# Pålitligare programvara med Java Modeling Language och dess stödverktyg

Benjamin Burman

Kandidatavhandling i datateknik
Institutionen för informationsteknologi
Åbo Akademi
Handledare: Johannes Eriksson
3.4.2011

# Referat

Den här avhandlingen kommer att behandla Java Modeling Language (JML), ett formellt gränssnittsspecifikationsspråk för programmeringsspråket Java, vilket man kan använda som ett verktyg för att i programvara implementera Design enligt kontrakt (eng. *Design by Contract*). Avhandlingen börjar med en allmän presentation av bakomliggande orsaker till design enligt kontrakt och fortsätter med att visa syntaxen för JML, vilka möjligheter språket ger utöver implementeringen av design enligt kontrakt i Java och vilka begränsningar språket har. Syntaxen följs upp med små exempel för att visa hur JML används i praktiken. Därefter introduceras några stödverktyg. Till slut visas ett exempelprogram som illustrerar en delmängd av all funktionalitet som kan uppnås med JML och vad några stödverktyg ger för utdata när de körs på exempelprogrammet.

**Sökord:** Java Modeling Language, Design by Contract, förvillkor, eftervillkor, invarianter.

# Innehåll

# 1 Inledning

Ett mål inom programvaruproduktion är att man skall skapa pålitliga och, till den mån det är möjligt, felfria system. Speciellt viktigt är detta vid produktion av komponenter i objektorienterade språk, eftersom objektorientering ger underlag för återanvändning och ett fel i någon komponent som återanvänts i flera system kan orsaka problem i alla dessa system. [1]

En metod som framkommer i många böcker som behandlar pålitlighet i programvara är defensiv programmering (eng. *Defensive programming*). Denna metod går ut på att man försöker göra programvarukomponenter säkra även om de utsätts för oväntad användning. Detta leder i sin tur till att utvecklare av komponenter lägger till mycket kod som endast utför granskningar i den egentliga funktionella koden. Ytterligare kan det också hända att redundanta granskningar skapas på klientsidan och hos tillhandahållaren av tjänsten (klient och tillhandahållare avser här programvaruklasser eller komponenter, och kommer att göra så i den här avhandlingen). Allt detta leder dock till ökad komplexitet hos programkoden, vilket är ett av de största problemen inom programvaruproduktion över huvudtaget. [1]

Ett annat sätt att uppnå dessa mål hos programvaran är design enligt kontrakt (eng. *Design by Contract*), vilket är ett begrepp som innefattar metoder och riktlinjer för hur man kan skapa pålitlig programvara [1]. Översiktligt kan man säga att det går ut på att man genom att använda olika typer av intyganden (eng. *assertions*) skapar kontrakt som klienter och tillhandahållare av tjänster kommer överens om och antas hålla [2].

Java Modeling Language (JML) är ett formellt gränssnittsspecifikationsspråk som kan användas för att specificera beteendet hos Java-moduler. Det kombinerar tillvägagångssättet som programmeringsspråket Eiffel [3] använder för att möjliggöra design enligt kontrakt och ett modellbaserat specifikationssätt som används i Larch-familjen av specifikationsspråk. [4]

Det finns många JML-stödverktyg, med vilka man som programvaruutvecklare lättare kan skapa pålitligare programvara. Vad dessa verktyg specifikt hjälper till med är granskning och testning av intyganden under körtid, statisk granskning och verifiering, generering av specifikationer och dokumentation [5]. Några exempel på dylika verktyg är JML-kompilatorn, *jmlc*, JML-enhetstestaren, *jmlunit*, den utvidgade statiska granskaren, *ESC/Java* [6], och JML-dokumentationsgeneratorn, *jmldoc* [5]. Dessa verktyg kommer att beskrivas översiktligt i avsnitt 5. Mera information om JML och nedladdningsbara stödverktyg kan hittas på http://www.eecs.ucf.edu/~leavens/JML/ .

Huvudmålet med denna avhandling kommer att vara se hur JML och tillhörande stödverktyg möjliggör produktion av bättre programvara.

# 2 Java Modeling Language

## 2.1 Allmänt

Java Modeling Language kan användas för att specificera Java-klassers detaljerade design genom annotationer i Java-programkod. Målet med JML är att man skall ha ett specifikationsspråk som för Java-programmerare är lätt att använda och som stöds av en stor uppsättning av verktyg. [5]

Kontrakt i JML går ut på att man skriver programkommentarer bland annat i form av för- och eftervillkor för metoder. JML stöder direkt att dessa villkor kompileras med JML-kompilatorn, *jmlc*, till Java-bytekod så att villkoren också kan maskintestas i praktiken.[7]

## 2.2 Notation

Annotationerna som används i JML är `@`-tecken inbakade i vanliga Javakommentarer, detta för att en vanlig Javakompilator (t.ex. *javac*) skall ignorera dem då vissa av dem liknar vanliga Javauttryck [5]. Ett korrekt intygande skrivs antingen efter ”`//@`” eller ”`/*@ <JML specifikation> @*/`”. Noggrannhet måste fästas vid skrivandet av dessa annotationer, eftersom de måste stå exakt så som ovan illustrerats. Ett mellanslag mellan `@`-tecknet och Java-kommentarsymbolerna ”`//`” eller ”`/*`” fungerar alltså inte [7].

```
//@ requires x >= 0.0;
/*@ ensures JMLDouble
  @  .approximatelyEqualTo
  @  (x, \result * \result, eps);
  @*/
public static double sqrt(double x) {
     /*...*/
}
```

Figur 2.1: Exempel på för- och eftervillkor i JML.
Källa: Gary T. Leavens, Yoonsik Cheon, Design by Contract with JML, 2006

För att beteckna förvillkor (eng. *precondition*) används nyckelordet `requires`. Eftervillkor (eng. *postcondition*) betecknas med nyckelordet `ensures`. Ett kort exempel för att belysa detta finns i Figur 2.1, kontraktet gäller i detta exempel en metod, `sqrt`, som beräknar kvadratroten av ett givet tal [7]. Förvillkor är att inputvärdet `x` är större än lika med noll och eftervillkoret är att ett tillräckligt noggrannt (och korrekt) värde har beräknats. Metoden `approximatelyEqualTo` i klassen `JMLDouble` testar ifall den relativa skillnaden mellan de två första argumenten är tillräckligt noggrann (är mindre än ett givet epsilon, `eps`. Parametern `\result` i exemplet syftar på det från metoden returnerade värdet. Fler nyckelord och deras betydelse illustreras i Tabell 2.1.

| Nyckelord | Betydelse |
|---|---|
| `signals` | Definierar ett eftervillkor i fall av att ett givet Java-undantag (`Exception`) har kastats. |
| `signals_only` | Definierar ett eftervillkor i fall av ett i metodens deklaration med `throws`-nyckelordet givet Java-undantag har kastats. |
| `invariant` | Definierar en klassinvariant. |
| `assignable` | Definierar vilka fält som får modifieras i den efterföljande metoden. |
| `pure` | Berättar att metoden inte ändrar på något fält. Synonymt med att skriva: `assignable \nothing` |
| `also` | Kombinerar flera specifikationsfall och kan deklarera att en metod ärver specifikation från sin(a) superklass(er). |
| `assert` | Definierar ett JML-intygande. |
| `spec_public` | Används för att definiera att en specifikation är publik, trots att fältet den beskriver kan vara definierat som privat eller privat för paketet (`protected`). |
| `\old(E)` | Syftar på värdet av `E` vid början av anropet av metod. |
| `\forall` | Allkvantor. |
| `\exists` | Existenskvantor. |
| `a==>b` | `a` implicerar `b`. |
| `a<==b` | `a` följer från `b`. |
| `a<==>b` | `a` om och endast om `b`. |
| `a<=!=>b` | inte `(a` om och endast om `b)`. |

Tabell 2.1: Några JML-nyckelord och deras betydelser.

Invarianter definieras med nyckelordet `invariant`, och är egenskaper som måste gälla alla synliga tillstånd för en klass. De måste gälla vid instantieringen av en klass, vid början av en metod och vid slutet på en metod i klassen. Mellan början och slutet på en metod behöver de nödvändigtvis inte gälla. Exempel på användningen av invarianter ges i Figur 2.2. [7]

```
/*@ public invariant !name.equals("")
  @ && weight >= 0;
  @*/
```

Figur 2.2: Exempel på användning av invariant.
Källa: Gary T. Leavens, Yoonsik Cheon, Design by Contract with JML, 2006

Allkvantorns användningssätt ser ut som följande: "`(\forall T x; b1; b2)`", där T är en typ, `b1` och `b2` är booleska uttryck och `x` är en variabel som finns i `b1` och `b2`. Detta betyder att för alla instanser av typ `x`, för vilka `b1` gäller, måste även `b2` gälla [8]. Existenskvantorn fungerar på liknande vis: "`(\exists T x; b1; b2)`", vilket betyder att det finns en instans `x` av typ `T` för vilken både `b1` och `b2` gäller [8]. Dessa kan även förkortas ifall man enbart har ett booleskt uttryck b vilket bör gälla för respektive kvantor: "`(\<kvantor> T x; b)`", vilket är samma sak som att skriva "`(\<kvantor> T x; true; b)`" [8].

Det finns också något som kallas informella beskrivningar i specifikationer. Dessa innebär att man, som namnet antyder, skriver icke-formella specifikationer i kommentarer för en metod, vilka av JML-kompilatorn tolkas som booleska uttryck med värdet `true` [7]. Vad detta medför är att man då kan kombinera dylika uttryck med andra (formella) beskrivningar, för att göra specifikationen tydligare för programmerare. Informella beskrivningar skrivs som följande: "`(* <någon informell specifikation> *)`", och givetvis inom ramarna av JML-annotationer. Figur 2.3 visar ett litet exempel på detta [7].

```
//@ requires (* x is positive *);
/*@ ensures (* \result is an
  @ approximation to
  @ the square root of x *)
  @ && \result >= 0;
  @*/
public static double sqrt(double x) {
  return Math.sqrt(x);
}
```

Figur 2.3: Exempel på informella beskrivningar.
Källa: Gary T. Leavens, Yoonsik Cheon, Design by Contract with JML, 2006

## 2.3 Begränsningar på specifikationsuttryck

JML strävar efter att vara så nära Java som möjligt. Specifikationerna är ofta enbart Java-uttryck. Den kanske största begränsningen uttryck i JML har gentemot vanliga Java-uttryck är att de inte får ha några följder i programvaran. Ifall metodanrop på metoder i klassen används måste alltså vara så kallade rena metoder (eng. *pure methods*), och alla former av tilldelningsoperationer, t.ex. ”`var++`” eller ”`var = värde;`”, är således förbjudna. [8]

Rena metoder måste vara utmärkta med annotationen ”`/*@ pure *@/`”. Ifall man t.ex. har en publik (och ren) metod `getSize()` som returnerar ett heltal, skall den definieras enligt följande: ”`public /*@ pure *@/ int getSize(){...}`”

En annan begränsning med att använda Java-uttryck i specifikationer är det faktum att ett uttryck i Java inte alltid kan beräknas, utan under vissa omständigheter kan undantag kastas. T.ex. ett uttryck som vid evaluering råkar innebära att något värde skall divideras med noll leder till att ett `java.lang.ArithmeticException` kastas och inte att intygandet misslyckas. Detta beror på att booleska uttryck i Java använder en slags trevärd logik med sanningsvärdena *sant*, *falskt* och *odefinierat* (undantag). [8]

Det är dock fördelaktigare att i specifikationer arbeta med tvåvärd logik, med sanningsvärdena *sant* och *falskt*, eftersom osäkerheten med undantag vid evaluering av specifikationer då undviks. JML använde tidigare

*underspecifikation*, vilket går ut på att man istället för sanningsvärdet *odefinierat* använder fasta men okända värden. Man utnyttjar sig av Javas ”lata” evaluering av booleska uttryck sammanbundna med konnektiven ”`||`” och ”`&&`” för att undvika att undantag kastas [8]. T.ex. uttrycket ”`x == 0 || y/x == 5`” evalueras till `true` ifall värdet på `x` är noll, medan uttrycket om det skrevs i omvänd ordning skulle leda till att ett undantag kastas.

Problemet här är att ansvaret för att specifikationsuttryck inte kan kasta undantag hamnar hos programmeraren, som oftast inte är ofelbar. Man har därför under senare hälften av 2000-talet i JML tagit i bruk något som kallas stark validitet (eng. *strong validity*) [8], vilket innebär att ett uttryck evalueras till `true` ifall det logiskt är sant och inte kastar något undantag. Ifall detta inte är fallet evalueras uttrycket till `false`.

Underspecifikation i samband med JML-specifikationer kan även avse specifikationer som med flit lämnas ofullständiga. Detta kan man göra för att ge utvecklare mera frihet när en metod skall implementeras. Det mest väsentliga bör dock alltid finnas med i specifikationen, ty gränssnittspecifikationen för en metod kan vara allt programmeraren har att tillgå när en implementation skall skapas. [7]

## 2.4 Undantag

JML har stöd för att hantera Java-undantag med hjälp av nyckelorden `signals` och `signals_only`. Dessa används för att definiera eftervillkor som bör gälla ifall metoden kastar ett undantag. Nyckelorden skrivs i programkoden enligt följande mönster: ”`signals(E e) b`”, där `E` är någon undertyp av klassen `Java.lang.Exception`, `e` är en i `b` bunden variabelidentifierare och `b` är ett JML-specifikationsuttryck. [8]

Vad ovanstående uttryck innebär är att ifall den med undantagsbeskrivningen försedda metodens förvillkor har hållit, men metoden avbrutits och kastat ett undantag av typ `E`, så måste `b` kunna evalueras till `true`.

## 2.5 Ramvillkor

Det går inte att garantera att endast variabler som nämns i ett eftervillkor kan ha ändrats under exekveringen av en metod [9], t.ex. eftersom någon form av hjälpvariabler ofta används i programkod: ”count”-variabler som inkrementeras under beräkningen av något annat (önskat) värde, eller någon annan godtycklig variabel som ändras.

För att specificera beteendet av en metod med JML kan man i sådana fall definiera så kallade ramvillkor (eng. *frame properties/frame conditions*). Dessa är definitioner av vilka fält som får vara modifierade efter exekvering av en metod. Nyckelordet `assignable` används för att specificera dessa. [9] Det används enligt följande: ”`assignable variabel1, variabel2, ... ;`”

När ramvillkor används behöver inte lika många fält nämnas i ett eftervillkor som annars. Vilket håller specifikationen simplare. En metod utan explicit specificerat ramvillkor gäller villkoret ”`assignable \everything`” och för en ren metod gäller ”`assignable \nothing`”. [9]

## 2.6 Modellfält

Ramvillkor är inte alltid tillräckliga, eftersom man vid användningen av dem ofta är tvungen att blotta en implementation av en metod, eller överhuvudtaget vara medveten om vilka variabler som kommer att existera, även om ett gränssnitt används.

En lösning på detta problem är abstraktion med modellfält (eng. *model fields*), vilka är fält som endast existerar i specifikationen, men som kan referera till ett konkret fält [7]. Nyckelordet `represents` används för detta ändamål. Figur 2.4 illustrerar ett programkodsexempel [7] på detta. I exemplet är även fältet `fullName` definierat så att det inte får ha värdet `null`.

```
//@ public model non_null String name;
private /*@ non_null @*/ String fullName;
//@ private represents name <- fullName;
```

Figur 2.4: Exempel på ett modellfält (`String name`).
Källa: Gary T. Leavens, Yoonsik Cheon, Design by Contract with JML, 2006

## 2.7 Spökfält

Spökfält (eng. *ghost fields*) och spökvariabler (eng. *ghost variables*) liknar modellfält i och med att de inte kan refereras till av den vanliga Java-koden. Spökfält kan användas för att modellera utvidgade tillstånd i programvaran [9]. De kompileras inte med i vanlig Java-kod men de kan användas på liknande sätt som vanliga fält i Java [8].

Spökfält definieras med nyckelordet `ghost`. De kan tilldelas värden med nyckelordet `set` och användas i specifikationsuttryck. I figur 2.5 illustreras ett exempel på användningen av spökfält. Klassen `Lightbulb` är en simplifierad modell av en glödlampa som antingen är tänd eller släckt. Klassen innehåller dock ingen normal Java-variabel för att hålla reda på vilket läge lampan är. JML-specifkation i form av en spökvariabel `isGlowing` och förvillkor för metoderna `turnOn()` och `turnOff()` finns dock, så att man under testning kan se ifall metoderna används korrekt i sitt klientprogram.

Spökfält blir, om inte annat anges, definierade som statiska. För att undvika detta kan nyckelordet `instance` användas. `Instance` skrivs då direkt efter `ghost` enligt följande mönster: ”`ghost instance`”.

```
class Lightbulb {
     //@ public ghost boolean isGlowing = false;

     //@ requires isGlowing == false;
     public void turnOn(){
          //@ set isGlowing = true;
     }
     //@ requires isGlowing == false;
     public void turnOff(){
          //@ set isGlowing = false;
     }
}
```

Figur 2.5: Exempel på spökfält.

## 2.8 JML och arv

JML stöder beteendemässig subtypning (eng. *behavioural subtyping*) [8]. Instanser av en given typ T måste därför följa samma specifikationer som gäller för alla superklasser till T [9]. JML stöder i och med detta *Liskovs substitutionsprincip* [10], vilket innebär att subklasser till någon klass skall gå att byta ut utan att någon skillnad i beteendet hos programmet där klasserna används märks. Nedärvning enligt denna princip kan därför leda till ökad pålitlighet i programvaran.

En subklass ärver i JML alla specifikationer så som för- och eftervillkor samt invarianter från sin superklass [7]. Det är tillåtet att i subklasser lägga till ytterligare villkor, så länge superklassens specifikation fortfarande håller [8].

Ur en formell synvinkel innebär beteendemässig subtypning att förvillkoret som en metod som överskrider en annan metod impliceras av förvillkoret hos den överskridna metoden. Eftervillkoret hos den överskridande metoden implicerar den överskridna metodens eftervillkor [8]. Man får endast försvaga ett förvillkor i en subklass och endast förstärka ett eftervillkor. Detta krav följer från Liskovs substitutionsprincip: om ett metodanrop görs med superklassens förvillkor kommer det fortfarande att vara inom ramarna för det svagare förvillkoret i subklassen, medan ett striktare eftervillkor i subklassen innebär att det fortfarande är inom ramarna för superklassens eftervillkor.

Invarianter kan dock leda till problem med tanke på kravet att ett förvillkor endast får försvagas i en subklass gentemot förvillkoret i den berörda superklassen. Detta eftersom en invariant medför något nytt krav på klassen som skall gälla vid början av varje metodanrop, vilket i princip är samma sak som ett förstärkt förvillkor. [8]

# 3 Möjligheter med JML

## 3.1 Allmänt

Enbart JML-annotationer i Java-programkod bidrar inte till förbättrad pålitlighet hos programvara. Det finns många stödverktyg till JML med vilka man kan uppnå olika mål. Det finns bland annat verktyg för att utföra granskningar av intyganden vid körtid (eng. *runtime assertion checking*), verktyg för att statiskt utföra granskningar av intyganden (eng. *static checking*), verktyg för att skapa specifikationer och verktyg för att skapa dokumentation [5]. Dessa begrepp kommer att beskrivas kort i detta avsnitt.

## 3.2 Skuldsättning

Design enligt kontrakt ger redan från början ett underlag för skuldsättning (eng. b*lame assignment*) [7]. Beroende på ifall det är ett för- eller eftervillkor som inte hålls, kan man få reda på var fel har uppstått. Brott mot förvillkor tyder på att det finns fel på klientsidan, eftersom det är den anropande sidan (klienten) som skall se till att alla anrop till en viss metod görs enligt den anropade metodens förvillkor. Brott mot eftervillkor tyder på att felet finns på tillhandahållarsidan, eftersom resultatet metoden då har kommit fram till inte följer specifikationen och implementationen av metoden därmed är felaktig.

## 3.3 Granskning vid körtid

Det enklaste sättet att se ifall programvaran fungerar enligt det specificerade kontraktet är att testa JML-intyganden (för- och eftervillkor samt invarianter) under exekvering av programvaran och rapportera om de inte håller [5]. Detta är vad granskning vid körtid går ut på. För att detta verkligen skall fungera krävs dock att informella specifikationer inte används, eftersom de inte är maskinkontrollerbara.

Verktyg som används för detta ändamål är JML-kompilatorn, *jmlc*. Med *Jmlc* kompilerar man Java-kod, men till skillnad från kompilation med *javac* kompilerar man även JML-annotationer till exekverbar kod [5]. *Jmlc* kommer att behandlas vidare i sektion 5.2

## 3.4 Statisk granskning

Statisk granskning (eng. *static checking*) är granskning av programkoden och intyganden utan att exekvera programvaran [11]. Om JML används, kan detta utföras med hjälp av Den utvidgade statiska granskaren, *ESC/Java,* eller dess efterföljare *ESC/Java2*. Dessa kommer att behandlas i sektion 5.3. Utöver dessa två verktyg finns det ett antal andra verktyg för utförande av statisk granskning vilka inte kommer att behandlas.

## 3.5 Enhetstestning

Enhetstestning (eng. *unit testing*) går ut på att varje enskild komponent (eller klass) testas enskilt och fristående från resten av komponenterna [11]. Ett känt ramverk för att utföra dylik testning av Java-komponenter är *JUnit* [12].

JML kan (tillsammans med ett verktyg) användas för att generera enhetstestklasser. Data måste dock framställas av programmeraren [5]. Detta minskar avsevärt på mängden arbete som går åt för programmeraren att skapa enhetstester jämfört med vid användning av enbart *JUnit*. Verktyg för detta är *jmlunit*, vilket kommer att behandlas i sektion 5.4.

## 3.6 Dokumentation

I och med att JML-specifikationen skrivs i programkoden, och direkt har påverkan på programvaran (åtminstone i fallet av granskning vid körtid), fungerar den också som bas för en dokumentation. Denna dokumentation är dessutom up-to-date, då den med ett lämpligt verktyg (*jmldoc*) enkelt kan omvandlas till html-format, inte olikt hur Java-verktyget *javadoc* fungerar. *Jmldoc* kommer att tas upp i sektion 5.5.

# 4 Begränsningar med JML

## 4.1 Allmänt

Trots att JML av programvaruutvecklare kan användas på många olika sätt för att få programvara testad och till viss grad verifierad för korrekthet finns det några frågor om JML som man kan ställa sig: är JML standardiserat och används det över huvudtaget av någon? Hur går det med den matematiska noggrannheten en specifikation kan kräva, med tanke på noggrannheten som de primitiva datatyperna i Java, speciellt flyttal, erbjuder? Lämpar sig JML för specifikation av ett programvarusystem i sin helhet? Dessa frågor kommer att gås igenom i detta avsnitt.

## 4.2 Standardisering

Angående standardiseringen av JML kan sägas att eftersom JML-referensmanualen [13] för tillfället inte är färdigskriven och språket fortfarande utvecklas så är JML inte helt standardiserat. Däremot om man ser på specifikationsspråk som möjliggör design enligt kontrakt för Java kan JML dock ses som standard [8]. JML har tillsammans med några stödverktyg använts i både reella system och i några fallstudier [5].

## 4.3 Noggrannhet gällande flyttal

I sektion 2.2 nämndes ett program för beräkning av kvadratrötter (den ofullständiga programkoden finns listad i Figur 2.1). I exempelprogrammet ser man hur man är tvungen att introducera en variabel som håller värdet på noggrannheten som man önskar beräkna svaret med. Dylika specifikationer kunde leda till problem ifall man har ett liknande krav på någon metod man skall implementera. T.ex. ifall en formell specifikation existerar, men en ny implementation av metoden skall skapas, med målet att snabba upp beräkningen. Här kunde man stöta på problemet att man genom uppsnabbningen av beräkningen är tvungen att sänka precisionen, vilket då skulle vara felaktigt enligt specifikationen. Givetvis kunde specifikationen ändras, men detta kan knappast alltid antas vara möjligt med tanke på resten av systemet.

## 4.4 Fokus på metoder

I och med att JML är ett formellt gränssnittsspecifikationsspråk för Java följer det att det är Java-klassers gränssnitt, metodspecifikationer, som man kan specificera med JML. Även om invarianter stöds så är det endast vid konstruktoranrop, vid början och vid slut av metodanrop som de granskas.

Vad denna fokus på metoders specifikation innebär är att en specifikation av ett helt Java-program, i vilket klasserna ingår, inte direkt är möjligt med JML. Detta är något som JML inte heller är designat för [13].

## 4.5 Trådar

Den nuvarande versionen av JML fokuserar sig på sekventiellt beteende hos Java-programkod. Det finns inga nyckelord eller konstruktioner i JML för att beskriva hur Java-trådar interagerar [13]. Arbete inom detta område har dock gjorts och det finns förslag till lösningar på problemen som uppstår när design enligt kontrakt används i samband med trådar i [14]. Dessa förslag verkar inte ännu ha blivit medtagna i referensmanualen. JML stöder alltså inte program som innehåller flera trådar.

# 5 Stödverktyg

## 5.1 Allmänt

Som tidigare nämnts, har enbart JML-annotationer i programkod ingen verklig påverkan på programvaran. För att få någon nytta av JML måste man använda något av de många stödverktygen som finns tillgängliga. I den här sektionen kommer följande verktyg samt deras målsättningar att beskrivas: JML-kompilatorn, även kallad körtidsgranskaren (eng. *runtime assertion checker*) [5], den utvidgade statiska granskaren *ESC/Java*, efterföljaren till *ESC/Java*, *ESC/Java2,* och *jmldoc*.

## 5.2 JML-kompilatorn jmlc

Målet med JML-kompilatorn, *jmlc*, är att finna fel i programkod som beror på felaktig implementation med avseende på specifikationen. Verktyget uppnår detta mål genom att möjliggöra granskning vid körtid [5]. *Jmlc* utvecklades vid Iowa State University som ett tillägg till MultiJava-kompilatorn [15][5].

Ett annat viktigt mål för detta verktyg är att användningen är transparent för användaren när inga intyganden bryts, ifall kostnaden i tid och utrymme inte beaktas [5]. Ett felfritt program kompilerat med *jmlc* bör bete sig som om det vore kompilerat med en vanlig Java-kompilator. Detta understöds av det faktum att JML-annotationer inte får ha några följder i programvaran.

Verktyget kompilerar Java-filer annoterade med JML till filer där JML-annotationerna finns med som körtidsintyganden (eng. *assertions*). Detta är likt hur *javac* fungerar med tillägget att granskning vid körtid går att utföra efter kompilering. [16]

JML-kompilatorn skapar en class-fil som av programmeraren kan hanteras likt utfilen från vilken Java-kompilator som helst, med reservation för att de av *jmlc* skapade class-filerna har beroenden till klasser i paket i JML-utgåvan. För att få dessa paket laddade kan kommandot *jmlrac* användas, detta kommando finns med

i JML-utgåvan. *Jmlrac* är ett skript som efter att ha laddat de nödvändiga paketen kör kommandot *java* och i och med detta utför granskning vid körtid. [16]

Problem här är informella specifikationer, som måste antas vara sanna av verktyget och det faktum att verktyget endast kan finna fel i programkod med hjälp av given specifikation. Ifall specifikationen är felaktig (i förhållande till vad t.ex. en kund vill ha) hjälper detta verktyg inte.

Verktyget är ett av de mest använda av alla JML-stödverktyg och stort sett ger det möjligheter åt Java-programmerare att använda JML-specifikationer i praktiken för debugging, testning och för att implementera design enligt kontrakt. [5]

## 5.3 Den utvidgade statiska granskaren ESC/Java och ESC/Java 2

Målet med den utvidgade statiska granskaren *ESC/Java* [17] (*Extended Static Checker for Java*) är, som namnet antyder, att möjliggöra statisk granskning av Java-program. Att verktyget kallas en *utvidgad* statisk granskare beror på att det klarar av mera än andra vanliga statiska granskare så som typgranskare [6]. Verktyget använder sig i grunden av verifikationsvillkorsgenerering (eng. *verification-condition generation*) och automatiska teorembevisningstekniker [6]. Verktyget utvecklades ursprungligen av Compaq Research [5].

Verktyget kan finna fel i relativt enkla JML-intyganden, men också vanliga programmeringsfel såsom användande av index som kan går utöver en räckas längd, möjliga referenser till `null`, eller felaktiga typkonverteringar i vanlig Java-kod. En viktig detalj här är att verktyget inte egentligen kräver att JML används men för mera avancerad granskning är det till fördel. Verktyget stöder endast en delmängd av JML och använder egentligen ett eget specifikationsspråk där en annotationerna kallas *pragman*. [17]

Vid körning av den utvidgade statiska granskaren ger den varningar för potentiella fel som kan uppstå exekvering av det granskade programmet. Den ger också felmeddelanden ifall fel upptäcks vid parsing av programkoden, namnmatchning (eng. *name resolution*) eller vid typgranskning av programkoden och eventuella pragman [17].

*ESC/Java* är inte perfekt, varningar och felmeddelanden behöver inte tyda på några verkliga fel och avsaknaden av sådana betyder inte nödvändigtvis att programvaran är felfri [17]. Detta är ett designbeslut med målet att göra verktyget mera kostnadseffektivt, i och med att fullständig granskning kunde vara väldigt svårt att implementera eller ta lång tid att exekvera i programvara, om fullständig granskning ens är möjligt att uppnå [6].

En intressant detalj med detta verktyg (och statisk granskning i allmänhet) är att det möjliggör detektering av fel i programkod där var de begås och inte där var de upptäcks under körtid [6]. Detta kan leda till ökad produktivitet hos utvecklare av programvara eftersom man direkt kan finna orsaken till felet och inte bara ”symptomen”.

Utvecklingsarbetet av *ESC/Java* upphörde en viss tid efter att Compaq Research blev en del av Hewlett-Packard Labs [5]. Verktygets efterföljare, *ESC/Java2* kommer att beskrivas kort som följande.

*ESC/Java2* bygger på *ESC/Java*s källkod. Ett mål med verktyget var att uppdatera det så att det var kompatibelt med nyare versioner av Java (1.4 vid början av utvecklingsarbetet) [5]. Ett annat mål var också att få specifikationsspråket som verktyget använder att överensstämma bättre med JML, som också utvecklats sedan *ESC/Java* lades ner [18]. En del förbättringar i granskningarna som verktyget utför har också lagts till samtidigt som verktyget är bakåtkompatibelt till en viss grad. Värt att notera är att *ESC/Java2* likt sin företrädare inte är perfekt, utan samma designbeslut som för *ESC/Java* har följts för att hålla verktyget kostnadseffektivt [5].

Ett konkret exempel på användning av verktyget är från år 2004 när ett Internetbaserat röstningssystem skapat för det nederländska parlamentet verifierades [5].

*ESC/Java2* finns tillgängligt i tre former: som ett plugin till Eclipse IDE, som ett kommandotolk-verktyg och inbyggt i programverifieringsmiljön Mobius [19].

## 5.4 Enhetstestningsverktyget jmlunit

Det enklaste sättet att skapa enhetstester är att ha en programmerare att manuellt se på en mjukvaruenhet och utgående från programkoden bestämma vilka testdata som skall användas samt avgöra hur man skall skilja på ett lyckat respektive ett misslyckat test [20]. Att skriva enhetstester för hand är dock väldigt arbetsdrygt, enformigt, besvärligt och ibland rent utav svårt. Dessutom kan förändringar i programkoden kräva att många enhetstester måste skrivas om, vilket ytterligare kräver tid och arbete [21].

För att undvika detta problem vill man automatisera enhetstestning. Detta kan till viss grad möjliggöras med verktyget *JUnit*. Men inte heller med *JUnit* löser man problemet med att skriva om testerna ifall programvaran ändras. Automatisering med enbart *JUnit* kan innebära att man stöter på samma problem som beskrevs ovan. [21]

Målet med enhetstestningsverktyget *jmlunit* är allmänt att möjliggöra enhetstestning av Java-programkod annoterad med JML-specifikationer och speciellt att automatisera testgenereringen. Man vill frigöra programmeraren för att skriva koden som avgör ifall ett test har lyckats eller misslyckats. Verktyget kan ses som en kombination av *JUnit* och JML [5].

Verktygets funktion bygger på generering av JUnit-testklasser och utnyttjandet av granskning vid körtid som möjliggörs av JML-kompilatorn och *jmlrac*. Testklasserna interagerar med de till programvaran hörande klasserna som skall testas genom att göra metodanrop till dem. Beroende på ifall förvillkor eller eftervillkor då bryts (detta upptäcks av körtidsgranskaren, se sektion 5.2) upptäcker man ifall klasserna möter sina specifikationer. Ifall testdata bryter mot ett förvillkor kan testet inte ses som ett misslyckande, i och med att metoden då anropats med felaktiga parametrar och därmed inte är skyldig att ge ett korrekt svar (design enligt kontrakt). Om däremot förvillkor hålls men ett intygande ändå bryts mot (invariant eller eftervillkor) tyder det på att ett fel finns i implementationen av metoden med avseende på specifikationen. Detta registreras som ett misslyckat enhetstest. [5]

Testdata som skall användas i enhetstesterna måste fortfarande skapas av programmeraren men detta underlättas av att testklasser automatiskt genereras av verktyget. Med verktyget följer också ett ramverk som underlättar skapandet av testdata. Även vanliga, manuellt skapade, JUnit-testmetoder kan användas samtidigt för att t.ex. tvinga fram mera komplicerade sekvenser av metodanrop, vilket de automatiskt genererade, specifikationsinriktade, testerna inte nödvändigtvis gör.[5]

Slutligen, enhetstester har länge varit en viktig teknik för validering av programvara i programvaruproduktionsprocesser [20] men skapandet av testerna har varit tids- och arbetskrävande. Det här verktyget underlättar och snabbar upp skapandet av enhetstester för programvara i vilken JML har använts.

## 5.5 Dokumentationsgeneratorn jmldoc

Dokumentationsgeneratorn *jmldoc* fungerar likt *javadoc* [22], den skapar människoläsliga html-sidor från Java-filer. Dessa sidor liknar väldigt mycket motsvarande dokumentation som *javadoc* framställer, med tillägget att även JML-specifikationer finns med.

Html-dokumentation skapad med *jmldoc* är nyttig, eftersom den skapas från en dokumentation som inte är skiljd från programkoden. Detta underlättar arbetet att hålla koden och dokumentationen konsekventa. Html-representationen av dokumentationen är bra för att få en överblick av den eftersom man inte behöver se på flera programkodsfiler utan kan läsa ett dokument som är formaterat på ett läsvänligt sätt och innehåller relevanta länkar för hänvisningar [23]. *Jmldoc* grupperar även JML-specifikationer så att t.ex. specifikationen för en vid arv överladdad metod finns tillgänglig på samma ställe som specifikationen för den överladdande metoden [5].

Verktyget ger alltså ungefär samma funktionalitet som *javadoc*, men erbjuder även ett sätt att få specifikationerna för metoder och klasser tillgängliga på ett bekvämare sätt än enbart i programkodskommentarer. Detta kan tänkas vara nyttigt speciellt ifall vidareutveckling eller underhåll av programkod görs av andra utvecklare än de ursprungliga.

## 5.6 Övriga verktyg

Utöver dessa fyra beskrivna verktyg finns det många andra verktyg med olika mål och användningssätt. Som exempel kan nämnas programverifikationsverktyget *LOOP*, det statiska verifikationsverktyget *JACK* och invariantdetektorn *Daikon.*

*LOOP* [24] är egentligen en kompilator som tar sekventiella Java-program och tillhörande JML-specifikation som indata och ger som utdata flera filer som beskriver betydelsen av Java-programmet och dess specifikation i syntaxen som används av teorembevisaren PVS [25]. Dessa filer kan därefter köras i PVS för att försöka bevisa att implementationen av Java-programmet följer specifikationen.

*JACK* (*the Java Applet Correctness Kit*) [26], är ett verktyg som kan användas för att förbättra kvaliteten speciellt på applikationer för diverse mobila enheter så som mobiltelefoner, handdatorer (eng. *PDA*, *Personal Digital Assitant*) och smartkort (eng. *smartcard*), vilka ofta implementerar JVM (Java Virtual Machine) eller en variant av den. På grund av detta har *JACK* gjorts kapabelt att även hantera Java-bytekod.

*JACK* genererar automatiskt bevisobligationer från Java-källor annoterade med JML vilka kan testas med någon teorembevisare. I verktyget följer ett teorembevis-toolkit med, i vilket bl.a. PVS ingår. Verktyget fungerar alltså likt *LOOP* med skillnaden att det själv kan köra en teorembevisare i bakgrunden, vilket underlättar användandet av verktyget för programmeraren. [26]

*Daikon* [27] är ett verktyg som hjälper till skapandet av specifikation genom att finna möjliga invarianter i programkod. Detta görs dynamiskt, efter exekvering av ett program rapporterar *Daikon* en mängd egenskaper vilka var sanna under exekveringen, av vilka några kan vara verkliga invarianter som kan läggas till i programkodens specifikation.

# 6 Exempelprogram

## 6.1 Allmänt

I det här avsnittet kommer ett exempelprogram att gås igenom. Syftet med exemplet är att illustrera hur ett program kunde specificeras med JML och att visa vad några av de tidigare nämnda stödverktygen ger för utdata när de körs på exempelprogrammet. Vad som kommer att visas är kompilering med *jmlc*, granskning vid körtid med *jmlrac* och generering av dokumentation med *jmldoc*.

Exempelprogrammet innehåller två Java-klasser: `StorageArea` och `StorageAreaDriver`. `StorageArea` (Figur 6.2) är en väldigt simplifierad modell av ett lagerutrymme som har en begränsning enbart i hur stor vikt (av godtycklig last) det kan innehålla. `StorageAreaDriver` (Figur 6.1) är enbart en ”hjälpklass” för att kunna exekvera programmet och därmed göra granskning vid körtid.

```
public class StorageAreaDriver {

      public static void main(String [] args) {
            int storageAreaID = 123;
            int maxWeight = 1400;
            StorageArea a = new StorageArea(saID, maxWeight);

            a.addMateriel(400);
            System.out.println("Weight is now: "+a.getWeight());
            a.addMateriel(800);
            System.out.println("Weight is now: "+a.getWeight());
            a.removeMateriel(200);
            System.out.println("Weight is now: "+a.getWeight());
            a.addMateriel(500);
            System.out.println("Weight is now: "+a.getWeight());
      }
}
```

Figur 6.1: Klassen StorageAreaDriver.

```
public class StorageArea {
	private /*@ spec_public @*/ int storageAreaID;
	private /*@ spec_public @*/ int maxWeight;
	private /*@ spec_public @*/ int currentWeight;

	/*@ invariant currentWeight >= 0
	  @   && currentWeight <= maxWeight;
	  @*/

	public StorageArea(int id, int maxWeight) {
		this.storageAreaID = id;
		this.maxWeight = maxWeight;
	}

	/*@ requires weight >= 0;
	  @ assignable currentWeight;
	  @ ensures currentWeight==\old(currentWeight)+weight
	  @ && \result == currentWeight;
	  @*/
	public int addMateriel(int weight){
		currentWeight = currentWeight + weight;
		return currentWeight;
	}

	/*@ requires weight >= 0;
	  @ assignable currentWeight;
	  @ ensures currentWeight==\old(currentWeight)-weight;
	  @*/
	public int removeMateriel(int weight){
		currentWeight = currentWeight - weight;
		return currentWeight;
	}

	public /*@ pure @*/ int getWeight(){
		return currentWeight;
	}
}
```

Figur 6.2: Klassen StorageArea.

## 6.2 Programbeskrivning

Klassen `StorageArea` innehåller tre privata fält, `storageAreaID`, `maxWeight` och `currentWeight` vilka är specificerade med nyckelordet `spec_public` för att gå att använda i annotationerna i resten av klassen. Det finns en klassinvariant som säger att vikten, `currentWeight`, av godtyckligt materiel som lagras i lagringsutrymmet inte får underskrida noll viktenheter och inte heller överstiga en viss gräns, `maxWeight`.

Konstruktorn har inget annat krav än att invarianten gäller. För metoden `addMateriel` gäller förvillkoret att den formella parametern `weight` inte får underskrida noll viktenheter. Det gäller också att endast fältet `currentWeight` får modifieras. Slutligen säger eftervillkoret att `currentWeight` skall ha värdet av `currentWeight` som gällde när metoden anropades adderat med värdet av parametern `weight` och att returvärdet från metoden skall ha samma värde som fältet `currentWeight`.

Specifikationen för metoden `removeMateriel` är väldigt lik den för `addMateriel` och kommer därför inte att beskrivas i detalj. Slutligen: metoden `getWeight` utmärks av nyckelordet `pure` vilket säger att den inte skall ha några effekter på klassens tillstånd.

Vad klassen `StorageAreaDriver` gör är helt enkelt att initialisera ett `StorageArea`-objekt och därefter köra metoderna `addMateriel` och `removeMateriel` några gånger för att slutligen orsaka ett fel när granskning vid körtid görs.

## 6.3 Kompilering och granskning vid körtid

Kompilering görs med *jmlc* enligt följande:

```
C:\JML\JML\bin>jmlc StorageArea.java StorageAreaDriver.java
```

Och resultatet är tämligen ointressant eftersom inget fel hittades i programkoden eller i annotationerna och en del av resultatet ser ut så här:

```
parsing StorageArea.java
parsing StorageAreaDriver.java
parsing ..\specs\java\lang\Object.jml
parsing ..\specs\java\lang\reflect\Array.refines-spec
...
```

Granskning vid körtid gjordes med *jmlrac* och resultatet är inte oväntat om man studerat programkoden: en avsiktlig felanvändning av klassen `StorageArea` förekommer i det sista `addMateriel`-anropet i klassen `StorageAreaDriver`.

Körning:

```
C:\JML\JML\bin>jmlrac StorageAreaDriver
```

Resultat:

```
Weight is now: 400
Weight is now: 1200
Weight is now: 1000
Exception in thread "main"
org.jmlspecs.jmlrac.runtime.JMLInvariantError: by method
StorageArea.addMateriel@post<File "StorageArea.java", line 15,
character 21> at
StorageArea.checkInv$instance$StorageArea(StorageArea.java:142)
at StorageArea.addMateriel(StorageArea.java:418)
at StorageAreaDriver.internal$main(StorageAreaDriver.java:19)
at StorageAreaDriver.main(StorageAreaDriver.java:292)
```

Körtidsgranskaren upptäckte alltså felet och exekveringen avbröts.

## 6.4 Dokumentationsgenerering med jmldoc

Detta gjordes enligt följande:

```
C:\JML\JML\bin>jmldoc StorageArea.java StorageAreaDriver.java
```

Se figur 6.3 för ett urklipp ur den resulterande html-sidan som genererades.

**Method Detail**

**addMateriel**

```
public int addMateriel(int weight)
```

**Specifications:**

**requires** weight >= 0;
**assignable** currentWeight;
**ensures** this.currentWeight == \old(this.currentWeight)+weight&&\result ==
this.currentWeight;

Figur 6.3: Urklipp från den genererade html-dokumentationen av exempelprogrammet.

Sidan som genererades ser i sin helhet ut precis som en sida genererad av *javadoc*, med specifikationen snyggt tillagd. Allmänt: verktygen fungerar alltså så som beskrivits i tidigare avsnitt. Nämnas bör att en del arbete krävdes för att få dem att fungera, men när det väl var gjort, gav de god feedback på diverse fel som avsiktligt lades till i progamkoden. Exempelprogrammet innehåller dock enbart det nämnda felet.

# 7 Slutsatser

Det formella gränssnittsspecifikationsspråket Java Modeling Language som används för att implementera design enligt kontrakt i Java med hjälp av speciella annotationer i Java-programkod är en av två delar som kan användas för att skapa pålitligare programvara. Den andra, lika viktiga, delen är mängden av alla JML-stödverktyg, vilka behövs för att i praktiken nå detta mål.

Inte enbart pålitligheten hos programvara som kan förbättras med hjälp av JML (och stödverktygen), utan även andra faktorer påverkas. Faktorer så som tiden och arbetsmängden som kan behövas att att få en välfungerande programvara kan minskas t.ex. genom användning av statisk granskning, enhetstestgenering och specifikationsgenerering. Även underhåll av programvara kan stödas med dokumentationen som verktyget *jmldoc* kan skapa.

I grunden bygger dock skapandet av pålitlig programvara fortfarande på att en täckande specifikation finns tillgänglig, så att en implementation som följer denna specifikation kan skapas i Java med hjälp av JML och diverse verktyg. Ansvaret för skrivandet av denna specifikation ligger fortfarande hos utvecklarna av programvaran.

JML utvecklas fortfarande med målet att minska på bristerna språket har, så som avsaknaden av stöd för flertrådade program. Verktygen (åtminstone några av dem) utvecklas också kontinuerligt. Detta kan ses som positivt eftersom användningsområdena därmed kan öka.

Avslutningsvis kan sägas att för de tillämpningar som för tillfället stöds verkar JML vara ett bra språk för Java-programmerare att använda för att specificera Java-klasser och därmed öka pålitligheten hos programvaran som i slutändan byggs upp av dessa klasser.

# Källor

[1] Bertrand Meyer, Applying "Design by Contract", 1992
[2] Bertrand Meyer, Object Oriented Software Construction, Prentice Hall, 1997.
[3] Bertrand Meyer, Eiffel: The Language, Prentice Hall, 1992.
[4] About JML, http://www.eecs.ucf.edu/~leavens/JML/, Hämtad: 18.2.2011
[5] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, Erik Poll, An overview of JML tools and applications,
[6] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, Raymie Stata, Extended Static Checking for Java, 2002
[7] Gary T. Leavens, Yoonsik Cheon, Design by Contract with JML, 2006
[8] Benjamin Weiss, Deductive verification of Object-Oriented Software, 2011
[9] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll, Beyond Assertions: Advanced Specification andVerification with JML and ESC/Java2, 2005
[10] Barbara Liskov, Data Abstraction and Hierarchy, 1987
[11] Ian Sommerville, Software Engineering, Addison-Wesley, 2007.
[12] Vincent Massol, Ted Husted, JUnit in Action, Manning Publications Co, 2003.
[13] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon,Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry,Patrice Chalin, Daniel M. Zimmerman, Werner Dietl, JML Reference Manual (DRAFT), 2011
[14] Wladimir Araujo, Lionel Briand och Yvan Labiche, Concurrent Contracts for Java in JML, 2008
[15] Curtis Charles Clifton, MultiJava: Design, implementation, and evaluation of a Java-compatible language supporting modular open classes and symmetric multiple dispatch, 2001
[16] JMLC, http://www.eecs.ucf.edu/~leavens/JML-release/docs/man/jmlc.html, Hämtad: 14.3.2011
[17] ESC/Java User's Manual, http://sort.ucd.ie/products/opensource/ESCJava2/ESCTools/docs/ESCJAVA-UsersManual.html, Hämtad: 15.3.2011
[18] David R. Cok och Joseph R. Kiniry, ESC/Java2: Uniting ESC/Java and JML, 2005
[19] ESC/Java2 Summary, http://secure.ucd.ie/products/opensource/ESCJava2/, Hämtad: 17.3.2011
[20] Daniel M. Zimmerman, Rinkesh Nagmoti, JMLUnit: The Next Generation,
[21] Yoonsik Cheon, Gary T. Leavens, The JML and JUnit Way of Unit Testing and its Implementation, 2004
[22] Javadoc Tool, http://www.oracle.com/technetwork/java/javase/documentation/index-jsp-135444.html, Hämtad: 21.3.2011
[23] Package org.jmlspecs.jmldoc , http://opuntia.cs.utep.edu/utjml/jml-javadocs/org/jmlspecs/jmldoc/package-summary.html, Hämtad: 21.3.2011
[24] Bart Jacobs, Erik Poll, Java Program Verification at Nijmegen: Developments and Perspective, 2004
[25] S. Owre, S. Rajan, J. M. Rushby, N.Shankar and M. Srivas, PVS: Combining

Specification, Proof Checking, and Model Checking*, 1996
[26] Gilles Barthe, Lilian Burdy, Julien Charles, Benjamin Grégoire, Marieke Huisman, Jean-Louis Lanet, Mariela Pavlova och Antoine Requet, JACK — a tool for validation of security andbehaviour of Java applications, 2007
[27] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco,Matthew S. Tschantz, Chen Xiao, The Daikon system for dynamic detection of likely invariants, 2007