Datavetenskap

Linh Nguyen och Enar Tångring

Javaeditor som stöder för- och eftervillkor

Examensarbete, C-nivå

2000:21

Javaeditor som stöder för- och eftervillkor

Linh Nguyen och Enar Tångring

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en kandidatexamen i datavetenskap. Allt material i denna rapport, vilket inte är vårt eget, har blivit tydligt identifierat och inget material är inkluderat som tidigare använts för erhållande av annan examen.

Linh Nguyen och Enar Tångring

Godkänd, 2000-05-31

Handledare: Martin Blom

Examinator: Stefan Lindskog

Tack

Vi skulle först och främst vilja tacka vår handledare Martin Blom som har gett oss fria händer att utveckla vårt program och för den handledning vi har fått. Tack till Eivind J Nordby som introducerade oss i ämnet och fångade vårt intresse, Jörgen Sigvardsson som tipsade oss om ANTLR [6] samt ANTLR och Jipe [7] för hjälp med parsing och syntax highlighting. Till sist vill vi tacka Object Insight för JVision [9]. Vi sparade många timmars arbete med att rita UML-diagram genom att använda JVision.

Sammanfattning

Idag finns det många bra kommersiella utvecklingsverktyg som ska underlätta för utvecklaren att skapa program. De flesta av dessa utvecklingsverktyg har någon form av syntaktisk kontroll. Något som saknas är utvecklingsverktyg som belyser den semantiska innebörden. Vår uppgift har varit att skapa en källkodseditor för Java som ska informera utvecklaren om den semantiska innebörden av de metoder som han/hon använder sig av. Detta ska ske i form av att för- och eftervillkor för en viss metod visas upp när metoden används. Utvecklaren ska då kunna se om han/hon har följt de för- och eftervillkor som gäller.

Java editor supporting pre- and postconditions

Abstract

There are many commercial tools available for making software development easier. Most of these have some kind of syntactical checking. What is missing though, is functionality to display semantic information. The aim of this project has been to develop a Java source code editor that has the capability of showing semantic information to the developer. This has been done by allowing developers to specify pre- and postconditions for methods. These conditions together with standard javadoc documentation are displayed when a call to the method is typed in. This will prevent misuse of methods.

Innehållsförteckning

Figurförteckning

1 Inledning

Det finns inget allmänt sätt att skriva felfria program och det kommer förmodligen aldrig att finnas det heller. Felsökning är inte direkt kreativt skapande och helst skulle man vilja kunna hoppa över det helt och hållet. Ska man då bara strunta i felsökningen och acceptera att det alltid kommer att finnas fel? Svaret är naturligtvis nej. Det finns nästan inga program som är perfekta, men det går att skriva bättre program. Det är det som vi (blivande) programutvecklare försöker göra. Den här uppsatsen handlar om att ta fram ett utvecklingsverktyg för Java. Detta verktyg ska uppmärksamma utvecklaren och få denne att förstå om han/hon följer de regler som specificerats för de metoder som han/hon använder sig av. Förhoppningen är att skapa bättre, effektivare och mer robusta program.

2 Bakgrund och syfte

Vid Karlstads universitet finns forskargruppen SKUTT som bland annat forskar i att utveckla bättre mjukvara genom bättre semantisk beskrivning. En del i deras arbete är att ta fram en editor, som bland annat ska kunna informera användaren om för- och eftervillkor för en viss metod innan de använder sig av metoden. Vi har då blivit erbjudna att få utveckla en prototyp till denna editor som examensarbete.

Syftet med arbetet är att försöka belysa för programmeraren vilka för– och eftervillkoren är för en metod när han/hon använder sig av denna. I och med att för– och eftervillkor klart framgår ska risken för att använda sig av metoder på ett felaktigt vis minska.

2.1 Syntax och semantik

Att beskriva syntax är relativt enkelt. För ett programspråk finns ett antal regler för hur man ska skriva ett program. Dessa kan beskrivas med ett formellt språk. Följer man inte dessa regler, så kommer kompilatorn att klaga och inget körbart program skapas. Syntaktiska fel uppstår vid kompileringstid, innan ett program skapas. Att beskriva semantik är svårare och görs vanligen med ett naturligt språk, t.ex. engelska. Det är svårt att beskriva semantik med ett formellt språk [1]. Semantiska fel uppstår oftast vid exekveringstid, men kan även upptäckas vid kompileringstid. Semantik som kan kontrolleras vid kompileringstid kallas statisk semantik [1]. Exempel på detta är när en heltalsvariabel tilldelas ett värde av en flyttalsvariabel. Hårt typade programmeringsspråk som Java upptäcker sådana fel vid kompileringstid, medan kompilatorn i mindre hårt typade språk som C ignorerar liknande fel. Vanligast är dock dynamisk semantik som inte upptäcks förrän vid exekveringstid. Med dynamisk semantik menas betydelsen av ett kodstycke [1]. Detta kodstycke skulle kunna vara en metod, en kodrad inom en metod eller ett helt program. Dynamisk semantiska fel uppstår därför att programmeraren inte riktigt förstått innebörden av den metod som han/hon använder. När vi hädanefter nämner semantik så menar vi dynamisk semantik om inget annat anges. Fel som uppstår vid exekveringstid är svårare att upptäcka och oftast mer kostsamma att korrigera, därför är det mycket viktigt att belysa semantiken för programmeraren. Detta görs i nuläget genom att bifoga dokumentation och att inkludera kommentarer i koden.

Följande är ett exempel på när man behöver den semantiska beskrivningen.

```
interface Container
{
    public void remove(java.lang.Object obj);
    public void add(java.lang.Object obj);
}
```

Om man för ett objekt av en klass som implementerar ovanstående gränssnitt utför metodanropet `remove(new GarbageBag())` gör man troligen fel eftersom objektet som ju skapas vid anropet inte redan kan finnas och därför inte borde kunna tas bort. Felet upptäcks inte av kompilatorn eftersom det är ett syntaktiskt korrekt anrop. Om metoden förväntar sig att objektet skall finnas, kommer programmet att krascha. Annars är det troligt att metoden är implementerad så att inget händer om objektet inte finns. För att slippa göra sådana antaganden skulle man kunna göra följande tillägg:

```
interface Container
{
    /** Removes an object from this Container.
     * @pre The object exists in the container. (obj ∈ this)
     * @post The object has been removed. (obj ∉ this)
     */
    public void remove(java.lang.Object obj);
    public void add(java.lang.Object obj);
}
```

När man har denna information vet man vad som gäller för metoden och risken att använda den fel är mindre. Här har för- och eftervillkor enligt kontraktskonceptet, som tas upp i avsnitt 2.2, angivits i javadoc kommentaren. Det är på detta sätt man skall dokumentera sin kod för att dra nytta av funktionaliteten i vår editor. Villkoren är alltså textsträngar som kan vara antingen en beskrivning med ett naturligt språk eller ett formellt uttryck som kan användas för att bevisa att ett program är korrekt. Vår editor gör inget mer med villkoren än att presentera dem som text. Det är upp till programmeraren att följa dem.

2.2 Programmera med kontrakt

Att programmera med kontrakt kan liknas vid ett avtal mellan programmeraren och de metoder som programmeraren använder sig av. En metod har då vissa villkor som måste gälla för att den ska kunna utföra sitt arbete på ett korrekt sätt. Gäller inte dessa villkor så vet man heller inte vad som har gjorts efter metodanropet. Om programmeraren garanterar att förvillkoren gäller innan han/hon använder sig av metoden så garanterar metoden att den ska utföra sin upp uppgift på ett korrekt sätt enligt vad som uttrycks i eftervillkoren [2]. Om man konsekvent följer detta mönster, kan man bevisa ett programs korrekthet.

Under samma tidsperiod som detta arbete utförts, har vi jobbat som laborationshandledare i kursen PUMA. Där har vi fått erfara att många studenter ser kontraktskonceptet som en dokumentationsdetalj som skall fixas till innan inlämning. Andra studenter specificerar för- och eftervillkor noggrant, men följer sedan inte dessa. Vår editor skall underlätta att göra detta på rätt sätt genom att påminna om villkoren när de behövs.

2.3 Tidigare arbeten inom samma område

Vi har tittat på SymantecVisual Café (SVC) och Microsoft Visual J++ (MVJ++). SVC saknar helt stöd för semantik. Den enda information om metoder som presenteras, är den som kan extraheras ur kompilerad javakod (”.class”-filer). I javabytekod finns inte ens namnen på metodparametrar kvar. Det som visas upp kan t.ex. se ut såhär:

```
public void setValues(int, float, char, java.lang.String, int)
```

Den informationen är oftast inte tillräcklig för att utvecklaren skall veta vad han gör när han anropar metoden. Vill man veta mer än syntaxen måste man ha ytterligare ett fönster med dokumentationen öppet samtidigt.

MVJ++ visar upp i princip samma information med tillägget den första meningen i javadoc för en metod när man skriver in den. Detta kan ge viss semantisk information, men är inte tillräckligt enligt oss.

3 Mål

Vi ska ta fram en prototyp till en Javaeditor som ska ha följande egenskaper:

1 Kunna visa för- och eftervillkor om dessa är ifyllda på ett korrekt sätt.

2 Kunna visa en lista med alla metoder och datamedlemmar för en viss klass när användaren refererar till ett objekt av klassen.

3 Ha syntax highlighting

4 Ha de mest grundläggande funktionaliteter som en editor ska ha (ex spara och öppna filer)

5 All funktionalitet ska gå att stänga av och sätta på.

6 Designen av editorn ska vara så att man vid senare tillfälle ska kunna utöka funktionaliteten utan stora problem.

4 Avgränsning

Detta är den avgränsning vi gjorde innan vi satte igång projektet.

- Editorn ska bara känna till klasser i samma paket.
- Ingen automatiskt kodgenerering för kontroll av att villkoren uppfylls ska ske.
- Effektivitet tar vi inte hänsyn till, eftersom vi utvecklar i Java som av sin natur inte är ett snabbexekverande språk.

5 Beskrivning av konstruktionslösningen

5.1 Översiktlig arbetsbeskrivning

Under arbetets gång har vi kontinuerligt, speciellt i början, haft möten med vår handledare. Vi båda har dessutom hela tiden diskuterat med varandra. Det har ej varit möjligt att dela upp arbetet, utom vid vissa speciella tillfällen då vi kunde komma överens om att vi på olika håll skulle satsa på varsin del en kortare tid. En kortfattad dagbok har skrivits av oss båda.

5.2 Val av språk

Det första vi fick bestämma var val av programmeringsspråk. De två språk det stod emellan var C++ och Java eftersom vi behärskar de språken bäst. Om vi valde C++ så skulle programmet gå mycket snabbare, men vi skulle då vara tvungna att lära oss att göra grafiskt gränssnitt i C++ vilket skulle ta tid och plattformsoberoendet skulle försvinna. Om vi valde Java så skulle programmet bli långsammare, men eftersom vi redan kunde det grafiska API som finns i Java, så hade vi det gratis. Det fanns ett tredje alternativ, nämligen att blanda de två språken och använda JNI (Java Native Interface) för att anropa C++ kod. Men i och med att vi blandar två språk så blir inte programmet lika "rent" plus att plattformsoberoendet försvinner med C++ (man behöver åtminstone kompilera om). Vi bestämde oss till slut för att använda oss av enbart Java eftersom vi hittade ett bra verktyg för generering av lexer- och parserkod, ANTLR.

5.3 Parsing

Det svåraste med uppgiften visade sig bli att utifrån en färdigskriven javakod och den fil man editerar extrahera den information som vi behövde. Det som vi behövde ta ut från en färdigskriven javafil var vilka metoder som finns i klassen, javadoc för varje variabel och metod och för- och eftervillkor för varje metod. I den fil som editeras behövde vi ta ut alla importerade klasser och alla deklarerade variabler samt det uttryck som skall avrefereras. Uttrycket som skall avrefereras måste till sist evalueras för att komma fram till vilken klass vi ska visa upp information om.

5.3.1 Lexikalisk analysator/parser

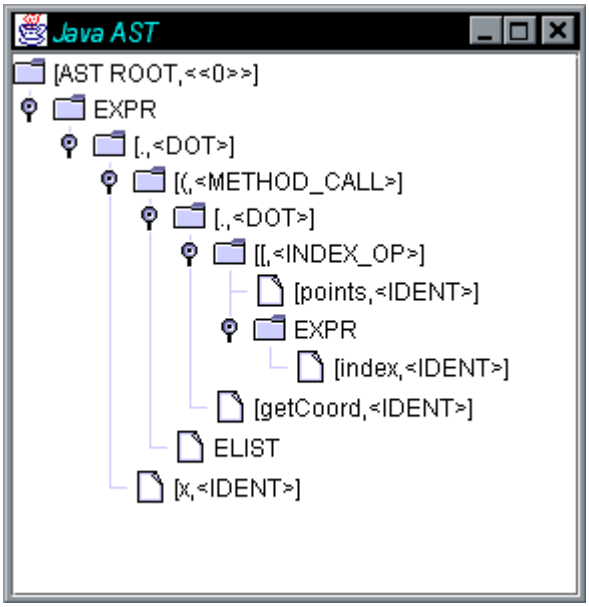
För att kunna underlätta parsingen av javakod behövde vi en lexikalisk analysator/parser. Lex och Yacc var de enda verktygen vi kände till, men de finns bara till C/C++. Vi började leta efter motsvarande Lex och Yacc till Java och hittade (med hjälp av tips från Jörgen Sigvardsson) ANTLR som är en lexer/parser-generator som kan skapa javakod.

Med i ANTLR följde även en grammatik för Java, vilket underlättade stort för oss eftersom vi då inte behövde översätta någon annan grammatikfil till rätt format. Grammatiken skrivs i EBNF (Extended Bachus Naur Form) [1]. Från denna grammatik kan man med ett verktyg generera den javakod som behövs för att parsa. Det som skapas är i princip en lexer och en parser. Lexern ser allt som en ström av tecken och delar upp den i tokens enligt angivna

regler. Detta visas enklast med ett exempel. Kodraden `if(b==true)  b=false;` kommer att delas upp i följande tokens:

```
'if' - IDENT
'(' - LPAREN
'b' - IDENT
'==' - EQUAL
'true' - IDENT
')' - RPAREN
' ' - WHITESPACE
'b' - IDENT
'=' - ASSIGN
'false' - IDENT
';' - SEMI
```

Det är parserns uppgift att läsa denna ström av tokens och kontrollera den mot grammatiken. Kombinationer av tokens matchas mot regler för hur de olika tokens på ett entydigt sätt bildar ett syntaxträd. Nedan visas ett syntaxträd för `points[index].getCoord().x;`.



Figur 1: Exempel på syntaxträd

ANTLR har även stöd för att generera en TreeWalker som traverserar ett syntaxträd och utför olika saker beroende på hur noderna ser ut. Vi har genererat en TreeWalker som vi använder för att rekursivt evaluera ett uttryck som kan avrefereras. Syntaxträdet i Figur 1 är

även ett exempel på ett träd för just ett sådant uttryck. Däremot utnyttjade vi inte en TreeWalker då vi skulle plocka ut information ur syntaxträdet för en komplett javafil.

5.3.2 Parsning av komplett javakodsfil

När vi vill visa upp tillgängliga metoder för en viss klass behöver vi kunna plocka ut den informationen ur en *”.java”*-fil. För att hitta filen låter vi klassladdaren i den virtuella javamaskinen ladda in klassen och leta efter en associerad resurs med javafilens namn. Om klassladdaren inte ens kan ladda in klassen betyder det att klassen inte är kompilerad eller inte finns i classpath. Om filen inte hittas med det förfarandet, så försöker vi hitta den i alla jarfiler som finns i classpath. Vi tyckte det kändes nödvändigt eftersom källkoden till hela JavaAPI ligger i src.jar i den katalog man installerat JDK i. Denna jarfil måste läggas till i classpath om man vill kunna få information om klasser i JavaAPI.

Eftersom ANTLRs grammatik inte inkluderade javadoc, var vi tvungna att lägga till regler om det. Vi har även modifierat hur parsern skall bete sig då den genererar ett träd. Vi vill inte att implementationen av metoder skall finnas med i trädet eftersom det inte är intressant i vårt fall. Ytterligare en fördel som det medför, är att parsningen blir effektivare. När trädet är uppbyggt stegar vi oss igenom det och plockar ut metoder och variabler. För det ändamålet skulle vi ha kunnat använda oss av ANTLRs stöd för att även generera en s.k. TreeWalker som traverserar syntaxträdet och utför olika åtgärder beroende på informationen i noderna. Anledningen till att vi inte gjorde det, var att vi upptäckte den möjligheten efter att redan ha löst problemet. Nu har vi istället skapat en klass, ASTReader, som innehåller metoder för att avläsa noder av vissa typer, t.ex. parameter eller typ. Resultatet från parsingen av en komplett javafil är i princip en lista med alla deklarationer i klassen.

5.3.3 Parsning i filen som editeras

Att plocka fram nödvändig information ur den fil som användaren håller på att skriva i är mycket komplicerat eftersom filen nästan aldrig har ett korrekt utseende. Det går därför inte att specificera några regler för hur den ska se ut. Vi har valt att åskådliggöra detta med ett exempel. Antag att en javakodfil ser ut så här:

```
class SomeClass extends SomeOtherClass
{
    static int count;
    public vojd setCount(int cnt) { count = cnt; }
    public void doAction()
    {
```

```
        int a = 2*count;
    static int count2;
}
```

I fallet ovan kan en mänsklig hjärna lista ut att metoden `setCount` finns i klassen och att `void` är felstavat. Dessutom borde det vara på det viset att man glömt avsluta metodkroppen för `doAction`, men det skulle egentligen lika gärna kunna vara klassens kropp som inte är avslutad. Det är indenteringen som får den som läser koden att ta det för givet.

Den information som vi behövde plocka fram ur ofullständig kod var följande:

- Vilket uttryck programmeraren vill avreferera när punkt trycks.
- Vad det uttrycket refererar till för klass.
- Vilka variabler som är synliga på aktuellt ställe i koden och deras typ.
- Vilka metoder som finns klara att anropa i aktuell klass.
- Vilka paket och klasser som är importerade.

Enligt resonemanget om ej korrekt utseende av editerad fil har vi inte gett oss på att försöka ta fram vilka metoder som finns i aktuell klass. Det verkar som om de andra utvecklingsverktygen vi tittat på har gjort nödlösningar på detta. I SVC blir metoder i aktuell klass synliga ibland, beroende på hur mycket fel det är i dem. MVJ++ verkar dock ha löst detta bättre. Detta är egentligen ett omöjligt problem eftersom det inte är någon metod om den inte är helt korrekt. Dessutom hör inte metoden till någon klass om inte hela klassen är korrekt, vilket den sällan är under editering.

När vi ska plocka ut vilka variabler som är synliga på aktuellt ställe i koden, gör vi en nödlösning. Vi spånade länge kring hur vi skulle kunna dela upp den ej kompletta filen i olika scopes och kom fram till att det är omöjligt. Vi går igenom hela filen och letar upp alla giltiga variabeldeklarationer och betraktar dessa som synliga. Hela filen delas upp i delsträngar med semikolon som skiljetecken och dessa försöker vi en efter en att matcha mot parserregeln för variabeldeklaration. Parametrar till metoder tas inte med, vilket är ytterligare en begränsning.

För att ta fram det uttryck som skall avrefereras plockar vi först ut hela den rad man befinner sig på i koden. Raden kapas successivt av i början tills det som återstår fram till punkten är ett korrekt uttryck som kan avrefereras. Det kändes först som att även detta var en nödlösning, men efter att ha funderat och testat en del, har vi kommit fram till att det alltid fungerar. För att kontrollera om ett uttryck kan avrefereras, har vi skrivit en speciell regel i en kopia av javagrammatikfilen och genererat ytterligare en lexer och parser. Anledningen till att

vi inte bara lade till regeln i samma fil som används för att parsa klara klasser, var att vi av effektivitetsskäl tog bort trädgenereringen för all kod som var onödig i det sammanhanget.

Att bestämma vad ett utryck refererar till för klass visade sig vara extremt svårt. Vi ville att man skulle kunna skriva alla möjliga uttryck och vår editor skulle hela tiden veta vad man refererar till. Svårigheten ligger i att man i Java kan skriva ganska invecklade uttryck. Ett exempel på ett sådant uttryck kan vara:

```
(Klass2)((klass1Variabel.method())[ARRAY_INDEX].objekt).
```

Parsingen av detta uttryck ska resultera i en referens till ett objekt av Klass2. Vi gjorde inledningsvis en avgränsning som innebar att vi alltid tolkade uttrycket som ett enkelt variabelnamn. När vi upptäckte möjligheten att generera en TreeWalker försökte vi oss på en lösning, men misslyckades. Några dagar innan inlämnandet av denna rapport gjorde vi ett nytt försök och lyckades lösa problemet till stor del. Det vi fortfarande inte klarar av är arrayer. Den regel vi lagt till för detta i grammatiken för DotExpTreeWalker återfinns i Bilaga B.

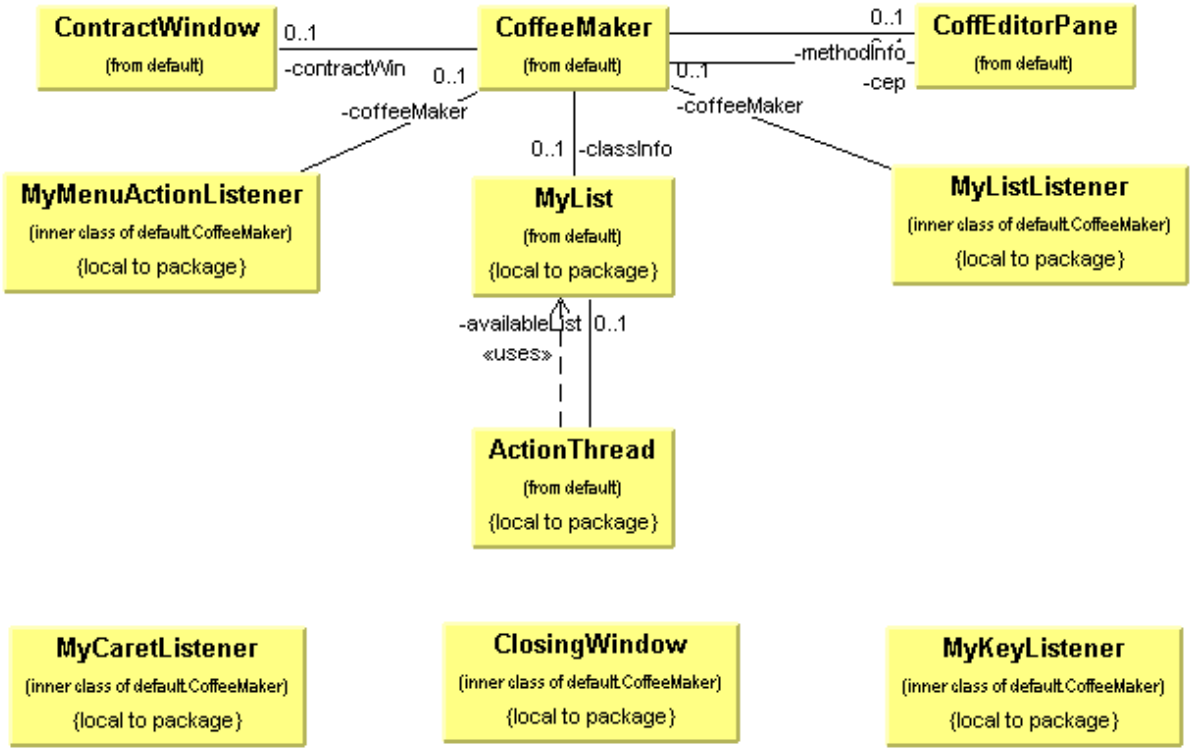
5.4 Design av editorn

Vi hade från början tänkt att göra en objektorienterad analys- och design och använda oss av några kända design patterns [3]. Vi ville använda oss av MVC (Model View Controller) pattern för att dela på programmets logik och data och det grafiska gränssnittet. Eftersom vi utvecklar programmet i Java vars grafiska API (Swing) [5] stödjer MVC så kändes naturligt att vi skulle försöka följa den designen. Av gammal vana från tidigare arbeten så började vi att koda väldigt tidigt utan att ha en design innan. Det grafiska gränssnittet har i princip vuxit fram genom att först vara ett verktyg för att debugga vid våra parsingexperiment. Ju mer vi kodade desto mer försvann tanken på att vi skulle planera och rita upp en klassdesign. Hade vi gjort en ordentlig design så hade det tagit mesta delen av tiden vilket inte hade gett oss så mycket tid över till att lära oss hur parsing går till. Då hade vi förmodligen kommit fram till en snygg design utan funktionalitet. Vi hade att välja mellan att göra en bra design eller snabbt ta fram en prototyp.

5.5 Programmets uppbyggnad

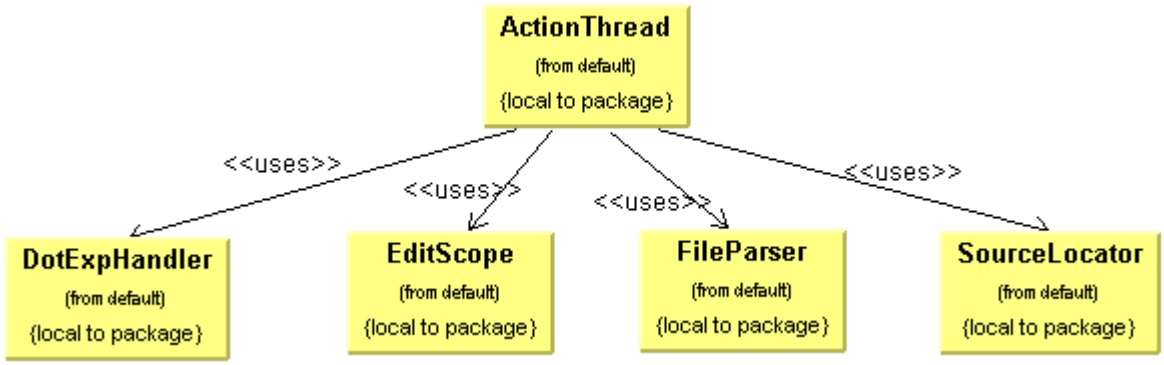
Vi har valt att inte inkludera hela källkoden i denna rapport. Det beslutet grundade vi på att det dels skulle bli alldeles för långt (ca 50 sidor) samt att den kommer att vidareutvecklas efter examensarbetets slut. I Bilaga C finns koden till den viktigaste klassen, ActionThread.

Källkoden tillsammans med det körbara programmet finns upplagt på Internet [10]. Vi försöker här endast översiktligt beskriva själva programmet.



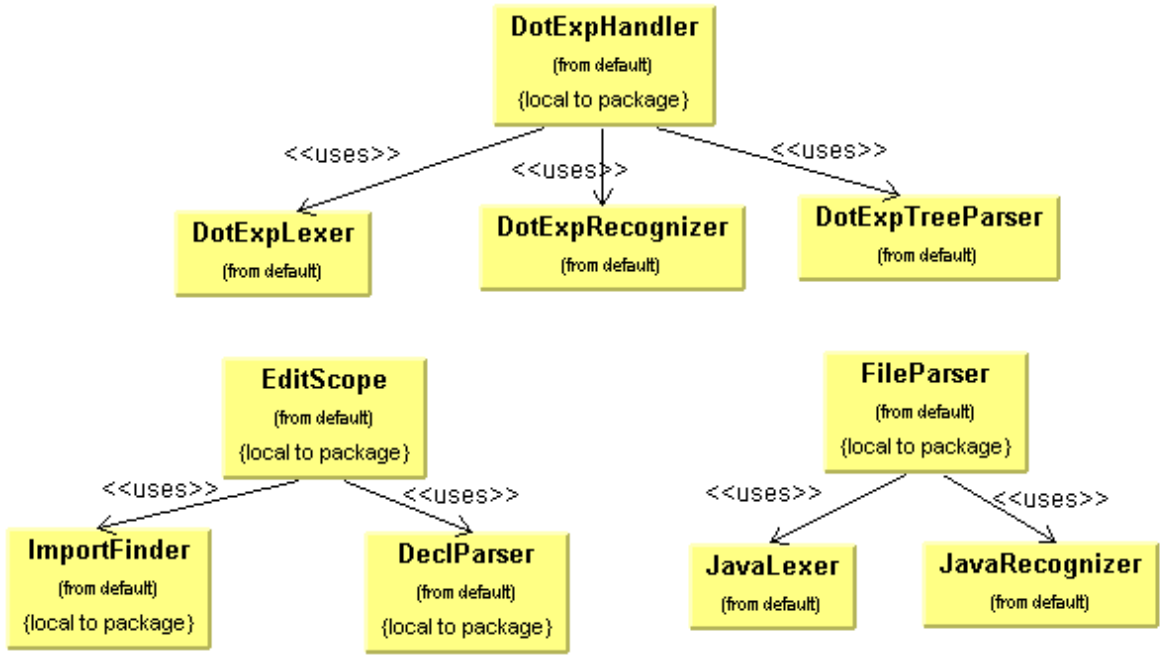
Figur 2: Klassdiagram över det grafiska gränssnittet.

Som Figur 2 visar är det grafiska gränssnittet uppbyggt av CoffeeMaker, ContractWindow, CoffEditorPane och MyList. Alla klasser som slutar på Listener och ClosingWindow är inre klasser vars uppgift är att ta hand om händelser. ContractWindow är ett litet fönster som visar för- och eftervillkor för en viss metod. CoffEditorPane tar hand om syntaxhighlightingen. MyList innehåller en lista över alla metoder och variabler för den klass som användaren har refererat till. ActionThread skapas varje gång användaren tryckt på punkt. ActionThread har en referens till MyList för att kunna uppdatera MyList med den senaste refererade klassens metoder och variabler.



Figur 3: Klassdiagram över de logiska klasserna.

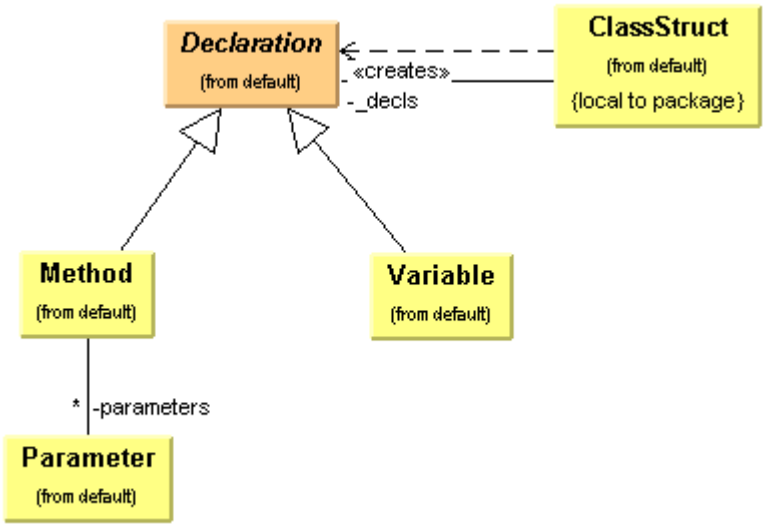
ActionThreads uppgift är att ta fram all den information som ska visas upp för användaren. ActionThread använder sig av DotExpHandler, EditScope, SourceLocator och FileParser för att utföra sin uppgift. DotExpHandler tar fram det refererande uttrycket och tar fram returtypen. SourceLocator tar reda på vart källkodsfilen finns för den returnerade klassen. FileParser parsar klassen.



Figur 4: Klasser som ActionThread använder sig av.

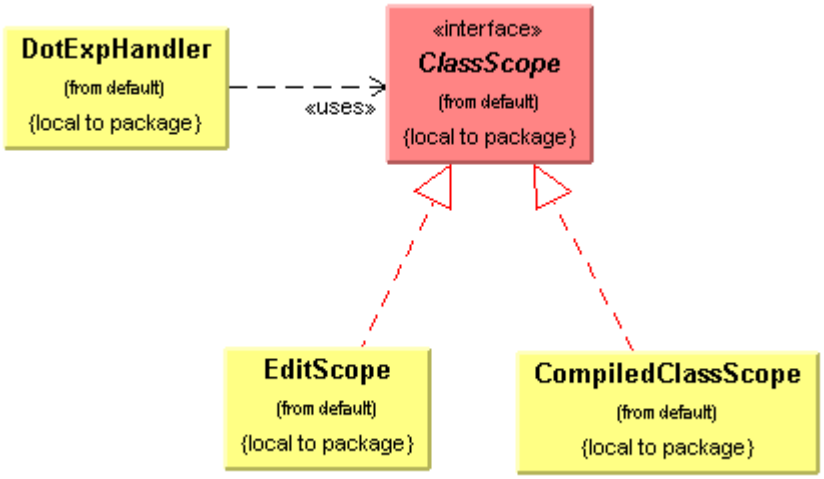
DotExpHandler använder sig av DotExpLexer, DotExpRecognizer och DotExpTreeParser, se Figur 4. DotExpHandler, DotExpRecognizer och DotExpTreeParser är klasser som genererats från grammatikfiler med hjälp av ANTLR. DotExpLexer och DotExpRecognizer bygger upp ett syntaxträd som DotExpTreeParser sedan traverserar. EditScope är en abstraktion över den klass man editerar och används som startargument då DotExpTreeParser

rekursivt ska ta fram returtypen för ett uttryck. FileParser använder sig av JavaLexer och JavaRecognizer för att parsa en javakällkodsfil.



Figur 5: Klassdiagram över hur information om parsade klasser lagras.

Information om alla parsade klasser läggs i en varsin ClassStruct, se Figur 5. Varje ClassStruct har en vector som lagrar en eller flera Declaration. Declaration är en abstraktion av en metod- eller variabeldeklaration.



Figur 6: Klassdiagram över hur en klass abstraheras.

ClassScope, se Figur 6, är en abstraktion av en javaklass. EditScope är den klass som editeras och CompiledClassScope är en färdigskriven klass.

Följande klasser genereras av ANTLR:

JavaLexer (coffee.g)

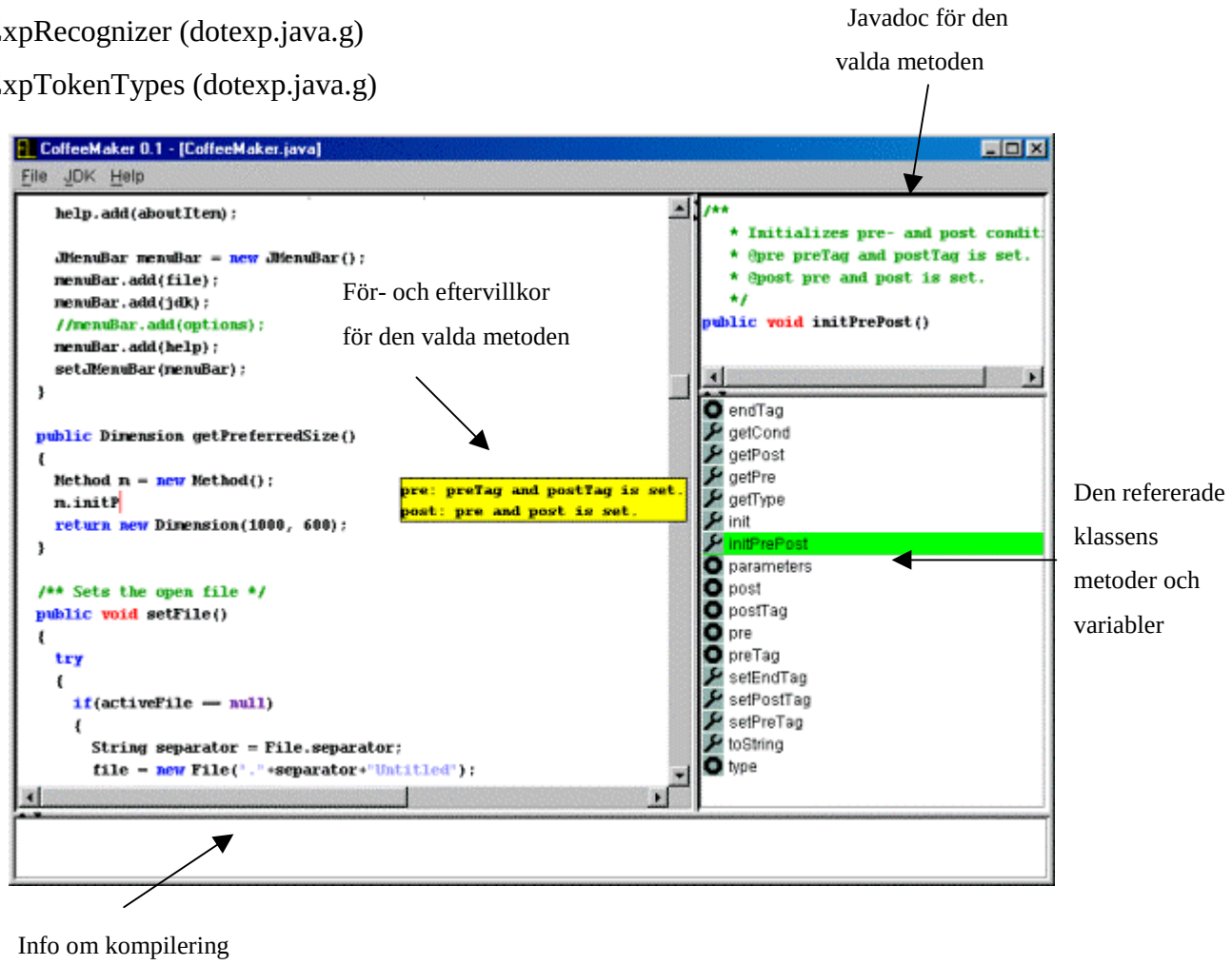
JavaRecognizer (coffee.g)

JavaTokenTypes (coffee.g)

DotExpLexer (dotexp.java.g)

DotExpRecognizer (dotexp.java.g)

DotExpTokenTypes (dotexp.java.g)



Figur 7: Skärmdump av editorn

Figur 7 visar en bild över hur editorn ser ut. För- och eftervillkoren visas i ett eget enskilt gult fönster. Det är meningen att detta fönster ska fånga användarens uppmärksamhet.

5.6 Sammanfattande beskrivning av programmets arbetssätt

När användaren skriver en punkt utförs alltid följande komplexa sekvens:

1. Allt innan punkten på aktuell rad i koden plockas ut som en sträng.
2. Första tecknet i denna sträng kapas iterativt bort tills resten av strängen är ett uttryck som kan avrefereras.
3. Ett syntaxträd bildas av uttrycket.

4. Alla variabeldeklarationer i den aktuella filen plockas ut och lagras i en lista.
5. Alla import-satser i den aktuella filen letas upp.
6. Ett EditScope som implementerar ClassScope skapas av deklarationerna och importsatserna.
7. Klassen för uttrycket som skall avrefereras tas fram genom att anropa en rekursiv DotExpTreeParser som tar som argument ett träd och ett ClassScope och returnerar den klass det refererar till. Den anropas med EditScope och trädet för uttrycket som ska avrefereras.
8. Om det är första gången klassen har blivit refererad så görs punkterna 9 till 11, annars görs bara 12.
9. En ström till källkodsfilen för klassen som refereras tas fram.
10. Strömmen parsas och ett syntaxträd byggs upp.
11. Från syntaxträdet skapas en lista av metoder och variabler i klassen dessa läggs sedan i minnet.
12. Klassens variabler och metoder visas upp för användaren i en lista.

Källkoden till klassen som utför ovanstående finns i Bilaga C.

6 Problem

6.1 Parsing

Vi visste inte så mycket om parsing när vi påbörjade examensarbetet, vilket gjorde att vi ofta körde Brute-force metoden [4]. Denna metod kännetecknas av att man går på och försöker lösa problemet direkt utan hänsyn till det antal operationer som krävs för att lösa uppgiften. Vi har dock i slutet av projektet optimerat koden en hel del. Vi har alltså ägnat oss åt inlärning genom intensivt experimenterande. Parsing är ett återkommande problem inom programmering och det finns algoritmer för hur parsing ska gå till väga på bästa sätt. Vi har inte hunnit göra annat än nödlösningar på vissa parsingproblem.

Vi spenderade två veckor heltid åt att lära oss ANTLR. Dokumentationen som följde med ANTLR förutsatte att man kunde det mesta om parsing och EBNF-grammatik [1]. Den var endast inriktad på att lära ut det som är ANTLR-specifikt. Eftersom det följde med många bra

exempel, kunde vi själva lista ut hur man skriver en grammatikfil och genererar en lexer och parser med deras verktyg.

Det hade känts mycket bättre med exempelvis en kurs i kompilatorkonstruktion bakom sig. Då hade vi vetat mer om vad som stod framför oss och kunnat planera projektet bättre.

6.2 Syntax Highlighting

Vi tillbringade över en hel vecka åt att försöka göra en egen syntax highlighting. Vi utgick från ett exempel som fanns på Suns hemsida [6]. Eftersom Suns egen syntax highlighting var ganska buggig, så bestämde vi oss för att inte följa deras exempel. Vi hittade istället Jipe, en javaeditor med syntax highlighting och källkod. Källkoden var väldigt hårt knuten till deras editor så vi fick göra vissa ändringar för att anpassa deras TextArea till vårt och alla andras eventuella program. Det vi gjorde var i princip att omvandla den till en fristående komponent från att ha varit en klass med diverse beroenden.

6.3 Arbetsformen

Vi jobbade ibland på varsitt håll och eftersom vi inte hade någon utarbetad design så var det svårt att alltid veta vad man då skulle göra. En av anledningarna att vi jobbat på varsitt håll är att vi inte fick ett ställe med dator på skolan att vara på och ibland passade det inte att sitta hemma hos någon av oss.

6.4 Versionshantering och JavaAPI

Det som gjorde att vi många gånger inte fick något gjort på en hel dag var problem med versionshantering av kod, Java virtual machine och Swing. Vi hade till en början olika versioner av Java så ibland fungerade programmet på den enas dator men inte på den andras. Vi hade en del problem med Swing (Javas grafiska API). Vi vet inte om det beror på att vi inte har tillräckliga kunskaper om Swing eller om Swing är buggigt, men är i stort sett säkra på att det är båda delarna.

7 Resultat

Vi uppnådde en lösning på de flesta uppsatta målen. Nedan följer en utförligare resultatbeskrivning för respektive mål.

1 Editorn kan visa upp för- och eftervillkor om de angivits korrekt.
Det korrekta sättet att ange dessa på är att skriva `@pre` eller `@post` någonstans i javadockommentaren. Då kommer resten av den raden att tolkas som respektive villkor. Villkoren för metoden visas upp i ett gult fönster på skärmen när man valt den i listan.

2 Editorn klarar att visa upp en lista över tillgängliga metoder och datamedlemmar i refererad klass. Med tillgängliga menar vi alla metoder som finns i klassen och inte bara de som är synliga. Man skulle givetvis kunna ta bort alla som har private som modifierare, men det är inget vi har prioriterat.

3 Vi har lyckats uppnå målet med syntax highlighting genom att modifiera en klass ifrån editorn Jipe.

4 Målet om grundläggande funktioner för en editor är uppnått. Vi har hunnit inkludera ett antal nödvändiga funktioner. Man kan öppna och spara filer samt klippa och klistra text. Dessutom är det möjligt att kompilera och köra sitt javaprogram. Detta har vi löst med stödet i Java för att exekvera andra program.

5 Vi har inte uppnått målet att all funktionalitet skall vara enkel att stänga av och sätta på. Det är en konsekvens av att vi inte hunnit lägga ner tillräckligt med tid för att lösa nästa mål.

6 Vi har inte lyckats se till att den objektorienterade designen är tillräckligt bra för att man väldigt enkelt ska kunna bygga vidare eller ändra i programmet. Vi tror dock inte att det är speciellt svårt att sätta sig in i koden.

Vi har lyckats få editorn att känna till alla importerade klasser. Detta problem löstes automatiskt eftersom vi var tvungna att veta vilka klasser och paket som är inkluderade i aktuell fil för att kunna veta att t.ex. Frame är en benämning på klassen java.awt.Frame. Denna avgränsning visade sig alltså onödig.

Avgränsningen om prestanda visade sig däremot nödvändig. Vi valde därför att kommentera bort den delen av koden som lade till information om ärvda metoder genom att rekursivt parsa superklassens källkod. Eftersom en klass kan ha väldigt många superklasser

blev det ibland outhärdligt att invänta resultatet från parsingen efter att man tryckt punkt. För att visa information om klassen javax.swing.JFrame och alla dess ärvda metoder krävdes 27 sekunder på en PentiumII 266Mhz.

Vi anser att vi har tagit fram en prototyp med godkänd funktionalitet. Med tanke på våra förkunskaper så är vi nöjda med resultatet. Vår prototyp klarar helt klart av huvudmålet att informera användaren om för- och eftervillkor. Enligt vår mening inser man snabbt när man leker med editorn att idén om denna funktionalitet verkligen hjälper till vid programutveckling. Om en metod har för- och eftervillkor specificerade visas de klart och tydligt. Om inga villkor har angivits, visas ändå javadoc upp i realtid och utvecklaren slipper bläddra i en separat javadocbrowser för att läsa dokumentationen.

8 Erfarenheter och rekommendationer

När man arbetar med ett större projekt kan man inte bara sätta sig och börja implementera utan noggrann design och planering. Vi planerade inte så mycket och gjorde heller ingen ordentlig design. Vi har dock varit tvungna att arbeta på det sättet med tanke på våra förkunskaper. Vi har heller inte dokumenterat klassdesignen under tiden vi utvecklade. Detta var det största misstaget vi gjorde och det ledde till att vi ibland inte hade koll på vilka klasser vi hade skrivit och hur de fungerade lång tid efter att vi skrivit dem. Det här är en erfarenhet som vi kommer att ha nytta av i framtiden då vi kommer att vara delaktiga i större projekt. I slutet av detta arbete gjorde vi en nödvändig upprensning av koden.

Vi gjorde ett tidsschema i början av projektet som vi endast lyckades följa fram till dess att en tredjedel av tiden gått. Vi visste redan då vi gjorde schemat att det var stor sannolikhet för att det skulle bli så. Vi borde ha upprättat ett nytt schema när det gamla inte kunde följas. Att vi inte gjorde så, ledde till att vi fick panik i slutet av projektet när vi upptäckte hur lite tid vi hade över till dokumentation.

Trots att vårt arbete handlar om att använda för- och eftervillkor, har vi inte själva gjort det fullt ut. Detta kan delvis bero på att vi valde att utveckla i Java som använder exceptions (undantag) flitigt, vilket ledde oss bort från kontraktskonceptet.

Vi kunde ha designat bättre så att det exempelvis skulle vara lätt och smidigt att byta det grafiska gränssnittet eller stänga av och sätta på all funktionalitet. Kanske hade vi hunnit om vi inte missat nästan två halvfartsveckor i början då vi valde att byta till den här uppgiften. En annan bidragande orsak till att tiden inte räckte, var att vi båda jobbade som labbhandledare

under de dagar som vi hade examensarbete. Detta gjorde att många hela dagar gick bort. Vi fick ta igen dessa timmar på helger, men det var svårt att jobba lika effektivt då. Att vi jobbade hemma kan också ha bidragit till resultatet, vissa dagar satt vi utan att få mycket gjort. Vi tror det är bra att kunna kombinera hemarbete med en arbetsplats.

9 Slutord

Svårigheten i den här uppgiften har varit parsingen. Vi hade kunnat tillägna all tid åt att göra en fullständig parsing och då kanske fått med allt vi velat ha med. Då hade vi inte haft något att visa upp mer än kod. Även om det grafiska gränssnittet och dess funktionaliteter är relativt lätt att lära sig så tar det tid och kan inte göras på bara några dagar.

Det här arbetet har varit mycket lärorikt. Vi vet nu hur det känns att arbeta i lag under ett större projekt. Vi har jobbat ihop med varandra förut, men inte under ett så här stort projekt. Vissa dagar har kreativiteten flödat medan andra dagar har det stått still. Vi har båda varit lika delaktiga i arbetet och vi skulle gärna vilja vara med att utveckla prototypen till en färdig produkt om vi får möjlighet i framtiden.

Referenser

[1] Robert W. Sebesta. *Concepts of Programming Languages.* Addison Wesley, 4th edition, 1999.

[2] Martin Blom. *Semantic Integrity in Program Development.* Karlstads universitet, 1999.

[3] Gamma, Helm, Johnson, Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[4] Heileman. *Data Structures, Algorithms, and Object-Oriented Programming.* McGraw-Hill, 1996.

[5] Kathy Walrath, Mary Campione. *The JFC Swing Tutorial.* Addison Wesley, 1999.

[6] http://www.antlr.org

[7] http://www.e-i-s.co.uk/Jipe

[8] http://www.sun.com

[9] http://www.objectinsight.com

[10] http://www.cse.kau.se/~di7linh

Bilagor

A Exempel på en grammatikfil

Följande exempel är taget från ANTLR för att demonstrera hur en grammatikfil för en enkel miniräknare kan se ut. Från denna grammatikfil genereras CalcParser.java, CalcLexer.java och CalcTreeWalker.java.

```
class CalcParser extends Parser;
options {
 buildAST = true; // uses CommonAST by default
}

expr
 :  mexpr (PLUS^ mexpr)* SEMI!
 ;

mexpr
 :  atom (STAR^ atom)*
 ;

atom: INT
 ;

class CalcLexer extends Lexer;

WS :  (' '
 |  '\t'
 |  '\n'
 |  '\r')
    { _ttype = Token.SKIP; }
 ;
```

```
LPAREN:  '('
 ;

RPAREN:  ')'
 ;

STAR: '*'
 ;

PLUS: '+'
 ;
a
SEMI: ';'
 ;

protected
DIGIT
 :  '0'..'9'
 ;

INT :  (DIGIT)+
 ;

class CalcTreeWalker extends TreeParser;

expr returns [float r]
{
 float a,b;
 r=0;
}
 :  #(PLUS a=expr b=expr)   {r = a+b;}
 |  #(STAR a=expr b=expr)   {r = a*b;}
 |  i:INT {r =
           (float)Integer.parseInt(i.getText());}
 ;
```

B Regeln för att hitta vad ett uttryck refererar till

```
evaluateDotExp[ClassScope scope] returns [ClassScope newScope]
{
   newScope=null;
   ClassScope a,b;
   String ne;
}
   :  i:IDENT
      {
         newScope=new
            CompiledClassScope(scope.getVarType(i.getText()));
      }
    |  ne=newExpression
      {
         newScope=new CompiledClassScope(scope.getFullClassName(ne));
      }
    |  #(DOT a=evaluateDotExp[scope] b=evaluateDotExp[a])
      {
         newScope=b;
      }
    |   (#(METHOD_CALL #(DOT evaluateDotExp[scope] IDENT) ELIST)) =>
      #(METHOD_CALL #(DOT a=evaluateDotExp[scope] m2:IDENT) ELIST)
      {
         newScope=new
            CompiledClassScope(a.getReturnType(m2.getText()));
      }
    |  #(METHOD_CALL m:IDENT ELIST)
      {
         newScope=new
            CompiledClassScope(scope.getReturnType(m.getText()));
      }
    |  #(TYPECAST t:TYPE expression)
      {
         newScope=new
            CompiledClassScope(
               scope.getFullClassName(type(t.getFirstChild())));
```

```
      }
  ;
```

C Kodexempel – ActionThread.java

```
import antlr.collections.AST;
import java.util.*;

/**
 * An instance of this class is created when the user presses
 * dot(.).
 * It takes care of all the steps in chapter 5.6 of the report.
 */
class ActionThread extends Thread
{
   private String editorText;
   private int caretPos;
   private MyList availableList;
   private static final String separators = "\n;";
   private HashMap classes;
   private static String[] errorInfo;

   static
   {
      errorInfo = new String[3];
      errorInfo[0] = "No";
      errorInfo[1] = "Info";
      errorInfo[2] = "Available";
   }

   ActionThread(String edTxt, int cPos, MyList jl, HashMap cl)
   {
      editorText = edTxt;
      caretPos = cPos;
      availableList = jl;
      classes = cl;
```

```
}

/** Extracts the expression to be unreferenced. */
private String getValidReference()
{
   int i = 0;//badly chosen name

   for(int j = 0; j < caretPos; j++)
   {
      if(editorText.charAt(i) == Character.LINE_SEPARATOR)
         i = i + 2;
      else
         i++;
   }
   caretPos = i;
   while(i > 0 && separators.indexOf((new
      Character(editorText.charAt(i-1))).toString()) == -1)
      i--;

   return DotExpHandler.getDotExp(
      editorText.substring(i, caretPos-1));
}

private void setListData(ClassStruct cs)
{
   Declaration[] availDecl = cs.getAvailable();
   availableList.setListData(availDecl);
   availableList.clearString();
}

/** Extracts all information to be shown to the user*/
public void run()
{
   String dotexp = getValidReference();
   System.out.println(dotexp);
   EditScope es=new EditScope(editorText);
  String fullClassName=DotExpHandler.evaluateDotExp(
```

```
                                                dotexp,es);
System.out.println("FullClassName = "+fullClassName);

 if(fullClassName != null)//the variable is declared
 {
    if(classes.containsKey((Object)fullClassName))
    //the class is already in memory
    {
       ClassStruct cs = (ClassStruct)classes.get(
                                    (Object)fullClassName);
       setListData(cs);
    }
    else
    {
       AST tree = FileParser.getTree(
          SourceLocator.getSourceInputStream(fullClassName));

       if(tree != null)//parsing OK
       {
          ClassStruct cs = new ClassStruct(tree,fullClassName);
          classes.put(fullClassName, cs);
          setListData(cs);
       }
       else  //ClassStruct could not be created,
             //show info from reflection instead
       {
          java.lang.Class c;
          try
          {
             c = java.lang.Class.forName(fullClassName);
             availableList.setListData(c.getDeclaredMethods());
          }
          catch(Exception e){
             availableList.setListData(errorInfo);
          }
       }
    }
```

```
      }
      else//the class is not found
      {
         availableList.setListData(errorInfo);
      }
   }

}
```