

Canica: An IDE for the Java Modeling Language

Angelica B. Perez, Yoonsik Cheon, and Ann Q. Gates

TR #06-36
August 2006

Keywords: Integrated development environment, specification tool, programming tool, unit test, Java Modeling Language.

1998 CR Categories: D.2.3 [*Software Engineering*] Coding Tools and Techniques—program editors; D.2.4 [*Software Engineering*] Software/Program Verification—assertion checkers, class invariants, formal methods, programming by contract; D.2.5 [*Software Engineering*] Testing and Debugging—testing tools (e.g., data generators, coverage testing); D.2.6 [*Software Engineering*] Programming Environments—graphical environments, integrated environments, interactive environments; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs—assertions, invariants, pre- and post-conditions, specification techniques.

Published in *Proceedings of the 10th IASTED International Conference on Software Engineering and Applications, November 13-15, 2006, Dallas, TX, USA*, pages 32-37, November, 2006.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

Canica: An IDE for the Java Modeling Language

Angelica B. Perez, Yoonsik Cheon, and Ann Q. Gates
Department of Computer Science
The University of Texas at El Paso
El Paso, Texas 79968, U.S.A.
{abperez, ycheon, agates}@utep.edu

ABSTRACT

Canica is an integrated development environment for the Java Modeling Language (JML), a formal behavioral interface specification language for Java. The JML distribution includes several support tools, such as a syntax checker, a compiler, and a document generator, and there are several third-party tools available for JML. However, most of these tools are command-line-based and work in isolation. Canica glues and streamlines these tools to provide a GUI-based, integrated environment for JML; for example, it automates unit testing including test data generation, test execution, and test result determination. In this paper, we describe the key features of Canica and explain its design and implementation. We also discuss the lessons that we learned from our development effort.

KEY WORDS

Integrated development environment, specification tool, programming tool, unit test, Java Modeling Language

1. Introduction

The Java Modeling Languages (JML) [1] [2] is a formal behavioral interface specification language for Java to document the behavior of Java classes and interfaces. JML is becoming popular both in academia and industry [3]. In academia, it is widely accepted as a research and teaching platform for formal specification and verification of Java programs. In industry, it is being adopted as a practical tool to specify the behaviour of Java program modules, e.g., as a design-by-contract tool for Java.

JML provides several support tools [3], some available from the standard JML distribution and others from researchers around the world. At the University of Texas at El Paso (UTEP), for example, we are developing a tool to automate unit testing of Java classes [4]. However, most of these tools are command-line-based and are not user-friendly, especially for beginning JML specifiers.

Just like writing Java code, writing JML specifications is an iterative process that includes activities such as editing, checking, testing, and debugging. As such, it also requires an integrated environment that supports the life cycle of

specification development. As JML specifications are typically annotated in Java source code, ideally such an environment should also include a Java programming environment. However, most JML tools work in isolation without supporting the specification development life cycle.

We present an integrated development environment for JML, called Canica. Canica is a GUI-based, specification and programming environment for JML and Java. It integrates several JML/Java tools to support the iterative nature of developing JML-annotated Java programs. One of the most distinguishing features of Canica is its support for JML-based unit testing of Java classes. Unit testing is automated, i.e., it automatically generates unit test cases, executes the tests, and determines the results of the tests. In the approach, JML specifications are used both to guide test data generation and as test oracles to determine test successes or failures [4]. The approach facilitates detecting errors not only in Java code but also in JML specifications and, thus, provides a valuable tool for debugging JML specifications.

Our approach for developing Canica was rigorous, producing documentation that included software requirements specification and software design documents. In the rest of this paper, we will present relevant portions of these documents to describe the key features of Canica and to explain its design and implementation. We will also share some of our experience by discussing the lessons that we learned from this development effort, hoping that our experiences are valuable to other developers designing and implementing GUI interfaces to command-line-based tools.

2. The JML Language and Its Tools

In this section, we introduce JML briefly and explain some of its tools, focusing mainly on those tools that were integrated into Canica.

JML [1] uses Hoare-style pre- and post-conditions and class invariants to describe the behavior of Java classes and interfaces. As in the design-by-contract approach, JML assertions are Java Boolean expressions with several

JML-specific extensions such as quantifiers, logical connectives, and `\old` expressions (see Figure 1). JML specifications are typically annotated directly in Java source code files as special forms of comments. This means that an ideal JML development environment should also support Java program development.

```

public class BankAccount {
    private /*@ spec_public */ int bal;
    //@ public invariant bal >= 0;

    /*@ requires amt > 0;
       @ assignable bal;
       @ ensures (\result <==> \old(bal) >= amt)
       @   && (\result => bal = \old(bal) - amt)
       @   && (!\result => bal = \old(bal)); */
    public boolean withdraw(int amt) { /* ... */

    // the rest of code ...
}

```

Figure 1. Sample JML specification

There is a number of freely-available JML tools [3]. The JML distribution available from the JML website (www.jmlspecs.org) includes “common” JML tools.

- checker (`jml`): checks for syntax and type errors
- compiler (`jmlc`): compiles JML specifications into runtime assertion checking code
- document generator (`jmldoc`): generates HTML documents
- test oracle generator (`jmlunit`): generates JUnit-based test oracles

At UTEP, we are developing a tool to automate unit testing of Java classes [4]. The underlying idea of this tool is to automate each step of unit testing by integrating JML, evolutionary testing, and JUnit. In essence, the tool generates test data automatically by applying evolutionary techniques (e.g., genetic algorithms), executes generated test data by using the JUnit testing framework [6], and determines test results by using JML specifications as test oracles.

There are also several JML tools available from other researchers around the world. These tools include static checkers (e.g., ESC/Java2 [7]), theorem provers, and model checkers.

3. Canica Features and Requirements

In this section, we describe some of the key features and requirements of Canica. These features and requirements provide a basis for our design that will be explained in the next section.

Our vision for Canica is to provide a teaching and research platform for JML-based formal program specification and verification. To this end, we have two ultimate goals for developing Canica: (1) to provide a lightweight integrated environment for JML/Java developers, especially for the beginning JML specifiers, and (2) to provide a tool integration platform for JML researchers.

To achieve the first goal, Canica must integrate individual JML/Java tools and streamline them to support the iterative nature of specification/program development, including editing, checking, compiling, testing, and debugging. We integrated into Canica the common JML tools included in the JML distribution and the essential Java tools included in the Sun’s Java SDK (e.g., `javac`, `java`, and `javadoc`). To achieve the second goal, Canica must provide a flexible and extensible framework so that other JML/Java tools can be easily integrated in the future. As an example, we integrated a Java, JML-based unit testing tool that is, being developed by our research group at UTEP.

We formulated Canica features and requirements into a software requirements specification (SRS) document [8]. This document provides a clear and precise description of the functionality of Canica. It serves as a reference for developers who need the user requirements, to design, implement, and test the tool. In the following two subsections, we will look at some portions of the Canica SRS: the general description, including the main use case diagram, and the functional requirements, including Statechart diagrams.

3.1 Use Case Diagram

Figure 2 shows the top-level use case diagram for Canica. The main use cases are:

- Edit: edit one or more Java/JML source code files.
- Check JML syntax: check a JML source code file for syntax and type errors.
- Compile: compile a source code file with either a Java compiler (e.g., `javac`) or the JML compiler (`jmlc`).
- Test: perform a unit test for a class either completely automatically or in a step-by-step fashion interacting with the user.
- Run: execute a compiled Java class.
- Generate documents: generate API documents in HTML by using either the Javadoc tool (`javadoc`) or the JMLdoc tool (`jmldoc`).

Detailed scenarios for these use cases can be found in the Canica SRS document [8].

In the next subsection, we describe the functional behavior of Canica, focusing on the Test use case.

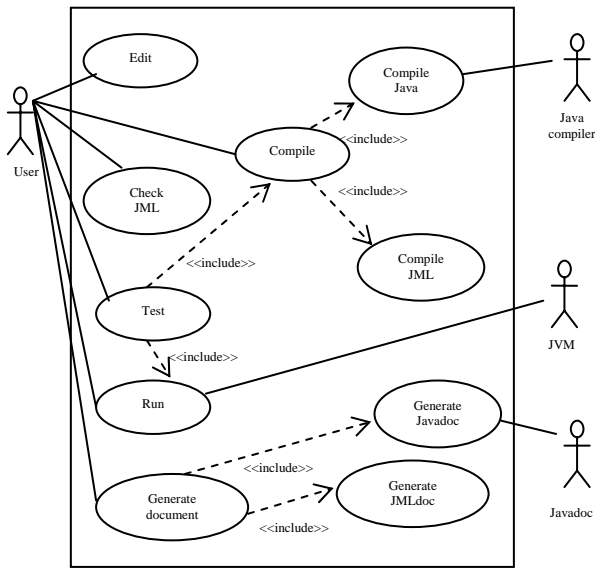


Figure 2. Use Case Diagram

3.2 Statechart Diagrams

Figure 3 shows the top-level Statechart diagram for Canica. It describes how the system interacts with the user by showing the events that initiate transitions from one system state to another. Essentially, the system allows the user to use all the functionalities of Canica, i.e., editing, checking, compiling, running, testing, and document generation.

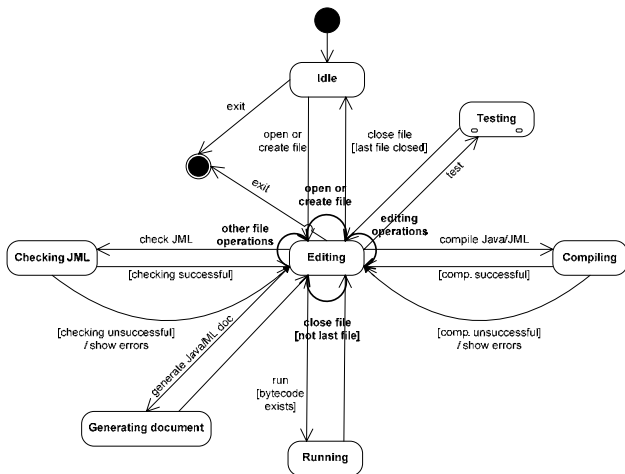


Figure 3. Top-level Statechart diagram

Figure 4 shows a second-level Statechart diagram, the composite state for testing. Each test consists of four steps [4]: (1) compiling source code for testing, (2) generating test data, (3) compiling generated tests, and (4) executing compiled tests. The test case generation step generates both a JUnit-based test oracle class and a test data class. The test execution step runs the compiled test data class that supplies to the test oracle class the generated test data. The test oracle class executes the methods to be

tested with the supplied test data and determines test results [5].

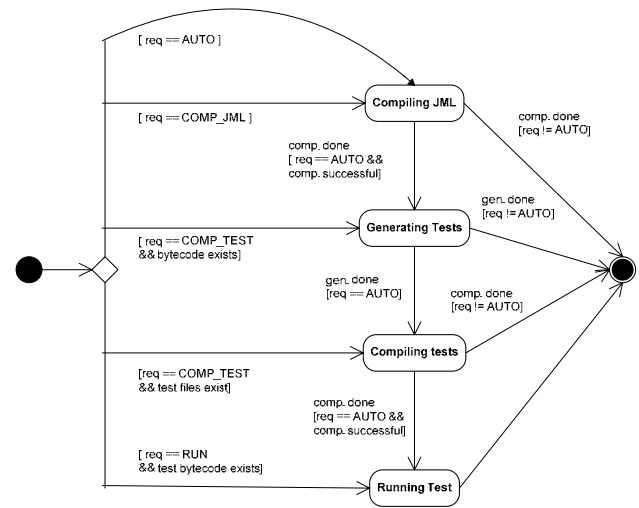


Figure 4. Statechart diagram for testing

As shown in the diagram, each test can be performed either automatically or in a step-by-step fashion. In the step-by-step mode, the system interacts with the user prior to proceeding to the next step. This may be useful when the user wants to perform the same step several times, e.g., to generate different sets of test data for the same class.

We identified about 40 requirements concerning the functional behaviour of Canica, all of which are documented in the Canica SRS [8]. The following are some of the functional requirement regarding testing, excerpted from the Canica SRS document.

[REQ-T-1] The system shall provide the user with the capability to automate unit testing of Java classes: compiling a class for testing, generating test data, compiling generated tests, and executing compiled tests.

[REQ-T-2] The system shall provide the user with the capability to perform automated testing in a step-by-step fashion in which the user initiates each step by supplying parameters appropriate for that step (refer to [REQ-T-3] below).

[REQ-T-3] The system shall provide the user with the capability to specify the following test parameters:

- An algorithm for test data generation, either a random algorithm or a genetic algorithm (refer to Section 3.2.4 for the specification of the algorithms);
- A test parameter to request test coverage;
- Test parameters to indicate the maximum number of iterations, population size, and

mutation rate (if the user selects a genetic algorithm).

In the next section, we will show how we mapped these requirements into a design by explaining the architectural design and some of the detailed design.

4. Canica Design

The most important design goal for Canica was to make the design flexible and extensible so that new JML tools can be easily integrated in the future. We tried to achieve this goal by identifying and analyzing variable parts of the system and separating them through well-defined interfaces. For the detailed design, we also used many well-known design patterns [10], e.g., Observer, Adaptor, Decorator, Abstract Factory, Singleton, and Façade.

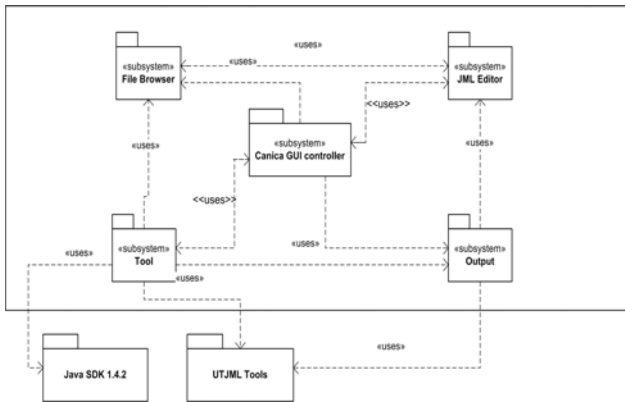


Figure 5. Architecture of Canica

The architecture of Canica is shown in Figure 5. In the design, both Java and JML tools reside outside Canica and both interact with Canica to provide an integrated environment for JML/Java developers. The Canica system consists of five major subsystems.

- Explorer: browses files and directories, and keeps track of previously opened files.
- JML editor: provides a JML/Java source code editor with syntax coloring. The editor is also used to locate and view source code lines corresponding to the error messages produced by the external tools such as javac and jmlc.
- Tool: interfaces with external JML and Java tools. It also provides a generic framework to invoke external command-line-based tools.
- Output: displays both console-based outputs, e.g., error messages from external command-line tools, and GUI-based outputs, e.g., test cases and test results.
- GUI controller: manages services and GUI views of other subsystems.

Our approach to support extensibility is to encapsulate tool-specific details in a separate module and provide a generic interface. The tool subsystem performs this role

by interfacing with external tools. For a new tool to be integrated into Canica, it has to implement the tool interface to supply, for example, views and controls. The rest of the system will use this interface to access the new tool.

Figure 6 shows the class diagram of the tool subsystem. The tool subsystem allows Canica to execute external command-line-based JML/Java tools. In addition, it tightly integrates the test automation tool being developed by our research group, in the sense that control, data, and view are all integrated into Canica.

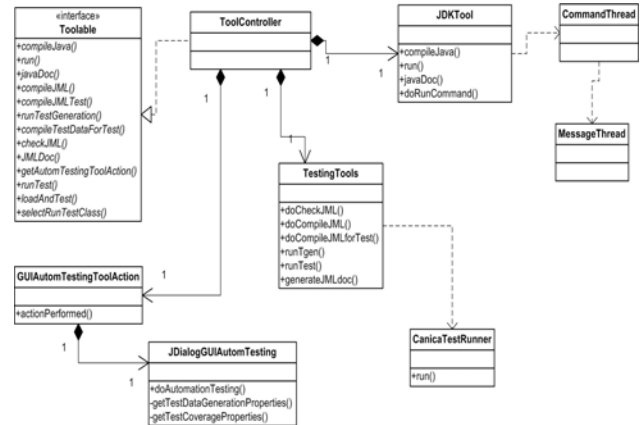


Figure 6. Class diagram of the tool subsystem

As shown in Figure 6, the services provided by the tool subsystem are exposed by the Toolable interface. The detailed design of each subsystem follows the Façade design pattern [10], where only a small subset of the operations are provided in the public classes for the public behaviour of the subsystem, and the rest of the classes have private visibility, i.e., they can be seen only by other classes in the same subsystem. This design pattern gives the flexibility to modify its internal behaviour or structure without affecting its main exposed services. The description of each class with its methods is found in the Canica software design document (SDD) [9].

5. Canica Implementation

The first version of Canica has been implemented by using Java SDK 1.4.2 and Swing, and it supports the requirements that we elicited. The implementation consists of about 20K lines of source code, including comments, in 81 Java source code files. A sample screen shot of Canica is shown in Figure 7.

Due to the detailed design document [9] that contains description of major classes and interfaces including their methods, the skeletal implementation of Canica was semi-automatically generated; it was a matter of translating UML notations into the corresponding Java language

constructs. However, we had to introduce many additional implementation classes, e.g., Swing GUI and control classes, and often had to refine the design to cope with implementation details and decisions. The system architecture, however, remains the same.

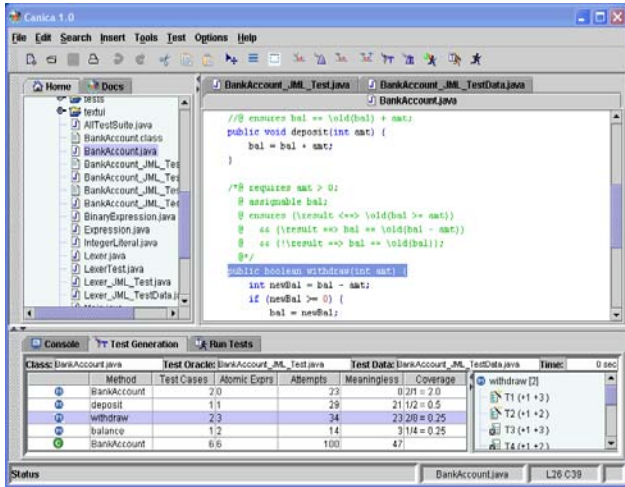


Figure 7. Sample screen of Canica

One of the challenges was to integrate JUnit for test execution; the JML-based test tool uses JUnit to execute generated test data. The challenge was to integrate the JUnit framework tightly into Canica from the control, view, and data aspects. We designed our own view and adapted some of the JUnit framework classes; e.g., we implemented our own test runner class as a subclass of JUnit test runner class to run test classes, interpret the test results, and feed them to our own view classes. Canica's testing facility can handle:

- Test oracle and data classes generated by the JML testing tool
- Bytecode of JUnit test classes supplied by the user
- Source code of JUnit test classes being edited by the built-in source code editor

Another challenge was, given a class file, to load its bytecode directly. For this, we had to write our own class loader as a subclass of the `java.lang.ClassLoader` by overriding some of the inherited methods.

6. Discussion

Canica is not the first GUI program for JML. The JML distribution includes a GUI launcher that can start the common JML tools. However, as the name indicates, it can start individual JML tools, but does not provide an integrated environment for writing and testing JML-annotated Java source code. There is an Eclipse plug-in for JML, called JMLeclipse¹, being developed by researchers at Kansas State University. The goal of this

tool is similar to that of Canica in that it provides a front-end to check JML specifications for back-end tools such as theorem provers and model checkers. It currently supports syntax highlighting, syntax checking (`jml`), and compilation (`jmlc`). Compared with JMLeclipse, Canica is lightweight; thus, it is more suitable for educational and research purposes, and it supports complete automation of unit testing of JML-annotated Java class.

In developing Canica, we tried to follow the traditional waterfall lifecycle model of software development. We started with requirement analysis and produced the SRS. Based on the SRS, we designed the system architecture and refined it into detailed designs, which are documented in SDD. The SDD provided a foundation for the implementation phase; however, we frequently had to go back to the earlier phases to address the changes that resulted from feedback in the later phases. So, in reality, we ended up using a mix of the waterfall model and the iterative model. We think this mix was helpful because, on one hand, it required us to write rigorous documents such as SRS and SDD and, on the other hand, it closed the feedback loop among development phases. In addition, the documents produced will be valuable assets for the maintenance and the future integration of new JML tools. We also found that it is a good practice to start prototyping early and, if possible, to refine it incrementally into the final product. This was particularly true for the GUI modules, partly because of a lack of familiarity with Java Swing programming.

The first version of Canica is one step toward achieving our ultimate goals of providing a teaching and research platform for JML-based formal program specification and verification. The future work includes:

- Further testing. We hope to use Canica itself to test some parts of its implementation (e.g., non-GUI modules) by writing formal specifications in JML.
- Complete implementation. There are some requirements that are not yet implemented completely.
- Application and evaluation. We would like to evaluate the effectiveness of Canica as a teaching and research platform for formal program specification and verification. For this, we plan to use Canica in our programming and software engineering courses at UTEP. We also plan to integrate other JML/Java tools, such as ESC/Java [7].

A significant portion of Canica design and its source code can be reused in developing GUIs to other command-line-based tools, not necessarily JML or Java tools. An interesting future effort will be to build such a reusable framework based on the source code of Canica.

7. Conclusion

¹ It is available from <http://jmleclipse.projects.cis.ksu.edu>.

Canica is a lightweight IDE for JML, integrating several command-line-based JML/Java tools such as compilers and testing tools; we think that other JML tools can be easily integrated into Canica. The two most distinguishing features of Canica are support for the iterative nature of specification and program development and the integration of complete automation of unit testing of Java classes. The goal of Canica is to provide a teaching and research platform for JML-based specification and verification. Although this needs to be evaluated, it is our belief that Canica is a valuable addition to the JML community

The development version of Canica is freely available in the source code form from <http://opuntia.cs.utep.edu>.

Acknowledgements

Cheon's work was supported in part by the National Science Foundation under Grant No. CNS-0509299.

References

- [1] G. T. Leavens, A. L. Baker, & C. Ruby, Preliminary Design of JML: A Behavioral Interface Specification Language for Java, *ACM SIGSOFT Software Engineering Notes*, 31(3):1-38, March 2006.
- [2] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, & D. R. Cok, How the Design of JML Accommodates Both Runtime Assertion Checking and Formal Verification. *Science of Computer Programming*, 55(1-3):185-208, March 2005.
- [3] L. Burdy, Y. Cheon, D. R. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, & E. Poll, An Overview of JML Tools and Applications, *International Journal on Software Technology Transfer*, 7(3):212-232, June 2005.
- [4] Y. Cheon, M. Kim, & A. Perumandla, A Complete Automation of Unit Testing for Java Programs, *Proceedings of the International Conference on Software Engineering Research and Practice, Las Vegas, Nevada, June 27-29, 2005*, pages 290-295.
- [5] Y. Cheon & G. T. Leavens, A Simple and Practical Approach to Unit Testing: The JML and JUnit Way, *Proceedings of the 10th European Object-Oriented Programming, Malaga, Spain, June 2002*, Volume 2374 of *Lecture Notes in Computer Science*, pages 231-255, Springer-Verlag, 2002.
- [6] K. Beck & E. Gamma, Test Infected: Programmers Love Writing Tests, *Java Report*, 1(3):37-50, 1998.
- [7] J. R. Kiniry & D. R. Cok, ESC/Java2: Uniting ESC/Java and JML, *Proceedings of Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, Volume 3362 of *Lecture Notes in Computer Science*, pages 108-128, Springer-Verlag, 2004.
- [8] A. B. Perez, *Canica Software Requirements Specification*, Dept. of Computer Science, University of Texas at El Paso, 2005.
- [9] A. B. Perez, *Canica Software Design Document*, Dept. of Computer Science, University of Texas at El Paso, 2006.
- [10] E. Gamma, R. Helm, R. Johnson, & J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.