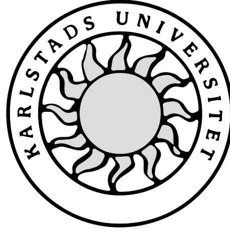


Datavetenskap

Håkan Bergmark och Tony Bergh

Utvecklingsmiljö för Java med stöd för kontraktsprogrammering

Master's Thesis

2004:02

Utvecklingsmiljö för Java med stöd för kontraktsprogrammering

Håkan Bergmark och Tony Bergh

© 2004 Håkan Bergmark, Tony Bergh och Karlstads universitet

Denna rapport är skriven som en del av det arbete som krävs för att erhålla en magisterexamen 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.

Håkan Bergmark

Tony Bergh

Godkänd, 2003-06-23

Handledare: Martin Blom

Examinator: Donald F. Ross

Sammanfattning

Denna D-uppsats syftar till att beskriva bakgrunden till och tillkomsten av en utvecklingsmiljö för Java som stöder kontraktsprogrammering. Uppsatsen specificerar vad syntax och semantik är. Denna kunskap krävs för att till fullo förstå uppsatsens mål och syfte. Begreppet kontraktsprogrammering förklaras ingående då detta är ett nyckelbegrepp i utvecklingsmiljön och därmed denna uppsats. En genomgång av ett antal verktyg som används inom området kontraktsprogrammering görs och testas för att demonstrera deras funktionalitet. Uppsatsen beskriver sedan ett idealsystem som vore det bästa möjliga resultatet av arbetet. Detta idealsystem används sedan för att jämföra det egentliga resultatet av arbetet med. Resultatet är en utvecklingsmiljö för Java som stöder kontraktsprogrammering genom att presentera för- och eftervillkor vid metodanrop. Utvecklingsmiljön stöder även presentation av klassinvarianter samt automatiskt infogande av förvillkorstest. Denna utvecklingsmiljö ska förenkla för programmeraren att använda sig av kontraktsprogrammering vid utveckling av mjukvara. Författarnas förhoppning är att denna utvecklingsmiljö ska kunna bidra till bättre och mer pålitlig mjukvara.

Development Environment for Java with Support for Programming by Contract

Abstract

The purpose of this master's thesis is to describe the background to and creation of a development environment for Java that supports programming by contract. The thesis specifies what syntax and semantics are since this is needed to understand the rest of the document. The concept of programming by contract is thoroughly explained since this is a key notion in the development environment and therewith also in this thesis. A review is made of some of the tools that are used within the area of programming by contract. These tools are also tested to demonstrate their functionality. The thesis then describes an ideal system that would be the best possible result of the project. This ideal system is later on used to compare the actual result of the project with. The result is a development environment for Java that supports programming by contract by showing pre- and postconditions when invoking a method. The development environment also supports showing class invariants and automatic insertion of tests on preconditions. The intent of this development environment is to simplify for the programmer to use contracts in software development. It is the authors' hopes that this will lead to better and more reliable software.

Innehållsförteckning

1 Inledning **3**

2 Bakgrund **3**

2.1 Motivering 3

2.2 Syfte och mål 3

2.3 Syntax och semantik 3

2.4 Kontraktsprogrammering 3

2.4.1 Olika nivåer av formalism 3

2.4.2 Kontrakt och arv 3

2.4.3 Kontrakt och undantag 3

2.4.4 Skillnader mellan kontraktsprogrammering och defensiv programmering 3

2.5 Tidigare arbeten 3

3 Befintliga verktyg **3**

3.1 Inledning 3

3.2 Testklass - MyStack 3

3.3 iContract 3

3.3.1 Provkörning av iContract 3

3.3.2 Fördelar med iContract 3

3.3.3 Nackdelar med iContract 3

3.4 Jass – Java with Assertions 3

3.4.1 Provkörning av Jass 3

3.4.2 Fördelar med Jass 3

3.4.3 Nackdelar med Jass 3

3.5 jContractor 3

3.5.1 Provkörning av jContractor 3

3.5.2 Fördelar med jContractor 3

3.5.3 Nackdelar med jContractor 3

3.6 Jcontract 3

3.6.1 Provkörning av Jcontract 3

3.6.2 Fördelar med Jcontract 3

3.6.3 Nackdelar med Jcontract 3

3.7 Sammanfattning 3

4 Idealsystem **3**

5 Implementation **3**

5.1 Inledning 3

5.2 Avgränsning 3

5.3 Val av implementationsmiljö 3

5.3.1 Javadoc 3

5.4 Editor-modulen i NetBeans 3

5.4.1 Filstruktur 3

5.5 Design 3

5.5.1 Kontraktstaggar 3

5.5.2 Syntaxbeskrivning för kontrakt 3

5.6 Realisering 3

5.6.1 Presentation av kontrakt i NetBeans 3

5.6.2 Inställning av kontraktspresentation i NetBeans 3

5.6.3 Infoga förvillkorstest i NetBeans 3

5.6.4 Kompilering 3

5.6.5 Stöd för kontraktstaggarna med Javadoc-verktyget 3

5.7 Filer som modifierats 3

5.7.1 org.netbeans.editor.Bundle.properties 3

5.7.2 org.netbeans.editor.ext.CompletionJavaDoc.java 3

5.7.3 org.netbeans.editor.ext.ExtSettingsDefaults.java 3

5.7.4 org.netbeans.editor.ext.ExtSettingsNames.java 3

5.7.5 org.netbeans.editor.ext.ExtSettingsInitializer.java 3

5.7.6 org.netbeans.editor.ext.ScrollJavaDocPane.java 3

5.7.7 org.netbeans.editor.ext.JavaDocPane.java 3

5.7.8 org.netbeans.editor.ext.JDCPopupPanel.java 3

5.7.9 org.netbeans.modules.editor.java.Bundle.properties 3

5.7.10 org.netbeans.modules.editor.java.JavaKit.java 3

5.7.11 org.netbeans.modules.editor.java.NbCompletionJavaDoc.java 3

5.7.12 org.netbeans.modules.editor.java.NbScrollJavaDocPane.java 3

5.7.13 org.netbeans.modules.editor.options.Bundle.properties 3

5.7.14 org.netbeans.modules.editor.options.JavaOptions.java 3

5.7.15 org.netbeans.modules.editor.options.JavaOptionsBeanInfo.java 3

5.8 Sammanfattning 3

6 Arbetets gång **3**

7 Resultat **3**

7.1 Demonstration 3

7.2 Utvärdering 3

7.3 Framtida arbete 3

8 Sammanfattning **3**

Referenser **3**

A Klasser vid provkörning av verktyg **3**

A.1 MyStack – iContract 3

A.2 MyStack – Automatgenererad av iContract 3
A.3 MyStack – Jass 3
A.4 MyStack – Automatgenererad av Jass 3
A.5 MyStack – jContractor 3
A.6 MyStack – Jcontract 3

B Taglet-filer 3
B.1 PreTaglet.java 3
B.2 PostTaglet.java 3
B.3 InvariantTaglet.java 3

C Klasser för provkörning av NetBeans IDE 3
C.1 StackInterface.java 3
C.2 StackImpl.java 3
C.3 StackApplication.java 3

D Systemkrav 3

E Källkod 3
E.1 org.netbeans.editor.Bundle.properies 3
E.2 org.netbeans.editor.ext.ExtSettingsNames.java 3
E.3 org.netbeans.editor.ext.ExtSettingsDefaults.java 3
E.4 org.netbeans.editor.ext.ExtSettingsInitializer.java 3
E.5 org.netbeans.editor.ext.JavaDocPane.java 3
E.6 org.netbeans.editor.ext.ScrollJavaDocPane.java 3
E.7 org.netbeans.editor.ext.JDCPopupPanel.java 3
E.8 org.netbeans.editor.ext.CompletionJavaDoc.java 3
E.9 org.netbeans.modules.editor.java.Bundle.properties 3
E.10 org.netbeans.modules.editor.java.NbScrollJavaDocPane.java 3
E.11 org.netbeans.modules.editor.java.NbCompletionJavaDoc.java 3
E.12 org.netbeans.modules.editor.options.Bundle.properties 3
E.13 org.netbeans.modules.editor.options.JavaOptions.java 3
E.14 org.netbeans.modules.editor.options.JavaOptionsBeanInfo.java 3

F SUN Public License 3

Figurförteckning

Figur 2.1: Exempel på grammatik i BNF. 3
Figur 2.2: Exempel på omarbetad grammatik i BNF. 3
Figur 2.3: Exempel på intuitiv semantik. 3
Figur 2.4: Exempel på strukturerad semantik. 3
Figur 2.5: Exempel på exekverbar semantik. 3
Figur 2.6: Exempel på formell semantik. 3
Figur 2.7: Parent's invariant rule [17]. 3
Figur 2.8: Assertion redefinition rule [17]. 3
Figur 3.1: Provkörning iContract. 3
Figur 3.2: Provkörning Jass. 3
Figur 3.3: Provkörning jContractor. 3
Figur 3.4: Provkörning jInstrument. 3
Figur 3.5: Jcontract inställningar - Instrumentation. 3
Figur 3.6: Jcontract inställningar - Monitor. 3
Figur 3.7: Jcontract inställningar - Editor. 3
Figur 3.8: Jcontract monitor med GUI. 3
Figur 3.9: Jcontract editor. 3
Figur 3.10: Jcontract monitor i konsol. 3
Figur 3.11: Jcontract med undantag. 3
Figur 4.1: Exempelklassen Sifr. 3
Figur 5.1: Paketstruktur. 3
Figur 5.2: Exempel @pre. 3
Figur 5.3: Exempel @post. 3
Figur 5.4: Exempel @invariant. 3
Figur 5.5: Exempel syntaxbeskrivning. 3
Figur 5.6: Exempel infogning av förvillkorstest. 3
Figur 5.7: Javadoc tag. 3

Figur 5.8: Javadoc taglet. 3
Figur 5.9: CompletionJavaDoc. 3
Figur 5.10: ExtSettingsDefaults. 3
Figur 5.11: ExtSettingsNames. 3
Figur 5.12: ExtSettingsInitializer. 3
Figur 5.13: ScrollJavaDocPane. 3
Figur 5.14: JDCPopupPanel. 3
Figur 5.15: JavaKit. 3
Figur 5.16: NbCompletionJavaDoc. 3
Figur 5.17: NbScrollJavaDocPane. 3
Figur 5.18: JavaOptions. 3
Figur 5.19: JavaOptionsBeanInfo. 3
Figur 6.1: Arbetets gång - översikt. 3
Figur 7.1: Inställning om endast kontrakt skall presenteras. 3
Figur 7.2: Javadoc popup-fönstrets fördröjning. 3
Figur 7.3: Mountning av filsystem. 3
Figur 7.4: Javadoc-fönster. 3
Figur 7.5: Knapp för infogande av förvillkorstest. 3
Figur 7.6: Förvillkorstest. 3

Tabellförteckning

Tabell D.1: Systemkrav. 3

1 Inledning

Det finns ett ständigt krav på bättre och pålitligare mjukvara. Trots detta förekommer inte sällan fel, så kallade buggar, i mjukvara som leder till ökade kostnader både för utvecklare och för användare av mjukvaran. Det kan finnas många faktorer och orsaker till att buggar förekommer i mjukvara. Det kan till exempel vara slarv, logiska fel, missförstånd, för komplex kod och så vidare. Skulle kodkvalitén kunna förbättras finns det stora pengabelopp att tjäna, både för utvecklare av mjukvara liksom användare av denna. Ett sätt att kunna förbättra kodkvalitén är att se till att kodens semantik, eller innebörd, kan förstås enkelt. Om metoder beskrivs på ett sådant sätt att deras funktionalitet klart framgår utan att koden behöver studeras, skulle detta säkerligen minska antalet missförstånd. Kodkvalitén skulle öka och det skulle finnas färre fel i mjukvaran. En utvecklingsmiljö som presenterar semantiken för metoder vid anrop skulle förenkla förståelsen för koden hos programmeraren. Om denna utvecklingsmiljö även skulle stöda kontraktsprogrammering, så att klassinvarianter och för- och eftervillkor presenterades, skulle det kunna öka pålitligheten hos mjukvaran och förhoppningsvis också öka kvalitén på koden.

2 Bakgrund

I detta kapitel presenteras först kortfattat motivering, syfte och mål med detta arbete. Därefter följer ett stycke om syntax och semantik, något som är väsentligt för resten av arbetet. Även begreppet kontraktsprogrammering redogörs i detta kapitel. Till sist följer en liten redogörelse av ett tidigare arbete som har utförts på Karlstads universitet inom samma område.

2.1 Motivering

Karlstads universitet har under ett flertal år bedrivit forskning inom området kontraktsprogrammering och dess inverkan på mjukvara. Vi har fått i uppdrag, av forskningsgruppen för programvaruutveckling (SERG) vid Karlstads universitet, att ta fram en utvecklingsmiljö som stöder kontraktsprogrammering.

Dagens utvecklingsmiljöer erbjuder oftast syntaktiskt stöd som till exempel *syntax highlighting*[1] och *code completion*[2], men semantiskt stöd finns sällan att tillgå frånsett några få undantag. Däremot saknar vi kännedom om någon utvecklingsmiljö där kontraktsprogrammering stöds i form av presentation av kontrakt. Det finns dock ett antal separata verktyg för att underlätta kontraktsprogrammering, varav några presenteras i kapitel 3.

Om projektet faller ut väl så är det i förlängningen tänkt att utvecklingsmiljön ska användas i laborationssalarna på Karlstads universitet. Det öppnar då för möjligheter att undersöka om denna funktionalitet och stöd underlättar för programmeraren och gynnar mjukvarans kvalité.

2.2 Syfte och mål

Syftet med detta arbete är att ta fram en utvecklingsmiljö. Denna utvecklingsmiljö skall tillhandahålla semantisk hjälp vid mjukvaruutveckling. Det är tänkt att denna utvecklingsmiljö skall användas tillsammans med programmeringsstilen kontraktsprogrammering. Fördelen med semantisk hjälp är att programmeraren får

---

[1] *Syntax highlighting* innebär att kodens olika delar särskiljs genom till exempel färgkodning.
[2] *Code completion* innebär att möjliga fortsättningar på koden, beroende på sammanhanget, presenteras.

information om en specifik metods uppgift och användningsområde. Därmed behöver inte programmeraren granska och tolka koden för att kunna förstå och använda metoden på ett korrekt sätt.

Målet är att utveckla ett tilläggsprogram till en utvecklingsmiljö som har till uppgift att presentera kontrakt för en viss metod, i form av för- och eftervillkor, för programmeraren vid editering av kod. Målet är också att detta tilläggsprogram skall vara så oberoende av utvecklingsmiljön som möjligt. Detta oberoende öppnar möjligheten att kunna använda detta tilläggsprogram tillsammans med ett flertal andra utvecklingsmiljöer.

2.3 Syntax och semantik

Språk kan definieras genom att beskriva hur meningar kan byggas upp och vad innebörden i dessa meningar är. Reglerna som specificerar hur meningarna i ett språk kan byggas upp kallas ett språks syntax. För att ett språk skall kunna användas bör dess meningar ha en betydelse, vilket kallas för språkets semantik. Tillämpar man ovanstående på programmeringsspråk kan man säga att syntaxen beskriver hur programkoden får konstrueras och semantiken beskriver programkodens innebörd.

Att beskriva syntax är enklare än att beskriva semantik, troligtvis på grund av att det finns en koncis och universellt accepterad notation för att beskriva syntax. Någon motsvarande har ej utvecklats när det gäller semantik, åtminstone inte i samma omfattning [21]. Den notation som vanligtvis används för att beskriva ett programmeringsspråk, vars grammatik är en så kallad *context-free grammar*[3], är *Backus-Naur form* (BNF). Figur 2.1 visar hur man kan definiera en heltalsoperation med de fyra räknesätten i BNF.

[3] En *context-free grammar* är en grammatik där varje mening är oberoende av sin omgivning.

```
<op> ::= <add> | <sub> | <mul> | <div>
<add> ::= <operand> "+" <operand>
<sub> ::= <operand> "-" <operand>
<mul> ::= <operand> "*" <operand>
<div> ::= <operand> "/" <operand>
<operand> ::= <integer> | <variable>
<variable> ::= "a" | "b" | "c"
<integer> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Figur 2.1: Exempel på grammatik i BNF.

Grammatiken i Figur 2.1 består av åtta produktioner eller regler. Vänstersidan i produktionerna utgörs av *non-terminals* och högersidan utgörs av en sekvens av *non-terminals* och/eller *terminals*. *Terminals* är bassymbolerna som strängar utgörs av. När man pratar om programspråk är *token* ett synonym för *terminal*. *Non-terminals* är syntaktiska variabler som betecknar mängder av strängar. *Non-terminals* bestämmer också en hierarkisk struktur i språket som är användbart vid både syntaktisk analys och översättning. Grammatiken måste även bestå av en startsymbol som i det här fallet är *<op>*. [1]

Om man kontrollerar Figur 2.1 ser man att även operationen division med noll är syntaktiskt korrekt. Eftersom division med noll inte är matematiskt definierat kan man anta att det inte skall vara tillåtet här heller. Detta har dock med semantiken i språket att göra. Man brukar skilja på statisk och dynamisk semantik, där statisk semantik kan kontrolleras vid kompilering medan dynamisk semantik inte kan kontrolleras förrän program exekveras. Det är önskvärt att så mycket som möjligt av semantiken kontrolleras vid kompilering eftersom fel då blir mycket enklare att upptäcka än om det sker under programexekvering. Den statiska semantiken för ett språk är bara indirekt kopplad till innebörden i program under exekvering, det har istället att göra med korrekt utformning av program (det vill säga syntax och inte semantik) [21]. I vissa fall kan man omforma den statiska semantiken så att den kan inkluderas i grammatiken. Figur 2.2 visar ett exempel på hur division med noll kan undvikas rent syntaktiskt genom att utöka grammatiken (gäller ej vid användning av variabler, detta förklaras lite senare). Figur 2.2 visar ett enkelt exempel och det är lätt att förstå att det kan bli mycket svårt att utöka grammatiken om den är omfattande och komplicerad. I andra fall är det omöjligt, som till exempel fallet när en variabel i ett programmeringsspråk först måste ha blivit deklarerad innan den kan refereras till i koden [21].

```
<op> ::= <add> | <sub> | <mul> | <div>
<add> ::= <operand> ”+” <operand>
<sub> ::= <operand> ”-” <operand>
<mul> ::= <operand> ”*” <operand>
<div> ::= <operand> ”/” <denominator>
<operand> ::= <integer> | <variable>
<denominator> ::= <pos_int> | <variable>
<variable> ::= ”a” | ”b” | ”c”
<integer> ::= ”0” | ”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”
<pos_int> ::= ”1” | ”2” | ”3” | ”4” | ”5” | ”6” | ”7” | ”8” | ”9”
```

Figur 2.2: Exempel på omarbetad grammatik i BNF.

Dynamisk semantik är den typ av semantiska regler som inte kan kontrolleras under kompilering utan endast under exekvering av program. Statisk semantik har med programmets utformning att göra, vilket dynamisk semantik inte har. Dynamisk semantik har endast att göra med programmets innebörd. Ett exempel på dynamisk semantik är användningen av variabler vilket illustreras i Figur 2.2. Dessa kan tilldelas värden under exekvering av program och därmed kan värdet noll erhållas som nämnare vid division. Detta kan alltså inte kontrolleras vid kompilering, därmed måste man skydda sig mot detta under programexekveringen. Denna typ av semantiska regler kan upprätthållas genom antingen tester i programmet eller genom att använda sig av kontrakt vid programmeringen.

2.4 Kontraktsprogrammering

Kontraktsprogrammering innebär kortfattat att man sätter upp kontrakt mellan klient (användaren av metoden) och metod. Kontraktet beskriver vad som skall gälla före metodanropet, ett så kallat förvillkor, samt vad som gäller efter metodanropet om förvillkoret uppfyllts, ett så kallat eftervillkor. Eftervillkoret kan endast garanteras om förvillkoret är uppfyllt. För att klienten skall uppfylla sin del av kontraktet måste klienten följa förvillkoret. Om förvillkoret är uppfyllt måste även eftervillkoret uppfyllas. Detta innebär att man alltid kommer att kunna garantera en metods beteende om kontrakten efterlevs, förutsatt att kontrakten är väldefinierade och korrekta. För att kontraktsprogrammering skall kunna fungera på ett tillfredsställande sätt så måste kontrakten specificeras i högsta möjliga mån på ett tydligt och otvetydigt sätt. Detta för att olika tolkningar om en metods uppgift och användningsområde inte skall vara möjlig.

2.4.1 Olika nivåer av formalism

Martin Blom anger i sin licentiatavhandling [4] olika nivåer av formalism när det gäller specifikation av semantik. Dessa är: intuitiv semantik, strukturerad semantik, exekverbar semantik samt formell semantik. I avhandlingen [4] inkluderas även nivån där ingen information om semantik angivits explicit och där endast rent syntaktisk information tillhandahålls. Anledningen till att denna nivå har inkluderats är för att göra beskrivningen av nivåerna komplett. Denna nivå har dock utelämnats i denna rapport.

Exempel på de olika nivåerna av formalism kommer här att illustreras i form av semantiska beskrivningar av metoden *pop*[4] för den abstrakta datatypen *stack*[5]. I de olika exemplen presenteras medvetet aningen skiftande innehåll i de semantiska beskrivningarna. Anledningen till detta är bland annat att belysa och förstärka nivåernas nack- respektive fördelar.

Intuitiv semantik är en beskrivning av en klass och dess metoder i form av ren text. Informationen är ostrukturerad till sin form och det finns ingen mall för vilken information beskrivningen skall innehålla. Därmed är det upp till den som författar beskrivningen att göra beskrivningen så komplett som möjligt. Detta kan ge upphov till tvetydighet och därmed flera olika tolkningar. Tvetydigheten kan ge upphov till att de egentliga förvillkoren av misstag utelämnas eller feltolkas. Denna nivå av semantikbeskrivning kan försvåra både för den som specificerar semantiken och även för de som skall tyda specifikationen. Detta illustreras i Figur 2.3.

Metoden `pop` avlägsnar det element som ligger överst på stacken. Stackens storlek minskas med ett.

Figur 2.3: Exempel på intuitiv semantik.

Beskrivningen i Figur 2.3 tar inte explicit upp förvillkoret om att stacken måste innehålla minst ett element före anrop till metoden *pop*. Genom att detta förvillkor utelämnats så kan metoden användas felaktigt vilket i sin tur kan leda till fel i systemet.

---

[4] Metoden *pop* avlägsnar det element som senast lagts till på stacken.

[5] Definitionen av en *stack* är en ordnad mängd, en sekvens, där man lägger till och tar bort element i samma ände. Principen är att det element som senast lagts till också är det element som först tas bort.

Genom att använda strukturerad semantik blir den semantiska informationen enklare att uttyda, men det är därmed inte sagt att den automatiskt blir enklare att förstå. Den senare utsagan baseras på att det fortfarande är upp till den som specificerar semantiken att uttrycka sig så tydligt som möjligt. Det är dock enklare för den som specificerar semantiken att veta vad denne skall skriva och skriva rätt saker tack vare att det finns rubriker, till exempel förvillkor och eftervillkor. Två fördelar med strukturerad semantik nämns i [4], nämligen att författaren av semantiken tänker igenom designen och att det underlättar förståelsen för semantiken hos klientprogrammerare och användare av metoden. Exempel på strukturerad semantik ges i Figur 2.4.

`pop` avlägsnar det element som ligger överst på stacken.

Förvillkor: stacken är inte tom.

Eftervillkor: det översta elementet har avlägsnats från stacken. Storleken på stacken har minskats med ett. Stacken är inte full.

Figur 2.4: Exempel på strukturerad semantik.

Exekverbar semantik är strukturerad semantik där villkoren kan översättas till exekverbar kod. Detta möjliggör att man kan kontrollera och verifiera kontrakten under exekvering av program. Det finns ett flertal verktyg som är anpassade för den här typen av semantiska beskrivningar, varav några beskrivs mer utförligt i kapitel 3. Figur 2.5 visar ett exempel på exekverbar semantik. Observera dock att eftervillkoret i kontraktet i Figur 2.5 inte uttryckligen anger att det är det översta elementet som har tagits bort, utan endast beskriver en konsekvens av att ett element har avlägsnats från stacken. Detta innebär i praktiken att eftervillkoret även uppfylls om till exempel det sista elementet på stacken avlägsnas.

`pop` avlägsnar det element som ligger överst på stacken.

Förvillkor: `!this.isEmpty()`

Eftervillkor: `this.size() == old.size()-1 && !isFull()`

Figur 2.5: Exempel på exekverbar semantik.

Formell semantik är när man beskriver metoder och klasser med hjälp av formella språk som matematiskt exakt beskriver vad som händer. Man kan i vissa fall använda sig av formella beskrivningar för att visa att ett program utför dess uppgift. Dessa språk brukar dock ha en stor nackdel i att de endast kan uttydas av personer med matematisk bakgrund, eftersom deras beskrivningar grundar sig på logiska formler. Detta har då medfört att dessa är svåra att uttyda

för en person utan en matematisk bakgrund, vilket i sin tur har lett till att deras användning har begränsats. Ett exempel på ett formellt specifikationsspråk, som är relativt svårt att uttyda för personer utan kunskaper i matematisk logik, är Z [5]. Ett formellt specifikationsspråk som har en betydligt enklare syntax än till exempel Z är språket Object Constraint Language (OCL) [6]. OCL använder sig av en syntax som påminner mycket om exekverbara uttryck blandat med nyckelord för logiska symboler. Figur 2.6 visar ett exempel på formell semantik, i figuren används OCL.

```
MyStack :: pop()
  pre : not isEmpty()
  post: size() = size()@pre - 1
  post: not isFull()
```

Figur 2.6: Exempel på formell semantik.

2.4.2 Kontrakt och arv

Bertrand Meyer skaparen av programmeringsspråket Eiffel, av många ansedd som den store pionjären inom området kontraktsprogrammering, anger två regler gällande kontrakt och arv i en av sina publikationer [17]. Dessa två regler är den så kallade *Parent's invariant rule* och *Assertion redefinition rule*.

Parent's invariant rule anger vad som skall gälla angående klassinvarianter vid arv. Regeln visas i Figur 2.7 och den säger att alla ärvda invarianter, antingen indirekt ärvda eller direkt ärvda, även skall gälla för klassen själv tillsammans med eventuella egna invarianter.

> **Parent's invariant rule:** The invariants of all the parents of a class apply to the class itself.

Figur 2.7: Parent's invariant rule [17].

Assertion redefinition rule anger vad som skall gälla vid arv av metoder med för- och eftervillkor. Regeln visas i Figur 2.8 och den säger att när en metod omdefinieras så måste den nya metodens förvillkor vara lika eller mindre restriktivt, samt dess eftervillkor måste vara lika eller mer restriktivt. Med andra ord så kan den nya metoden endast specificera ett lika starkt eller svagare förvillkor än den omdefinierade metoden, samt att den måste garantera ett lika starkt eller starkare eftervillkor än den omdefinierade metoden. Anledningen till detta är den dynamiska bindningsmekanismen för objekt. Ett objekt ska kunna anropas

genom dynamisk bindning genom dess superklass. Regeln garanterar då att den nya metodens kontrakt inte bryts vid dynamisk bindning, vilket är fullt möjligt om regeln inte efterlevs.

Assertion redefinition rule: Let *r* be a routine in class *A* and *s* a redefinition of *r* in a descendant of *A*, or an effective definition of *r* if *r* was deferred. Then pre_s must be weaker than or equal to pre_r, and $post_s$ must be stronger than or equal to $post_r$.

Figur 2.8: Assertion redefinition rule [17].

2.4.3 Kontrakt och undantag

När något har inträffat som inte anses som normalt, ett så kallat undantag, hoppar man helt enkelt ut ur metoden och hamnar i en programdel som hanterar undantaget. Slarvigt använt degenereras undantagshantering till en form av *goto*-instruktion. Undantagshantering är dessutom mycket krävande i form av processorkraft och en ökad komplexitet i programmen. Man bör således i möjligaste mån undvika användning av undantag vid programmering.

Det finns dock tillfällen när det kan vara lämpligt att använda sig av undantagshantering. Ett exempel är att i utvecklingsfasen använda sig av undantagshantering för att upptäcka fel. Annars kan man med fördel kombinera kontrakt och undantagshantering när man inte på förväg kan veta hur det kommer att gå. Exempel på detta är när det handlar om fel med kommunikationslänkar, operativsystemet eller hårdvaran, som *input*- eller *output*-fel [17]. Det som är viktigt att tänka på när man kombinerar kontrakt och undantagshantering, är att även undantagen klart och tydligt skall dokumenteras i kontraktspecifikationen, så att inga oklarheter uppstår.

2.4.4 Skillnader mellan kontraktsprogrammering och defensiv programmering

Den stora skillnaden mellan kontraktsprogrammering och defensiv programmering är hur man ser på användaren av en metod. I kontraktsprogrammering så krävs det av användaren eller klienten att uppfylla vissa kriterier för att kunna få använda metoden. Endast om dessa kriterier uppfylls så garanteras ett korrekt resultat. Således bygger kontraktsprogrammering på ett ömsesidigt förtroende mellan klient och metod.

Vid defensiv programmering saknas däremot helt förtroende mellan klient och metod. Varje värde måste testas och kontrolleras så att eventuella fel kan upptäckas. Detta innebär att mycket extra kod i form av att tester måste implementeras både i klienten och i metoden.

Detta leder ofta till att koden ökar i omfattning och storlek samt att läsbarheten minskar. Det kan också vara mycket svårt att täcka in och förutse alla fel som kan tänkas förekomma.

En stor fördel med kontraktsprogrammering är att man inne i en metod inte behöver täcka in alla felaktiga fall, utan endast behöver koncentrera sig på de fall där metoden används korrekt. Detta medför att koden oftast blir mer kompakt eftersom den inte behöver innehålla tester på värden i samma omfattning, vilket i sin tur leder till ökad läsbarhet. Risken att metoder används felaktigt minskar eftersom användare respekterar förvillkoren och tar del av eftervillkoren. Nackdelen med kontraktsprogrammering är dock att detta inte kan tillämpas i alla fall. Exempel på detta kan vara mjukvara som inte kan tolerera att systemen slutar att fungera bara för att ett förvillkor inte uppfylls. Exempel på sådan mjukvara kan vara mjukvara som ingår i kritiska system såsom styrsystem till kärnreaktorer och fordon. Sådan mjukvara måste innehålla mängder med tester för att kunna upptäcka fel vid onormala och oväntade situationer, så att eventuella katastrofer förhindras.

2.5 Tidigare arbeten

Våren 2000 gavs två studenter på dataingenjörsprogrammet vid Karlstads universitet, uppdraget att utveckla en Java-editor som skulle presentera semantisk information för användaren. Detta arbete ingick i dessa studenters C-uppsats [15] och utvecklingstiden låg därför på totalt 20 veckor. Den semantiska informationen skulle vara i form av för- och eftervillkor för metoder, vilka skulle presenteras i skrivande stund.

Man löste uppgiften genom att utveckla en ny editor, CoffeeMaker, som baserades på en redan befintlig editor, Jipe [13]. Även editorn Jipe baserade sin lösning på en redan befintlig editor nämligen jEdit [12]. Anledningen till att man valde att basera sin lösning på en redan befintlig editor var att själva slippa implementera *syntax highlighting*.

Man lyckades med uppgiften att utveckla en funktionalitet som presenterade kontrakt i skrivande stund. Dock hade denna lösning vissa begränsningar. Dessa begränsningar omfattade bland annat att endast kontrakt för klasser placerade i samma paketstruktur, kunde presenteras. Dessutom kunde inte ärvda metoders kontrakt presenteras och funktionaliteten kunde ej heller stängas av. Prestandan var också ett stort problem vilket dels berodde på att editorn implementerades i Java men till största delen på ineffektiva algoritmer.

På grund av editorns begränsade funktionalitet och ovan nämnda begränsningar, kunde CoffeeMaker aldrig tas i bruk i någon större utsträckning. Därför slutade CoffeeMaker vid att bara vara en prototyp utan egentligt användarvärde, dock med vissa utvecklingsmöjligheter.

Förutom CoffeMaker så saknar vi kännedom om någon annan editor eller utvecklingsmiljö som stöder kontraktsprogrammering i form av presentation av kontrakt för programmeraren. Däremot finns en mängd separata verktyg och program, både kommersiella och icke kommersiella, för att underlätta kontraktsprogrammering. Vi har valt att titta närmare på några av dessa verktyg nämligen iContract, Jass, jContractor och Jcontract, vilka samtliga är utvecklade för att användas vid kontraktsprogrammering i Java. Dessa fyra verktyg presenteras i kapitel 3.

3 Befintliga verktyg

I detta kapitel presenteras ett antal externa verktyg, vars syfte är att underlätta användningen av kontraktsprogrammering. De verktyg som presenteras i kapitlet är gjorda för att användas vid kontraktsprogrammering i programmeringsspråket Java. Dessa verktyg underlättar dock inte direkt kontraktsprogrammeringen vid editering av källkod, utan de försöker spåra kontrakt som ej efterlevs först under exekvering av program. Verktygen används därför främst i testningssyfte för att kunna upptäcka kontraktsbrott i program, men de erbjuder ingen direkt hjälp för att förhindra att kontraktsbrotten uppstår under utveckling av program.

3.1 Inledning

De flesta programmeringsspråk saknar direkt stöd för kontraktsprogrammering och detta måste därför tillföras som en extra del. Ett programmeringsspråk som däremot har ett inbyggt stöd för kontraktsprogrammering är Eiffel. I Eiffel skrivs kontrakten direkt i koden med nyckelorden *require* för förvillkor, *ensure* för eftervillkor och *invariant* för klassinvariant.

Programmeringsspråket Java har i version 1.4 infört en form av semantikkontroll, så kallade *assertions*. Ett *assertion* är ett uttryck i Java som ger programmeraren möjligheter att kontrollera de antaganden man har gjort för programmet. Om man till exempel skriver en metod som skall beräkna en partikels hastighet kan man skriva ett *assertion* om att den uträknade hastigheten är mindre än ljusets hastighet. Varje *assertion* innehåller ett booleskt uttryck som antas vara sant när detta *assertion* exekverar. Om det inte är sant kommer systemet att kasta ett undantag. Genom att verifiera att det booleska uttrycket verkligen är sant, bekräftar detta *assertion* programmets beteende vilket ökar tilltron till att programmet är felfritt. Erfarenheter har visat, enligt Sun Microsystems [20], att användandet av *assertions* när man programmerar är ett av de snabbaste och mest effektiva sätten att hitta och korrigera fel på. Som en extra bonus dokumenterar *assertions* programmets funktion och underlättar därmed underhållningsarbetet av programmet. [20]

Tyvärr så är *assertion*-mekanismen i Java väldigt primitiv och möjlighet att definiera för- och eftervillkor finns ej. Förespråkare för kontraktsprogrammering har därför valt att gå

ytterligare ett steg för att underlätta kontraktsprogrammering i Java genom att tillhandahålla olika externa verktyg. Några av dessa verktyg är iContract, Jass, jContractor och Jcontract.

Provkörningar av dessa verktyg har genomförts där enkla kontrakt bryts för att se hur verktygen hanterar sådana situationer. Klassen som har använts i testerna presenteras i kapitel 3.2. Klassen i kapitel 3.2 har anpassats för varje verktyg där det specifika verktygets syntax har använts, de anpassade klasserna återfinns i bilaga A.

3.2 Testklass - MyStack

```
/**
 * Invariant: storleken på stacken ligger alltid mellan 0 och MAX (0<=size<=MAX).
 */
public class MyStack {
    private java.util.Vector myStack;
    private final int MAX = 10;
    private int size;

    /**
     * Förvillkor: true
     * Eftervillkor: stacken är initierad och har storleken 0.
     */
    public MyStack() {
        this.myStack = new java.util.Vector();
        this.size = 0;
    }

    /**
     * Förvillkor: stacken är inte tom.
     * Eftervillkor: det översta elementet är borttaget, storleken har minskat med 1 samt
     *               stacken är inte full.
     */
    public void pop() {
        this.myStack.remove(this.getSize() - 1);
        this.size--;
    }

    /**
     * Förvillkor: stacken är inte full och elementet e är ett giltigt objekt.
     * Eftervillkor: elementet e har lagts överst på stacken, storleken har ökat med 1 samt
     *               stacken är inte tom.
     */
    public void push(Object e) {
        this.myStack.add(e);
        this.size++;
    }
```

```
/**
 * Förvillkor: stacken är inte tom.
 * Eftervillkor: det översta elementet på stacken har returnerats.
 */
public Object top() {
    return this.myStack.get(this.getSize() - 1);
}

/**
 * Förvillkor: true
 * Eftervillkor: storleken på stacken har returnerats.
 */
public int getSize() {
    return this.size;
}

/**
 * Förvillkor: true
 * Eftervillkor: det returnerade värdet har samma sanningsvärde som påståendet att stacken
 *               är tom dvs size == 0.
 */
public boolean isEmpty() {
    return this.getSize() == 0;
}

/**
 * Förvillkor: true
 * Eftervillkor: det returnerade värdet har samma sanningsvärde som påståendet att stacken
 *               är full dvs size == MAX.
 */
public boolean isFull() {
    return this.getSize() == this.MAX;
}

/**
 * Förvillkor: true
 * Eftervillkor: stacken är tom.
 */
public void makeEmpty() {
    this.myStack.clear();
    this.size = 0;
}

/**
 * Förvillkor: true
 * Eftervillkor: true
 */
public static void main(String[] args) {
    MyStack stack = new MyStack();

    if(!stack.isFull()) {
```

```
            stack.push(new Integer(0));
        }
        if(!stack.isEmpty()) {
            Object o = stack.top();
            stack.pop();
        }
        for(int i = 0; i < 11; i++) {
            stack.push(new Integer(i));
        }
        stack.makeEmpty();
    }
}
```

3.3 iContract

iContract är ett verktyg för Java som ger utvecklare stöd vid kontraktsprogrammering. Principen som används är att kontrakten specificeras som integrerade delar av källkoden. Likheten med den princip som används i programmeringsspråket Eiffel är tydlig, se kapitel 3.1. Utvecklarna av iContract nämner också denna likhet i sin beskrivning av verktyget. iContract är en gratis så kallad källkodspreprocessor. Att iContract är en källkodspreprocessor innebär att iContract går igenom koden före kompileringen och gör en egen källkodsfil där klassinvarianter och för- och eftervillkor för metoderna finns integrerade i koden. Kontrakt skrivs i Javadoc (se kapitel 5.3.1) med hjälp av specialtaggarna @*pre* för förvillkor, @*post* för eftervillkor och @*invariant* för klassinvariant. För mer ingående beskrivning hänvisas läsaren till [7] och [16].

3.3.1 Provkörning av iContract

Kontrakten för klassen *MyStack* i kapitel 3.2, har modifierats för att följa syntaxen för kontraktsbeskrivningar i iContract, se bilaga A.1. iContract följer till viss del OCL's [6] syntax och är därmed relativt enkelt att använda samt erbjuder en del kraftfulla operatorer. Symbolen *return* är motsvarigheten till *result* i OCL, det vill säga en metods returvärde, vilken kan användas i eftervillkoret för en metod. På samma sätt som i OCL så kan man i iContract referera till tillståndet före ett metodanrop genom att lägga till @*pre* på typen som skall kontrolleras. När man använder iContract så skapas en ny version av .java-filen/-erna vilka inkluderar testerna av de specificerade kontrakten, se bilaga A.2. Även en fil som används av iContract för att möjliggöra arv av kontrakt genereras. För att sedan testa om applikationen uppfyller kontrakten, så kompileras de av iContract genererade .java-filerna. De resulterande .class-filerna kan sedan exekveras av iContract för att kontrollera om kontraktsbrott förekommer.

Figur 3.1 visar resultatet av att kompilera och exekvera den av iContract genererade *MyStack*-filen, innehållande kontraktstesterna, med iContract. Verktyget meddelar användaren att applikationen bryter mot ett kontrakt på rad 599 i main-metoden, genom att inte uppfylla förvillkoret (`!isFull() && e != null`) för metoden *push()*. Metoden vars förvillkor bryts återfinns på rad 212. Båda dessa radnummer refererar till den av iContract automatgenererade källkodsfilen MyStack.java (bilaga A.2). Metoden *push()* finns deklarerad på rad 35 i källkodsfilen MyStack.java (bilaga A.1), vilket också kan utläsas ur Figur 3.1.

```
D:\program\iContract>java -cp C:\WINDOWS;C:\WINDOWS\COMMAND;D:\PROGRAM\J2SDK1~1.
1_0\BIN;D:\PROGRAM\J2SDK1~1.1_0\LIB\TOOLS.JAR;D:\PROGRAM\JAVACC~1\JAVACC2.1\BIN;
D:\PROGRAM\ANT\BIN;D:\PROGRAM\ICONTR~1\ICONTR~3.JAR;src;_contract_db;instr;. com
.reliablesystems.iContract.Tool -Z -a -v -minv,pre,post -b"javac -classpath d:\p
rogram\icontract\iContract.jar;src;." -c"javac -classpath d:\program\icontract\i
Contract.jar;instr;." -n"javac -classpath d:\program\icontract\iContract.jar;_co
ntract_db;instr;." -oinstr/@p/@f.@e -k_contract_db/@p src/MyStack.java
iContract:progress iContract, Version for JDK 1.2, 0.3d2
iContract:progress Copyright (C) 1997-2000 Reto Kramer <info@reliable-systems.co
m>
iContract:progress parsing input files to determine dependencies.
.

iContract:progress Analyzed 1 files in 0.44 s.

iContract:progress found 1 relavant types referenced in the 1 files.
iContract:progress starting dependency analysis (among 1 types)
iContract:progress unconditionally instrumenting all files (-a).
iContract:progress dependency analysis completed (1 levels).
iContract:progress javac -classpath d:\program\icontract\iContract.jar;src;. src
/MyStack.java
iContract:progress javac -classpath d:\program\icontract\iContract.jar;instr;. i
nstr\\MyStack.java
iContract:progress javac -classpath d:\program\icontract\iContract.jar;_contract
_db;instr;. _contract_db\__REP_MyStack.java

D:\program\iContract>java -cp instr/ MyStack
Exception in thread "main" java.lang.RuntimeException: java.lang.RuntimeExceptio
n: src/MyStack.java:35: error: precondition violated (MyStack::push(java.lang.Ob
ject)): (/*declared in MyStack::push(java.lang.Object)*/ (!isFull())) && ((e !=
null))
        at MyStack.push(MyStack.java:212)
        at MyStack.main(MyStack.java:599)

D:\program\iContract>
```

Figur 3.1: Provkörning iContract.

3.3.2 Fördelar med iContract

- iContracts kontraktssyntax är enkel att använda samt innefattar även kraftfulla uttryck beroende på att syntaxen till viss del följer OCL-specifikationen.
- Eftersom kontrakten uttrycks i form av Javadoc, så hålls källkoden kompatibel med standard Java.
- Det är enkelt att generera dokumentation av kontrakten med hjälp av Javadoc-verktyget.
- Möjlighet att styra vilka typer av villkor (förvillkor, eftervillkor, invarianter) som skall testas och i vilka klasser.
- iContract stöder arv av kontrakt.
- iContract är gratis att använda.

3.3.3 Nackdelar med iContract

- Inställningarna för iContract är relativt krångliga och det är helt nödvändigt att skapa en script-fil för att slippa skriva de långa och krångliga uttrycken om och om igen.
- I iContract kan inte ett kontrakt innehålla en referens till det tidigare tillståndet för den givna metoden. Till exempel *@post return == top()@pre* för metoden *top()* resulterar i en rekursiv "evighets" loop, vilket innebar att detta eftervillkor fick utelämnas i kontraktet för metoden *top()*.
- En annan nackdel är att en ny .java-fil måste skapas speciellt för kontraktstesterna. Detta medför att felmeddelandena om kontraktsbrott som visas upp i iContract refererar till denna fil, vilket innebär att det kan vara svårare att spåra kontraktsbrottet i originalkällkodsfilen.
- Ytterligare en nackdel är att om en metod har flera förvillkor och/eller eftervillkor så kommer dessa att slås samman till ett förvillkor respektive eftervillkor i koden. Detta resulterar i att det blir svårare att spåra vilket specifikt för- respektive eftervillkor som inte uppfylls i kontraktet. Detta ses i Figur 3.1 där det står *precondition violated ... (!isFull() && (e != null))...*. Användaren får här ingen information om det är förvillkoret *!isFull()* eller om det är förvillkoret *e != null* som inte uppfylls.
- Bara det första kontraktsbrottet presenteras. Detta beror på att undantag som kastas vid kontraktsbrott inte fångas, vilket leder till att applikationen terminerar.

3.4 Jass – Java with Assertions

Jass är ett verktyg som erbjuder kontraktsprogrammering i Java enligt samma princip som programspråket Eiffel. Jass är en källkodspreprocessor som går igenom källkoden innan den kompileras och ändrar Jass-specifika nyckelord till integrerad kod. Den slutliga koden är en ren Javakod där kontrakt kontrolleras dynamiskt under programexekvering.

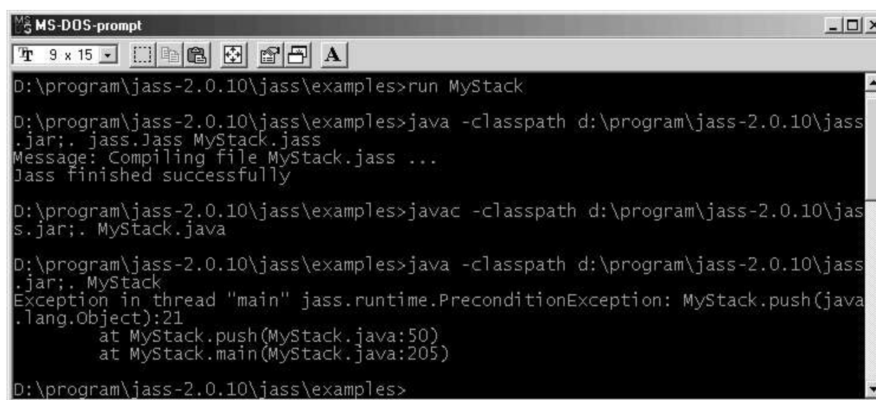
Verktyget Jass använder sig av .jass-filer, vilka innehåller vanlig Java-kod samt kontraktsbeskrivningar som kan tolkas av verktyget. Dessa kontraktsbeskrivningar utgörs av de tre nyckelorden *require*, *ensure* samt *invariant*, vilka representerar förvillkor, eftervillkor respektive invariant (jämför med Eiffel i kapitel 3.1). Kontraktsbeskrivningarna skrivs mellan tecknen /** och **/, vilka placeras på olika ställen i källkoden. Nyckelordet *require* måste stå först i en metod, medan *ensure* skall stå sist i metoden. Klassinvarianten anges sist i klassen med nyckelordet *invariant*. I Jass syntax ingår även bland annat nyckelorden *Result* och *Old*,

där *Result* representerar en metods returvärde och *Old* refererar till objektets tillstånd före metodanropet. *Old* är helt enkelt en kopia av objektet och därför måste klassen som använder sig av *Old* implementera interfacet *Cloneable* för att denna kopia skall kunna skapas.

När man kompilerar .jass-filen med hjälp av Jass så genereras en .java-fil med samma namn som .jass-filen. Denna fil innehåller förutom den ursprungliga koden i .jass-filen också testerna av kontrakten i form av Java-kod. Denna fil kan sedan kompileras och den resulterande .class-filen exekveras med hjälp av verktyget. Jass kommer att meddela användaren, om ett kontraktsbrott förekommer, genom att kasta ett undantag för respektive villkor. För ytterligare information om Jass hänvisas läsaren till [8] och [3].

3.4.1 Provkörning av Jass

För att kunna testa klassen *MyStack* i kapitel 3.2, så måste motsvarande .jass-fil skapas. Filen MyStack.jass återfinns i bilaga A.3 och som kan ses här implementerar klassen interfacet *Cloneable* för att möjliggöra användandet av *Old*. I Figur 3.2 visas resultatet av användningen av Jass på filen MyStack.jass och den av Jass genererade filen MyStack.java, den senare återfinns i bilaga A.4. Som kan ses i figuren så har ett undantag, jass.runtime.PreconditionException, kastas. Detta eftersom ett förvillkor för metoden *push()* i MyStack.java på rad 205 brutits. Även på vilken rad det brutna förvillkoret specificerats, i både MyStack.java (rad 50) samt MyStack.jass (rad 21), kan utläsas i figuren. Dock ges ingen information om vad detta förvillkor innefattar.



Figur 3.2: Provkörning Jass.

3.4.2 Fördelar med Jass

- Verktyget är enkelt att använda och kräver inga krångliga inställningar.
- Valmöjlighet finns om undantag skall kastas eller om endast en varning skall utfärdas vid kontraktsbrott. Detta möjliggör att exekvera hela applikationen färdigt och på så sätt kunna lokalisera flera kontraktsbrott utan att behöva exekvera applikationen om och om igen.
- Man har valmöjligheten att välja vilka typer av villkor som skall testas eller alternativt att inga tester skall infogas i den genererade .java-filen.
- Stöd för arv av kontrakt.
- Det är möjligt att lägga in kommentarer i kontrakten utan att testerna påverkas.
- Jass är gratis att använda.

3.4.3 Nackdelar med Jass

- Verktyget kräver att man skapar en särskild sorts källkodsfiler, .jass-filer.
- Kontraktsbeskrivningen följer inte Javadoc-beskrivningen samt att istället för OCL används en mer Eiffel-orienterad kontraktsbeskrivning.
- Vilket specifikt villkor som brutits presenteras ej och felmeddelandet refererar till kontraktsbrottet i den av Jass genererade .java-filen, vilket försvårar spårningen av brottet i källkoden (.jass-filen).
- Alla filer som använder sig av *Old* i kontraktsbeskrivningen måste implementera interfacet *Cloneable*, vilket innebär att klasserna utökas.
- På grund av att kontrakten delas upp och specificeras i olika delar av källkoden, minskar överblicken samt läsbarheten av kontrakten.
- Jass kräver ett separat verktyg för att generera dokumentation över kontrakten.

3.5 jContractor

jContractor ger en programmerare stöd för kontraktsprogrammering i Java. Detta sker genom att kontrakt skrivs som metoder som följer en enkel benämningskonvention. På så sätt behövs ingen förbehandling av källkoden, den är redan skriven i ren Java.

Verktyget jContractor arbetar med .java-filer i vilka kontrakt specificeras som booleska metoder. Dessa metoder följer benämningskonventionen enligt följande: förvillkor för en

metod specificeras genom att definiera en metod med samma namn som den kontrakterade metoden + suffixet *_Precondition*. Förvillkorsmetoden tar in samma parametrar som metoden den gäller för och dess returvärde är ett booleskt uttryck som anger om det specificerade förvillkoret uppfylls eller ej. Eftervillkor specificeras på samma sätt som förvillkor med den skillnaden att suffixet hos dessa är *_Postcondition*. Dessa metoder tar in, förutom de parametrar som metoden den gäller för, även parametern *RESULT*. Denna parameter representerar returvärdet av den kontrakterade metoden. Om metoden saknar returvärde anges parametern att vara av typen *Void*. Klassinvarianter anges i metoden *_Invariant* vilken saknar inparametrar. Liksom Jass (se kapitel 3.4) använder sig jContractor av objektreferensen *OLD*, vilken måste deklareras explicit, för att kunna referera till tillståndet före ett metodanrop. Därmed måste även interfacet *Cloneable* implementeras av den kontrakterade klassen, för att möjliggöra kloning av objekt.

Den kontrakterade källkodsfilen kan köras antingen genom jContractor eller genom jInstrument. Skillnaden mellan de båda är att jContractor kontrollerar kontrakten utan att testerna behöver infogas i .class-filen vilket de måste om de körs genom jInstrument. jInstrument kan också användas för att avlägsna kontraktstester i .class-filen genom parametern *–s*. För mer ingående information om jContractor eller jInstrument hänvisas läsaren till [11] och [14].

3.5.1 Provkörning av jContractor

I bilaga A.5 återfinns klassen *MyStack* vilken har anpassats till att följa jContractors kontraktsbeskrivning. I Figur 3.3 visas resultatet av att köra klassen *MyStack* genom jContractor. Som kan utläsas ur figuren så har ett undantag, *PreconditionViolationError*, kastats eftersom ett förvillkor har brutits på rad 135 i källkodsfilen. Förvillkoret gäller för metoden *push()* men ingen information om förvillkoret ges i felmeddelandet. I Figur 3.4 visas resultatet av att köra klassen *MyStack* genom jInstrument. Samma undantag kastas och samma information ges till användaren om kontraktsbrottet som vid jContractor-exekveringen. Skillnaden ligger i att här finns alla tester inkluderade i MyStack.class vilket innebär att felmeddelandet blir betydligt kortare än det i Figur 3.3.

```
D:\program\jcontractor-0.1>javac MyStack.java

D:\program\jcontractor-0.1>java -classpath d:\program\jcontractor-0.1\jcontracto
r.jar;. jContractor --verbose MyStack
Instrumenting MyStack at 'all' level
Instrumenting edu.ucsb.ccs.jcontractor.InvariantViolationError at 'all' level
Instrumenting edu.ucsb.ccs.jcontractor.PreconditionViolationError at 'all' level

Instrumenting edu.ucsb.ccs.jcontractor.PostconditionViolationError at 'all' leve
l
Instrumenting edu.ucsb.ccs.jcontractor.jContractorRuntime at 'all' level
jContractor Exception occured!
edu.ucsb.ccs.jcontractor.PreconditionViolationError: jContractor Exception: Prec
ondition Violated
        at MyStack.push(MyStack.java)
        at MyStack.main(MyStack.java:135)
        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
        at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.
java:39)
        at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAcces
sorImpl.java:25)
        at java.lang.reflect.Method.invoke(Method.java:324)
        at edu.ucsb.ccs.jcontractor.jContractor.runInstrumented(jContractor.java
:141)
        at edu.ucsb.ccs.jcontractor.jContractor.main(jContractor.java:266)
        at jContractor.main(jContractor.java:20)

D:\program\jcontractor-0.1>
```

Figur 3.3: Provkörning jContractor.

```
D:\program\jcontractor-0.1>javac MyStack.java

D:\program\jcontractor-0.1>java -classpath d:\program\jcontractor-0.1\jcontracto
r.jar;. jInstrument --verbose MyStack.class
Instrumenting MyStack at 'all' level

D:\program\jcontractor-0.1>java -classpath d:\program\jcontractor-0.1\jcontracto
r.jar;. MyStack
Exception in thread "main" edu.ucsb.ccs.jcontractor.PreconditionViolationError:
jContractor Exception: Precondition Violated
        at MyStack.push(MyStack.java)
        at MyStack.main(MyStack.java:135)

D:\program\jcontractor-0.1>
```

Figur 3.4: Provkörning jInstrument.

3.5.2 Fördelar med jContractor

- Den största fördelen med jContractor är att man kan specificera kontrakt för klasser vars källkod inte är tillgänglig. Detta görs genom att skapa en klass med samma namn som den klass som kontraktet skall gälla för plus suffixet *_Contract*. I denna klass kan sedan kontrakten specificeras på samma sätt som beskrivits tidigare.
- Arv av kontrakt stöds av verktyget.
- Möjlighet att påverka vilka villkor i kontrakten som skall testas samt i vilka klasser dessa tester skall infogas.
- jContractor är gratis att använda.

3.5.3 Nackdelar med jContractor

- Kräver att man skriver kontraktsmetoderna själv i källkoden, vilket innebär mer arbete och mycket extra kod, vilket kan leda till mindre läsbara program.

- Kräver att alla klasser som använder sig av referensvariabeln *OLD* implementerar interfacet *Cloneable*, vilket innebär att klassen utökas.
- Felmeddelandet anger inte vilket specifikt villkor som inte uppfylls i kontraktet.
- Endast det första kontraktsbrottet visas eftersom applikationen avslutas när ett undantag kastas.
- Eftersom kontrakten inte specificeras i form av Javadoc, så krävs extern dokumentation av kontrakten.

3.6 Jcontract

Jcontract är ett verktyg som underlättar kontraktsprogrammering i Java. Kontrakt anges i Javadoc-kommentarerna med speciella taggar. Jcontract har en egen kompilator, *dbc_javac*, vilken tolkar kontraktsspecifikationen i Javadocen och omvandlar den till exekverbar kod, vilken infogas i .class-filerna. De resulterande .class-filerna innehåller extra byte-kod som kontrollerar kontrakten under programexekvering. Eventuella kontraktsbrott presenteras i ett grafiskt användargränssnitt, i ett konsolfönster eller i en fil enligt konfiguration.

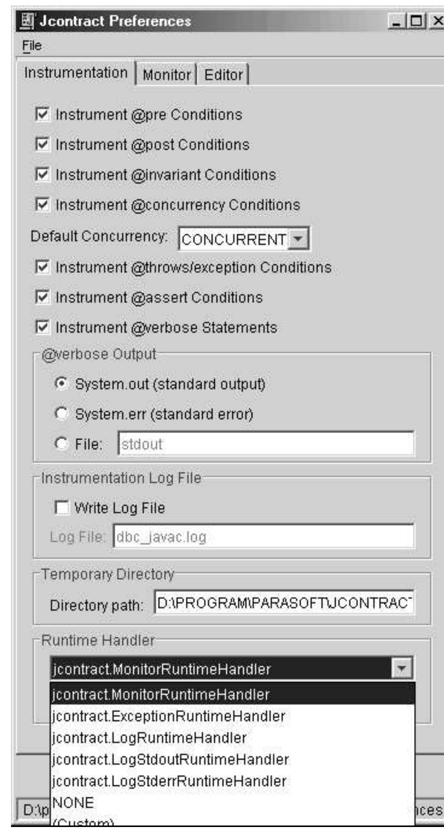
Jcontract använder sig av taggarna @*pre*, @*post* samt @*invariant* för att definiera förvillkor, eftervillkor respektive klassinvarianter i Javadocen. Syntaxen påminner om den syntax som beskrivs för OCL [6]. Symbolen *$result* används för att referera till metodens returvärde och symbolen *$pre* används till att referera till objektets tillstånd före anropet av metoden. Jcontract använder även symbolen *$none* för att specificera ”sanna” kontraktsvillkor, det vill säga där inga kontrakt finns eller villkor som alltid är uppfyllda. Genom att skriva *$none* efter kontraktstaggarna så kommer Jcontract att ignorera dessa och därmed genereras inga test för dessa villkor.

För ytterligare information om Jcontract hänvisas läsaren till [19].

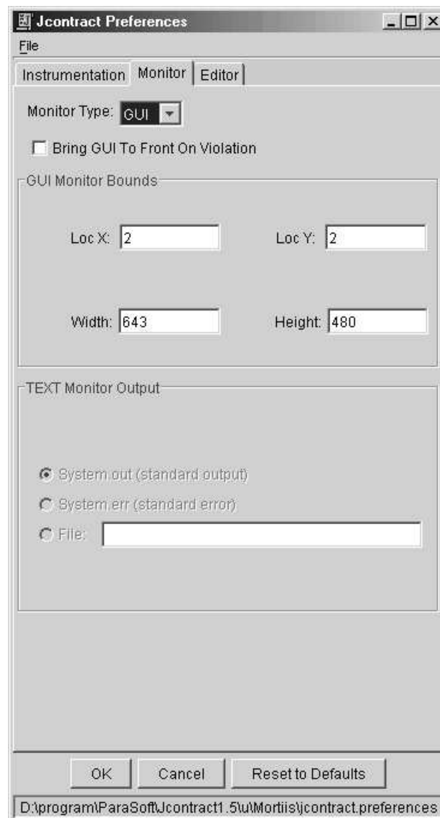
3.6.1 Provkörning av Jcontract

Alla inställningar sker genom ett grafiskt användargränssnitt *Jcontract Preferences*, som i sin tur utgörs av tre vyer. Dessa tre vyer är *Instrumentation*, *Monitor* och *Editor*. I vyn *Instrumentation*, se Figur 3.5, görs inställningar om vad som skall testas i kontrakten och hur dessa test skall behandlas under exekvering. I vyn *Monitor*, se Figur 3.6, anges om

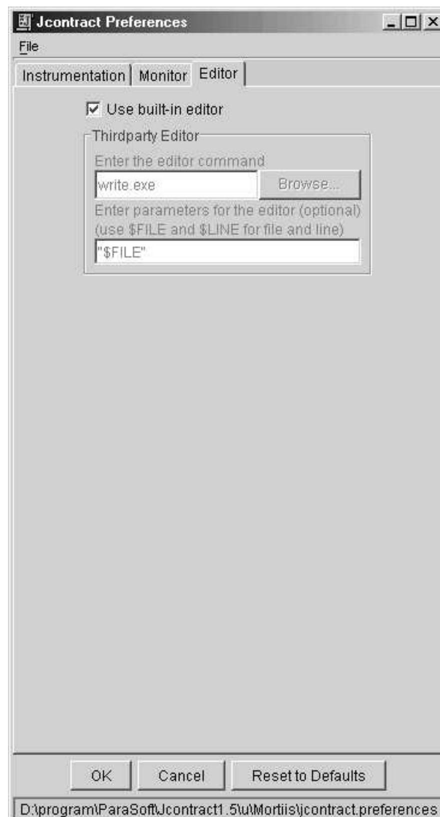
felmeddelanden skall presenteras i textform eller i ett grafiskt användargränssnitt. I vyn *Editor*, se Figur 3.7, så anges vilken editor som skall kopplas samman med Jcontract.



Figur 3.5: Jcontract inställningar - Instrumentation.

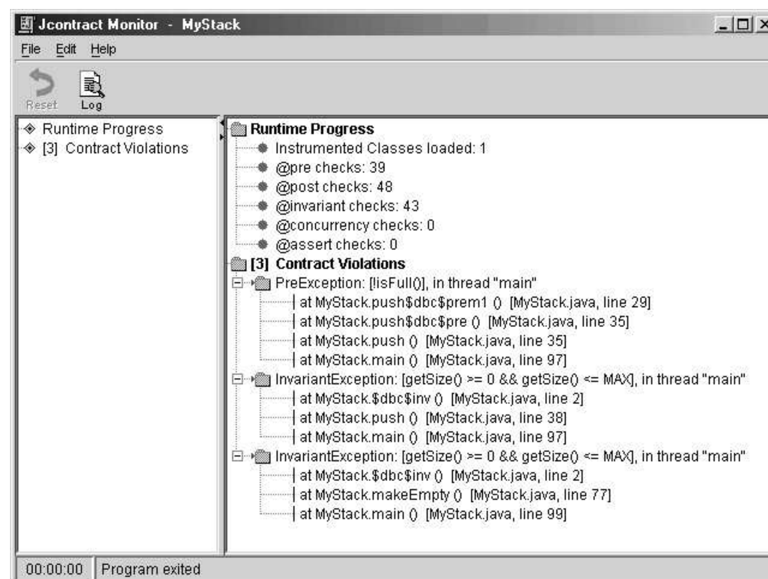


Figur 3.6: Jcontract inställningar - Monitor.



Figur 3.7: Jcontract inställningar - Editor.

Figur 3.8 visar hur Jcontract beter sig när man exekverar klassen *MyStack*, vilken återfinns i bilaga A.6, i det grafiska monitorläget. I figuren kan utläsas hur många test av respektive villkor som har införts i .class-filen. Man kan också utläsa att applikationen bryter mot förvillkoret `(!isFull())` för metoden *push()*, på rad 97 i main-metoden. Som en följd av detta kontraktsbrott bryts även klassinvarianten för klassen på två ställen i koden, vilket också framgår av figuren. Om man önskar kan man gå direkt till källkoden för att kunna rätta till kontraktsbrottet, vilket visas i Figur 3.9. Jcontract kan även köras så att informationen presenteras direkt i konsolen istället för i ett grafiskt användargränssnitt, vilket illustreras i Figur 3.10. Jcontract kan även köras i exception-läget, vilket innebär att ett undantag kastas för varje kontraktsbrott. Detta innebär att endast det första kontraktsbrottet presenteras, eftersom applikationen avslutas på grund av att detta undantag inte fångas. Detta läge presenteras i Figur 3.11.



Figur 3.8: Jcontract monitor med GUI.

```
D:\program\ParaSoft\Jcontract1.5\examples\MyStack.java
File  Edit  Help
81
82      /**
83       * @pre $none
84       * @post $none
85       */
86      public static void main(String[] args) {
87          MyStack stack = new MyStack();
88
89          if(!stack.isFull()) {
90              stack.push(new Integer(0));
91          }
92          if(!stack.isEmpty()) {
93              Object o = stack.top();
94              stack.pop();
95          }
96          for(int i = 0; i < 11; i++) {
97              stack.push(new Integer(i));
98          }
99          stack.makeEmpty();
100     }
101 }
                                            Ln 97, Col 1 | Bytes selected: 0
```

Figur 3.9: Jcontract editor.

```
MS-DOS-prompt
D:\program\ParaSoft\Jcontract1.5\examples>dbc_javac -Zverbose MyStack.java
dbc_javac: Version 1.5 -- Copyright (C) 2000-2002 ParaSoft
dbc_javac: compiling original classes
dbc_javac: instrumenting files
dbc_javac:     D:\program\ParaSoft\Jcontract1.5\examples\MyStack.java: instrumen
ted
dbc_javac: compiling instrumented classes
dbc_javac: patching .class files
dbc_javac: -Zstatistics:
dbc_javac:     .java files compiled: 1
dbc_javac:     .java files instrumented: 1
dbc_javac:     .class files generated: 1
dbc_javac: -Ztimings:
dbc_javac:     Total Time: 4,9 seconds
dbc_javac:         Time compiling original classes     : 1,7 seconds
dbc_javac:         Time instrumenting                  : 0,7 seconds
dbc_javac:         Time compiling instrumented classes: 2,2 seconds
dbc_javac:         Time patching .class files          : 0,1 seconds
dbc_javac: -Zusage: memory usage = 669032
dbc_javac: files compiled: 1, files instrumented: 1

D:\program\ParaSoft\Jcontract1.5\examples>java -classpath d:\program\parasoft\jc
ontract1.5\bin\jcontract.jar;. MyStack
Jcontract: Version 1.5 -- Copyright (C) 2000-2002 ParaSoft
Jcontract: PreException: [!isFull()], in thread "main"
        at MyStack.push$dbc$prem1(MyStack.java:29)
        at MyStack.push$dbc$pre(MyStack.java:35)
        at MyStack.push(MyStack.java:35)
        at MyStack.main(MyStack.java:97)
Jcontract: InvariantException: [getSize() >= 0 && getSize() <= MAX], in thread "
main"
        at MyStack.$dbc$inv(MyStack.java:2)
        at MyStack.push(MyStack.java:38)
        at MyStack.main(MyStack.java:97)
Jcontract: InvariantException: [getSize() >= 0 && getSize() <= MAX], in thread "
main"
        at MyStack.$dbc$inv(MyStack.java:2)
        at MyStack.makeEmpty(MyStack.java:77)
        at MyStack.main(MyStack.java:99)
Jcontract: Runtime Statistics:
    Instrumented classes loaded: 1
    @pre         checks: 39
    @post        checks: 48
    @invariant   checks: 43
    @concurrency checks: 0
    @assert      checks: 0
```

Figur 3.10: Jcontract monitor i konsol.

```
MS-DOS-prompt
D:\program\ParaSoft\Jcontract1.5\examples>dbc_javac -Zverbose MyStack.java
dbc_javac: Version 1.5 -- Copyright (C) 2000-2002 ParaSoft
dbc_javac: compiling original classes
dbc_javac: instrumenting files
dbc_javac:     D:\program\ParaSoft\Jcontract1.5\examples\MyStack.java: instrumen
ted
dbc_javac: compiling instrumented classes
dbc_javac: patching .class files
dbc_javac: -Zstatistics:
dbc_javac:     .java files compiled: 1
dbc_javac:     .java files instrumented: 1
dbc_javac:     .class files generated: 1
dbc_javac: -Ztimings:
dbc_javac:     Total Time: 4,9 seconds
dbc_javac:         Time compiling original classes    : 1,7 seconds
dbc_javac:         Time instrumenting                 : 0,7 seconds
dbc_javac:         Time compiling instrumented classes: 2,2 seconds
dbc_javac:         Time patching .class files         : 0,0 seconds
dbc_javac: -Zusage: memory usage = 669032
dbc_javac: files compiled: 1, files instrumented: 1

D:\program\ParaSoft\Jcontract1.5\examples>java -classpath d:\program\parasoft\jc
ontract1.5\bin\jcontract.jar;. MyStack
Exception in thread "main" jcontract.PreException: [!isFull()]
        at MyStack.push$dbc$prem1(MyStack.java:29)
        at MyStack.push$dbc$pre(MyStack.java:35)
        at MyStack.push(MyStack.java:35)
        at MyStack.main(MyStack.java:97)

D:\program\ParaSoft\Jcontract1.5\examples>_
```

Figur 3.11: Jcontract med undantag.

3.6.2 Fördelar med Jcontract

- Enkelt att skriva kontrakt eftersom dessa införs i övrig Javadoc för metoden.
- Inställningar är mycket enkla att genomföra genom det grafiska gränssnittet.
- Möjlighet att välja vilka villkor som skall testas i kontrakten.
- Mycket användbar funktion som möjliggör användande av den inbyggda editorn eller valfri editor för att kunna gå direkt till källkoden där villkoret bryts.
- Ett särskilt nyckelord för ”sanna” villkor, det vill säga sådana villkor som alltid är uppfyllda, erbjuds i form av *$none*. Denna deklaration vid en viss kontraktstagg innebär att inget test genereras för detta villkor.
- Mycket bra presentation av kontraktsbrottet genom att de olika villkoren i ett kontrakt särskiljs. Detta innebär att användaren får information om vilket specifikt villkor som inte uppfylls.

3.6.3 Nackdelar med Jcontract

- Den enda egentliga nackdelen med Jcontract är att detta verktyg kräver en licens för att brukas, vilket gör det till ett icke kostnadsfritt verktyg.

3.7 Sammanfattning

Jcontract är det verktyg som fungerar bäst och är det mest lättanvända av de testade verktygen, det är dock det enda av verktygen som kräver licens för att användas vilket kan ses

som en nackdel. Det av de kostnadsfria verktygen som fungerar bäst är iContract med den nackdelen att verktyget kräver krångliga inställningar vid användning. Inget av de testade verktygen stöder dock en programmerare vid programmering. De ger endast stöd för kontraktskontroll efter det att programmen och kontrakten redan är skrivna. Detta innebär att testerna inte kan utföras förrän programmen är så pass färdigutvecklade att de kan kompileras och exekveras. Meningen med vårt tilläggsprogram är att programmeraren skall få stöd under tiden denne programmerar.

4 Idealsystem

Ett idealsystem vore ett system som ger programmeraren både semantisk och syntaktisk hjälp vid mjukvaruutveckling. Här följer kortfattade beskrivningar på idealsystemets funktionalitet. När programmeraren skall anropa till exempel metoden *getValue()* i klassen *Sifr* (se Figur 4.1) så skall den syntaktiska hjälpen dyka upp när man (i detta fall i main-metoden) skriver ”sifrInstance.”, det vill säga när man skriver punkten som avgränsar objektet från metoden. Det skall presenteras en lista med alla tänkbara fortsättningar på koden. Det vill säga alla metodnamn som klassen känner till och som den kan anropa. Fortsätter man att skriva till exempel ”sifrInstans.ge”, så skall listan exkludera de alternativ som inte längre stämmer överens med den inmatade koden. Denna funktionalitet kallas *code completion*.

```
public class Sifr {
    private int value;

    public Sifr(int startVal) {
        value = startVal;
    }

    public int getValue() {
        return value;
    }

    public static void main(String[] args) {
        Sifr sifrInstance = new Sifr(3);
    }
}
```

Figur 4.1: Exempelklassen Sifr.

När väl den syntaktiska hjälpen har presenterats så skall även semantisk hjälp i form av för- och eftervillkor för den metod som anropas presenteras. Med hjälp av detta kan programmeraren se direkt om denne bryter mot kontraktet som är uppsatt för den anropade metoden. En stor fördel med detta är att en programmerare inte behöver sitta och kontrollera mot ett API vid sidan av, utan endast behöver koncentrera sig på programmeringen, eftersom

all relevant information kommer upp på skärmen. Systemet skall även kunna stödja kompilering och naturligtvis *syntax highlighting*. En programmerare skall kunna infoga förvillkorstest i sin kod automatiskt, för att på så sätt försäkra sig om att uppfylla sin del av kontraktet. Även verifiering av att kontraktsbeskrivningarna stämmer överens med implementationen, skall vara möjlig i systemet. Detta skulle kunna leda till att kvalitén på kontraktsbeskrivningarna ökar och att tvetydigheter i specifikationen elimineras samt att implementationsfel upptäcks på ett tidigt stadium. Det skall även finnas möjlighet att testa om programmen uppfyller kontrakten under exekvering, liknande de verktyg som beskrivs i kapitel 3. På så sätt minskar risken för att klasser och metoder används på ett felaktigt sätt.

5 Implementation

I detta kapitel behandlas lösningen och implementationen av uppgiften. Motiveringar till de olika valen på lösningen presenteras också. Kapitlet behandlar även hur man möjliggör presentation av kontrakt med hjälp av Javadoc-verktyget.

5.1 Inledning

Det finns ett stort antal utvecklingsmiljöer som fungerar klanderfritt. Det känns därför onödigt att ta fram en helt ny utvecklingsmiljö eftersom det redan finns ett flertal sådana att tillgå vilka dessutom har öppen källkod vilket möjliggör förändringar i den befintliga koden. Eftersom detta arbete är tidsbegränsat till endast 20 veckor, så är risken att en egenutvecklad utvecklingsmiljö blir betydligt sämre än redan befintliga utvecklingsmiljöer. Det som detta arbete därför borde koncentrera sig på är själva huvuduppgiften, det vill säga att se till så att resultatet blir en utvecklingsmiljö som ger en programmerare semantisk information i form av för- och eftervillkor, lämpligen baserad på en befintlig utvecklingsmiljö.

Det finns dock minimikrav som ställs på utvecklingsmiljön. Först och främst så skall den ha öppen källkod och vara tillgänglig för allmänheten. Det skall inte finnas några licensproblem som inskränker en utvecklare att använda sig av källkoden. Den skall ha stöd för *syntax highlighting* och helst *code completion*. Detta för att den skall vara användbar och effektiv att arbeta i.

5.2 Avgränsning

Förutom de avgränsningar som följer av det som nämns i inledningen så måste även andra avgränsningar göras för att arbetet skall kunna färdigställas inom de uppsatta tidsramarna. Om man utgår ifrån det beskrivna idealsystemet i kapitel 4, så är lösningen avgränsad till att endast innefatta presentation av kontrakt samt möjlighet att infoga tester på förvillkor i koden. Något som utelämnas är verifiering av att kontrakt följs under exekvering. Denna avgränsning görs på grund av att det redan finns ett flertal sådana verktyg att tillgå, vilket visades i kapitel 3. Huvuduppgiften är just att presentera semantisk information i form av kontrakt för programmeraren i skrivande stund, inte verifiering av kontrakt.

5.3 Val av implementationsmiljö

Lösningen implementeras i och för programmeringsspråket Java. Det finns en rad anledningar till varför Java har valts:

- Koden är plattformsoberoende och kan därför användas på olika plattformar utan att först modifieras.
- Språket är objektorienterat. Används objektorienteringen korrekt medför det att metoder och klasser har klara avgränsningar. Detta förenklar kontraktsbeskrivningen.
- I Java har man definierat ett sätt att skriva kommentarer i koden på ett strukturerat sätt i form av dokumentationskommentarer, eller mer allmänt kallat Javadoc.
- Man erbjuder ett verktyg att presentera denna dokumentation, Javadoc, i form av till exempel HTML-sidor.
- Dessutom finns även ett flertal kostnadsfria utvecklingsmiljöer för språket, som har öppen källkod, vilket underlättar utvecklingen betydligt.
- I övrigt så finns mycket färdig kod i form av standardbibliotek, vilket gör språket mycket enkelt att använda.

Utvecklingsmiljön, som har valts ut att implementera lösningen i och till, är NetBeans IDE. Anledningen till valet av NetBeans är dels för den öppna källkoden och dels för att redan kraftfulla funktioner som till exempel *code completion* redan finns att tillgå. Dessutom finns stöd för automatisk presentation av Javadoc via ett så kallat *popup*-fönster. Detta innebär att allt som krävs runt omkring uppgiften redan finns implementerat och lösningen blir därmed mer renodlad till den ursprungliga uppgiften, nämligen att erbjuda presentation av kontrakt i en utvecklingsmiljö. Dessutom är NetBeans' källkod skriven i Java, vilket innebär att IDE:n är plattformsoberoende.

5.3.1 Javadoc

Java stöder kommentarer i koden enligt samma konventioner som i C/C++, nämligen enkelradskommentarer genom teckenkombinationen // och flerradskommentarer genom /* */. Det som står efter enkelradskommentartecknen på samma rad kommer att ignoreras av Javakompilatorn och ej översättas till byte-kod. Det samma gäller för allt som står mellan flerradskommentartecknen också, med den skillnaden att den kan sträcka sig över flera rader.

Java har även infört så kallade dokumentationskommentarer, /** */, vilka har samma funktion som flerradskommentarerna med den skillnad att dessa kommentarer kan tolkas av Javas

verktyg *Javadoc*. Genom att dokumentera koden i form av dokumentationskommentarer, eller mer allmänt kallat Javadoc, kan man på ett enkelt sätt skilja ut dokumentationen från koden med hjälp av Javadoc-verktyget, vilket bidrar till att förbättra läsbarheten samt medför att källkoden inte behöver lämnas ut i dokumentationssyfte. Javadoc-verktyget genererar dokumentation i form av till exempel HTML-sidor som beskriver klassen/-erna och dess metoder. En dokumentationskommentar består av en ”huvud”-beskrivning följd av en så kallad taggsektion. ”Huvud”-beskrivningen börjar direkt efter starttecknen för dokumentationskommentaren, /**, och pågår tills den första taggen påträffas. En tagg specificeras med tecknet @ före namnet på taggen. Alla blanktecken och asterisker i början på en rad kommer att filtreras bort liksom nyradstecken. En dokumentationskommentar behöver inte innehålla någon huvudbeskrivning, liksom den ej heller måste innehålla en taggsektion. Alla dokumentationskommentarer placeras direkt före antingen en klass-, interface-, konstruktor-, metod- eller fältdeklaration om de skall kunna tolkas av Javadoc-verktyget. Varje deklaration kan endast ha en dokumentationskommentar, om den har flera så kommer dessa att ignoreras liksom de som inte är placerade enligt ovanstående.
Java har ett flertal fördefinierade taggar vilka finns angivna på [10]. När Javadoc-verktyget stöter på ett taggtecken, så kommer verktyget först att kontrollera om det är en standardtagg som stötts på. Om så är fallet så kommer taggen att bytas ut mot en fördefinierad rubrik i HTML-koden. Om inte taggen är en standardtagg, så kommer den att ignoreras eller alternativt används taggens namn som rubrik i HTML-koden, beroende på vilka inställningar man använder för Javadoc-verktyget. Något som är nytt för Java version 1.4, är att man kan definiera egna taggar, genom valen *–tag* och *–taglet* till Javadoc-verktyget, se [10].

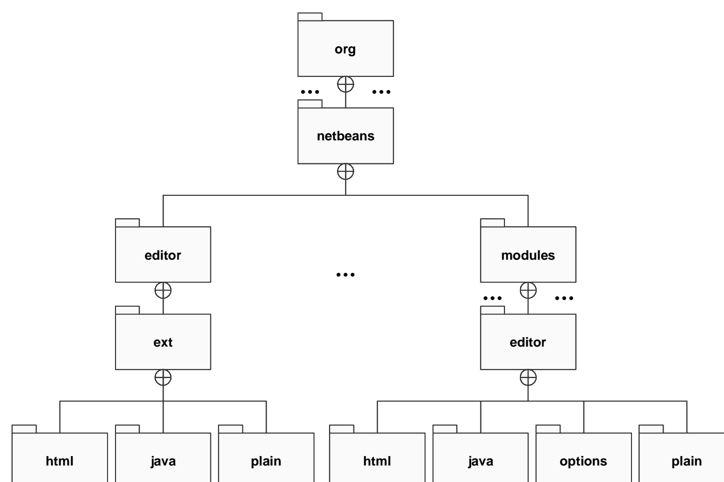
Som sammanfattning kan sägas att Javadoc-verktyget erbjuder en enkel och användbar funktionalitet genom att lyfta ut dokumentationen av koden i separata dokument, som kan vara i form av HTML-sidor. Den stora fördelen med detta verktyg är att man kan erbjuda implementationsdokumentation eller API-dokumentation redan innan koden är färdigutvecklad, eftersom inte Javadoc-verktyget tar någon hänsyn till implementationen, utan endast använder sig av dokumentationskommentarerna tillsammans med respektive deklaration.

5.4 Editor-modulen i NetBeans

Koden i NetBeans är organiserad i olika moduler för att strukturera upp koden på ett bra sätt. En av dessa moduler är editor-modulen, vilken är den modul där ändringarna införts för att tillhandahålla lösningen. Källkoden för denna modul återfinns i katalogen *editor*, i huvudkatalogen för källkoden, *netbeans-src*. Källkoden till NetBeans IDE:n kan laddas ned från NetBeans' hemsida [18].

5.4.1 Filstruktur

Katalogen *editor* innehåller ett flertal filer och underkataloger varav två är *src* och *libsrc*. Dessa två underkataloger innehåller den egentliga källkoden för editorn i NetBeans. *Libsrc*-katalogen innehåller ett bibliotek av klasser som är oberoende av IDE:n och dess implementation. *Src*-katalogen däremot innehåller IDE:ns implementation av editor-modulen, och dessa klasser använder sig bland annat av klasserna i *libsrc*. Filerna i *libsrc* ingår i paketet *org.netbeans.editor* och filerna i *src* ingår i paketet *org.netbeans.modules.editor*. Figur 5.1 visar paketstrukturen.



Figur 5.1: Paketstruktur.

5.5 Design

För att presentationen av kontrakt skall kunna möjliggöras så måste man definiera en syntax för hur ett kontrakt skall se ut. Det här arbetet definierar tre nya Javadoc-taggar som skall representera förvillkor, eftervillkor samt klassinvariant.

5.5.1 Kontraktstaggar

Vi har skapat tre nya Javadoc-taggar som representerar förvillkor, eftervillkor samt klassinvariant.

@pre: vid denna tagg så kan förvillkoret för metoden uttryckas antingen i textform som exekverbart uttryck eller på ett formellt sätt. I Figur 5.2 visas ett exempel på användningen av denna tagg.

```
/**
   * …
   * @pre stacken är inte full.
   * …
   */
public void push(Object element)
```

Figur 5.2: Exempel @pre.

@post: vid denna tagg så kan eftervillkoret för metoden uttryckas i antingen textform som exekverbart uttryck eller på ett formellt sätt. I Figur 5.3 visas ett exempel på användningen av @*post*-taggen.

```
/**
   * …
   * @post Objektet element har lagts överst på stacken och storleken
   * på stacken har ökats med ett.
   * …
  */
public void push(Object element)
```

Figur 5.3: Exempel @post.

@invariant: vid denna tagg så kan invarianten för en klass uttryckas i antingen textform som exekverbart uttryck eller på ett formellt sätt. I Figur 5.4 visas ett exempel på användningen av @*invariant*-taggen.

```
/**
   * …
   * @invariant size >= 0 && size <= MAX
   * …
   */
public class MyStack {
```

Figur 5.4: Exempel @invariant.

Anledningen till att just *@pre*, *@post* och *@invariant* har valts som kontraktstaggar, är dels att dessa finns specificerade i OCL och dels för att det redan finns befintliga verktyg som stöder dessa taggar, se kapitel 3. Detta innebär att dessa verktyg då kan användas tillsammans med den tänkta lösningen, vilket ytterligare ökar programmerarens möjligheter att utveckla robusta och pålitliga program.

5.5.2 Syntaxbeskrivning för kontrakt

Det är tillåtet att skriva kontrakten i form av ren text, exekverbar kod eller formella uttryck, eller en kombination av dessa former. Anledningen till denna valfrihet är att kontraktsskrivaren själv skall kunna välja och avgöra vilken form som passar bäst för den givna situationen. I vissa fall kan det vara bra att uttrycka kontrakten i textform för att ge tydliga och ingående förklaringar av dessa. I andra fall kanske ett kontrakt uttryckt i exekverbar kod, eller i ett formellt språk, passar bättre. Det viktiga är inte att kontrakten skrivs efter en viss syntax, utan det viktiga är att de är korrekta och otvetydiga samt överensstämmer med implementationen. En rekommendation är dock att varje villkor skrivs vid en separat tagg. Detta för att det skall vara enklare att urskilja de olika villkoren i kontrakten och på så sätt öka läsbarheten av dessa. Eftersom inga krav på syntaktisk utformning av villkoren i kontrakten ställs, så är det upp till kontraktsskrivaren att själv välja syntax för innehållet. På så sätt kan kontrakten anpassas till att eventuellt användas tillsammans med ett verktyg, till exempel något av de verktyg som nämns i kapitel 3. I Figur 5.5 visas ett exempel på ett kontrakt som blandar exekverbara och icke exekverbara uttryck.

```
/**
  * @pre !isEmpty()
  * @pre Stacken måste innehålla minst ett element.
  * @post Det översta elementet har tagits bort och storleken på stacken
  * har minskats med ett.
  */
    public Object pop()
```

Figur 5.5: Exempel syntaxbeskrivning.

Det exekverbara uttrycket *!isEmpty()* kan på ett enkelt sätt särskiljas från övrig beskrivning i kontraktet, genom att det uttrycks vid en separat tagg. Detta gör det enklare för programmeraren att uppfylla förvillkoret, genom att denne nu vet hur förvillkoret skall kunna testas och uppfyllas.

5.6 Realisering

I detta delkapitel beskrivs de olika steg som måste utföras för att kunna realisera implementationen.

5.6.1 Presentation av kontrakt i NetBeans

NetBeans IDE:n har en egen uppsättning med standardtaggar och använder sig alltså inte av Javadoc-verktygets standardtaggar när det gäller presentationen av dokumentationen i Javadoc-fönstret. Därför är man tvungen att lägga till de tidigare presenterade kontrakts-taggarna i NetBeans om man vill ha en korrekt presentation av kontrakt i IDE:n. Detta möjliggörs genom att modifiera filen *Bundle.properties*, i paketet *org.netbeans.editor*. Genom att här infoga följande rader, så kan kontraktstaggarna tolkas och ersättas med lämpliga rubriker:

javadoc-tag-@pre=Precondition:
javadoc-tag-@post=Postcondition:
javadoc-tag-@invariant=Invariant:

Detta tillägg i filen *Bundle.properties*, innebär att NetBeans kommer att ersätta taggarna *@pre*, *@post*, och *@invariant* mot rubrikerna *Precondition:*, *Postcondition:*, samt *Invariant:* respektive, när de presenteras i Javadoc-fönstret. Denna modifiering innebär att kontrakts-taggarna nu kan tolkas av NetBeans.

Eftersom NetBeans sorterar alla taggarna i bokstavsordning så betyder detta att @*post*-taggen kommer presenteras före @*pre*-taggen. Detta blir ett logiskt fel eftersom det som skall vara uppfyllt efter metoden har använts presenteras före skyldigheterna som måste uppfyllas innan man använder metoden. Därför är det nödvändigt att samla ihop och sortera om taggarna, så att de kan presenteras i en logisk ordning. Denna sortering implementeras på ett sådant sätt att kontraktstaggarna placeras överst i taggsektionen, så att dessa är de första som presenteras för en metod. Denna sortering utförs i metoden *sortContractTags()* i klassen *CompletionJavaDoc.java* i paketet *org.netbeans.editor.java*. Dessa modifieringar medför att kontrakt nu skall kunna presenteras i NetBeans IDE:n, på ett tillfredställande sätt.

5.6.2 Inställning av kontraktspresentation i NetBeans

För att kunna styra innehållet i Javadoc-fönstret gällande om enbart kontrakt eller om all dokumentation skall presenteras, så måste en sådan inställningsmöjlighet införas i IDE:n. Denna inställningsmöjlighet kan erbjudas genom att modifiera ett flertal filer i editor-modulen, nämligen *ExtSettingsDefaults.java*, *ExtSettingsNames.java* och *ExtSettingsInitilizer.java* i paketet *org.netbeans.editor.ext*. Dessa klasser definierar inställningar för editorn, och genom att lägga till parametrar för kontraktspresentationen i dessa, så möjliggörs inställningsmöjligheter för innehållet i Javadoc-fönstret. Dessa filer är som tidigare nämnts oberoende av IDE:n och därför måste även de filer som har med konfigurering av inställningar inifrån IDE:n modifieras. För att kunna ändra inställningarna inne i IDE:n så måste även filerna *Bundle.properties*, *JavaOptions.java* och *JavaOptionsBeanInfo.java*, i paketet *org.netbeans.modules.editor.options*, modifieras.

5.6.3 Infoga förvillkorstest i NetBeans

Nästa steg efter att ha infört presentation av kontrakt i NetBeans, är att möjliggöra infogande av tester på exekverbara förvillkor i koden. Detta skall fungera på ett sådant sätt att när en metod har ett exekverbart förvillkor så skall det vara möjligt att infoga ett test på detta förvillkor. Detta för att på ett enkelt sätt kunna försäkra sig om att förvillkoret uppfylls före anrop till metoden. Föregående innebär att när en @*pre*-tagg finns deklarerad i kontraktet för den anropade metoden, så skall dess uttryck kunna kontrolleras genom en *if-sats* för att försäkra sig om att förvillkoret uppfylls innan anrop sker. Figur 5.6 visar ett scenario när detta används. Infogandet av det exekverbara förvillkoret resulterar i testet *if(!isFull())*.

```
/**
   * ...
   * @pre !isFull()
   * ...
   */
  public void push(Object element) {
  ...
  }
  ...
  if(!isFull()) {
      push(object);
  }
```

Figur 5.6: Exempel infogning av förvillkorstest.

Detta infogande av förvillkorstest skall antingen kunna väljas från Javadoc-fönstret, genom en enkel knapptryckning, eller genom ett snabbkommando.

Ett flertal filer måste modifieras för att denna funktionalitet skall kunna erbjudas i NetBeans IDE:n. Själva bearbetningen av kontrakten sker som tidigare nämnts i *CompletionJavaDoc.java* och *NbCompletionJavaDoc.java*, i paketen *org.netbeans.editor.ext* respektive org.*netbeans.modules.editor.java*. Därför måste dessa filer modifieras för att denna funktionalitet skall kunna införas. Även *JDCPopupPanel.java*, *ScrollJavaDocPane.java*, *JavaDocPane.java*, i paketet *org.netbeans.editor.ext*, samt filerna Bundle.*properties* och *NbScrollJavaDocPane.java*, i paketet *org.netbeans.modules.editor.java*, måste modifieras för att kunna infoga test av förvillkor från Javadoc-fönstret. Detta test införs antingen genom ett snabbkommando eller genom en knapptryckning i Javadoc-fönstret.

5.6.4 Kompilering

För att nu skapa en IDE-version där tidigare nämnda ändringar införs och brukas, så måste en ny IDE-*build* skapas. Detta görs genom att använda verktyget Ant vilket kan laddas ned gratis från Ants hemsida[2]. Ant är Javas motsvarighet till Make i Linux och använder sig av XML-filer på samma sätt som Make använder sig av Make-filer. När man laddar ner källkoden för NetBeans IDE:n så finns dessa XML-filer inkluderade och de har namnet *build.xml*. Det enda man behöver göra är att köra kommandot *ant* så sköter verktyget resten. För att bygga en IDE, gå till mappen *nbbuild*, i huvudmappen för källkoden *netbeans-src,* och kör kommandot *ant.*

Resultatet blir en *zip*-fil med namnet *NetBeans-release[versionsnummer]-[datum].zip*, vilken är den nya IDE-*build:en*. Sedan är det bara att packa upp denna *zip*-fil och starta IDE:n.

5.6.5 Stöd för kontraktstaggarna med Javadoc-verktyget

Om man vill att kontraktstaggarna, som definierades i kapitel 5.5.1, skall stödjas även av Javadoc-verktyget, så finns här flera möjligheter beroende på vilken *JRE*-version (Java Runtime Environment) man använder sig av. *JRE 1.4* har nämligen stöd för egendefinierade taggar genom *–tag* respektive *–taglet* som parametrar till verktyget. Genom att använda *–tag* parametern så kan man här lista upp sina egendefinierade taggar samt vilka rubriker som skall ersätta dessa i de genererade HTML-sidorna. Detta sätt kräver att man specificerar de egendefinierade taggarna varje gång Javadoc-verktyget används för att få stöd för dessa taggar. I Figur 5.7 visas vilka parametrar som måste ges till Javadoc-verktyget för att detta skall kunna tolka och generera HTML-kod för kontraktstaggarna. Lite förklaring till figuren är på sin plats: *-d* parametern används för att specificera var de genererade HTML-sidorna skall läggas, *-tag* parametern består av ett uttryck i flera delar. Den första delen anger namnet på taggen det vill säga det namn som följer direkt efter @-symbolen. Nästa del vilken kommer efter det första kolonet, *:*, och anger var i Javadocen taggen får förekomma. Som kan ses i figuren så har @*pre*-taggen och @*post*-taggen bokstäverna *c* och *m* satta, vilket innebär att dessa taggar endast får förekomma i konstruktor- och metodbeskrivningar. Invariant-taggen får däremot endast förekomma i typbeskrivningar såsom klass- och interfacebeskrivningar, vilket anges med bokstaven *t*. Sista delen i tagguttrycket är vilket text som skall ersätta taggen i HTML-koden. Något som är viktigt att observera är att taggarna kommer att placeras i den ordning som de anges till Javadoc-verktyget. Det är därför viktigt att @*pre*-taggen anges före @*post*-taggen om man vill att kontraktsbeskrivningen skall ha en logisk ordning.

```
$javadoc –d html-directory –tag pre:cm:”Precondition:” –tag
post:cm:”Postcondition:” –tag:invariant:t:”Invariant:”
klassnamn.java
```

Figur 5.7: Javadoc tag.

Om man vill slippa att ange egenskaperna för egendefinierade taggar, såsom rubrik och giltig placering i dokumentationen, varje gång man använder sig av Javadoc-verktyget, kan man skapa så kallade *Taglet* för dessa taggar. Dessa filer fungerar på så sätt att de anger vad en tagg har för egenskaper, med andra ord så är det själva implementationen av taggen. I bilaga

B finns tre Taglet-filer, en för varje kontraktstagg vilket kan underlätta när Javadocen skall genereras. Figur 5.8 visar ett exempel på hur man kan använda dessa tillsammans med Javadoc-verktyget. Även här så spelar ordningen, i vilken Taglet-filerna anges, roll för hur taggarna placeras i de genererade HTML-sidorna. På samma sätt som tidigare anges var de genererade HTML-filerna skall placeras med *–d* parametern, *-tagletpath* parametern anger var de olika Taglet-filerna är lokaliserade. Denna sökväg kan innehålla flera olika vägar, vilka skiljs åt med kolon, *:*. Vid *–taglet* parametern anges namnet på Taglet-filen.

```
$javadoc -d html-directory -tagletpath path –taglet PreTaglet –
taglet PostTaglet –taglet InvariantTaglet klassnamn.java
```

Figur 5.8: Javadoc taglet.

Om man använder sig av en äldre *JRE* än 1.4, så finns varken *–tag* eller *–taglet* parametrarna att tillgå. För att här få stöd för egendefinierade taggar krävs att man antingen modifierar de befintliga filerna i *tools.jar* eller skapar en ny så kallad *Doclet*, en mall som används för att bestämma och formatera innehållet i utdatan från Javadoc-verktyget. Denna information finns att tillgå på Suns hemsida [10] liksom övrig information om Javadoc-verktyget. I bilaga B bifogas *Taglet*-filerna för kontraktstaggarna.

5.7 Filer som modifierats

I detta kapitel följer en kortfattad beskrivning av de filer som har modifieras för att kunna implementera lösningen. Dock bör nämnas att beskrivningarna av de olika filerna är våra egna tolkningar av dessa. Detta på grund av att egentlig dokumentation av dessa filer saknades. Vi hoppas dock att våra beskrivningar till större delen stämmer överens med dessa filers syfte. Även de klassdiagram som presenteras är mycket förenklade genom att kardinaliteter har utelämnats och att arvsrelationerna är ofullständiga. Anledningen till dessa förenklingar är att syftet med dessa klassdiagram inte är att ge en fullständig bild av systemet utan endast en överblick. Kompletta klassdiagram skulle bli allt för stora och oöverskådliga och därmed inte uppfylla vårt syfte, nämligen att ge en översikt över de klasser som berörs av modifieringarna.

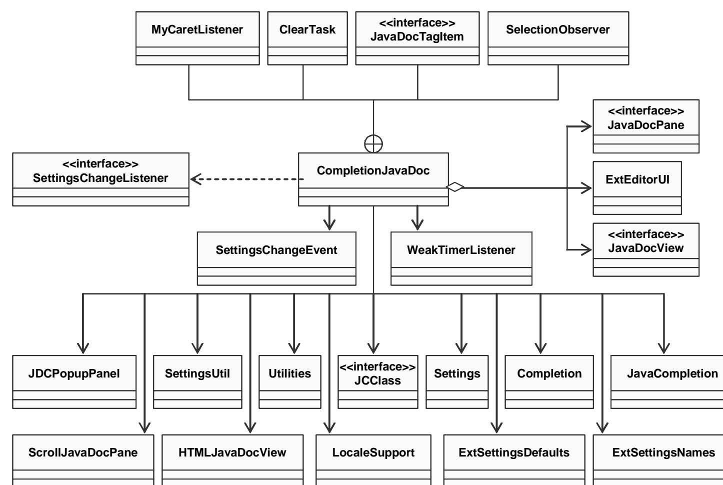
Filerna som har paketprefixet *org.netbeans.editor* återfinns i underkataloger till katalogen *libsrc* och filerna med paketprefixet *org.netbeans.modules.editor* återfinns i underkataloger till katalogen *src*, vilka båda återfinns i *editor*-katalogen i *netbeans-src*. De modifierade filerna har inkluderats i bilaga E.

5.7.1 org.netbeans.editor.Bundle.properties

I denna fil specificeras och definieras bland annat de Javadoc-taggar som NetBeans använder sig av. Denna fil har modifierats så att de tre kontraktstaggarna @*pre*, @*post* samt @*invariant* kan tolkas av NetBeans.

5.7.2 org.netbeans.editor.ext.CompletionJavaDoc.java

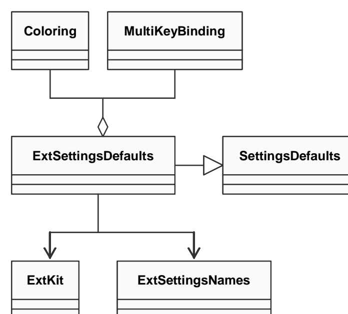
Denna klass är den klass som implementerar Javadoc-funktionaliteten. Det är här som Javadocen bearbetas och presenteras i editorn. Ändringarna som har införts här innebär att kontrakten specificerade med kontraktstaggarna kan presenteras på ett tillfredställande sätt. Figur 5.9 visar klassdiagrammet där klassen ingår.



Figur 5.9: CompletionJavaDoc.

5.7.3 org.netbeans.editor.ext.ExtSettingsDefaults.java

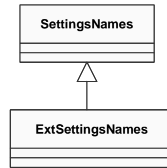
I denna klass anges *default*-inställningarna för editorn, det vill säga de inställningar som ska gälla första gången man använder sig av editorn i NetBeans. Den modifiering vi har infört i denna klass består i att valet för *default*-värdet för presentation av endast kontrakt i Javadoc-fönstret är satt till falskt, det vill säga all dokumentation presenteras i Javadoc-fönstret. Figur 5.10 visar klassdiagrammet där klassen ingår.



Figur 5.10: ExtSettingsDefaults.

5.7.4 org.netbeans.editor.ext.ExtSettingsNames.java

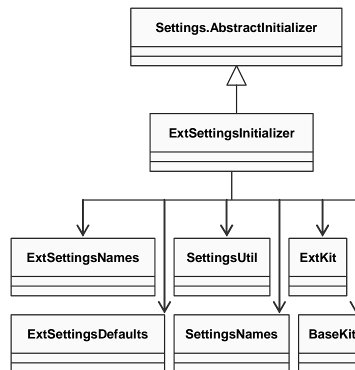
Denna klass innehåller namnen på de olika inställningarna i editorn och när dessa ska presenteras. Namnet på inställningen för kontraktspresentationen har införts i denna klass. Figur 5.11 visar klassdiagrammet där klassen ingår.



Figur 5.11: ExtSettingsNames.

5.7.5 org.netbeans.editor.ext.ExtSettingsInitializer.java

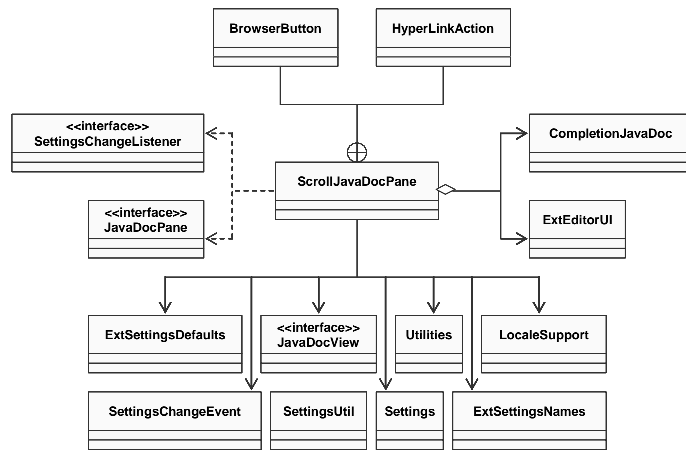
Denna klass används för att initiera inställningarna i editorn. Här har inställningen för presentation av kontrakt lagts till. Figur 5.12 visar klassdiagrammet där klassen ingår.



Figur 5.12: ExtSettingsInitializer.

5.7.6 org.netbeans.editor.ext.ScrollJavaDocPane.java

Denna klass är implementationen av ytan på vilken Javadocen presenteras i editorn. En metod som möjliggör aktivering/inaktivering av infogande av förvillkorstest har definierats i klassen. Figur 5.13 visar klassdiagrammet där klassen ingår.



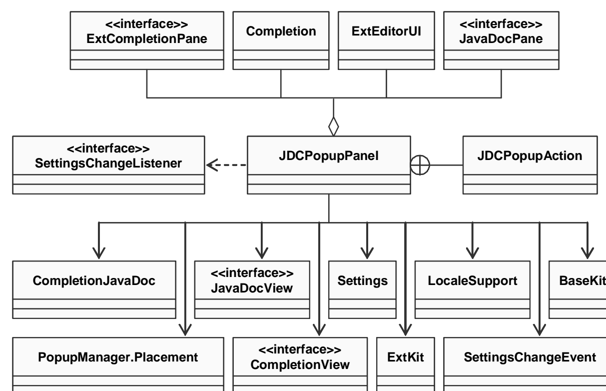
Figur 5.13: ScrollJavaDocPane.

5.7.7 org.netbeans.editor.ext.JavaDocPane.java

Detta är ett interface vilket används av *ScrollJavaDocPane* det vill säga ett interface för ytan på vilken Javadocen presenteras. En metod som möjliggör aktivering/inaktivering av infogande av förvillkorstest har deklarerats i interfacet.

5.7.8 org.netbeans.editor.ext.JDCPopupPanel.java

Denna klass representerar ett osynligt fönster vilket innehåller information om *code completion*-fönstret och Javadoc-fönstret. Modifieringarna som har införts i klassen syftar till att möjliggöra infogande av förvillkorstest genom snabbkommandot *alt+p*. Figur 5.14 visar klassdiagrammet där klassen ingår.



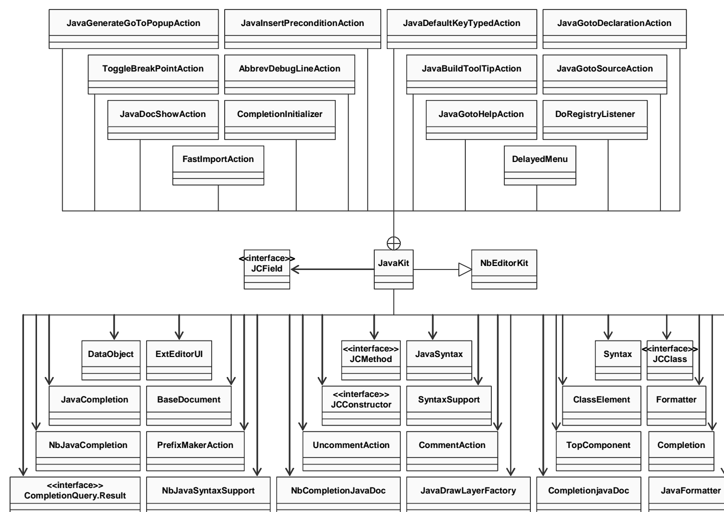
Figur 5.14: JDCPopupPanel.

5.7.9 org.netbeans.modules.editor.java.Bundle.properties

Denna fil innehåller bland annat texter som kan presenteras i IDE:n, till exempel så kallade *tool-tips* eller *hints*. Filen har modifierats så att information ges om vad som händer om knappen för infogande av förvillkorstest i Javadoc-fönstret används.

5.7.10 org.netbeans.modules.editor.java.JavaKit.java

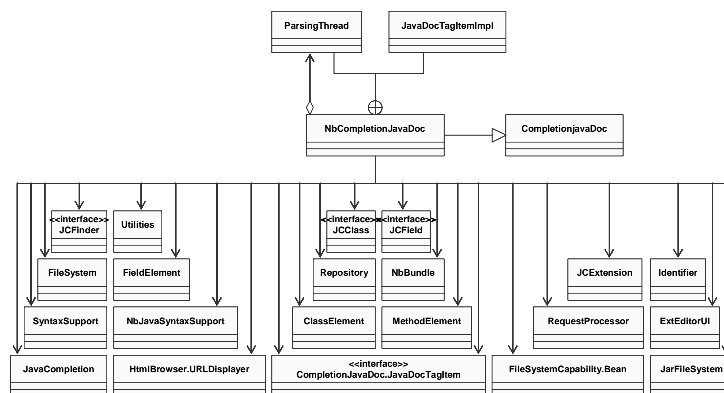
Denna klass används för att tillhandahålla egenskaper som krävs för att en textkomponent skall kunna fungera som en Java-editor. Här definieras en mängd så kallade *actions*, det vill säga operationer, för Java-editorn i NetBeans. Denna klass har ej modifieras utan anledningen till varför den beskrivs i detta kapitel är att den är den centrala klassen i editor-modulen och är därför viktig för översikten av lösningen. Figur 5.15 visar klassdiagrammet där klassen ingår.



Figur 5.15: JavaKit.

5.7.11 org.netbeans.modules.editor.java.NbCompletionJavaDoc.java

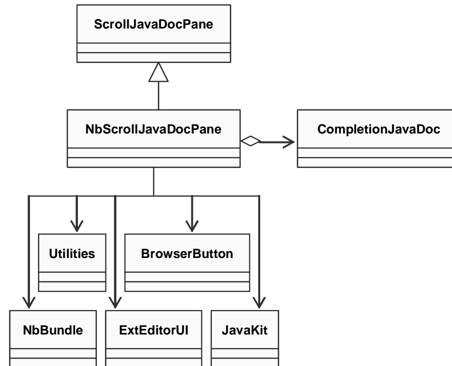
Klassen är IDE:ns implementation av Javadoc-funktionaliteten. Denna klass ärver från klassen *CompletionJavaDoc*. I denna klass så har funktionalitet som krävs för att infoga förvillkorstest i koden införts. Figur 5.16 visar klassdiagrammet där klassen ingår.



Figur 5.16: NbCompletionJavaDoc.

5.7.12 org.netbeans.modules.editor.java.NbScrollJavaDocPane.java

Klassen är IDE:ns implementation av *ScrollJavaDocPane* vilken också är dess superklass. I denna klass så har knappen för infogande av förvillkorstest lagts till så att den presenteras i verktygsfältet i Javadoc-fönstret. Figur 5.17 visar klassdiagrammet där klassen ingår.



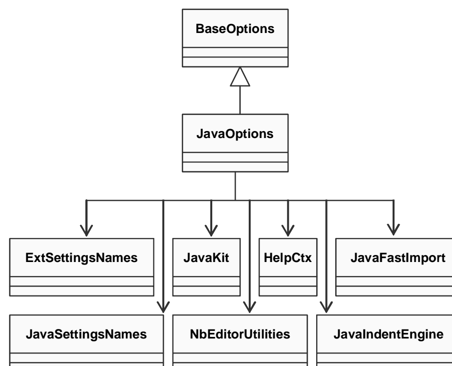
Figur 5.17: NbScrollJavaDocPane.

5.7.13 org.netbeans.modules.editor.options.Bundle.properties

Här finns inställningar och egenskaper för editorerna i IDE:n. Modifieringen av denna fil innebär att man kan styra innehållet som presenteras i Javadoc-fönstret. Antingen kan man nu välja att endast presentera kontraktstaggarna och deras specifikationer, eller så kan man presentera övrig Javadoc tillsammans med kontraktstaggarna.

5.7.14 org.netbeans.modules.editor.options.JavaOptions.java

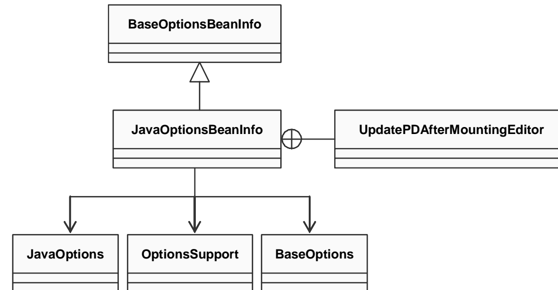
I denna klass definieras en mängd val som gäller för Java-editorn i IDE:n. Modifieringen av denna klass innebär att man får möjlighet att styra innehållet i Javadoc-fönstret, se kapitel 5.7.13, genom att detta val nu finns att tillgå som en inställning i editorn. Figur 5.18 visar klassdiagrammet där klassen ingår.



Figur 5.18: JavaOptions.

5.7.15 org.netbeans.modules.editor.options.JavaOptionsBeanInfo.java

Denna klass används för att sätta valen i NetBeans IDE:n för Java-editorn. Modifieringen av denna klass möjliggör att värdet för vad som ska presenteras i Javadoc-fönstret kan ändras. Figur 5.19 visar klassdiagrammet där klassen ingår.



Figur 5.19: JavaOptionsBeanInfo.

5.8 Sammanfattning

Detta kapitel har visat hur man inför ökat stöd till kontraktsprogrammering i NetBeans IDE genom att tillhandahålla presentation av kontrakt och infogande av test på exekverbara förvillkor. I kapitlet har tre nya taggar definierats och deras användningsområde har specificerats för att kunna skriva kontrakt på ett strukturerat sätt i form av Javadoc. Dessa tre taggar är *@pre*, *@post*, samt *@invariant*. Dessa taggar representerar förvillkor, eftervillkor och klassinvarianter respektive. Dessa taggar har sedan införts i NetBeans IDE:n. Även några lösningar på hur man får Javadoc-verktyget att tolka de egendefinierade kontraktstaggarna, och därmed kunna generera kontraktsbeskrivningar i HTML-format, har presenterats. Till sist har även en kortfattad beskrivning av syfte och ändringar av de filer som modifierats presenterats.

6 Arbetets gång

Arbetet startade med en analys av det tidigare arbete som hade gjorts på universitetet, se kapitel 2.5. Det arbetet hade använt sig av editorn Jipe [13]. Jipe studerades översiktligt och det framgick att Jipe var baserat på en annan editor, jEdit [12].

Koncentrationen lades nu på jEdit. Detta på grund av att det verkade som att utvecklingen av Jipe hade avstannat plus att jEdit verkade ha ett mycket bra stöd för programmering av tilläggsprogram. jEdit hade inga inskränkande licensavtal och källkoden var öppen. jEdit hade stöd för *syntax highlighting*, det var positivt. Användandet av jEdit skulle dock kräva utveckling av *code completion*. En sådan lösning krävde att man skulle parsa källkoden för att se vilka syntaktiska möjligheter som fanns. Detta var tänkt att lösas genom att använda verktyget JavaCC [9] för att generera en parser. Tilläggsprogrammet skulle tillhandahålla den funktionalitet som projektet krävde. Tilläggsprogrammet skulle kunna parsa koden som skrevs i jEdit och kunna hantera *code completion* och till slut även presentation av för- och eftervillkor som skrevs i Javadoc-kommentarerna.

jEdit visade sig dock vara svårt att ha som bas för att implementera all funktionalitet som beskrivs i föregående stycke. Något tilläggsprogram blev aldrig realiserat på grund av att den hjälp som jEdit gav för programmering av tilläggsprogram inte var tillräcklig. Beslutet att avbryta arbetet med jEdit fattades och sökandet efter en annan editor med öppen källkod påbörjades. Tyvärr hade redan flera veckors arbete lagts ner på att granska jEdit.

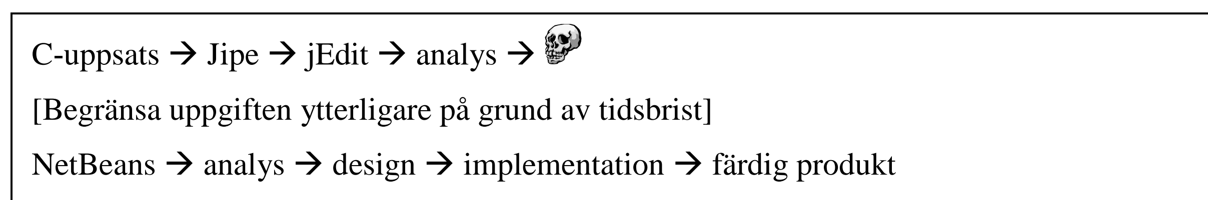
En avgränsning var nu nödvändig att göras för att det skulle vara möjligt att slutföra projektet i tid. Sökandet efter en editor som erbjöd ännu mer funktionalitet som skulle vara till nytta för projektet påbörjades.

En sökning på Internet gav resultat. NetBeans IDE [18] var en utvecklingsmiljö som hade öppen källkod. Denna IDE erbjöd redan från början kraftfulla funktioner som var tätt sammankopplade med uppgiften. Detta omfattade förutom *syntax highlighting* bland annat *code completion*, vilken erbjuder en syntaktisk hjälp, och en integrerad Javadoc funktionalitet. Denna funktionalitet erbjuder en semantisk hjälp i form av presentation av

dokumentationskommentarerna eller mer allmänt kallat Javadoc. Dessutom var det en stor fördel att Sun Microsystems, skaparna av programmeringsspråket Java, var projektsponsorer till IDE:n eftersom den då alltid skulle anpassas efter eventuella förbättringar i språket. Detta innebar att NetBeans borde vara en av de bästa Java IDE:rna som fanns att tillgå. En annan fördel var att koden var under Sun Public License (se bilaga F) vilket innebar att man kunde få tillgång till koden, modifiera den samt använda den i kommersiella syften om man så önskade. IDE:n var även kostnadsfri. Den nackdel som kunde ses med NetBeans var att denna IDE var väldigt resurskrävande, vilket hade att göra med att den var helt skriven i Java. Detta medförde att IDE:n inte var lika effektiv prestandamässigt, som den skulle kunna vara om dess kod vore anpassad för en specifik plattform. Detta var dock ett mindre problem som kunde lösas rent hårdvarumässigt genom snabbare hårdvara. Fördelen med att NetBeans var skriven i Java, var att IDE:n blev plattformsoberoende och därmed kunde köras på ett flertal plattformar utan att koden måste modifieras. Eftersom mycket användbara hjälpmedel som underlättar programutvecklingen redan fanns implementerade i NetBeans, så föll valet på denna IDE.

En analys av NetBeans' källkod genomfördes. Analysen visade sig vara ganska krävande och tog mycket tid i anspråk. Ett designarbete påbörjades efter analysen av NetBeans' källkod där det bestämdes i stort hur själva implementationen skulle gå till väga. Implementationsfasen påbörjades då den utökade funktionaliteten skulle läggas in i NetBeans IDE.

Figur 6.1 visar arbetets gång mer kortfattat och abstrakt.



Figur 6.1: Arbetets gång - översikt.

7 Resultat

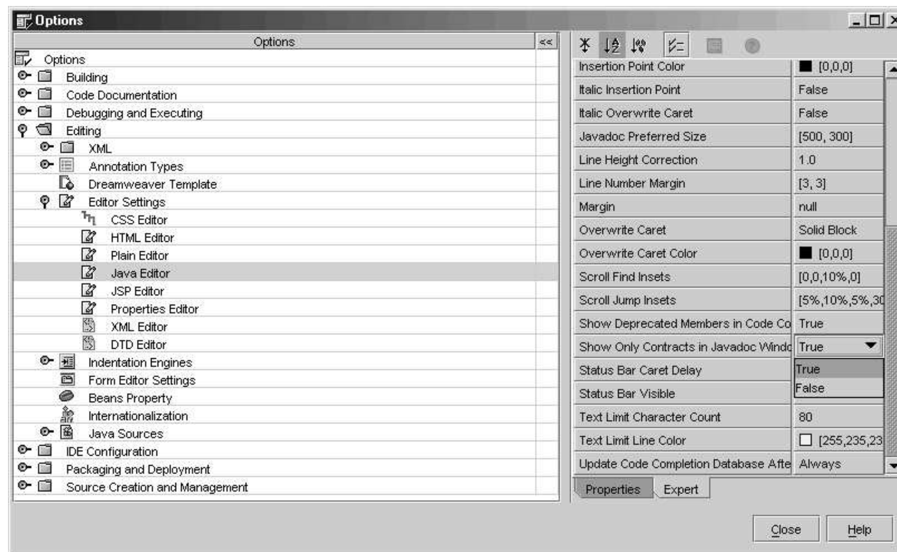
I detta kapitel demonstreras den implementerade lösningen i NetBeans IDE:n. För att demonstrera kontraktspresentationen används de tre klasserna *StackInterface.java*, *StackImpl.java* samt *StackApplication.java* vilka återfinns i bilaga C. Demonstrationen inkluderar bara de inställningar som har med själva lösningen att göra. Övrigt användande av IDE:n lämnas åt läsaren att själv utforska.

Sist i kapitlet beskrivs även hur IDE:n ytterligare kan anpassas för att bättre stödja programmeraren vid kontraktsprogrammering.

Se bilaga D för de systemkrav som ställs för att NetBeans IDE skall fungera.

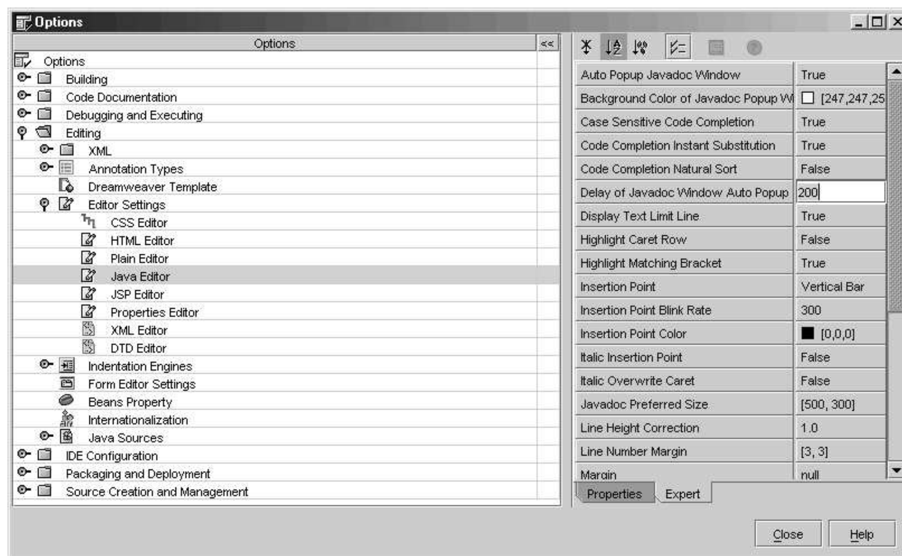
7.1 Demonstration

Den första inställning som måste göras är vad som skall presenteras i Javadoc-fönstret. Vill man att endast kontrakt skall presenteras eller skall även resterande dokumentation presenteras? Default-inställningen är att all dokumentation skall presenteras (med undantag för vissa taggar). Denna inställning görs genom att gå in i menyn *Tools->Options*, under *Editing* välj *Java Editor* under *Editor Settings*, gå in på fliken *Expert* och ändra till önskat värde för fältet *Show Only Contracts in Javadoc Window*, se Figur 7.1. Värdet *true* innebär att endast kontraktstaggarna presenteras, medan värdet *false* innebär att all Javadoc presenteras.



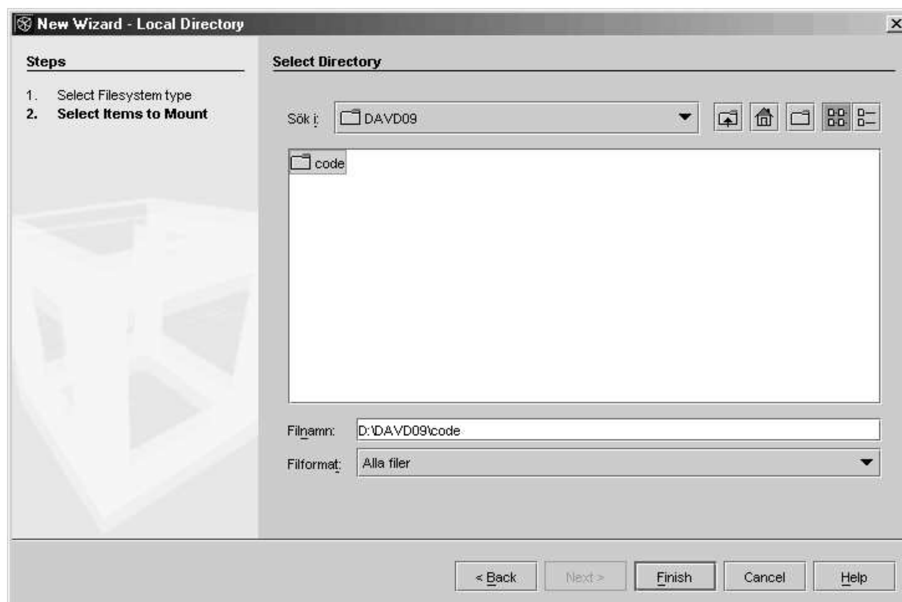
Figur 7.1: Inställning om endast kontrakt skall presenteras.

Nästa steg är att ställa in vilken fördröjning man vill ha för *code completion*-fönstret och Javadoc-fönstret innan dessa aktiveras. Dessa inställningar görs genom att gå in i menyn *Tools->Options*, välj *Java Editor* under *Editor Settings* i huvudkatalogen *Editing*. Sätt önskad fördröjning för *code completion*-fönstret genom att klicka på fliken *Properties* och ändra värdet för *Delay of Completion Window Auto Popup*. Javadoc-fönstrets fördröjning sätts under *Expert*-fliken vid värdet för *Delay of Javadoc Window Auto Popup*, se Figur 7.2. Anledningen till dessa värden är att man skall kunna kontrollera om de olika fönstren skall visas eller ej beroende på hur snabbt man skriver. Om man skriver snabbare än fördröjningstiderna kommer fönstren ej att aktiveras och om man skriver långsammare än dessa tiden så aktiveras fönstren. Detta är användbart när man inte vill ha någon hjälp från fönstren utan vet vad som gäller. Då är det bara att skriva på och man slipper därför se fönstret. Om man däremot är osäker på vad som gäller så kan man vänta tills fönstren aktiveras för att få den nödvändiga hjälp man behöver. En fördröjning satt till 0 kommer alltså att innebära att fönstret aktiveras omedelbart.



Figur 7.2: Javadoc popup-fönstrets fördröjning.

För att få upp dokumentationen för en klass så måste klassen vara *mountad*, det vill säga man måste inkludera filsystemet innan den känns igen av IDE:n. Detta görs genom att välja *File->Mount Filesystem* och mappen som skall *mountas*, observera att man väljer mappen i vilken paketstrukturen ligger, se Figur 7.3.



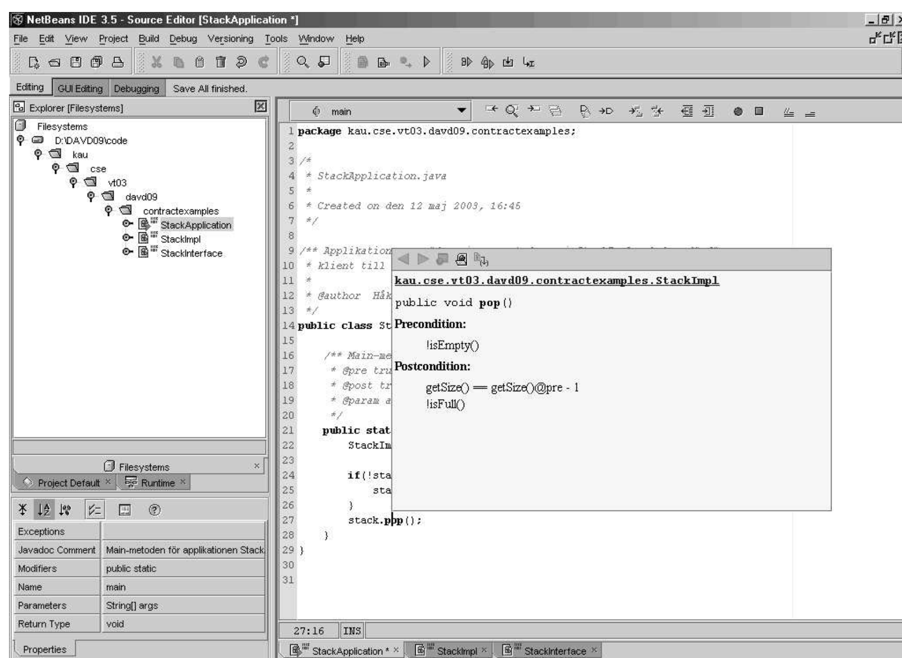
Figur 7.3: Mountning av filsystem.

Man kan få upp Javadoc-fönstret på flera olika sätt:

- Genom att skriva nyckelordet *new* och vänta ut fördröjningen. Detta alternativ leder till att *code completion*-fönstret aktiveras (förutsatt att funktionen inte är inaktiverad i inställningarna) och efter en stund aktiveras även Javadoc-fönstret. Javadoc-fönstret presenterar dokumentationen för den markerade datatypen i *code completion*-fönstret.

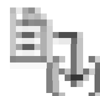
- Genom att skriva en punkt efter objektsreferensen och vänta ut fördröjningen. Detta kommer att leda till samma resultat som i den ovanstående punkten.
- Markera eller placera markören över den datatyp man vill se Javadocen för och sedan trycka *ctrl+space*. Detta leder till att *code completion*-fönstret omedelbart aktiveras varav sedan även Javadoc-fönstret aktiveras.
- Markera eller placera markören över den datatyp man vill se Javadocen för och sedan trycka *ctrl+shift+space*. Detta leder till att Javadoc-fönstret omedelbart aktiveras utan att *code completion*-fönstret först aktiveras.

I Figur 7.4 visas ett aktiverat Javadoc-fönster vilket har aktiverats genom snabbkommandot *ctrl+shift+space*. Fönstret uppvisar kontraktet för metoden *pop()* i klassen *StackImpl*.



Figur 7.4: Javadoc-fönster.

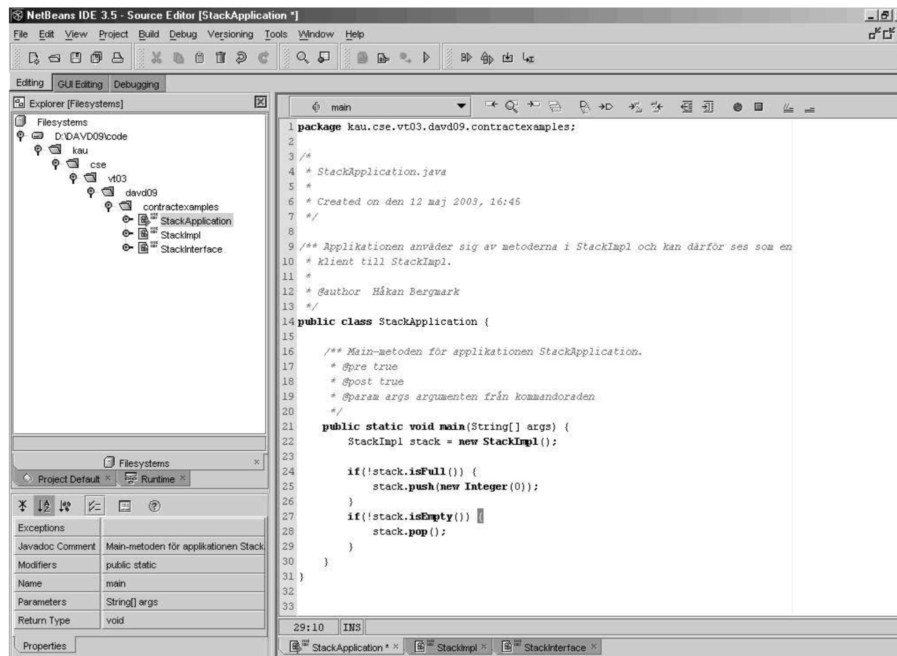
När ett Javadoc-fönster har aktiverats, så kan förvillkorstest infogas, förutsatt att sådana existerar. Detta kan göras antingen genom att klicka på knappen längst ut till höger i Javadoc-fönstrets knappmeny (se Figur 7.5) eller genom snabbkommandot *alt+p*. Båda resulterar i att en *if-sats,* innehållande förvillkoret, infogas i koden.



Figur 7.5: Knapp för infogande av förvillkorstest.

I Figur 7.6 ses att i kontraktet för *pop()* anges det exekverbara uttrycket *!isEmpty()* som förvillkor till metoden. I Figur 7.6 visas att detta förvillkor placerats i en *if-sats* genom att

snabbkommandot *alt+p* har använts. Den aktuella objektreferensen, *stack*, har infogats automatiskt framför metodanropet.



Figur 7.6: Förvillkorstest.

7.2 Utvärdering

Resultatet är en utvecklingsmiljö som erbjuder presentation av kontrakt vid programmering. Utvecklingsmiljön erbjuder även en möjlighet för en programmerare att genom en knapptryckning eller ett snabbkommando infoga test på förvillkor till en anropad metod. Om man jämför resultatet med det beskrivna idealsystemet (se kapitel 4) så stämmer resultatet överens med idealsystemet på ett flertal punkter. Idealsystemet skulle ha stöd för *syntax highlighting*, *code completion* och kompileringsmöjlighet. Detta hade den utvecklingsmiljö resultatet bygger på stöd för redan från början. Idealsystemet skulle ha stöd för presentation av kontrakt vid programmering. Detta stöd har införts i utvecklingsmiljön. Tillsammans med detta så har även möjligheten till infogande av förvillkorstest införts. Detta var också specificerat att ingå i ett idealsystem. I idealsystemet skulle även möjligheten till att verifiera att uppsatta kontrakt stämmer överens med implementationen finnas. Detta stöd finns inte i utvecklingsmiljön. Det finns inte heller stöd för att kunna kontrollera att programmen följer uppsatta kontrakt under exekvering, vilket ett idealsystem skulle kunna. Dock finns möjligheten att utforma kontrakten på ett sådant sätt att de följer den syntax som något av de testade verktygen i kapitel 3 använder. Då kan man använda sig av ett sådant verktyg

tillsammans med utvecklingsmiljön och på så sätt verifiera att kontrakt efterlevs under exekvering av program.

Enligt de mål som var uppsatta skulle ett tilläggsprogram till en utvecklingsmiljö skapas. Detta tilläggsprogram skulle vara så oberoende av utvecklingsmiljön som möjligt. Detta för att öppna möjligheten till att använda tilläggsprogrammet tillsammans med andra utvecklingsmiljöer utan att behöva ändra allt för mycket i koden (se kapitel 2.2). Tilläggsprogrammet skulle stödja presentation av kontrakt vid programmering. Resultatet blev inte ett tilläggsprogram till en utvecklingsmiljö även om detta var det som författarna strävade mot, se kapitel 6. Resultatet blev dock en fungerande utvecklingsmiljö som erbjuder presentation av kontrakt vid programmering. Utvecklingsmiljön är plattformsoberoende, en liten kompensation för att det inte blev ett tilläggsprogram med så svaga band till utvecklingsmiljön som möjligt.

7.3 Framtida arbete

Presentationen av kontrakt anses fungera väl och det finns därför ingen anledning till att förändra denna något ytterligare. Däremot så finns vissa brister när det gäller infogandet av förvillkorstest. Ingen skillnad görs här på exekverbara och icke exekverbara uttryck, vilket innebär att allt som står angivet som förvillkor kommer att infogas i testet. Detta betyder att programmeraren själv måste ta bort de icke exekverbara uttrycken ur testet, för att testet skall vara syntaktiskt korrekt. En annan begränsning är att markören måste befinna sig till vänster om öppningsparantesen i metodanropet när Javadoc-fönstret aktiveras. Om markören befinner sig till höger om denna parantes så kommer inte den korrekta objektreferensen att kunna urskiljas och infogas i testet. Ytterligare en begränsning är att objektreferensen för konstruktor-anrop inte urskiljs och därmed måste programmeraren själv ange denna i testet. Ett förslag till hur detta automatiska infogande av förvillkorstest, skulle kunna förbättras är att möjliggöra automatisk identifiering av exekverbara uttryck. Om en sådan funktionalitet finns tillgänglig innebär det att man kan förbättra det automatiska infogandet av förvillkorstestet, till att endast vara möjlig för exekverbara uttryck. Även tillhandahållandet av en funktionalitet som verifierar kontraktsbeskrivningarna mot implementationen kan ses som ett framtida arbete liksom möjligheten att kontrollera om program bryter mot kontrakt under exekvering.

8 Sammanfattning

I detta arbete har en lösning gällande presentation av kontrakt vid programmering tagits fram och presenterats till NetBeans IDE:n. Även andra verktyg som förenklar kontrakts-programmering har presenterats och en beskrivning av kontrakt och deras betydelse för programvaruutvecklingen har lagts fram. Ett idealsystem har beskrivits, detta skulle fungera som något att sträva mot och jämföra med i slutet. Även möjligheten till test av förvillkor har implementerats. En programmerare har möjlighet att infoga test av förvillkoret för den metod denne anropar. Denna funktionalitet med infogande av förvillkorstest är dock inte fullt utvecklad. Den skall därför ses mer som en hjälp än som ett komplett stöd vid programmeringen. Det är tänkt att programmeraren skall kunna infoga testet på ett enkelt sätt och endast behöva göra mindre modifieringar av testet. Presentationen av kontrakt fungerar tillfredställande. Hela nyttan med presentationen av kontrakt bygger på att kontrakten är korrekta. Detta arbete tar ingen hänsyn till om kontrakten är korrekta eller ej, utan det är upp till kontraktsskrivaren att tillhandahålla väldefinierade och korrekta kontrakt.

Referenser

[1] A. V. Aho, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[2] Apache Ant
http://ant.apache.org/
(2003-05-15)

[3] D. Bartetzko, C. Fischer, M. Möller, H. Wehrheim. *Jass – Java with Assertions*. Från *Workshop on Runtime Verification, 2001 held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01*, Paris, Frankrike, juli 2001.

[4] M. Blom. *Semantic Aspects in Software Development*. Licentiatavhandling Karlstads universitet, Karlstad 2002.

[5] A. Diller. *Z: An Introduction to Formal Methods, Second edition.* Wiley & Sons Ltd., 1995.

[6] IBM – OCL
http://www-3.ibm.com/software/awdtools/library/standards/ocl.html
(2003-05-20)

[7] iContract
http://www.reliable-systems.com/tools/iContract/iContract.htm
(2003-05-20)

[8] Jass
http://csd.informatik.uni-oldenburg.de/~jass/index.html
(2003-05-20)

[9] JavaCC
http://www.experimentalstuff.com/Technologies/JavaCC/
(2003-05-21)

[10] Javadoc Tool
http://java.sun.com/j2se/1.4.1/docs/tooldocs/javadoc/index.html
(2003-05-15)

[11] jContractor
http://jcontractor.sourceforge.net/
(2003-05-20)

[12] jEdit
http://www.jedit.org/
(2003-05-20)

[13] Jipe
http://jipe.sourceforge.net/
(2003-05-20)

[14] M. Karaorman, U. Hölzle, J. Bruno. *jContractor: A Reflective Java Library to Support Design By Contract*. Från *In Preceedings of Meta-Level Architectures and Reflection, 2nd International Conference, Reflection '99*, Saint-Malo, Frankrike, juli 1999.

[15] L. Nguyen, E. Tångring. *Javaeditor som stöder för- och eftervillkor*. C-uppsats Karlstads universitet, Karlstad 2000.

[16] R. Kramer. i*Contract the Java Design by Contract™ Tool*. Från *Proceedings of the TOOLS'98 Conference*, Santa Barbara, USA, 1998.

[17] B. Meyer. *Object-oriented Software Construction*. Precentice Hall, 1988.

[18] NetBeans
http://www.netbeans.org/
(2003-05-15)

[19] Parasoft – Jcontract
http://www.parasoft.com/
(2003-05-20)

[20] Programming With Assertions
http://java.sun.com/j2se/1.4.1/docs/guide/lang/assert.html
(2003-05-20)

[21] R. W. Sebasta. *Concepts of Programming Languages*. Addison Wesley, 1999.

A Klasser vid provkörning av verktyg

A.1 MyStack – iContract

```
/**
 * @invariant getSize() >= 0 && getSize() <= MAX
 */
public class MyStack {
    private java.util.Vector myStack;
    private final int MAX = 10;
    private int size;

    /**
     * @pre true
     * @post isEmpty()
     */
    public MyStack() {
        this.myStack = new java.util.Vector();
        this.size = 0;
    }

    /**
     * @pre !isEmpty()
     * @post getSize() == getSize()@pre - 1
     * @post !isFull()
     */
    public void pop() {
        this.myStack.remove(this.getSize() - 1);
        this.size--;
    }

    /**
     * @pre !isFull()
     * @pre e != null
     * @post getSize () == getSize()@pre + 1
     * @post top() == e
     * @post !isEmpty()
     */
    public void push(Object e) {
        this.myStack.add(e);
        this.size++;
    }

    /**
     * @pre !isEmpty()
     * @post return != null
```

```
     */
    public Object top() {
        return this.myStack.get(this.getSize() - 1);
    }

    /**
     * @pre true
     * @post return == size
     */
    public int getSize() {
        return this.size;
    }

    /**
     * @pre true
     * @post return == (getSize() == 0)
     */
    public boolean isEmpty() {
        return this.getSize() == 0;
    }

    /**
     * @pre true
     * @post return == (getSize() == MAX)
     */
    public boolean isFull() {
        return this.getSize() == this.MAX;
    }

    /**
     * @pre true
     * @post isEmpty()
     */
    public void makeEmpty() {
        this.myStack.clear();
        this.size = 0;
    }

    /**
     * @pre true
     * @post true
     */
    public static void main(String[] args) {
        MyStack stack = new MyStack();

        if(!stack.isFull()) {
            stack.push(new Integer(0));
        }
        if(!stack.isEmpty()) {
            Object o = stack.top();
            stack.pop();
```

```
        }
        for(int i = 0; i < 11; i++) {
            stack.push(new Integer(i));
        }
        stack.makeEmpty();
    }
}
```

A.2 MyStack – Automatgenererad av iContract

```
/**
 * @invariant getSize() >= 0 && getSize() <= MAX
 */
public class MyStack {
/*|*/ //#*#---------------------------------------------------------------------------
/*|*/ // Keeps track of calling chain to avoid recursive invariant checks.
/*|*/ // Avoids inv checks in public methods that are called from private ones.
/*|*/ // Stores bookkeeping information -- key: thread, value: call level
/*|*/ protected transient java.util.Hashtable __icl_ = new java.util.Hashtable(1);
/*|*/ // Update bookkeeping of method nesting and check the invariant if appropriate
/*|*/ private synchronized void __inv_check_at_entry__MyStack(Thread thread, String loc)
throws RuntimeException {
/*|*/   // perform lazy initialization after de-serialization (transient __icl_)
/*|*/   if (__icl_ == null) __icl_ = new java.util.Hashtable(1);
/*|*/   if ( !__icl_.containsKey(thread) ) {
/*|*/     __icl_.put(thread, new Integer(1));
/*|*/     __check_invariant____MyStack(loc);
/*|*/   }
/*|*/   else
/*|*/     __icl_.put(thread, new Integer(((Integer)__icl_.get(thread)).intValue()+1));
/*|*/ }
/*|*/ // Update bookkeeping of method nesting and check the invariant if appropriate
/*|*/ private synchronized void __inv_check_at_exit__MyStack(Thread thread, String loc) throws
RuntimeException {
/*|*/   // perform lazy initialization after de-serialization (transient __icl_)
/*|*/   if (__icl_ == null) __icl_ = new java.util.Hashtable(1);
/*|*/   if (((Integer)__icl_.get(thread)).intValue() == 1 ) {
/*|*/     try {
/*|*/       __check_invariant____MyStack(loc);
/*|*/     } finally {
/*|*/       __icl_.remove(thread); // remove from bookkeeping, before checking (resoliant wrt
exceptions)
/*|*/   }}
/*|*/   else
/*|*/     __icl_.put(thread, new Integer(((Integer)__icl_.get(thread)).intValue()-1));
/*|*/ }
/*|*/ // Update bookkeeping of method nesting DO NOT check the invariant (i.e. for non-default
constr.)
/*|*/ private synchronized void __inc_icl_at_entry__MyStack(Thread thread)  {
/*|*/   // perform lazy initialization after de-serialization (transient __icl_)
/*|*/   if (__icl_ == null) __icl_ = new java.util.Hashtable(1);
```

```
/*|*/   if ( !__icl_.containsKey(thread) ) {
/*|*/     __icl_.put(thread, new Integer(1));
/*|*/   }
/*|*/   else
/*|*/     __icl_.put(thread, new Integer(((Integer)__icl_.get(thread)).intValue()+1));
/*|*/ }
/*|*/ // Tests the invariants of the class and its superclasses.
/*|*/ // This method is public (see note below) to give subclasses (potentially in different
packages),
/*|*/ // acccess to the inv of superclasses (and to let the reflection API find the.
/*|*/ // method)
/*|*/ //
/*|*/ public synchronized void __check_invariant____MyStack( String location ) throws
RuntimeException {
/*|*/ try {
/*|*/ if (!(getSize() >= 0 && getSize() <= MAX))
/*|*/   throw new RuntimeException ( location + "error: invariant violated (MyStack): "+
/*|*/   "getSize() >= 0 && getSize() <= MAX");}
/*|*/ catch ( RuntimeException ex ) {
/*|*/   String msg = "";
/*|*/   if (ex.getClass()==RuntimeException.class) { msg = ex.toString(); }
/*|*/   else msg = location + " exception <<"+ex+">> occured while evaluating the class
INVARIANT.";
/*|*/   throw new RuntimeException(msg);}
/*|*/
/*|*/ }
/*|*/ //-------------------------------------------------------------------------------#*#
/*|*/
    private java.util.Vector myStack;
    private final int MAX = 10;
    private int size;

    /**
     * @pre true
     * @post isEmpty()
     */
    public MyStack() {
        this.myStack = new java.util.Vector();

/*|*/   //#*#---------------------------------------------------------------------------
/*|*/   __inc_icl_at_entry__MyStack(Thread.currentThread());
/*|*/   try {
/*|*/   //---------------------------------------------------------------------------#*#
/*|*/
/*|*/   //#*#---------------------------------------------------------------------------
/*|*/   try {
/*|*/   boolean __pre_passed = false; // true if at least one pre-cond conj. passed.
/*|*/   // checking MyStack::MyStack()
/*|*/   if (! __pre_passed ) {
/*|*/   if ( (true /* do not check prepassed */ ))   __pre_passed = true; // succeeded in:
MyStack::MyStack()
```

```
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::MyStack()
/*|*/   }
/*|*/   if (!__pre_passed) {
/*|*/     throw new RuntimeException ("src/MyStack.java:13: error: precondition violated
(MyStack::MyStack()): (/*declared in MyStack::MyStack()*/ (true)) "
/*|*/   ); }}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:13:  exception <<"+ex+">> occured while evaluating PRE-
condition in src/MyStack.java:13:  MyStack::MyStack()): (/*declared in MyStack::MyStack()*/
(true)) "
/*|*/   ;
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/   this.size = 0;

/*|*/   //#*#-------------------------------------------------------------------------------
/*|*/   try {
/*|*/   if (!(isEmpty()))
/*|*/     throw new RuntimeException ("src/MyStack.java:13: error: postcondition violated
(MyStack::MyStack()): "+
/*|*/     "isEmpty()");}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:13:  exception <<"+ex+">> occured while evaluating
POST-condition in src/MyStack.java:13:  MyStack::MyStack()";
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-------------------------------------------------------------------------
------
/*|*/   } finally {
/*|*/   __inv_check_at_exit__MyStack(Thread.currentThread(),"src/MyStack.java:13:  just before
exit MyStack::MyStack() ");}
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/
}

    /**
     * @pre !isEmpty()
     * @post getSize() == getSize()@pre - 1
     * @post !isFull()
     */
    public void pop() {
/*|*/   //#*#-------------------------------------------------------------------------------
/*|*/   this.__inv_check_at_entry__MyStack(Thread.currentThread(),"src/MyStack.java:23:  just
after entry MyStack::pop() ");
```

```
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#----------------------------------------------------------------------
------
/*|*/   int getSize___pre = getSize(); // save getSize()@pre (in MyStack::pop())
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#----------------------------------------------------------------------
------
/*|*/   try {
/*|*/   boolean __pre_passed = false; // true if at least one pre-cond conj. passed.
/*|*/   // checking MyStack::pop()
/*|*/   if (! __pre_passed ) {
/*|*/   if ( (!isEmpty() /* do not check prepassed */ ))   __pre_passed = true; // succeeded
in: MyStack::pop()
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::pop()
/*|*/   }
/*|*/   if (!__pre_passed) {
/*|*/     throw new RuntimeException ("src/MyStack.java:23: error: precondition violated
(MyStack::pop()): (/*declared in MyStack::pop()*/ (!isEmpty())) "
/*|*/   ); }}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:23:  exception <<"+ex+">> occured while evaluating PRE-
condition in src/MyStack.java:23:  MyStack::pop()): (/*declared in MyStack::pop()*/
(!isEmpty())) "
/*|*/   ;
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#----------------------------------------------------------------------
------
/*|*/   try {
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/
        this.myStack.remove(this.getSize() - 1);
        this.size--;

/*|*/   //#*#----------------------------------------------------------------------------
/*|*/   try {
/*|*/   if (!(getSize() == getSize___pre - 1))
/*|*/     throw new RuntimeException ("src/MyStack.java:23: error: postcondition violated
(MyStack::pop()): "+
/*|*/     "getSize() == getSize()@pre - 1");
/*|*/   if (!(!isFull()))
/*|*/     throw new RuntimeException ("src/MyStack.java:23: error: postcondition violated
(MyStack::pop()): "+
/*|*/     "!isFull()");}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
```

```
/*|*/     else txt = "src/MyStack.java:23:  exception <<"+ex+">> occured while evaluating
POST-condition in src/MyStack.java:23:  MyStack::pop()";
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //---------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#------------------------------------------------------------------------
------
/*|*/   } finally {
/*|*/     this.__inv_check_at_exit__MyStack(Thread.currentThread(),"src/MyStack.java:23:  just
before exit MyStack::pop() ");
/*|*/   }
/*|*/   //---------------------------------------------------------------------------#*#
/*|*/
}

    /**
     * @pre !isFull()
     * @pre e != null
     * @post getSize () == getSize()@pre + 1
     * @post top() == e
     * @post !isEmpty()
     */
    public void push(Object e) {
/*|*/   //#*#------------------------------------------------------------------------
/*|*/   this.__inv_check_at_entry__MyStack(Thread.currentThread(),"src/MyStack.java:35:  just
after entry MyStack::push(java.lang.Object) ");
/*|*/   //---------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#------------------------------------------------------------------------
------
/*|*/   int getSize___pre = getSize(); // save getSize()@pre (in
MyStack::push(java.lang.Object))
/*|*/   //---------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#------------------------------------------------------------------------
------
/*|*/   try {
/*|*/   boolean __pre_passed = false; // true if at least one pre-cond conj. passed.
/*|*/   // checking MyStack::push(java.lang.Object)
/*|*/   if (! __pre_passed ) {
/*|*/   if ( (!isFull() /* do not check prepassed */ ))   __pre_passed = true; // succeeded
in: MyStack::push(java.lang.Object)
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::push(java.lang.Object)
/*|*/
/*|*/   if ( (__pre_passed && (e != null /*...*/)))   __pre_passed = __pre_passed && true; //
conditionally succeeded in: MyStack::push(java.lang.Object)
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::push(java.lang.Object)
/*|*/   }
/*|*/   if (!__pre_passed) {
```

```
/*|*/     throw new RuntimeException ("src/MyStack.java:35: error: precondition violated
(MyStack::push(java.lang.Object)): (/*declared in MyStack::push(java.lang.Object)*/
(!isFull())) && ((e != null)) "
/*|*/   ); }}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:35:  exception <<"+ex+">> occured while evaluating PRE-
condition in src/MyStack.java:35:  MyStack::push(java.lang.Object)): (/*declared in
MyStack::push(java.lang.Object)*/ (!isFull())) && ((e != null)) "
/*|*/   ;
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //-----------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-------------------------------------------------------------------------
------
/*|*/   try {
/*|*/   //-----------------------------------------------------------------------------#*#
/*|*/
        this.myStack.add(e);
        this.size++;

/*|*/   //#*#-------------------------------------------------------------------------------
/*|*/   try {
/*|*/   if (!(getSize () == getSize___pre + 1))
/*|*/     throw new RuntimeException ("src/MyStack.java:35: error: postcondition violated
(MyStack::push(java.lang.Object)): "+
/*|*/     "getSize () == getSize()@pre + 1");
/*|*/   if (!(top() == e))
/*|*/     throw new RuntimeException ("src/MyStack.java:35: error: postcondition violated
(MyStack::push(java.lang.Object)): "+
/*|*/     "top() == e");
/*|*/   if (!(!isEmpty()))
/*|*/     throw new RuntimeException ("src/MyStack.java:35: error: postcondition violated
(MyStack::push(java.lang.Object)): "+
/*|*/     "!isEmpty()");}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:35:  exception <<"+ex+">> occured while evaluating
POST-condition in src/MyStack.java:35:  MyStack::push(java.lang.Object)";
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //-----------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-------------------------------------------------------------------------
------
/*|*/   } finally {
/*|*/     this.__inv_check_at_exit__MyStack(Thread.currentThread(),"src/MyStack.java:35:  just
before exit MyStack::push(java.lang.Object) ");
/*|*/   }
/*|*/   //-----------------------------------------------------------------------------#*#
```

```
/*|*/
}

    /**
     * @pre !isEmpty()
     * @post return != null
     */
    public Object top() {
/*|*/   //#*#---------------------------------------------------------------------------
/*|*/   this.__inv_check_at_entry__MyStack(Thread.currentThread(),"src/MyStack.java:44:  just
after entry MyStack::top() ");
/*|*/   //---------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-----------------------------------------------------------------------
------
/*|*/   java.lang.Object __return_value_holder_;
/*|*/   //---------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-----------------------------------------------------------------------
------
/*|*/   try {
/*|*/   boolean __pre_passed = false; // true if at least one pre-cond conj. passed.
/*|*/   // checking MyStack::top()
/*|*/   if (! __pre_passed ) {
/*|*/   if ( (!isEmpty() /* do not check prepassed */ ))   __pre_passed = true; // succeeded
in: MyStack::top()
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::top()
/*|*/   }
/*|*/   if (!__pre_passed) {
/*|*/     throw new RuntimeException ("src/MyStack.java:44: error: precondition violated
(MyStack::top()): (/*declared in MyStack::top()*/ (!isEmpty())) "
/*|*/   ); }}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:44:  exception <<"+ex+">> occured while evaluating PRE-
condition in src/MyStack.java:44:  MyStack::top()): (/*declared in MyStack::top()*/
(!isEmpty())) "
/*|*/   ;
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //---------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-----------------------------------------------------------------------
------
/*|*/   try {
/*|*/   //---------------------------------------------------------------------------#*#
/*|*/

/*|*/   //#*#---------------------------------------------------------------------------
/*|*/   /*
return this.myStack.get(this.getSize() - 1);
```

```
/*|*/
/*|*/   __return_value_holder_ =  this.myStack.get(this.getSize() - 1);
/*|*/   //----------------------------------------------------------------------------------#*#
/*|*/
/*|*/   //#*#-------------------------------------------------------------------------------
/*|*/   try {
/*|*/   if (!(__return_value_holder_ != null))
/*|*/     throw new RuntimeException ("src/MyStack.java:44: error: postcondition violated
(MyStack::top()): "+
/*|*/     "(/*return*/  this.myStack.get(this.getSize() - 1)) != null");}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:44:  exception <<"+ex+">> occured while evaluating
POST-condition in src/MyStack.java:44:  MyStack::top()";
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //----------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-----------------------------------------------------------------------
------
/*|*/   } finally {
/*|*/     this.__inv_check_at_exit__MyStack(Thread.currentThread(),"src/MyStack.java:44:  just
before exit MyStack::top() ");
/*|*/   }
/*|*/   //----------------------------------------------------------------------------------#*#
/*|*/
/*|*/   //#*#-------------------------------------------------------------------------------
/*|*/   return __return_value_holder_;
/*|*/   //----------------------------------------------------------------------------------#*#

}

    /**
     * @pre true
     * @post return == size
     */
    public int getSize() {
/*|*/   //#*#-------------------------------------------------------------------------------
/*|*/   this.__inv_check_at_entry__MyStack(Thread.currentThread(),"src/MyStack.java:52:  just
after entry MyStack::getSize() ");
/*|*/   //----------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-----------------------------------------------------------------------
------
/*|*/   int __return_value_holder_;
/*|*/   //----------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-----------------------------------------------------------------------
------
/*|*/   try {
/*|*/   boolean __pre_passed = false; // true if at least one pre-cond conj. passed.
/*|*/   // checking MyStack::getSize()
/*|*/   if (! __pre_passed ) {
```

```
/*|*/   if ( (true /* do not check prepassed */ ))   __pre_passed = true; // succeeded in:
MyStack::getSize()
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::getSize()
/*|*/   }
/*|*/   if (!__pre_passed) {
/*|*/     throw new RuntimeException ("src/MyStack.java:52: error: precondition violated
(MyStack::getSize()): (/*declared in MyStack::getSize()*/ (true)) "
/*|*/   ); }}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:52:  exception <<"+ex+">> occured while evaluating PRE-
condition in src/MyStack.java:52:  MyStack::getSize()): (/*declared in MyStack::getSize()*/
(true)) "
/*|*/   ;
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //---------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-----------------------------------------------------------------------
------
/*|*/   try {
/*|*/   //---------------------------------------------------------------------------------#*#
/*|*/

/*|*/   //#*#-----------------------------------------------------------------------------------
/*|*/   /*
return this.size;

/*|*/
/*|*/   __return_value_holder_ =  this.size;
/*|*/   //---------------------------------------------------------------------------------#*#
/*|*/
/*|*/   //#*#-----------------------------------------------------------------------------------
/*|*/   try {
/*|*/   if (!(__return_value_holder_ == size))
/*|*/     throw new RuntimeException ("src/MyStack.java:52: error: postcondition violated
(MyStack::getSize()): "+
/*|*/     "(/*return*/  this.size) == size");}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:52:  exception <<"+ex+">> occured while evaluating
POST-condition in src/MyStack.java:52:  MyStack::getSize()";
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //---------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-----------------------------------------------------------------------
------
/*|*/   } finally {
```

```
/*|*/     this.__inv_check_at_exit__MyStack(Thread.currentThread(),"src/MyStack.java:52:  just
before exit MyStack::getSize() ");
/*|*/   }
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/
/*|*/   //#*#-------------------------------------------------------------------------------
/*|*/   return __return_value_holder_;
/*|*/   //-------------------------------------------------------------------------------#*#

}

    /**
     * @pre true
     * @post return == (getSize() == 0)
     */
    public boolean isEmpty() {
/*|*/   //#*#-------------------------------------------------------------------------------
/*|*/   this.__inv_check_at_entry__MyStack(Thread.currentThread(),"src/MyStack.java:60:  just
after entry MyStack::isEmpty() ");
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-------------------------------------------------------------------------
------
/*|*/   boolean __return_value_holder_;
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-------------------------------------------------------------------------
------
/*|*/   try {
/*|*/   boolean __pre_passed = false; // true if at least one pre-cond conj. passed.
/*|*/   // checking MyStack::isEmpty()
/*|*/   if (! __pre_passed ) {
/*|*/   if ( (true /* do not check prepassed */ ))   __pre_passed = true; // succeeded in:
MyStack::isEmpty()
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::isEmpty()
/*|*/   }
/*|*/   if (!__pre_passed) {
/*|*/     throw new RuntimeException ("src/MyStack.java:60: error: precondition violated
(MyStack::isEmpty()): (/*declared in MyStack::isEmpty()*/ (true)) "
/*|*/   ); }}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:60:  exception <<"+ex+">> occured while evaluating PRE-
condition in src/MyStack.java:60:  MyStack::isEmpty()): (/*declared in MyStack::isEmpty()*/
(true)) "
/*|*/   ;
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#-------------------------------------------------------------------------
------
```

```
/*|*/   try {
/*|*/   //--------------------------------------------------------------------------------#*#
/*|*/

/*|*/   //#*#--------------------------------------------------------------------------------
/*|*/   /*
return this.getSize() == 0;

/*|*/
/*|*/   __return_value_holder_ =  this.getSize() == 0;
/*|*/   //--------------------------------------------------------------------------------#*#
/*|*/
/*|*/   //#*#--------------------------------------------------------------------------------
/*|*/   try {
/*|*/   if (!(__return_value_holder_ == (getSize() == 0)))
/*|*/     throw new RuntimeException ("src/MyStack.java:60: error: postcondition violated
(MyStack::isEmpty()): "+
/*|*/     "(/*return*/  this.getSize() == 0) == (getSize() == 0)");}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:60:  exception <<"+ex+">> occured while evaluating
POST-condition in src/MyStack.java:60:  MyStack::isEmpty()";
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //--------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#--------------------------------------------------------------------------
------
/*|*/   } finally {
/*|*/     this.__inv_check_at_exit__MyStack(Thread.currentThread(),"src/MyStack.java:60:  just
before exit MyStack::isEmpty() ");
/*|*/   }
/*|*/   //--------------------------------------------------------------------------------#*#
/*|*/
/*|*/   //#*#--------------------------------------------------------------------------------
/*|*/   return __return_value_holder_;
/*|*/   //--------------------------------------------------------------------------------#*#

}

    /**
     * @pre true
     * @post return == (getSize() == MAX)
     */
    public boolean isFull() {
/*|*/   //#*#--------------------------------------------------------------------------------
/*|*/   this.__inv_check_at_entry__MyStack(Thread.currentThread(),"src/MyStack.java:68:  just
after entry MyStack::isFull() ");
/*|*/   //--------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#--------------------------------------------------------------------------
------
```

```
/*|*/   boolean __return_value_holder_;
/*|*/   //---------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#----------------------------------------------------------------------------
------
/*|*/   try {
/*|*/   boolean __pre_passed = false; // true if at least one pre-cond conj. passed.
/*|*/   // checking MyStack::isFull()
/*|*/   if (! __pre_passed ) {
/*|*/   if ( (true /* do not check prepassed */ ))   __pre_passed = true; // succeeded in:
MyStack::isFull()
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::isFull()
/*|*/   }
/*|*/   if (!__pre_passed) {
/*|*/     throw new RuntimeException ("src/MyStack.java:68: error: precondition violated
(MyStack::isFull()): (/*declared in MyStack::isFull()*/ (true)) "
/*|*/   ); }}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:68:  exception <<"+ex+">> occured while evaluating PRE-
condition in src/MyStack.java:68:  MyStack::isFull()): (/*declared in MyStack::isFull()*/
(true)) "
/*|*/   ;
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //---------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#----------------------------------------------------------------------------
------
/*|*/   try {
/*|*/   //---------------------------------------------------------------------------------#*#
/*|*/

/*|*/   //#*#---------------------------------------------------------------------------------
/*|*/   /*
return this.getSize() == this.MAX;

/*|*/
/*|*/   __return_value_holder_ =  this.getSize() == this.MAX;
/*|*/   //---------------------------------------------------------------------------------#*#
/*|*/
/*|*/   //#*#---------------------------------------------------------------------------------
/*|*/   try {
/*|*/   if (!(__return_value_holder_ == (getSize() == MAX)))
/*|*/     throw new RuntimeException ("src/MyStack.java:68: error: postcondition violated
(MyStack::isFull()): "+
/*|*/     "(/*return*/  this.getSize() == this.MAX) == (getSize() == MAX)");}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
```

```
/*|*/     else txt = "src/MyStack.java:68:  exception <<"+ex+">> occured while evaluating
POST-condition in src/MyStack.java:68:  MyStack::isFull()";
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#--------------------------------------------------------------------------
------
/*|*/   } finally {
/*|*/     this.__inv_check_at_exit__MyStack(Thread.currentThread(),"src/MyStack.java:68:  just
before exit MyStack::isFull() ");
/*|*/   }
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/
/*|*/   //#*#--------------------------------------------------------------------------------
/*|*/   return __return_value_holder_;
/*|*/   //-------------------------------------------------------------------------------#*#

}

    /**
     * @pre true
     * @post isEmpty()
     */
    public void makeEmpty() {
/*|*/   //#*#--------------------------------------------------------------------------------
/*|*/   this.__inv_check_at_entry__MyStack(Thread.currentThread(),"src/MyStack.java:76:  just
after entry MyStack::makeEmpty() ");
/*|*/   //-------------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#--------------------------------------------------------------------------
------
/*|*/   try {
/*|*/   boolean __pre_passed = false; // true if at least one pre-cond conj. passed.
/*|*/   // checking MyStack::makeEmpty()
/*|*/   if (! __pre_passed ) {
/*|*/   if ( (true /* do not check prepassed */ ))   __pre_passed = true; // succeeded in:
MyStack::makeEmpty()
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::makeEmpty()
/*|*/   }
/*|*/   if (!__pre_passed) {
/*|*/     throw new RuntimeException ("src/MyStack.java:76: error: precondition violated
(MyStack::makeEmpty()): (/*declared in MyStack::makeEmpty()*/ (true)) "
/*|*/   ); }}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:76:  exception <<"+ex+">> occured while evaluating PRE-
condition in src/MyStack.java:76:  MyStack::makeEmpty()): (/*declared in
MyStack::makeEmpty()*/ (true)) "
/*|*/   ;
/*|*/     throw new RuntimeException(txt);}
```

```
/*|*/
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#------------------------------------------------------------------------
------
/*|*/   try {
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/
        this.myStack.clear();
        this.size = 0;

/*|*/   //#*#----------------------------------------------------------------------------
/*|*/   try {
/*|*/   if (!(isEmpty()))
/*|*/     throw new RuntimeException ("src/MyStack.java:76: error: postcondition violated
(MyStack::makeEmpty()): "+
/*|*/     "isEmpty()");}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:76:  exception <<"+ex+">> occured while evaluating
POST-condition in src/MyStack.java:76:  MyStack::makeEmpty()";
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/   /*|*/   //#*#------------------------------------------------------------------------
------
/*|*/   } finally {
/*|*/     this.__inv_check_at_exit__MyStack(Thread.currentThread(),"src/MyStack.java:76:  just
before exit MyStack::makeEmpty() ");
/*|*/   }
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/
}

    /**
     * @pre true
     * @post true
     */
    public static void main(String[] args) {
/*|*/   //#*#----------------------------------------------------------------------------
/*|*/   try {
/*|*/   boolean __pre_passed = false; // true if at least one pre-cond conj. passed.
/*|*/   // checking MyStack::main(java.lang.String[])
/*|*/   if (! __pre_passed ) {
/*|*/   if ( (true /* do not check prepassed */ ))   __pre_passed = true; // succeeded in:
MyStack::main(java.lang.String[])
/*|*/   else
/*|*/     __pre_passed = false; // failed in: MyStack::main(java.lang.String[])
/*|*/   }
/*|*/   if (!__pre_passed) {
```

```
/*|*/     throw new RuntimeException ("src/MyStack.java:85: error: precondition violated
(MyStack::main(java.lang.String[])): (/*declared in MyStack::main(java.lang.String[])*/
(true)) "
/*|*/   ); }}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:85:  exception <<"+ex+">> occured while evaluating PRE-
condition in src/MyStack.java:85:  MyStack::main(java.lang.String[])): (/*declared in
MyStack::main(java.lang.String[])*/ (true)) "
/*|*/   ;
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/
        MyStack stack = new MyStack();

        if(!stack.isFull()) {
            stack.push(new Integer(0));
        }
        if(!stack.isEmpty()) {
            Object o = stack.top();
            stack.pop();
        }
        for(int i = 0; i < 11; i++) {
            stack.push(new Integer(i));
        }
        stack.makeEmpty();

/*|*/   //#*#----------------------------------------------------------------------------
/*|*/   try {
/*|*/   if (!(true))
/*|*/     throw new RuntimeException ("src/MyStack.java:85: error: postcondition violated
(MyStack::main(java.lang.String[])): "+
/*|*/     "true");}
/*|*/   catch ( RuntimeException ex ) {
/*|*/     String txt = "";  if (ex.getClass()==RuntimeException.class) { txt = ex.toString();;
}
/*|*/     else txt = "src/MyStack.java:85:  exception <<"+ex+">> occured while evaluating
POST-condition in src/MyStack.java:85:  MyStack::main(java.lang.String[])";
/*|*/     throw new RuntimeException(txt);}
/*|*/
/*|*/   //----------------------------------------------------------------------------#*#
/*|*/
}
}
```

A.3 MyStack – Jass

```
public class MyStack implements Cloneable {
    private java.util.Vector myStack;
```

```
private final int MAX = 10;
private int size;

public MyStack() {
    /** require true; **/
    this.myStack = new java.util.Vector();
    this.size = 0;
    /** ensure isEmpty(); **/
}

public void pop() {
    /** require !isEmpty(); **/
    this.myStack.remove(this.getSize() - 1);
    this.size--;
    /** ensure getSize() == Old.getSize() - 1; !isFull(); **/
}

public void push(Object e) {
    /** require !isFull(); e != null; **/
    this.myStack.add(e);
    this.size++;
    /** ensure getSize() == Old.getSize() + 1; top() == e; !isEmpty(); **/
}

public Object top() {
    /** require !isEmpty(); **/
    return this.myStack.get(this.getSize() - 1);
    /** ensure Result == Old.top(); Result != null; changeonly{}; **/
}

public int getSize() {
    /** require true; **/
    return this.size;
    /** ensure Result == size; changeonly{}; **/
}

public boolean isEmpty() {
    /** require true; **/
    return this.getSize() == 0;
    /** ensure Result == (getSize() == 0); changeonly{}; **/
}

public boolean isFull() {
    /** require true; **/
    return this.getSize() == this.MAX;
    /** ensure Result == (getSize() == MAX); changeonly{}; **/
}

public void makeEmpty() {
    /** require true; **/
    this.myStack.clear();
    this.size = 0;
    /** ensure isEmpty(); changeonly{myStack, size}; **/
}

protected Object clone() {
    Object b = null;
    try {
        b = super.clone();
    }
    catch (CloneNotSupportedException e){;}
    return b;
}

public static void main(String[] args) {
    /** require true; **/
    MyStack stack = new MyStack();

    if(!stack.isFull()) {
        stack.push(new Integer(0));
    }
    if(!stack.isEmpty()) {
        Object o = stack.top();
        stack.pop();
    }
    for(int i = 0; i < 11; i++) {
        stack.push(new Integer(i));
```

```
        }
        stack.makeEmpty();
        /** ensure true; **/
    }
    /** invariant getSize() >= 0; getSize() <= MAX; **/
}
```

A.4 MyStack – Automatgenererad av Jass

```
public class MyStack implements Cloneable {
    private java.util.Vector myStack;
    private final int MAX = 10;
    private int size;

    public MyStack() {
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true;
jass.runtime.traceAssertion.Parameter[] jassParameters; jassParameters = new
jass.runtime.traceAssertion.Parameter[] {};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "MyStack()", true),
jassParameters);


              /* precondition */
              if (!(true)) throw new
jass.runtime.PreconditionException("MyStack","MyStack()",7,null);
        this.myStack = new java.util.Vector();
        this.size = 0;
              /* postcondition */
              if (!(jassInternal_isEmpty())) throw new
jass.runtime.PostconditionException("MyStack","MyStack()",10,null);
              /* invariant */
              jassCheckInvariant("at end of method MyStack().");
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true; jassParameters =
new jass.runtime.traceAssertion.Parameter[] {};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "MyStack()", false),
jassParameters);

    }

    public void pop() {
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true;
jass.runtime.traceAssertion.Parameter[] jassParameters; jassParameters = new
jass.runtime.traceAssertion.Parameter[] {};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "pop()", true),
jassParameters);
```

```
              /* invariant */
              jassCheckInvariant("at begin of method pop().");
              MyStack jassOld = (MyStack)this.clone();
              /* precondition */
              if (!(!jassInternal_isEmpty())) throw new
jass.runtime.PreconditionException("MyStack","pop()",14,null);
        this.myStack.remove(this.getSize() - 1);
        this.size--;
              /* postcondition */
              if (!(jassInternal_getSize()==jassOld.jassInternal_getSize()-1)) throw new
jass.runtime.PostconditionException("MyStack","pop()",17,null);
              if (!(!jassInternal_isFull())) throw new
jass.runtime.PostconditionException("MyStack","pop()",17,null);
              /* invariant */
              jassCheckInvariant("at end of method pop().");
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true; jassParameters =
new jass.runtime.traceAssertion.Parameter[] {};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "pop()", false),
jassParameters);

    }

    public void push(Object e) {
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true;
jass.runtime.traceAssertion.Parameter[] jassParameters; jassParameters = new
jass.runtime.traceAssertion.Parameter[] {new jass.runtime.traceAssertion.Parameter(e)};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "push(java.lang.Object)",
true), jassParameters);


              /* invariant */
              jassCheckInvariant("at begin of method push(java.lang.Object).");
              MyStack jassOld = (MyStack)this.clone();
              /* precondition */
              if (!(!jassInternal_isFull())) throw new
jass.runtime.PreconditionException("MyStack","push(java.lang.Object)",21,null);
              if (!(e!=null)) throw new
jass.runtime.PreconditionException("MyStack","push(java.lang.Object)",21,null);
        this.myStack.add(e);
        this.size++;
              /* postcondition */
              if (!(jassInternal_getSize()==jassOld.jassInternal_getSize()+1)) throw new
jass.runtime.PostconditionException("MyStack","push(java.lang.Object)",24,null);
              if (!(jassInternal_top()==e)) throw new
jass.runtime.PostconditionException("MyStack","push(java.lang.Object)",24,null);
              if (!(!jassInternal_isEmpty())) throw new
jass.runtime.PostconditionException("MyStack","push(java.lang.Object)",24,null);
```

```
            /* invariant */
            jassCheckInvariant("at end of method push(java.lang.Object).");
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true; jassParameters =
new jass.runtime.traceAssertion.Parameter[] {};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "push(java.lang.Object)",
false), jassParameters);

    }

    public Object top() {
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true;
jass.runtime.traceAssertion.Parameter[] jassParameters; jassParameters = new
jass.runtime.traceAssertion.Parameter[] {};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "top()", true),
jassParameters);
            java.lang.Object jassResult;

            /* invariant */
            jassCheckInvariant("at begin of method top().");
            MyStack jassOld = (MyStack)this.clone();
            /* precondition */
            if (!(!jassInternal_isEmpty())) throw new
jass.runtime.PreconditionException("MyStack","top()",28,null);
            jassResult = ( this.myStack.get(this.getSize() - 1));
            /* postcondition */
            if (!(jassResult==jassOld.jassInternal_top())) throw new
jass.runtime.PostconditionException("MyStack","top()",30,null);
            if (!(jassResult!=null)) throw new
jass.runtime.PostconditionException("MyStack","top()",30,null);
            if (!(jass.runtime.Tool.referenceEquals(myStack,jassOld.myStack) && MAX ==
jassOld.MAX && size == jassOld.size)) throw new
jass.runtime.PostconditionException("MyStack","top()",-1,"Method has changed old value.");
            /* invariant */
            jassCheckInvariant("before return in method top().");
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true; jassParameters =
new jass.runtime.traceAssertion.Parameter[] {new
jass.runtime.traceAssertion.Parameter(jassResult)};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "top()", false),
jassParameters);


            return jassResult;
    }

    public int getSize() {
```

```
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true;
jass.runtime.traceAssertion.Parameter[] jassParameters; jassParameters = new
jass.runtime.traceAssertion.Parameter[] {};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "getSize()", true),
jassParameters);
              int jassResult;

              /* invariant */
              jassCheckInvariant("at begin of method getSize().");
              MyStack jassOld = (MyStack)this.clone();
              /* precondition */
              if (!(true)) throw new
jass.runtime.PreconditionException("MyStack","getSize()",34,null);
              jassResult = ( this.size);
              /* postcondition */
              if (!(jassResult==size)) throw new
jass.runtime.PostconditionException("MyStack","getSize()",36,null);
              if (!(jass.runtime.Tool.referenceEquals(myStack,jassOld.myStack) && MAX ==
jassOld.MAX && size == jassOld.size)) throw new
jass.runtime.PostconditionException("MyStack","getSize()",-1,"Method has changed old value.");
              /* invariant */
              jassCheckInvariant("before return in method getSize().");
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true; jassParameters =
new jass.runtime.traceAssertion.Parameter[] {new
jass.runtime.traceAssertion.Parameter(jassResult)};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "getSize()", false),
jassParameters);


              return jassResult;
    }

    public boolean isEmpty() {
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true;
jass.runtime.traceAssertion.Parameter[] jassParameters; jassParameters = new
jass.runtime.traceAssertion.Parameter[] {};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "isEmpty()", true),
jassParameters);
              boolean jassResult;

              /* invariant */
              jassCheckInvariant("at begin of method isEmpty().");
              MyStack jassOld = (MyStack)this.clone();
              /* precondition */
```

```
            if (!(true)) throw new
jass.runtime.PreconditionException("MyStack","isEmpty()",40,null);
            jassResult = ( this.getSize() == 0);
            /* postcondition */
            if (!(jassResult==(jassInternal_getSize()==0))) throw new
jass.runtime.PostconditionException("MyStack","isEmpty()",42,null);
            if (!(jass.runtime.Tool.referenceEquals(myStack,jassOld.myStack) && MAX ==
jassOld.MAX && size == jassOld.size)) throw new
jass.runtime.PostconditionException("MyStack","isEmpty()",-1,"Method has changed old value.");
            /* invariant */
            jassCheckInvariant("before return in method isEmpty().");
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true; jassParameters =
new jass.runtime.traceAssertion.Parameter[] {new
jass.runtime.traceAssertion.Parameter(jassResult)};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "isEmpty()", false),
jassParameters);


            return jassResult;
    }

    public boolean isFull() {
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true;
jass.runtime.traceAssertion.Parameter[] jassParameters; jassParameters = new
jass.runtime.traceAssertion.Parameter[] {};
jass.runtime.traceAssertion.CommunicationManager.internalAction = false;
jass.runtime.traceAssertion.CommunicationManager.communicate(this, new
jass.runtime.traceAssertion.MethodReference("null", "MyStack", "isFull()", true),
jassParameters);
            boolean jassResult;

            /* invariant */
            jassCheckInvariant("at begin of method isFull().");
            MyStack jassOld = (MyStack)this.clone();
            /* precondition */
            if (!(true)) throw new
jass.runtime.PreconditionException("MyStack","isFull()",46,null);
            jassResult = ( this.getSize() == this.MAX);
            /* postcondition */
            if (!(jassResult==(jassInternal_getSize()==MAX))) throw new
jass.runtime.PostconditionException("MyStack","isFull()",48,null);
            if (!(jass.runtime.Tool.referenceEquals(myStack,jassOld.myStack) && MAX ==
jassOld.MAX && size == jassOld.size)) throw new
jass.runtime.PostconditionException("MyStack","isFull()",-1,"Method has changed old value.");
            /* invariant */
            jassCheckInvariant("before return in method isFull().");
      jass.runtime.traceAssertion.CommunicationManager.internalAction = true; jassParameters =
new jass.runtime.traceAssertion.Parameter[] {new
jass.runtime.traceAssertion.Parameter(jassResult)};
```