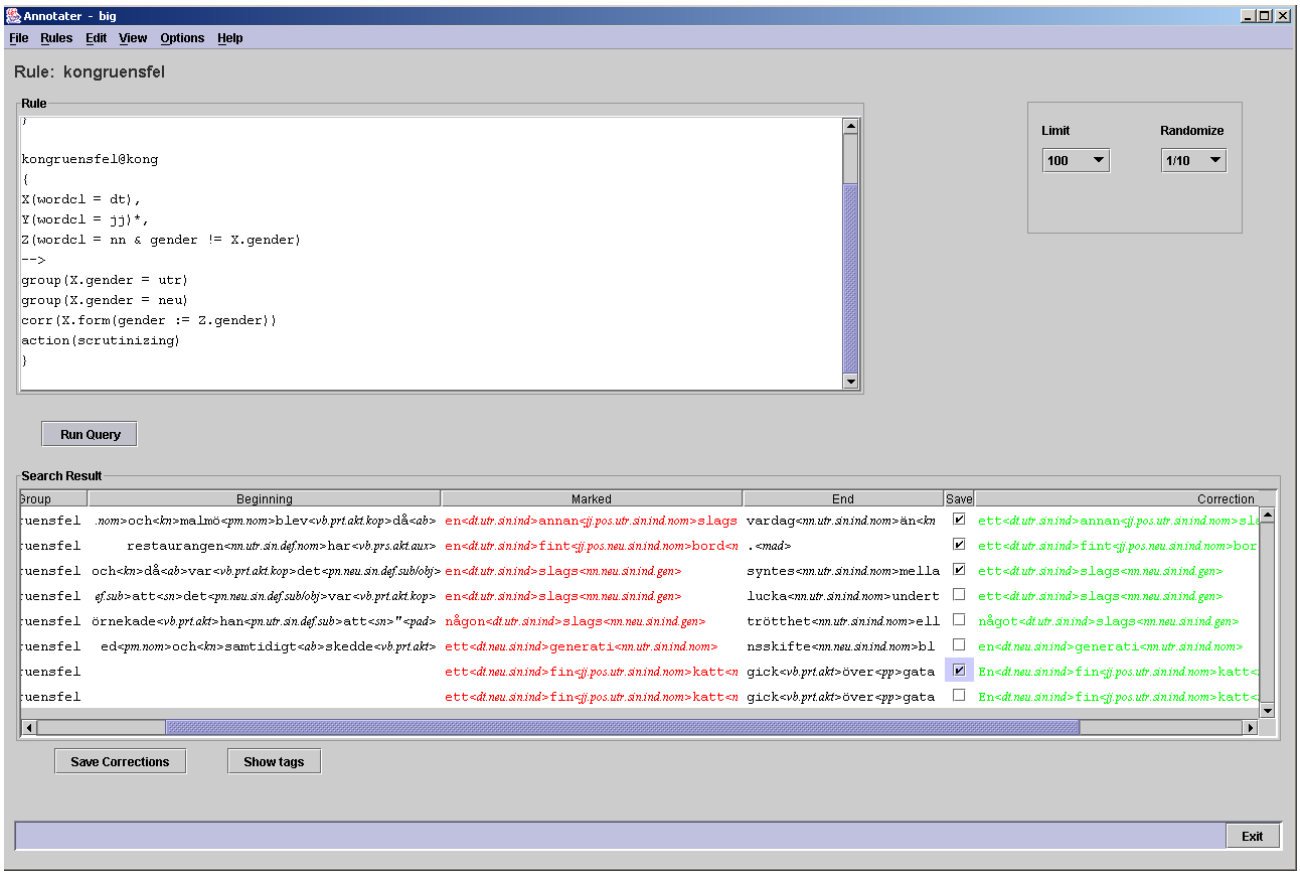



*Bild 12 visar svarsfönstret med tagginformation till varje ord. Taggen visas alltid med avvikande stil.*

## 4.5.9 Loggfil

De svar (ett svar motsvaras av en rad i tabellen) en användare vill rätta i korpusen rättas inte i korpusen direkt utan sparas på en loggfil istället. Loggfilen används sedan för automaträttning av korpusen, vid ett senare tillfälle. Loggfilen är skriven på XML-format och ligger fysiskt på servern. De svar en användare vill rätta adderas sedan till loggfilen i sekventiell ordning. I loggfilen sparas varje ord som skall ändras i meningen med det felaktiga ordet (`before`) och det rätta ordet (`after`), vem som gjorde rättningen (`user`) samt var och i vilken text det felaktiga ordet finns.

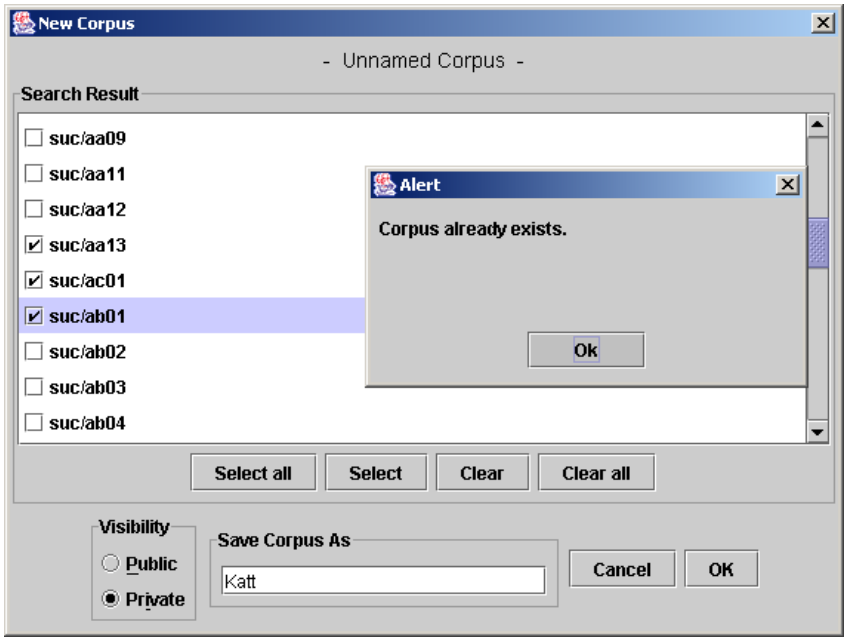
```
<?xml version="1.0" encoding="UTF-8" ?>
<save_corrections>
```

```
  <correction>
     <user>andreas</user>
     <date>2002-12-07 15:09:51</date>
     <file> suc/txt4711 </file>
     <id>txt4711.66</id>
     <before>fint</before>
     <after>fin</after>
  </correction>
  <correction>
     <user>andreas</user>
     <date>2002-12-07 15:09:51</date>
     <file> suc/txt4711 </file>
     <id>4711.67</id>
     <before>stort</before>
     <after>stor</after>
  </correction>
  <correction>
      <user>helena</user>
      <date>2002-12-15 13:10:05</date>
      <file>suc/txt4713</file>
      <id>4713.145</id>
      <before>blod</before>
      <after>blodsockret</after>
  </correction>
  <correction>
      <user>helena</user>
      <date>2002-12-15 13:10:05</date>
      <file>suc/txt4713</file>
      <id>4713.146</id>
      <before>sockret</before>
      <after> </after>
  </correction>
</save_corrections>
```

*Bild 14 visar loggfilen, där rättningarna till ”En fint stort katt” och ”blod sockret” visas. Om ett ord skall tas bort i texten markeras detta med ett tomt afterfält. Om ord skall läggas till i texten visas detta som tomt beforefält och suffix i idfältet.*

### 4.5.10 Modal meddelandebox

Jag har valt att låta systemet presentera felmeddelanden till användaren i form av en meddelandebox i det grafiska användargränssnittet. Meddelandeboxen är modal, d.v.s. användaren måste trycka ”ok” för att komma tillbaka till huvudfönstret. På så vis blir användaren uppmärksammad på vad som gick fel i systemet och var felet uppstod. Ett fel kan ju uppstå var som helst i systemet: GUI, server eller logikdel. Om ett fel uppstår vid ett anrop till server eller logikdel returneras ett felobjekt med felinformationen till klienten. Att använda en dialogruta för felmeddelanden ger användaren förståelse vad som sker i systemet. Jag har försökt göra felmeddelandena meningsfulla och förståliga enligt riktlinjer [10].




*Figur 15. Användaren har fyllt i ett namn på korpus som redan existerar. Ett meddelande om detta presenteras.*


*Bild 16. Kontakten med servern är nere.*

## 4.6 Server

Servern är implementerad med hjälp av Java. Den består i huvudsak av serverfunktionalitet och kataloghanteringsfunktionalitet.

Servern är konstruerad enligt bild 17. Den lyssnar efter klienter som ansluter sig. När detta sker skapas en *Task* som läser XML-data från klienten och skapar ett kommandoobjekt som exekveras av servern. Ett kommandoobjekt kan vara en av subtyperna: *GetRule, LoadCorpus, RunQuery*, etc (se bild 5). När ett kommando är klart returneras ett svarsobjekt. Ett svarsobjekt kan vara en av subtyperna; *Result, Error, Text, CorpusList, Profile, RuleList, GramerrorList* (se bild 5). Varje *Task* är en egen tråd vilket gör att servern kan klara av flera kommandon samtidigt. Eftersom kommunikationen mellan klient och server sker med XML över sockets och XML har en ganska stor overhead även om man designar med detta i åtanke, så har för att minimera mängden bytes som skicka till och från servern, XML-data komprimerats till zip-format. Indata mappas med hjälp av JAXB till ett objekt som sedan ett kommandoobjekt ärver av.

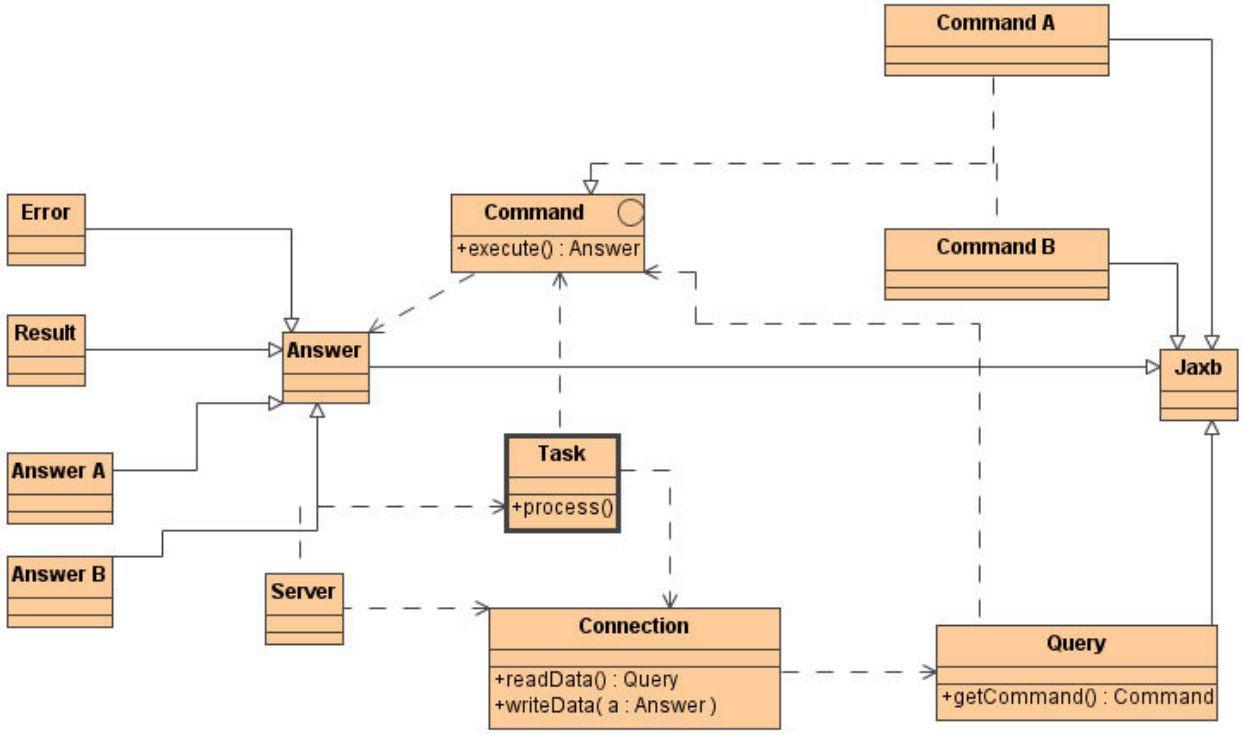



*Bild 17 visar ett klassdiagram med de viktigaste klasserna för servern. De olika kommandona t.ex. Get Corpus, Get Rule och Login som en klient kan ställa till servern visas i diagrammet ovan som Command.*

Bild 18 visar ett sekvensdiagram som visar huvuddragen i hur klienten kommunicerar med servern. De olika svarsobjekten, eller kommandona, som returneras visas som *Answer* i diagrammet. Alla kommandona (*command)* innehåller en *execute()*-metod som beroende på subtyp exekverar sin uppgift och returnerar tillhörande svarsobjekt (*Answer)* tillbaka till klienten. Exempelvis kommer execute()-metoden i ett *RunQuery* kommando att anropa logikdelen, vänta på svar och returnera ett *GrammerrorList*-kommando tillbaks till klienten.

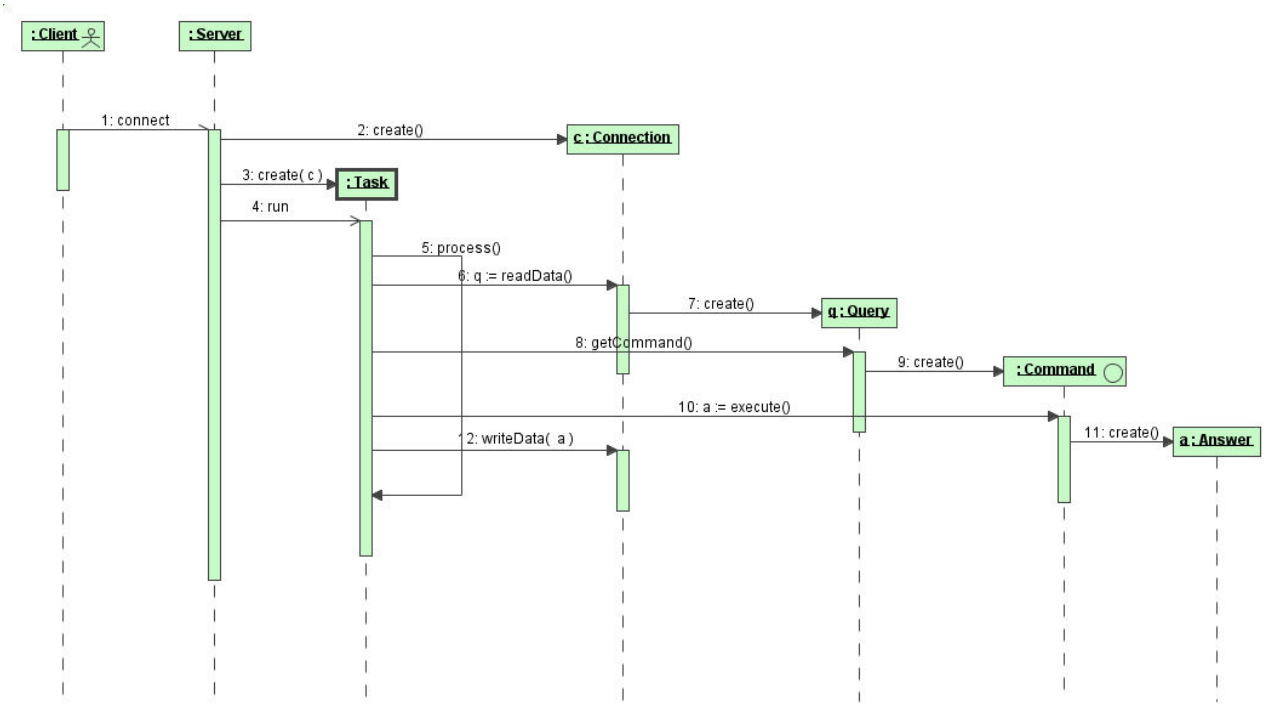



*Bild 18 visar analyssekvensdiagram av servern.*

För de kommandon (object) som behöver anropa logikdelen, sker detta också med XML och över sockets.

# 5 Körning av prototypen

Det finns naturligtvis många sätt att mäta hur bra ett system fungerar. Jag har dock valt att begränsa mig till att endast göra ett enkelt prestandatest på de tidskritiska och viktigaste operationerna i systemet. Vid en tidigare analys fann jag att den stora flaskhalsen kan komma att bli nätverksöverföringen av två skäl:

- Valet att låta servern vara en platformsoberoende, separat, komponent ökar nätverksaktiviteten.
- Storleken på data kan bli stort. En virtuell korpus kan ha i storleksordningen 100000 ingående texter samt en körning av sökfråga kan resultera i många svar.

Vid testet har jag låtit klienten kommunicera med servern via ett 1024 Kbit nätverk. De tider som presenteras innefattar nätverksöverföring och presentationsevaluering på GUI:t. Jag har exkluderat de exekveringstider som logikdelen behöver.

**Hämta texter**
Tabellen visar tiden för en användare att skapa ett nytt korpus från GUI:t. Tiderna inkluderar nätverksöverföring logikdel–server, server– klient och presentation på GUI:t.

*Tabell 1. Tiden att hämta texter från korpus-och indexkatalogen.*

| **Antal texter** | **Storlek** | **Tiden (sek)** |
|---|---|---|
| 5000 | 164 KB | 1.5 |
| 10000 | 348 KB | 1.9 |
| 100000 | 3256KB | 7 |

**Editera texter**
Tabellen visar tiden för en användare att ändra i ett befintligt korpus från GUI:t. Tiderna inkluderar nätverksöverföring samt presentation.

*Tabell 2. Tiden att ändra en befintlig virtuell korpus.*

| **Antal texter** | **Storlek** | **Tiden (sek)** |
|---|---|---|
| 5000 | 164 KB | 1.6 |
| 10000 | 348 KB | 2.1 |

**Exekvering**
Tabellen visar tiden för att köra en matchningsregel mot en korpus. Tiderna inkluderar nätverksöverföring och presentation på GUI:t.

*Tabell 3. Tiden att köra och presentera en matchning.*

| **Antal svar** | **Storlek** | **Tiden (sek)** |
|---|---|---|
| 100 | 20 KB | 2.5 |

# 6 Slutsats

Målet med detta examensprojekt var att skapa ett helt nytt verktyg för utveckling inom datorstödd språkgranskning, med tillhörande grafiskt gränssnitt. En viktig del av examensarbetet bestod i att välja lämplig utvecklingsmiljö till verktyget. Att skapa ett eget protokoll med XML och sockets för nätverksöverföringen visade sig vara en snabb kombination. Svarstiderna tillbaks till GUI:t var tillfredställande. Att hålla svarstiden kort vid körning av operationerna var för mig en mycket viktig riktlinje att följa [11].

Innan implementationen påbörjades gjorde jag en analys och design av systemet som jag formaliserade med hjälp av UML. Att jag valde att lägga ner tid på analysen resulterade i att både server och GUI är utvecklade med avseende på utbyggbarhet, återanvändbarhet och skalbarhet. Att sedan göra implementeringen gick därför mycket smärtfritt.

Ett annat val var att implementera servern som en separat komponent som kommunicerar med klient och logikdel via nätverk. Fördelen med denna arkitektur är att servern blir platformsoberoende.

För det grafiska gränssnittet har även användarvänlighet varit en viktig aspekt. Att skapa ett användarvänligt grafiskt gränssnitt är en kreativ process med mycket inslag av konstnärlighet. Implementationen visade sig vara mycket svårare och mer tidskrävande än jag från början förutspått. Resultatet blev inte helt tillfredställande. Framförallt tycker jag designen brister i användbarhet.

# 7 Litteraturförteckning


[1] Grady Booch, James Rumbaugh & Ivar Jacobson, 1999, *The Unified Modeling Language User Guide*, Addison-Wesley, Massachusetts, ISBN 0-201-57168-4

[2] Fundamentals of swing, part 1 [www]. Tillgängligt på <http://developer.java.com/developer/onlineTraining/GUI/swing>. Hämtat 5 september 2002.

[3] Fundamentals of swing, part 2 [www]. Tillgängligt på <http://developer.java.com/developer/onlineTraining/GUI/swing2/ >. Hämtat 5 september 2002.

[4] Nils-Erik Gustafsson & Yngve Sundblad, 1997, *Material om interaction mellan människa och dator*, Nada KTH

[5] Cay s.Horstmann & Gary Cornell, 2002, *Core java, volume II-advanced features,* Sun Microsystems, Palo Alto, ISBN 0-13-092738-4

[6] Darrel Ince, 2002, *Developing Distributed and E-commerce Applications*, Addison Wesley, Edinburgh Gate, ISBN 0 201 73046 4

[7] Introduction to the Java Architecture for XML Binding (JAXB) [www], Tillgängligt på <http://sunflach.sun.com/49/4/ja/index.shtml>. Hämtat 20 oktober 2002

[8] Java Technology and XML [www]. Tillgängligt på http://java.sun.com/xml/. Hämtat 14 september 2002.

[9] Ola Knutsson, 2001, *Automatisk språkgranskning av svensk text,* Kungliga Tekniska Högskolan, Stockholm, ISBN 91-7283-052-2

[10] Craig Larman, 2002, *Applying uml and patterns, second edition*, Prentice Hall, Upper Saddle River, ISBN 0-13-092569-1

[11] Peter van der Linden, 2002, Just java, fifth edition, Sun Microsystems, Palo Alto, ISBN 0-13-032072-2

[12] Jakob Nielsen 1993, *Visability Engineering*, Academic Press, Boston, ISBN 0-12-518405-0

[13] Programmer to programmer, 2002, *Java Server Programming, J2EE edition*, Wrox Press Ltd, Arden House, ISBN 1-861004-65-6

[14] SUC [www]. Tillgängligt på http://www.ling.su.se/DaLi/Projects/SUC/. Hämtat 15 november 2002.

[15] Ann Wollrath & Jim Waldo, An Overview of RMI Applications [www]. Tillgängligt på http://java.sun.com/docs/books/tutorial/xml/overview.html. Hämtat 4 juli 2002.

# Bilagor

## Bilaga A: Användarfall

Aktörer: Klient, server, logikdel, användare

**Logga in:**

Huvudflöde:
1. Användaren fyller i lösenord och användarnamn och startar ny inloggning.

2. Klienten skickar ett meddelande till servern för lösenord och användarkontroll.

3. Meddelande skickas tillbaks till GUI:t med resultat av inloggningen.

4. Användaren erhåller huvudfönstret.

Alternativa flöden:
4a Klienten signalerar att användaren ej är behörig genom att presentera ett meddelande på inloggningsfönstret.

Förhandsvillkor: Användaren har erhållit inloggningssidan.

Efterhandsvillkor Användaren är inloggad och har erhållit huvudfönstret.

**Hämta texter:**

Huvudflöde
1. Användaren går in under menyn "file" och väljer "new corpus".

2. Klienten skickar meddelande till servern att hämta alla tillgängliga texter.

3. Meddelandet skickas vidare till logikdelen, som returnerar alla tillgängliga texter.

4. Servern returnerar meddelandet med alla tillgängliga texter.

5. Texterna presenteras för användaren.

Alternativa flöden
4a. Felmeddelande returneras
1. Felmeddelandet presenteras i ett pop-up-fönster.

Förhandsvillkor: Användaren har loggat in och erhållit huvudfönstret.

Efterhandsvillkor: Användaren har erhållit alla tillgängliga texter i systemet.

**Skapa virtuell korpus**

Huvudflöde:
1. Användaren väljer de texter som skall ingå i korpusen och sparar korpusen med ett namn.

2. Klienten skickar ett meddelande till servern att spara korpusen med texterna.

3. Servern sparar korpusen i användarkatalogen, innehållande namnet på alla ingående texter.

4. Ett resultatmeddelande skickas tillbaks till klienten.

5. Om operationen lyckades informeras användaren om detta i en statuspanel.

Alternativa flöden:
5a Resultatmeddelandet
om operationen misslyckades informeras användaren genom ett pop-up-fönster.

Förhandsvillkor: Användaren har erhållit namnen på alla tillgängliga texter i en lista.

Efterhandsvillkor: En ny korpus har sparats i användarkatalogen.

**Markera/avmarkera texter**

Huvudflöde:
1. Användaren går in under menyn "file" och väljer "edit corpus".

2. Klienten skickar meddelande till servern att hämta alla tillgängliga texter.

3. Meddelandet skickas vidare till logikdelen, som returnerar alla tillgängliga texter.

4. Servern returnerar meddelandet med alla tillgängliga texter till klienten.

5. Klienten skickar ett meddelande att hämta alla texter tillhörande det valda korpusen till servern.

6. Servern anropar kataloghanteringssystemet som hämtar namnen på alla texter som tillhör korpusen och returnerar dessa.

7. En beräkning görs på klienten för att avgöra vilka texter som ingår i korpusen respektive inte ingår i korpusen.

8. Klienten presenterar en lista med alla texter, där de texter som ingår i korpusen markerade och som ej ingår avmarkerade.

Förhandsvillkor: Användaren har loggat in och erhållit huvudfönstret.
Användaren har laddat en korpus.

Efterhandsvillkor: Användaren har erhållit en lista med markerade/avmarkerade texter.

**Editera korpus:**

Huvudflöde:
1. Användaren markerar/avmarkerar de texter som nu skall ingå i korpusen, och trycker "save".

2. Ett meddelande skickas till servern att spara korpusen med de nya texterna.

3. Servern sparar korpusen på användarkatalogen. Ett resultatmeddelande skickas tillbaks till klienten.

4. Resultatet presenteras på en statuspanel.

Alternativt flöde:

4a Resultatmeddelandet är ett felmeddelande
1. Felmeddelandet visas i ett pop-up-fönster.

Förhandsvillkor: Användaren har erhållit en lista med markerade/avmarkerade texter.

Efterhandsvillkor: Användaren har ändrat korpusen och sparat på nytt i användarkatalogen.

**Ladda korpus**

Huvudflöde:
1. Användaren går in under menyn "file" och väljer "Load corpus".

2. En lista med alla tillgängliga korpusar visas.

3. Användaren väljer vilken korpus som skall laddas i listan och trycker ”ok”.

4. Ett meddelande skickas till servern innehållande korpus och användarnamn.

5. Alla texter som tillhör korpusen hämtas i användarkatalogen och skickas över till logikdelen för laddning.

6. Ett meddelande skickas tillbaks till GUI:t om att korpusen laddats.

7. Meddelandet presenteras i statuspanelen för användaren.

Alternativt flöde:
7a Resultatmeddelandet är ett felmeddelande
1. Felmeddelandet visas på ett pop-up-fönster.

Förhandsvillkor: Användaren har loggat in och erhållit huvudfönstret.

Efterhandsvillkor: Användaren har laddat en korpus på logikdelen.

**Radera korpus**

Huvudflöde:
1. Användaren går in under menyn "file" och väljer "Remove corpus" och

trycker sedan "ok".

2. Ett meddelande skickas till servern att ta bort korpusen i användarkatalogen.

3. Servern tar bort korpusen och returnerar ett meddelande om hur det gick.

4. Meddelandet presenteras i statuspanelen för användaren.

Alternativt flöde:
4a Resultatmeddelandet är ett felmeddelande
1. Felmeddelandet visas i ett pop-up-fönster.

Förhandsvillkor: Användaren har loggat in och erhållit huvudfönstret.

Efterhandsvillkor: Användaren har tagit bort en korpus i katalogen.

**Skapa ny regel**

Huvudflöde:
1. Användaren går under menyn ”rule” och väljer ”new rule”. Denne har då möjlighet att skriva in en ny regel i regel fönstret.

2. Användaren skriver in regeln och sparar den genom att välja ”save as” och trycker ”ok”.

3. Ett meddelande skickas till servern att spara regeln.

4. Servern sparar regeln under användarnamnet i användarkatalogen.

5. Ett resultatmeddelande skickas tillbaks till klienten.

6. Om operationen lyckades informeras användaren om detta i en statuspanel

Alternativa flöden:
6a Resultatmeddelandet
1. Om operationen misslyckades informeras användaren genom ett Pop-up-fönster.

Förhandsvillkor: Användaren har loggat in och erhållit huvudfönstret.

Efterhandsvillkor: En ny regel har sparats i användarens katalog.

**Välja regel**

Huvudflöde:
1. Användaren går under menyn ”rule” och väljer ”open rule”.

2. Klienten presenterar alla regler i en lista.

2. Användaren kan välja en regel och trycka ”ok”.

3. Klienten skiver ut regeltexten och namnet i regelfönstret.

Precondition: Användaren har loggat in och erhållit huvudfönstret.

Postcondition: En vald regel visas i regelfönstret

**Editera regel**

Huvudflöde:
1. Användaren går under menyn ”rule” och väljer ”edit rule”.

2. Användaren kan editera regeln i regelfönstret och spara regeln genom att välja ”save” i ”rule” menyn.

3. Klienten skickar ett meddelande till servern som sparar regeln i användarkatalogen.

3. Ett meddelande returneras om resultatet.

4. Meddelandet presenteras i statuspanelen för användaren.

Alternativt flöde:
4a Resultatmeddelandet är ett felmeddelande
  1. Felmeddelandet visas i ett pop-up-fönster.

Precondition: Användaren har valt en regel.

Postcondition: En regel har ändrats och sparats under användarkatalogen.

**Ta bort regel**

Huvudflöde:
1. Användaren går under menyn ”rule” och väljer ”remove rule”.

2. Klienten skickar ett meddelande till servern som tar bort regeln i

användarkatalogen.

3. Ett meddelande returneras om resultatet.

4. Meddelandet presenteras i statuspanelen för användaren.

Alternativt flöde:
4a Resultatmeddelandet är ett felmeddelande
1. Felmeddelandet visas på ett pop-up fönster.

Precondition: Användaren har valt en regel.

Postcondition: En regel har tagits bort i användarens katalog.

**Köra:**
Huvudflöde:
1. Användaren trycker på ”run” i huvudfönstret.

2. Klienten skickar ett meddelande till servern med parametrar.

3. Servern tar emot meddelandet och skickar vidare till logikdelen.

4. Logikservern returnerar resultatet av körningen.

5. Servern skickar vidare resultatet till klienten.

6. Klienten presenterar svaren i ett svarsfönster.

Alternativt flöde:
6a Logikdel hittar inga matchningar.
1 Ett meddelande presenteras i status panelen.
6b. Ett fel har uppstått
1. Felmeddelandet visas i ett pop-upfönster.

Precondition: Användaren har skrivit in en regel i regelfönstret samt laddat en korpus.

Postcondition: Svaren är presenterade i svarsfönstret.

**Spara ändringar**
Huvudflöde:
1. Användaren markerar de ändringar hon/han vill göra i svarsfönstret och trycker ”save corrections”.

2. Klienten plockar ut de markerade svaren i svarsfönstret och
räknar ut vilka ord som skall ändras.

3. Klienten skickar svaren till servern.

4. Servern sparar svaren i en loggfil i användarkatalogen.

5. Servern returnerar resultatet av sparningen.

6. Meddelandet presenteras i statuspanelen för användaren.

.
Alternativt flöde:.
6a. Ett fel har uppstått
1. Felmeddelandet visas i ett pop-up fönster.

Precondition: Användaren har kört en fråga och erhållit ett svar.

Postcondition: Svaren har sparats i en loggfil.

# Bilaga B: API

**Metoder**

---

**saveRule**

public ResultObject **saveRule**(String user, String rulename, String ruletext)

Sparar en regel i katalogen. Returnerar resultatet av sparningen.

**Parametrar:**
user – användarnamn
rulename – namnet på regeln
ruletext – regeltexten

**Returnerar:**
Resultat med meddelande

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**
1.4

---

**loadCorpus**

public ResultObject **loadCorpus**(String user, String corpusName)

Laddar en korpus tillfälligt hos Logikserver under en temporär användarplats

**Parametrar:**
user – användarnamn
corpusName – namnet på den korpus som skall laddas.

**Returnerar:**
Resultat med meddelande

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**
1.4

---

**getCorpusTexts**

public String[] **getCorpusTexts**(String user, String corpusName)

Hämtar från katalogen namnet på de texter som ingår i ett korpus. Returnerar en lista med alla texterna

**Parametrar:**
user – användarnamn
corpusName – namnet på korpuset

**Returnerar:**
Lista med de ingående texterna.

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**
1.4

---

**saveCorpus**

public ResultObject **saveCorpus**(String user, String CorpusName, String[] texts)

Sparar en korpus i katalogen med tillhörande texter. Returnerar resultatet av sparningen.

**Parametrar:**
user – användarnamn

coprusName – namnet på korpuset
texts – namnen på de texter som ingår.

**Returnerar:**
Resultat med meddelande.

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**
1.4

---

**saveCorrections**

public ResultObject **saveCorrections**(List corrections)

Sparar svaren i katalogen i en log fil. Returnerar resultatet av sparningen.

**Parametrar:**
correction – Lista med svar som skall införas i loggfil.

**Returnerar:**
Resultat med meddelande.

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**
1.4

---

**saveProfile**

public ResultObject **saveProfile**(String user, String name, String height,
String width, String ruleText, String corpusName)

En användare kan spara utseende på gränssnittet i sin katalog.

**Parametrar:**
user – användarnamn
name – namnet på profilen
height – höjden på fönstret
Width – bredden på fönstret
corpusName – namnet på regeln
ruleText – regeltexten

**Returnerar:**

Resultat med meddelande.

**Kastar:**

**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**

1.4

---

**removeRule**

public ResultObject **removeRule**(String user, String ruleName)

Tar bort en sparad regel i användarens katalog.

**Parametrar:**

user – användarnamn
ruleName – namnet på regeln

**Returnerar:**

Resultat med meddelande.

**Kastar:**

**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**

1.4

---

**removeCorpus**

public ResultObject **removeCorpus**(String user, String name)

Tar bort ett sparat korpus i användarens katalog.

**Parametrar:**

user – användarnamn
ruleName – namnet på korpuset

**Returnerar:**

Resultat med meddelande.

**Kastar:**

**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**

1.4

---

**getCorpus**

public CorpusObject[] **getCorpus**(String user)

Hämtar alla tillgängliga korpus för en användare i katalogen. Returnerar en lista med alla namnen på korpus.

**Parametrar:**
user – användarnamn

**Returnerar:**
Lista med tillgängliga korpus för en specifik användare.

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**
1.4

---

**getProfile**

public ProfileObject **getProfile**(String user, String name)

Hämtar en profil i användarens katalog. Returnerar de data som är sparade om gränssnittet.

**Parametrar:**
user – användarnamn
name – namnet på profilen

**Returnerar:**
En profil innehållande data om gränssnittet.

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**
1.4

---

**getRules**

public RuleObject[] **getRules**(String user)

Hämtar de regler en användare sparat i sin katalog. Returnerar en lista med alla regler innehållande namn och regeltext.

**Parametrar:**
user – användarnamn

**Returnerar:**
Alla regler en användare har i sin katalog.

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**
1.4

---

**getTexts**

public String[] **getTexts**()

Hämtar alla tillgängliga texter i systemet.

**Returnerar:**
Lista med namnen på alla tillgängliga texter i systemet.

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error object genererats.

**Sen:**
1.4

---

**run**

public GramerrorObject[] **run**(String user, String corpusName, String ruleText,
(String random, String size)

Kör en regel på en tidigare laddat korpus. Returnerar en lista med alla meningar där en matchning hittats, med rättningsförslag.

**Parametrar:**
user – användarnamn
corpusName – namnet på det korpus som skall köras
ruleText – regeln som skall matchas
random – slumpar vilka av svaren som skall returneras
size – anger en gräns på antalet svar som returneras

**Returnerar:**
Lista med alla matchningar med rättningsförslag.

**Kastar:**

**AnnotateException –** om något fel uppstått och ett Error-object genererats.

**Sen:**
1.4

---

**login**

public ResultObject **login**(String user, String password)

Kontrollerar om en användare är behörig att använda systemet.

**Parametrar:**
user – användarnamn
password - lösenord

**Returnerar:**
Resultat med meddelande.

**Kastar:**
**AnnotateException –** om något fel uppstått och ett Error-object genererats eller om användaren ej behörig.

**Sen:**
1.4

---

# Bilaga C: DTD

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ELEMENT name (#PCDATA)>
<!ELEMENT user (#PCDATA)>
<!ELEMENT date (#PCDATA)>
<!ELEMENT info (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT corpusname (#PCDATA)>
<!ELEMENT textname (#PCDATA)>
<!ELEMENT origin (#PCDATA)>
<!ELEMENT errormessage (#PCDATA)>
<!ELEMENT rulename (#PCDATA)>
<!ELEMENT passwd (#PCDATA)>
<!ELEMENT ruleline (#PCDATA)>
<!ELEMENT height (#PCDATA)>
<!ELEMENT width (#PCDATA)>
<!ELEMENT random (#PCDATA)>
<!ELEMENT resultmessage (#PCDATA)>
```

```
<!ELEMENT sugg (#PCDATA)>
<!ELEMENT sentence (#PCDATA)>
<!ELEMENT size (#PCDATA)>
<!ELEMENT file (#PCDATA)>
<!ELEMENT before (#PCDATA)>
<!ELEMENT after (#PCDATA)>

<!ELEMENT ruletext (ruleline+)>

<!ELEMENT gramerror (sentence, suggestions,id, marked_section)>
<!ATTLIST gramerror
  group CDATA #REQUIRED>
<!ELEMENT profile (height,width, ruletext, corpusname)>
<!ELEMENT correction (user, date, file, id, before, after)>
<!ELEMENT gramerrorlist (gramerror+)>
<!ELEMENT suggestions (sugg+)>
<!ELEMENT marked_section (mark)>
<!ELEMENT mark EMPTY>
<!ATTLIST mark
  begin CDATA #REQUIRED
  end CDATA #REQUIRED>

<!ELEMENT rulelist (rule+)>
<!ELEMENT rule (ruletext)>
<!ATTLIST rule
  name CDATA #REQUIRED>

<!ELEMENT corpuslist (corpus+)>
<!ELEMENT text (textname+)>
<!ELEMENT error (origin, errormessage)>
<!ELEMENT result (resultmessage)>
<!ELEMENT corpus (corpusname)>
<!ATTLIST corpus
  visibility (common | user) #REQUIRED>
<!ELEMENT load_corpuscc (textname+)>
<!ATTLIST load_corpuscc
  user CDATA #REQUIRED
  corpusname CDATA #REQUIRED>
<!ELEMENT load_corpus (corpusname)>
<!ATTLIST load_corpus
  user CDATA #REQUIRED>
<!ELEMENT save_rule (rule)>
<!ATTLIST save_rule
  user CDATA #REQUIRED>
<!ELEMENT save_corpus (text)>
<!ATTLIST save_corpus
```

```
  user CDATA #REQUIRED
  name CDATA #REQUIRED>
<!ELEMENT save_profile (profile)>
<!ATTLIST save_profile
  user CDATA #REQUIRED
  name CDATA #REQUIRED>
<!ELEMENT save_corrections (correction+)>
<!ELEMENT get_corpus EMPTY>
<!ATTLIST get_corpus
  user CDATA #REQUIRED>
<!ELEMENT get_profile EMPTY>
<!ATTLIST get_profile
  user CDATA #REQUIRED
  name CDATA #REQUIRED>
<!ELEMENT login EMPTY>
<!ATTLIST login
  name CDATA #REQUIRED
  password CDATA #REQUIRED>
<!ELEMENT get_corpustexts (corpusname)>
<!ATTLIST get_corpustexts
  user CDATA #REQUIRED>
<!ELEMENT get_rules EMPTY>
<!ATTLIST get_rules
  user CDATA #REQUIRED>
<!ELEMENT get_text EMPTY>
<!ELEMENT remove_rule (name)>
<!ATTLIST remove_rule
  user CDATA #REQUIRED>
<!ELEMENT remove_corpus (name)>
<!ATTLIST remove_corpus
  user CDATA #REQUIRED>

<!ELEMENT users (login+)>
<!ELEMENT log (save_corrections+)>

<!ELEMENT run_query (ruletext, corpusname, random, size)>
<!ATTLIST run_query
  user CDATA #REQUIRED>

<!ELEMENT answer (gramerrorlist | corpuslist | text | error | rulelist | profile |
result)>

<!ELEMENT query (run_query | save_rule | save_corrections | save_corpus |
save_profile | get_corpus | get_profile | login | get_rules | get_text | remove_rule |
remove_corpus | load_corpus  | load_corpuscc| get_corpustexts )>
```

# Bilaga D: Klassdiagram

Klassdiagram över de viktigaste klasserna för klienten.