

# Integration of Java Generics Into The `jmle` Tool Within The Eclipse IDE

by  
Adrian Kostrubiak

Submitted in partial fulfillment of Honors Requirements  
For the Computer Science Major,  
Dickinson College, 2008-09.

Professor Tim Wahls, Supervisor.  
Professor Grant Braught, Reader.  
Professor John MacCormick, Reader.

May 11, 2009.

The Department of Mathematics and Computer Science at Dickinson College hereby accepts this senior honors thesis by Adrian Kostrubiak, and awards departmental honors in Computer Science.

---

Professor Tim Wahls (Advisor)

---

Date

---

Professor Grant Braught (Committee Member)

---

Date

---

Professor John MacCormick (Committee Member)

---

Date

---

Professor David Richeson (Department Chair)

---

Date

Department of Mathematics and Computer Science  
Dickinson College

May 2009

## Abstract

# Integration of Java Generics Into The `jmle` Tool Within The Eclipse IDE

by  
Adrian Kostrubiak

The Java Modeling Language (JML) is a mathematical notation that can be included in Java programs, often through the use of in code annotations. JML is used to formally define the behavior of Java classes and modules. Due to their nature, formal specifications cannot actually be executed. However, the `jmle` tool created by Professor Tim Wahls has allowed the execution of these formal specifications through converting them into constraint programs. Unfortunately, previous to this work, the `jmle` tool was not compatible with many of the newer Java 5 JDK features, such as generics and `enum` types. The goal of this research is to bring the functionality of generics into the `jmle` tool. Although there were numerous unexpected complications, limited generic functionality has been brought to the `jmle` tool. The aforementioned success has shown that the full integration of generics into the `jmle` tool should not require much more implementation. However, for the `jmle` tool to acquire complete Java 5 JDK compatibility, more research needs to be performed, especially in the areas not researched in this paper (for example, `enum` types).

## **Acknowledgements**

Firstly, I would like to thank Professor Tim Wahls, my advisor on this project. Without his help and the many hours he has already spent working with JML and the `jml` tool, my project could never have been possible. Secondly, I would like to thank Professors Grant Braught and John MacCormick, both of whom were gracious enough to sign up as readers for this thesis. Lastly, I would like to thank my peers, my family and all of my friends who have supported me while I have spent many hours working on this massive project.

## Table Of Contents

Title Page	.....	i
Signature Page	.....	ii
Abstract	.....	iii
Acknowledgements	.....	iv
Table Of Contents	.....	v
<b>Chapter 1: Introduction</b>	.....	<b>1</b>
1.1. Introduction	.....	1
<b>Chapter 2: Background Information</b>	.....	<b>3</b>
2.1. Java Generics	.....	3
2.2. Design By Contract Design Pattern	.....	3
2.3. Java Modeling Language (JML)	.....	3
2.4. JML Model Classes	.....	7
2.5. The <code>jm1e</code> Tool	.....	10
2.6. The <code>jm1e</code> Tool In Eclipse	.....	11
2.7: Relevant Literature	.....	11
2.7.1. <code>jm1e</code> : A tool For Executing JML Specifications Via Constraint Programming	.....	11
2.7.2. JACK: A Java Constraint Kit	.....	12
2.7.3. Executing JML Specifications of Java Card Applications: A Case Study	.....	15
<b>Chapter 3: Design Decisions</b>	.....	<b>16</b>
3.1. Two Versions Of JML Model Classes	.....	16
3.2. Parameter Types On <code>equals ( )</code> Methods	.....	17
3.3. Genericizing Static Methods	.....	19
3.4. Immutable Iterators	.....	20
<b>Chapter 4: Results</b>	.....	<b>23</b>
4.1 Implementation Changes Supporting Generics in Specifications	.....	23
4.2. JML Model Classes	.....	23
<b>Chapter 5: Conclusions</b>	.....	<b>26</b>
<b>References</b>	.....	<b>27</b>

# Chapter 1

## Introduction

### **1.1 Introduction**

In the recent movement towards smarter and more efficient software development, many software engineers have been looking for an effective means to verify the correctness of the software modules that they have written. One of the directions that this search has taken is in the area of formal specifications, which are mathematical notations describing what a module should do as opposed to how it should complete its task. Due to the popularity of Java and the lack of a formal specification language (for use in Java programs), the Java Modeling Language (JML) was created.

Numerous tools have been created to work with JML, including tools for runtime assertion checking and general code documentation (Leavens et al., 2000), amongst many others. Although JML and the accompanying suite of tools are a powerful toolset for a developer, a few large hurdles remain to be dealt with before some of the associated tools can effectively be used in industry. For instance, some tools lack the implementation of generics, which were added to Java with the release of the Java 5 JDK. Other researchers working on the JML4 project have also outlined the need for generic capabilities (Cok, 2008; Krause & Wahls, 2007), further providing evidence of the necessity of this work for the success of the JML4 project.

To this end, the goal of this research has been focused on the integration of Java generics into the `jmlc` tool, a tool that is used for creating executable versions of the specifications. This paper aims to document the process of adding generic functionality into

the jml<sub>e</sub> tool, specifically through the discussion and justification of the design decisions that have been made in the generification of the jml<sub>e</sub> tool.

An overview of the relevant technologies will be presented first, followed by a brief overview of some of the more relevant literature. After this, a discussion of design decisions will be presented. Following the discussion of design decisions, the initial results will be presented. Finally, conclusions and the consequences of the aforementioned results will be presented.

## **Chapter 2**

### **Background Information**

#### **2.1 Java Generics**

In Java, generics were added to the platform with the release of the Java 5 JDK. Since this addition, generics have become an indispensable tool in Java programming, allowing for methods to operate on variables of any class type while at the same time supporting compile time type checking. This ultimately results in fewer run time errors.

#### **2.2 Design By Contract**

Design by contract is an approach to developing software. This approach dictates that software engineers should define formal specifications on code modules. These specifications are based on the idea of a business contract. In a business contract between a client and supplier, each party has certain obligations and benefits. To extend this metaphor of the business contract to software, any given module has obligations and benefits. The contract hinges on these obligations and benefits, which can be summarized through three questions: what does a module expect? What does a module guarantee? And lastly, what does the module maintain? The specifications are simply a formalization of the informal idea of a contract for any given software module (Cheon & Leavens, 2006).

#### **2.3 Java Modeling Language**

The Java Modeling Language (JML) is a mathematical notation that can be included in Java programs, often through the use of in-code annotations. JML is used to formally

specify the behavior of modules, such as a method, in a Java program (Leavens et al., 2008). As such, JML follows the ideal of design by contract. In following the design by contract design pattern, any methods specified with JML should be constrained by both pre and post conditions (assuming the code is correct and the method upholds its specified contract). The following is a trivial example of a Java method and its JML specification:

```
/*@ assignable \nothing;                               (2.1)
   ensures \result == j + k; */

public int sumOfTwo(int j, int k) { return(j+k); }
```

In this example (2.1), the `assignable` clause tells us that this method does not assign any of instance variables defined by the class. The `ensures` clause tells us that the result (that is, the return value) is going to be the sum of `j` and `k`. There is no `requires` clause, as most methods would have, as this `sumOfTwo` method has no preconditions to be satisfied. However, should one wish to implement the `sumOfTwo` method with preconditions, it would be as easy as simply adding a `requires` clause into the JML specification before the method, such as `requires j > 0 && k > 0;`. This would change the method's contract such that it would mandate that the method be called with positive integers only rather than any possible integer.

Example 2.1 is a relatively basic and trivial one, however a more realistic example can be seen in example 2.2:

```

public class GenericArray<T> {
    public T[] elems;
    public int next;
    // further details omitted

    /*@ requires t != null &&
        next < elems.length - 1;
        assignable elems[*], next;
        ensures next == \old(next) + 1 &&
            elems[\old(next)] == t &&
            (\forallall int i; 0 <= i && i < \old(next);
                \old(elems[i]) == elems[i]);
    */
    public void add(T t) { /* implementation omitted */ }
    // further details omitted
}

```

(2.2)

This is an example of the generic class `GenericArray` of the generic type `T` with an array used as the underlying data structure that is also of the generic type `T`. The specification present in this example describes the contract for the `add` method of the `GenericArray` class. As described in section 2.2, this contract details the obligations and benefits of a specified method, as well as what is not maintained by the method in question. In detailing the pre-conditions with the use of the `requires` clause, we have specified what benefits the method can expect. Likewise, by specifying the post-conditions with the use of the `ensures` clause, we have specified the obligations of the method. And lastly, with the use of the `assignable` clause, we have specified what this method does not maintain (ie: what it changes).

In the specification from example 2.2, we can see that the `add` method can only be called if the parameter is not null and if the `next` pointer (a pointer determining the next open position in the array `elems`) is at a legitimate position in the array (so as to avoid an `ArrayIndexOutOfBoundsException` exception). In the `assignable` clause, the specification details that both `elems[*]` (all elements in the array) and `next` (the next open array index pointer) will be modified by this method. Finally, the `ensures` clause is broken up into

three separate sections in order to fully specify the post-conditions that should hold true after the execution of the `add` method. The first part of the `ensures` clause specifies that the `next` pointer should be increased by one with respect to its previous value. The second part of the `ensures` clause specifies that the first open position in the array `elems` (the previous value of the `next` pointer) should be equal to the object to be inserted, or the parameter used in the call to the `add` method. This is equivalent to saying that the object was actually inserted into the first open position in the array. The final part of the `ensures` clause specifies that every position in the `elems` array up to the position at which the new element is to be inserted (that position being the old value of `next`, `\old(next)`) has maintained its value. This is done by checking that the current value at any position (`elems[i]`) is equal to the old value at this position (`\old(elems[i])`). Thus by formally specifying this `add` method, any developer can look at the specification and understand exactly what the pre and post-conditions for the method are. By understanding the contract of the `add` method, developers should be able to easily create their own implementations of this method.

It is also useful to note that JML has a number of quantifiers and operators in the language, each of which serves a different purpose. For example, JML contains a universal quantifier and an existential quantifier, `\forall`, and `\exists`, each of which is very useful in specifications. Amongst the operators in JML, one of the most highly used is the `\old` quantifier. This operator is used to refer to the value of the enclosed expression in the pre-state of the specified method. Although these are just some of the available quantifiers and operators in JML, there are numerous other quantifiers and operators available for use in specifications.

Aside from checking contracts, JML has a multitude of additional uses. Amongst these other uses of JML are general code documentation, runtime assertion checking, and the ability to create proofs of program correctness based on the JML specifications (Leavens et al., 2000).

## 2.4 JML Model Classes

One of the key components to the JML4 project is JML model classes. This is a collection of classes that are used to represent the type of traditional mathematical notation that one would have found in previous specification languages not associated with Java. Such mathematical notations include, but are not limited to: sets, sequences, bags and mathematical functions. One of the goals of JML is to have a very Java-like syntax, both making the learning curve an easy one and as well allowing Java programmers to easily pick up this formal language due to the familiarity of the syntax. These model classes are the Java implementations of the traditional mathematical notations, allowing JML to continue to use a syntax very similar to Java's own syntax.

There are a number of different model classes, however the ones that I have worked with the most in order to genericize, are the classes representing sets, bags and sequences, and the associated node classes, as well as the associated implemented interfaces. These classes are `JML#Set`, `JML#Bag`, `JML#Sequence`, `JMLList#Node`, where `#` is one of `Object`, `Equals`, or `Value`. These classes are all collection classes, and that being the case, have many of the same methods. These sort of collection methods include, but are not limited to: `has()`, `size()`, `insert()`, `remove()`, and `equals()`. As well, there are

various utility methods that are used convert a collection of one sort to a collection of another type. For example, in the `JMLObjectSet` class, there are methods to convert to a bag and to convert to a sequence. Lastly, these classes also include methods that are used to perform various comparisons with the collections. Such methods include, but are not limited to: `isSubBag()`, `isSuperSet()`, and `isSubBag()`.

As well, it is worth mentioning that there is a subset of the model classes that is designed to exclusively contain objects implementing the `JMLType` interface. The `JMLType` interface simply specifies that any class which implements this interface must implement at a minimum the `clone()`, `equals()`, and `hashCode()` methods. The classes which have been created to hold object implementing `JMLType` are all of the form `JMLValue#`, where `#` is one of `Set`, `Bag`, or `Sequence`.

The JML model classes are especially useful because they can be used in specifications. These classes were designed with this use in mind, and as such they have been designed as immutable, or pure classes. This is due to the fact that only pure classes can be used in specifications because these sorts of mathematical assertions (the specifications) do not have side effects. For an example of one of the JML model classes being used in a specification, see example 2.3.

```

public class IntList {
    //@ public model JMLOBJECTSEQUENCE<Integer> theList;

    /*@ assignable theList;
       ensures (theList.first()).intValue() == e &&
              theList.trailer() == \old(theList); */
    public void addElem(int e) {}

    /*@ assignable theList;
       ensures theList.int_size() == \old(theList.int_size()) &&
              (\forall Integer i; \old(theList.has(i));
               theList.count(i) == \old(theList.count(i))) &&
              (\forall int j; 0 <= j &&
               j < \old(theList.int_size()) - 1;
               (theList.itemAt(j)).intValue() <=
                (theList.itemAt(j + 1)).intValue()); */
    public void sort() {}
}

```

This example is a small section of a class called `IntList`. For the sake of space, other methods and the implementations have been omitted. In this example, there are two methods which have been specified using JML: `addElem()`, and `sort()`. As we can see, the specifications are also making use of a model object (an object that can only be used in specifications), `theList`, which is of the type `JMLOBJECTSEQUENCE<Integer>`, in order to more thoroughly specify the underlying data structure of this `IntList` class.

In specifying the `addElem()` method, we declare that the model object, `theList`, will be changed by this method. The post-condition for this method is that the first item in the list (`theList.first()`) has the same integer value as the parameter to the method call (`e`), and that the trailer of the list (everything after the first element) is the same as the old version of the list (`\old(theList)`), before the `addElem()` method was run.

In specifying the `sort()` method, again we declare that the model object, `theList`, will be changed by this method. The post-conditions for the `sort()` method are that the size of the list must remain constant (`theList.int_size() == \old(theList.int_size())`), that the list contains all the same elements as it did

```
before the execution of this method (\forall Integer i; \old(theList.has(i));
theList.count(i) == \old(theList.count(i)), and lastly that every single object
is less than or equal to the next object in the list ((\forall int j; 0 <= j && <
\old(theList.int_size()) - 1; (theList.itemAt(j)).intValue() <=
(theList.itemAt(j + 1)).intValue())).
```

Lastly, it is worth noting that before the implementation of generics into the `jmlc` tool, this specification for the `sort()` method utilized a number of type casts (casting objects to be of type `Integer`). However, now that generics are supported in this tool (and the JML model classes), the type casts are completely unnecessary. Therefore, it is clear that by implementing generic capabilities in the `jmlc` tool, specifications can now be written with type-safe code no longer requiring type casts.

## 2.5 The `jmlc` Tool

The `jmlc` tool was designed to turn JML specifications into constraint programs. As JML is a formal specification language, these specifications themselves are not executable; however the constraint programs created through the use of the `jmlc` tool can be executed as normal Java byte code utilizing the libraries of the Java Constraint Kit (JACK) (Abdennadher et al., 2002) (for further information on JACK, see section 2.7.2). Having an executable formal specification for an application greatly simplifies the development process for software engineers. With the ability to actually execute the specification, the developer is not only able to confirm the meaning of the specification, but can also validate whether or not any given software module fits its specification. Furthermore, having an executable version of the specification greatly simplifies the process of debugging the specification in order to determine its correctness (Wahls and Krause, 2007).

## 2.6 The `jmlE` Tool In Eclipse

In its current version, the `jmlE` tool has been ported into the Eclipse architecture (Eclipse being a very popular and extremely powerful Java based integrated development environment (IDE) (Eclipse Foundation, 2009)) through the work of Professor Tim Wahls. Many other researchers are currently working on the JML4 project, which is the integration of JML into Eclipse (Chalin et al., 2007), but Professor Tim Wahls is the only one to be working with the `jmlE` tool. Within this Eclipse-based context, when a program is run via `jmlE`, the program's code is not actually being executed; rather, the `jmlE` tool translates the JML specifications from the methods in this program into a constraint program. Then these constraint programs are executed utilizing the JACK library (Abdennadher et al., 2002). The constraint program can then be run in its own right and checked for accuracy. The developer's ability to check the meaning of the specification against what was truly meant through this type of "hands on" technique not only makes developing the specification easier, but as well greatly simplifies the very understanding of the specification.

## 2.7 Relevant Literature

### 2.7.1 *jmlE: A tool For Executing JML Specifications Via Constraint Programming*

The `jmlE` tool is a Java based tool created by Professor Tim Wahls and Ben Krause ('08). This tool is an adaptation of the earlier `jmlC` tool (a JML tool to create code that performs runtime assertion checking (Burdy et al., 2005)) in order to compile JML specifications to constraint programs. These constraint programs are then executable via the Java Constraint Kit (JACK) (see section 2.7.2 for further information about JACK)

(Abdennadher et al., 2002). This is a powerful tool, though unfortunately it lacks implementation of many of the new and powerful features introduced into Java with the release of the Java 5 JDK. Amongst the lacking features is the capacity to deal with generics. To this end, I have added generic functionalities to this tool over the course of my honors project.

### 2.7.2 *JACK: A Java Constraint Kit*

Constraint programming has made substantial progress in recent years. It is now at a point at which many of the essential features are available as libraries, or these features are even embedded in some languages. The need for a constraint library in the Java language is underscored by the ever-growing popularity of the language. To this end, a group of researchers created the Java Constraint Kit (JACK). JACK is made up of three main parts: JCHR, VisualCHR, and JASE. In this system, JCHR is a high level language used to write the constraint handling rules, VisualCHR is a tool to aid in the visualization of these rules, and JASE is an abstract search engine used by JCHR to solve the constraint problems (Abdennadher et al., 2002).

In any constraint system, the ultimate goal is to remove all the constraints (in the form of relations between variables) from the constraint store. This constraint system has two stores, one of user-defined constraints and the other of built-in constraints. When a user-defined constraint (from the JCHR file) is activated from the store, the constraint checks for applicability in all of the rules that it appears in (from the `rules` section of the constraint handler). The rules are used in this system to simplify the constraint store through the propagation and simplification of constraints. Depending on the type of the firing rule,



```

handler leq {
  class IntUtil;
  constraint leq(java.lang.Integer, java.lang.Integer);
  rules {
    variable java.lang.Integer X,Y,Z;
    {leq(X,X)} <=> {true}          reflexivity;
    {leq(X,Y) && leq(Y,X)} <=> {X=Y}  antisymmetry;
    {leq(X,Y) && leq(Y,Z)} ==> {leq(X,Z)} transitivity;
    {leq(X,Y) &\& leq(X,Y)} <=> {true}  idempotence;
    if (IntUtil.ground(X) && IntUtil.ground(Y))
      {leq(X,Y)} <=> {IntUtil.le(X,Y)}  ground;
  }
  goal g1 {
    variable java.lang.Integer X,Y,Z;
    leq(X,Y) && leq(Z,X) && leq(Y,Z)
  }
}

```

This small example shows a constraint system for the constraint less than or equal. The constraint itself is defined using the keyword `constraint` and expects two parameters, both of type `Integer`. Within the `rules` section of this constraint handler, we declare three variables, all of type `Integer`, which are to be used within the constraint rules. The rules themselves define reflexivity, antisymmetry, transitivity, idempotence, and a ground rule (that is, the value of the variable is known).

The goal section of this constraint handler is where the constraints are solved. The first (`leq(X,Y)`) and second (`leq(Z,X)`) constraints cause the transitivity rule to fire, which adds `leq(Z,Y)` into the constraint store. When coupled with the third rule, `leq(Y,Z)`, this newly added rule causes the antisymmetry rule to fire, which therefore replaces these two constraints with `Z=Y`. This new equality can now be applied to the remaining goals, `leq(X,Y)` and `leq(Z,X)`, which results in the constraints `leq(X,Y)` and `leq(Y,X)`. Again, the antisymmetry rule can be applied here, and doing so results in the equality `X=Y`. Since the goal contains no more constraints or inequalities, the process of

solving the constraints is stopped, and we are left with the results of  $X=Y$  and  $Y=Z$  (Abdennadher et al., 2002).

### 2.7.3 *Executing JML Specifications of Java Card Applications: A Case Study*

This article is about the use of the `jmlc` tool in order to execute JML specifications in a Java Card application. Java Card is a dialect of Java that was designed to run in situations where resources are extremely limited, such as on mobile phones. The goals for this study were to determine the actual efficacy of the `jmlc` tool on a real, production level program, and one that is moderately large and rather complex at that (Cataño and Wahls, 2009). Overall, the project was a success, leading to changes in both the implementation of the `jmlc` tool and in the specifications found in the Java Card electronic purse application. Some were problems encountered, including the significant amount of time needed to diagnose problems. This was partly due to the lack of information in error messages from the `jmlc` tool, wherein the only message given on failure was that the failure came in evaluating either the `requires`, `modifies`, or `ensures` clauses.

This article is useful in determining what changes can be made to the `jmlc` tool to make it a more powerful and fully useable tool, especially as it relates to actual use in industry and with larger and more complex applications.

## Chapter 3

### Design Decisions

#### **3.1 Two Versions of JML Model Classes**

At the outset of this project, a need to maintain the previous, non-generic versions of the JML model classes was speculated. This was thought to be necessary in order to maintain backwards compatibility with the large base of already existing JML specifications. However, research into the implementation of generics in Java showed that this is emphatically not the case.

When a compiler is generating code for generic methods, there are two possible options for the generated byte code: the first option is code specialization, whereupon the compiler creates specific executables for every single instantiation using a different type parameter. The second option is that the compiler creates one executable that is to be shared by every possible generic type. Clearly, the first option is impractical due to quickly bloating executable size (although interestingly enough this is the approach taken by C++ (Langer, 2009)), and it follows that the Java compiler follows the second strategy, creating only one executable that is to be used by every possible generic type.

The process through which the compiler is able to create only one executable is called type erasure. The compiler goes through the generic code, and removes any generic types. The removed generic types are then replaced with their uppermost bound. In most cases, this means that the generic type will simply be replaced with `Object`, as every single object inherits from `Object` and has `Object` as its uppermost bound (Langer, 2009). Through

this process, it is easy to see that once compiled, a class `JMLObjectBag<String>` uses the same executable as `JMLObjectBag<Object>`. Likewise using the parameterized type of `Object` would be equivalent to using a non-parameterized type of the class. As well, it is worth mentioning that using a non-parameterized instance of the class, such as a `JMLObjectBag` is equivalent to using a generic instance of the class that has been simply parameterized with `Object` (ie: `JMLObjectBag<Object>`). Therefore it clearly follows that there is no need to have two versions of the JML model classes in order to maintain backwards compatibility; the generic versions of these classes will satisfy the needs of both generic uses as well as non-generic uses of these classes.

### 3.2 Parameter Types On `equals()` Methods

In the generification of the JML model classes, the question of collection equality came up, specifically with regards to the implementation of the `equals()` methods. Can two collections with differing actual type parameters possibly be equivalent? A concrete example of this situation would be asking whether a `JMLObjectSet<String>` could possibly be equivalent to a `JMLObjectSet<Object>`.

Initially, we speculated that the correct choice for the method's formal parameter would be to use the type of the class, with the parameter. An example of such an implementation, within the `JMLObjectSequence<T>` class can be seen in example 3.1.

```
public boolean equals(JMLObjectSequence<T> obj) {...} (3.1)
```

However, we quickly recognized that this implementation yielded unwanted results. This design left us in a scenario where it would only be possible to call the `equals()` method using a parameter of the same type as the calling object. This would therefore mean that it would be impossible to determine whether a `JMLObjectSequence<String>` would be

equal to an `Integer`. Although clearly this method call would return false, we could not actually execute it, as the `equals()` method had been defined (in this specific example) to only allow `JMLObjectSequence` objects that had been parameterized with `Strings` as parameters (assuming that the instantiation of the `JMLObjectSequence` was parameterized with `Strings`).

Coming back to the initial question of whether two collections with differing actual type parameters could possibly be equivalent, research showed that in fact this scenario is possible. That is to say that it is possible for two collections, parameterized with different types, to be equal. An example of this can be seen in example 3.2. However, the scenarios in which two collections with differing parameterized types could be equal are rather contrived, and are unlikely to occur in most programs. Nevertheless, the results of this research are that the parameter of the `equals()` method should be of type `Object`, not the type of the class, with the parameter. Further backing this decision is the fact that this design corresponds with the design of the `equals()` methods in Java's own collection classes (those found in the `java.util` package).

```
JMLObjectSet<String> str = new JMLObjectSet<String>(); (3.2)
JMLObjectSet<Object> obj = new JMLObjectSet<Object>();

Object o1 = "one"; Object o2 = "two";
String s1 = "one"; String s2 = "two";
str = str.insert(s1); str = str.insert(s2);
obj = obj.insert(o1); obj = obj.insert(o2);

System.out.println("JOS<String> equal to JOS<Object>? "
    +str.equals(obj));
System.out.println("JOS<Object> equal to JOS<String>? "
    +obj.equals(str));

/* Output:
JOS<String> equal to JOS<Object>? true
JOS<Object> equal to JOS<String>? true */
```

Furthermore, the same logic and thought processes applying to the formal parameter types on `equals()` methods in the JML model classes applies to the `has()` methods of these classes.

### 3.3 Genericizing Static Methods

Static methods are methods in classes that can be invoked without a specific instance of the class that contains it. There are multiple static methods throughout the JML model classes, and in adding generic capabilities to the JML model classes, all of these methods required generification as well. However, genericizing these static methods proved more difficult than expected. This trouble comes from the fact that non-static variables, including type parameters, cannot be accessed from static contexts because non-static variables only exist when there is an instance of the class.

A good example of one such static method can be found in the JML model class `JMLObjectSequence<E>`. In this class there is a static `singleton()` method, which is used to create a new `JMLObjectSequence` with the supplied element (the single parameter). Although logically it would seem reasonable to use the parameterized type of the class, `E` (as seen in example 3.3), as the type of the formal parameter for the method call, this is unfortunately not allowed by Java syntax due to the fact that non-static classes (and variables) cannot be referenced in static contexts (which this is) because of the aforementioned reasons.

```
public class JMLObjectSequence<E> {                                     (3.3)
    // further details and implementation omitted
    public static JMLObjectSequence<E> singleton(E obj) {}
    /* This fails, as non-static E is being referenced in a
       static context, which is not allowed in Java */
}
```

Some research into the finer details of the implementation of generics in Java yielded interesting results: just as a class or variable can be generic, so too can a method be generic (Langer, 2009). One can declare a method with a formal type parameter, therefore genericizing the method with the supplied parameter. A reimplementaion of example 3.3 with the `singleton()` method being declared generic can be seen in example 3.4:

```
public class JMLObjectSequence<E> {                               (3.4)
    // further details and implementation omitted
    public static <F> JMLObjectSequence<F> singleton(F obj) {}
    /* This compiles fine and produces the expected results */
}
```

In this example, the `singleton()` method has been declared as a generic method with the inclusion of the generic type parameter, `<F>`, before the return value of the method.

### 3.4 Immutable Iterators

The JML model classes have always been immutable, or by other nomenclature, pure. This is to say that once an object has been instantiated, it cannot be changed in any way. The impetus for such a design decision lies in the fact that only immutable methods can be used in formal specifications. Such a design decision has many implications. One example of the implications that an immutable object design can have on a class can be seen easily if we are to examine any of Java's collection classes (these classes are certainly not immutable, however they serve as a good example for the implications of immutability). In the normal Java implementation of `LinkedList` (that being the non-immutable implementation), the `add()` method returns either a `boolean` (dependent on success of the `add()` call), or simply `void`, depending on the parameter(s) used when calling the `add()` method. However, if we wanted to create an immutable version of this `add()` method, this clearly poses certain problems, as adding an object to a `LinkedList` is by definition changing the

original object. Thusly, in an immutable version of the `LinkedList`, the `add()` method would have to produce no side effects. This is accomplished by having the `add` method instantiate a new copy of the `LinkedList` containing all the elements from the previous `LinkedList` and of course the new element to be added. This in turn means that the `add()` method must have a return type of `LinkedList`, as opposed to either `boolean` or `void`.

All the model classes have iterators in order to be able to iterate through the elements of the collection in a straightforward manner. In converting the model classes into generic model classes, the question of immutable iterators was raised: should iterators be immutable, and if so, how exactly would this work? Preliminary work by other researchers associated with the JML4 project had the outlined generic model classes with iterators that were as well immutable (Cok, 2008). After considering this issue for quite a while and discussing it with one of the researchers (David Cok), it was decided that immutable iterators were not only unnecessary, but as well simply an unreasonable as a concept. As iterators inherently have side effects, they cannot possibly be immutable. Amongst the reasons for which it was decided that iterators need not be immutable is included the fact that should one ever want to use an iterator in a specification, it would be just as easy, if not even easier to simply use the JML universal quantifier, `\forall`. This effectively covers the use of the iterator within specifications, and makes the use of an iterator in a specification completely unnecessary. An example of the JML universal quantifier being used in place of an iterator can be seen in example 3.5. However this is not to say that there is no need for iterators in the JML model classes, as they can be used in a variety of other applications.

```

/*@ requires s != null;                                     (3.5)
   assignable \nothing;
   ensures \result == (\forall int j; j >= 0 &&
      j < \old(s.int_size()); s.itemAt(j).intValue() > 3);
*/
public boolean allGT3(JMLObjectSequence<Integer> s) {}
/* Implementation omitted. This method will take a
JMLObjectSet<Integer> and return whether or not all the
elements of the supplied JMLObjectSet are greater than 3 */

```

The implementation of the iterators for the JML model classes was done via an inner class. The inner class design was chosen due to the highly coupled relationship between the iterator and the related collection. Further, using the inner class design allows for greater encapsulation. Lastly, from a purely practical standpoint, an iterator for a given collection, for example `JMLObjectSet`, is only useful to the `JMLObjectSet` class and no others. In this way, the use of the inner class design helps reduce clutter in the JML model class package. It is also useful to note that even though these inner class iterators are mutable, this does not change the immutability of the encapsulating model class, and therefore it can still be used in JML specifications, which is one of the goals for the JML model classes.

## Chapter 4

### Results

#### **4.1 Implementation Changes Supporting Generics in Specs**

One of the first things necessary to complete for the successful implementation of generics in the `jmlc` tool was to implement changes in the underlying Eclipse code, which is used by the `jmlc` tool to translate specifications into executable programs. The code in question, that being the `ExecVisitor` class, was originally written without consideration of generics, and thus the changes implemented in this class were done in order to allow for the use of generics in specifications. Professor Tim Wahls originally wrote this class, and as such it is not part of the base JML4 project. It is this class that is primarily responsible for the translation of JML specifications into constraint programs. Due to the fact that the `ExecVisitor` class was originally written without consideration for generics, upon initial testing with generics, multiple syntactical errors were encountered with the generated executables (that being the constraint programs). In the created executable, many methods were found that were using the full name of the class, including the generic type parameter(s), as a prefix for the full method name. Fixing these syntactical errors in order to allow for the use of generics in specifications necessitated numerous small code changes.

#### **4.2 JML Model Classes**

As mentioned previously (in section 2.4), the JML model classes play an integral role both for the `jmlc` tool and as well for the entire JML4 project. As such, one of the biggest

requirements for adding generic functionality to the `jmlc` tool is the creation and testing of generic versions of these model classes.

Initially, just two of the most frequently used of the model classes, `JMLObjectSet` and `JMLObjectSequence`, were both successfully genericized, along with the necessary node class, `JMLListNode`, as well as all the directly related interfaces, super classes, and so forth. After successfully creating the generic versions of these two model classes, both were thoroughly tested and adjusted to be sure that the implementations were correct using the `jmlc` tool within the Eclipse environment. This process continued until both of these classes were able to successfully pass a rigorous test suite, ensuring the correctness of their respective implementations.

The creation and testing of these two generic model classes proved that creating the rest of the model classes via some sort of template and related script should be relatively straightforward, due to the nature of many of the JML model classes being very similar to one another and the fact that many of these classes have common interfaces. A template for the automatic creation of non-generic versions of the model classes via shell scripting had already existed. Therefore, we were able to successfully genericize the majority of the most highly used classes from the JML model classes by utilizing the same design patterns from the previously implemented classes (these being `JMLObjectSet` and `JMLObjectSequence`) and applying these to the template files. Some minor changes were necessary in order to have correct implementations for some of the classes in this hierarchy, such as `JMLValueSet` and `JMLValueSequence`. This is due to the fact that these classes were designed specifically to hold objects that exclusively extend the `JMLType` interface.

However, beyond minor complications arising from the issues from the `JMLValue#` (where `#` is one of `Set`, `Sequence`, or `Bag`) classes, the implementation of the JML model classes via templates and their related shell scripts has been successful. The correctness of these classes has been further validated by running a very thorough suite of tests on these new generic JML model classes.

## Chapter 5

### Conclusions

#### **5.1 Current Progress and Moving Forward**

All in all, after many a stray path, this project has been successful thus far in the implementation of generics into the `jmle` tool within the Eclipse IDE. Even though some of the model classes remain to be genericized, the work completed thus far has proven that at this point it is simply enough to genericize the model classes and they will work with the tool in its current implementation. This is a valid assumption as the model classes all share common interfaces.

Of course, bringing the `jmle` tool up to Java 5 JDK compatibility will still require some work, as there are numerous features that were added to Java 5 which have yet to be implemented in JML. Amongst these features lacking in the `jmle` tool are included `enum` types and autoboxing (automatically creating an `Integer` object around a primitive `int` when necessary), to name just two.

This project has laid a solid groundwork for the completion of Java 5 compliance in the `jmle` tool. Only once this migration has occurred can the `jmle` tool, along with all the other tools associated with the JML4 project make the move away from mere academic pursuits to being very powerful and robust suites that developers would be able to leverage in the implementation of the design by contract ideals in their programs.

## References

1. S. Abdennadher, E. Krämer, M. Saft and M. Schmauss. JACK: A Java Constraint Kit. In Electronic Notes For Theoretical Computer Science, volume 64, pages 1-17. September 2002.
2. L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan, M. Leino, and E. Poll. An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer, 7(3):212-232, June 2005.
3. N. Cataño and T. Wahls. Executing JML Specifications of Java Card Applications: A Case Study. Proceedings of the 24th ACM Symposium on Applied Computing, Waikiki Beach, Honolulu, Hawaii. Vol 1. 404-408. March 8 - 12, 2009.
4. P. Chalin, P. R. James, and G. Karabotsos. An Integrated Verification Environment for JML: Architecture and Early Results. Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007), pages 47-53, September 2007.
5. Y. Cheon, G. T. Leavens. (2006). Design By Contract With JML. Retrieved from <http://www.jmlspecs.org/jmldbc.pdf>.
5. D. R. Cok. Adapting JML to generic types and Java 1.6. Seventh International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008), pages 27-34, November 2008.
6. Eclipse Foundation. (2009). Eclipse.org Home. Retrieved April 1, 2009, from <http://www.eclipse.org/>.
7. B. Krause and T. Wahls. jmle: A tool for executing JML specifications via constraint programming. In L. Brim, editor, Formal Methods for Industrial Critical Systems (FMICS '06), volume 4346 of Lecture Notes in Computer Science, pages 293 – 296. Springer-Verlag, August 2007.
8. A. Langer. (2009). Java Generics FAQs - Under The Hood Of The Compiler. Retrieved April 9, 2009 from <http://www.AngelikaLanger.com/GenericsFAQ/FAQSections/TechnicalDetails.html>
8. G. T. Leavens, K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In OOPSLA 2000 Companion, Minneapolis, Minnesota, pages 105–106. ACM, Oct. 2000.
9. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. (2008). JML Reference Manual. Retrieved

December 2, 2008,  
[http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman\\_toc.html](http://www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_toc.html).

from