Imperial College of Science,

Technology and Medicine

(University of London)

Department of Computing

The RJCE System for

Dynamically Changing Object Methods

By

James D Bloom

Submitted in partial fulfilment of the requirements for the MSc

Degree in Computing Science of the University of London and for the

Diploma of Imperial College of Science, Technology and Medicine.

September 2003

Abstract

The purpose of the Runtime Java Class Editor (RJCE) project was to enable a program to be altered at runtime. This is a revolutionary, new approach to developing, debugging or learning Java. Several approaches were investigated including byte code alteration and class loading. The chosen approach was to combine the use of interpreters and parsers. Through this, the speed of compiled code is combined with the flexibility of interpreted code. A powerful Integrated Development Environment was implemented that can be embedded into any application in only two lines of code. This allows not only source code editing but also provides a class browser, which can be used to introspect on variables, methods, inner-classes, interfaces and super-classes. The project achieved its purpose – as confirmed by various test. The potential of the RJCE system is demonstrated by over 500 downloads within the first week of being released as an open source project[1].

[1] http://class-editor.sourceforge.net

Acknowledgments

I would like to thank my supervisor Simon Colton, for his enthusiasm, interest and encouragement. I would also like to thank both Terence Parr and Pat Niemeyer, because without their products ANLTR and BeanShell, this project would not have been possible.

Table of Contents

1. Introduction

1.1 Motivation

Java has made a huge impact on the world of programming. It has shown that good language design can free programmers from many burdens, such as pointer errors, memory allocation problems and cumbersome, platform-specific libraries. The design of Java avoids these issues while also providing many new powerful features. Java is platform independent and has a large API including multithreading, class loading, reflection, graphical user interfaces, I/O, xml and regular expressions.

Although Java is a well-designed language, it suffers from a problem seen with all compiled languages. In general, it is not possible to alter the underlying code of a running program. The ability to alter a running program is an extremely useful technique, particular for frequently changing code. The Java solution is class loading. Class loading enables classes to be reloaded with new versions at runtime. However, class loading has major limitations, which are discussed in detail in chapter three.

Alternatively, an interpreter can be embedded into an application to execute scripts dynamically at runtime, but this approach has several major drawbacks. For example, before a running program can be altered, it must be reengineering to incorporate the use of an interpreter. Every method that might be altered at runtime must be changed to allow for this.

The most powerful and effective approach is to combine both the virtues of compilation with the flexibility of interpretation. Our system, called Runtime Java Class Editor (RJCE), combines both interpretation and compilation to simulate the alteration of underlying code in a running program. RJCE allows the editing of methods at runtime in any Java program. Method edits can then be applied to a combination of objects, such as an array, a complete class or a single instance.

RJCE offers a revolutionary new approach to developing, debugging or learning Java, enabling a program to be written from within itself at runtime. This ensures high coupling between testing and development, with no delay before the outcome of any alterations. RJCE improves the efficiency of programming by providing an interactive approach to development. Code can be tested instantly after any alteration allowing errors to be found immediately, reducing the time before resolving a problem. Code can also be repeatedly altered and tested until it is correct using trial and error. This makes RJCE a useful system to help students learn Java.

RJCE provides a powerful Integrated Development Environment that can be embedded into any application in two lines of code. This allows not only source code editing but also provides a class browser, which can be used to introspect on variables, methods, inner-classes, interfaces and super-classes.

The main aims of the RJCE project are:

- To simulate alteration of the underlying code in a running program, by allowing alteration of methods in classes.
- To do this in a complete, stable and reliable way, with no limitation on Java code used.
- To provide a system, with all features reasonably expected by users, that is easy and efficient to use.
- To hide from the user any technical aspects required to make this system to work.
- To improve:
    - The practice of building Java programs
    - The debugging of such programs
    - The education of students in the use of Java

1.2 Report Structure

The background section provides information and background to all the technologies considered for the project. The areas covered in the background section include compilation, interpretation, Java, parser generators and XML. The next three chapters contain the design and implementation of the complete system. The three areas covered are the design of the overall RJCE system and the design of two sub components javaprec and ROM. The following chapter considers several hypotheses and test programs that are subsequently used to test the hypotheses. The final chapter is a wish list of ideas that would be explored given more time.

2. Background

During the design stage of RJCE, a wide range of technologies was considered to achieve the desired runtime editing. At each point of the design, an effort was made to explore as many of the different approaches as possible. This background section covers the information required to understand the technologies used in the RJCE system.

We first review and compare compiled and interpreted languages. Then we introduce the Java language and the Java virtual machine (JVM). Several other aspects of Java are also explored; these include reflection, debugging, the JVM method area and class loading.

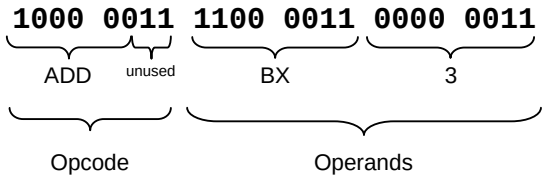
The next section examines two interpreters, BeanShell and DynamicJava. After this, human computer interaction is considered by examining existing programs. This is followed by a review of regular expressions and parser generators, before finally finishing with XML.

2.1 Compiled Languages

The first true digital computer was designed by the English mathematician Charles Babbage (1792-1871) [1]. Although Babbage spent most of his life and money trying to build his machine, he never actually succeeded. Babbage's machine was purely mechanical but the gears and cogs of the day did not have the required precision. Babbage realized that he would need an operating system to run on his computer so he hired the world's first programmer, a young woman called Ada Lovelace [1]. After Babbage, no real progress was made in electronic computers until the 1930s when several visionaries, including Vannevar Bush and Jon Vincent Atanasoff, created the first electronic programmable machines. Since these early days of computing it has been necessary to instruct computers with sequences of codes or programs. The earliest programming required plug boards to control the machine's basic functions. By the early 1950s, the introduction of punched cards improved the situation, so that it was possible to write programs in machine code on cards and load them into a computer.

2.1.1 Machine Code

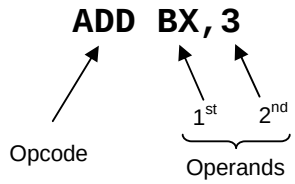
Machine code was designed primarily to be easy for a computer to read and understand. It consists only of a string of numeric codes that represent computer operations. Each computer has a set of available operations such as adding two numbers, multiplying two numbers or subtracting one number from another. Every operation is given an unique code called an operand; for example on the Intel 8086 processor, the add operand is 100000. Some operations also require parameters called opcodes. The add operation, for example, requires two numbers to add together and a memory location for storing the result. A complete add instruction including the operand and opcodes for an Intel 8086 chip is as follows:



This instruction tells the chip to add three to a special memory location on the chip, called the BX register. Writing programs in machine code is extremely difficult and time consuming. Machine code is also difficult to understand.

2.1.2 Assembly Language

It was not long before machine code was replaced by assembly language. Assembly language is the symbolic representation of a computer's machine code. Assembly language is more readable than machine language because it uses symbols instead of numeric codes. The symbols name commonly occurring numeric patterns such as opcodes and specific memory addresses. The previous add instructions, for example, in assembly language is:



The ADD symbol has replaced the bit pattern 1000 00, the special memory address BX has replaced the bit pattern 1100 0011 and the number 3 has replaced its binary form 0011. When assembly language was introduced, the speed and accuracy of writing programs greatly improved. Before executing a program written in assembly language, it must first be converted into machine code. A program called an assembler is used to translate assembly language files into executable machine code files.

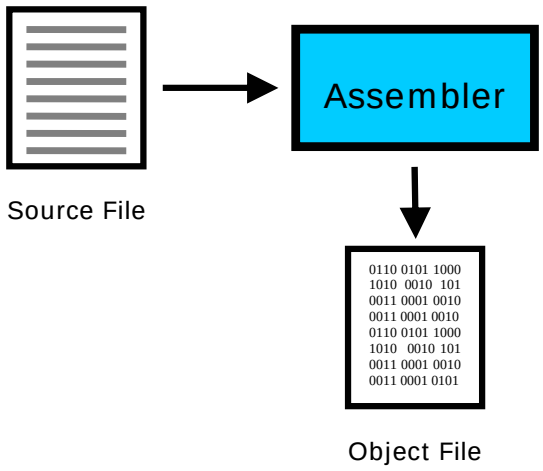


Figure 1, Assembling an assembly language file

Although assembly language improved the ease of programming, it was still not easy to write or understand and was dependent on the particular machine for which it was written. Code written for one machine must be completely rewritten for another machine.

2.1.3 FORTRAN

The next major step in computing was to develop a machine independent language that was easy and natural to write. At first, it was feared that this might not be possible, or if it were, then the code would be extremely inefficient. The development of the FORTRAN language and its compiler by a team at IBM led by John Backus between 1954 and 1957, showed initial fears to be unfounded. Since, the invention of FORTRAN, there have been many developments in compiled languages [1].

1951 Wheeler, Wikes and Gill invent the subroutine.

1958 The Algol programming language was designed by a team of European and American computer scientists in 8 days. Algol was the basis for Niklaus Wirth's later work on Pascal.

1967 Simula was invented introducing most of the key concepts of object-oriented programming including: objects, classes, inheritance and namespaces.

1968 Niklaus Wirth starts work on Pascal

1970 The UNIX operating system is introduced written by Ken Thompson and Dennis Ritchie of Bell Labs. Ken Thompson writes the “B” language to allow UNIX to be written in a machine independent way.

1971 Pascal is released by Niklaus Wirth.

1972 Dennis Ritchie creates the “C” languages, inspired from “B”.

1972 Smalltalk a completely interactive development environment is developed by Alan Kay. Smalltalk is based on Simula and therefore is fully object-oriented.

1983 Bjarne Stroustrup evolves C++ from his earlier work on “C with Classes”.

1991 James Gosling as Sun Microsystems’ starts work on Oak by taking C++ as a starting point. Oak is later renamed to Java after trademark problems.

1995 Java language is officially launched.

2.1.4 Modern Compilers

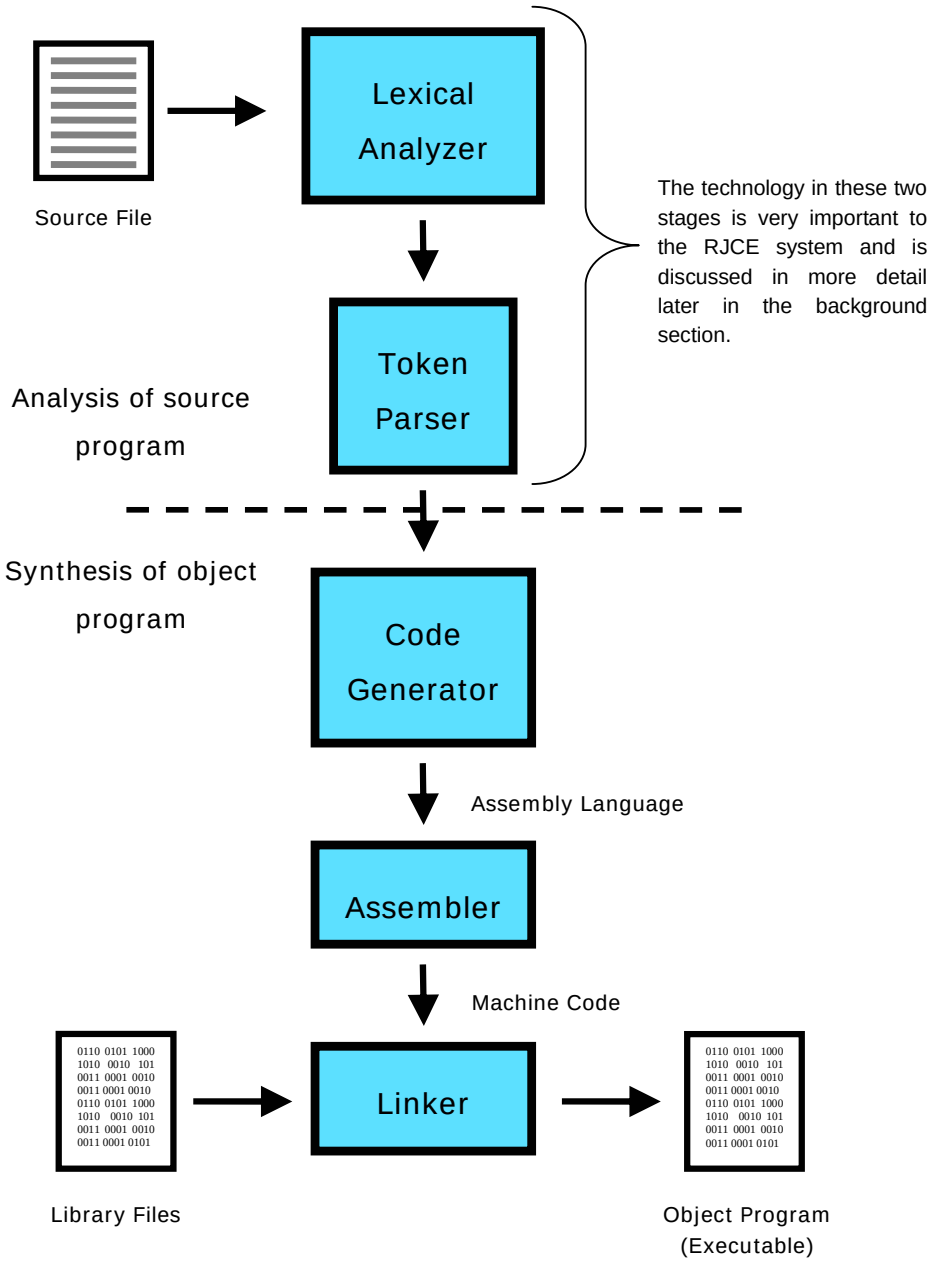


Figure 2, The main stages in compilation

Compilation has two subtasks: analysis of the source program and synthesis of the object program. This division can be seen by the dotted line in Figure 2.

The first step in compilation is lexical analysis. A lexical analyser or lexer reads a stream of characters such as a source file and produces tokens. Each token is an indivisible set of characters such as a keyword or a variable identifier, for example "int", "number", "=", "1" and ";" are all tokens found on the first line of Figure 3.

```
int number = 1;
number++;
System.out.println("The number equals " + number);
```

Figure 3, Example code[2]

If the code in Figure 3 was feed into a lexical analysis it would start scanning at 'i'. When the first space between "int" and "number" has been reached the lexer would output the first token which is the primitive type "int". This would go on until the last token ';' had been reached on the last line. As each token is produced it is passed to the token parser where it is stored until it has been matched against a set of rules, called productions. For example, there would be a production for a variable definition such as on the first line of Figure 3. This production would match the "int" "number" "=" "1" ";" sequence of tokens. These two steps contain important technology used in RJCE, and are discussed in more detail latter in section 2.7.

As the token parser matches the tokens against its productions, a special intermediate model of the source file, called an Abstract Syntax Tree (AST), is created. The AST is then used by the code generator to generate the assembly language. Once this is complete, the assembler converts the assembly language into machine code. The final stage of compilation called linking is used to link together separate compiled files. This stage allows files that have already been compiled, such as library files, to be included into programs. For example, a compiled library file might be used for writing text files to a hard disk. Such a library file could contain several methods useful written text files

[2] This and all the remaining code examples are screen shots taken from ROM.

such as creating a new file, writing text in a file or creating a directory. To enable a new program to use this library file the code from the library file has to be included into the new program. This is done during the linking stage of compilation.

2.2 Interpreted Languages

An interpreter directly executes its source language, without first translating it into an object program. This means there is no compilation stage, instead the interpreter acts much like a CPU, with a fetch-execute cycle. This runs in a loop repeatedly reading one instruction after another from a program and deciding what is necessary to carry out that instruction. In general, any programming language can be either compiled or interpreted. The optimum choice depends on the language and situation in which it is in use. Interpreters do not required a compilation stage so all the steps in Figure 2 are avoided.

2.2.1 Compilation - Interpretation

Some languages use combinations of compilation and interpretation. There are various compromises between the development speed when using an interpreter and the execution speed when using a compiler. Some systems, such as LISPs, allow interpreted and compiled code to call each other and to share variables. This means that once a routine has been tested and debugged under the interpreter it can be compiled and thus benefit from faster execution while other routines are being developed. Many interpreters do not execute the source code directly, instead, they convert it into some more compact internal form. The compiler first translates the source language into an intermediate language, which is then interpreted. The intermediate language is the object code for the compiler and the source language for the interpreter. An intermediate language is very important for languages such as Java as it plays an important role in portability, and security.

Compilers, on the other hand, are usually preferred if speed of execution is the primary consideration. It takes longer to run a program under an interpreter than to run an equivalent compiled version. Interpreters are most useful when the time for compilation is a significant amount of the total time for execution. This can be the case when many

alterations have to be made such as during education, software development or debugging situations. The experience gained during the creation of the RJCE system showed the time for compilation can often be significant, particularly during debugging.

Some examples of interpreted languages include BASIC, a language that is more often interpreted than compiled, and functional languages such as LISP, which generally tend to be interpreted [2]. There are also three interpreted languages found on most UNIX systems: Perl, a comprehensive interpreted language, sed, a non-interactive stream editor, and awk, a pattern scanning and processing language [3].

2.3 Java

2.3.1 History

Java, or Oak as it was originally named, was developed by "the Green Project" of Sun Microsystems. The secret team was directed to anticipate and plan for the next wave in computing [4]. This initially led them to develop the Java platform by exploring digital networked consumer devices. Java's initial design, therefore, supported multiple host architectures and allowed secure delivery of software components. To guarantee these targets are met, compiled code has to survive transport across networks, operate on any system and be safe to run. The first product the *7 ("StarSeven") was too far ahead of its time. *7 was targeted at the growing cable TV business that was still trying to find a viable business model. *7 ultimately failed to secure a target market. After a key meeting, in 1993, the team realised that there was no viable business in digital cable television. The Java team then changed to work with internet applications [5] hacking a Java browser together, called HotJava, over one weekend. In May 1995, Sun formally announced Java, previously Oak, and the HotJava browser. Sun's HotJava browser demonstrated the power of Java by allowing embedded applets to run inside web pages. Later that year, Netscape and Microsoft

Figure 4, Sun's *7 device

announced plans to license Java applets in their browsers [6]. The following year, Sun released Version 1.0 of the Java Development Kit (JDK) free of charge [7].

Java has had an intimate relationship with the Internet that few programming languages can boast, growing and benefiting from the Internet's rapid growth. Like the Internet itself, it does not discriminate against any specific platform, so Java can reach wherever the internet can. Due to its compiled-interpreted nature, Java programs can also be small and fast to download, particularly when compared to C++.

2.3.2 The Java Virtual Machine

You can run the same Java programs on most operating systems including Windows 95/98/NT/2000/XP, Linux, UNIX or Solaris. This is possible because a Java program does not execute directly on a computer. Instead, it runs on a standardized hypothetical computer called the Java Virtual Machine (JVM). The JVM is emulated by a program on your computer. It is like a real computer chip with an instruction set, machine code and various memory areas such as a heap and a stack [8]. The JVM is the cornerstone of the Java platform being responsible for system independence, small size of compile code and security. One of the main strengths of the JVM design is that not only can a JVM be emulated on most operating systems, but it can also be built as a real hardware chip [9].

An analogy can be draw between a JVM and a human language translator, as used in the United Nations. Every time an English person speaks to someone in a different foreign language, a different translator is normally used. For Java, a different translator or JVM is need on each platform, but this enables one language to be used everywhere.

2.3.3 Compiled or Interpreted?

Java source code is in general compiled to byte code, although Java can be executed without any compilation for example with the BeanShell [10] or DynamicJava [11] interpreters, these are discussed in more detail later. Byte code is machine code for a JVM and will execute on any hardware or software that conforms to the JVM specification. This includes emulated JVMs that are available for most operating systems, and the picoJavaII chip. When an emulated JVM reads byte code, a program called the Java interpreter inspects and deciphers the byte code, checking it is in the

correct form and is safe to execute, before deciding which actions need to be performed. These actions are then translated into the correct commands for the platform that the JVM is running on. Java can instead be executed directly without interpretation. When the JVM is running as a chip, for example the picoJavaII, it is not emulated and therefore the byte code is not interpreted but executed directly instead, just as in a normal CPU. Java can also be compiled to machine code more commonly called native code. There are several native Java compilers such as NativeJ [12], a Java compiler for the Windows family of operating systems. Table 1 shows a summary of the different methods of executing a Java program.

Table 1, Methods of executing Java

Method of Execution	Environment
Direct Interpretation	The BeanShell or DynamicJava applications
Compilation - Interpretation	Normal Java execution, such as a Java application running on a PC.
Compilation - Execution	Java running on any machine with a picoJavaII chip design, or Native compilation as with the NativeJ compiler.

2.3.4 References and Garbage Collection

Class loading is an important technology considered during the design of RJCE. To understand class loading and its limitations fully, it is important to understand garbage collection and how Java references work.

Java is a general-purpose object-oriented concurrent language. Its syntax is similar to C and C++, but some features are omitted in an attempt to make Java less complex, confusing, and unsafe. One such omitted feature is pointers. Pointers are used in C and C++ to store the memory address of a variable or object. In Java, the memory pointer does not exist; instead, Java only uses references to refer to objects. References are

used to name objects; they are similar to pointers but are generally less likely to cause memory problems, such as memory leaks.

An important reason why traditional languages, like C and C++, need memory pointers is that they do not have automatic garbage collection. Developers instead allocate and free memory manually, by using the *malloc()* and *free()* functions in C for instance, or the *new* and *delete* operators in C++. In Java applications, memory is allocated by the *new* operator, but developers need not free this memory explicitly. Instead, the garbage collector's job is to identify which objects are no longer in use. Once such an object has been identified, the memory it occupies can then be reclaimed. An object is considered to be still in use if it can be accessible or reachable by a program in its current state. A Java program's state is determined by the set of executing threads. Each thread is in turn executing a set of methods (one having called the next). All object defined in this set of methods as well as those defined in classes and static objects are said to be the root set of references for a program [13]. If an object is referenced by this set of references then it is said to be reachable and therefore cannot be reclaimed by the garbage collector.

2.3.5 Reflection

Java reflection enables discovery of the Metadata associated with a Java class at runtime.

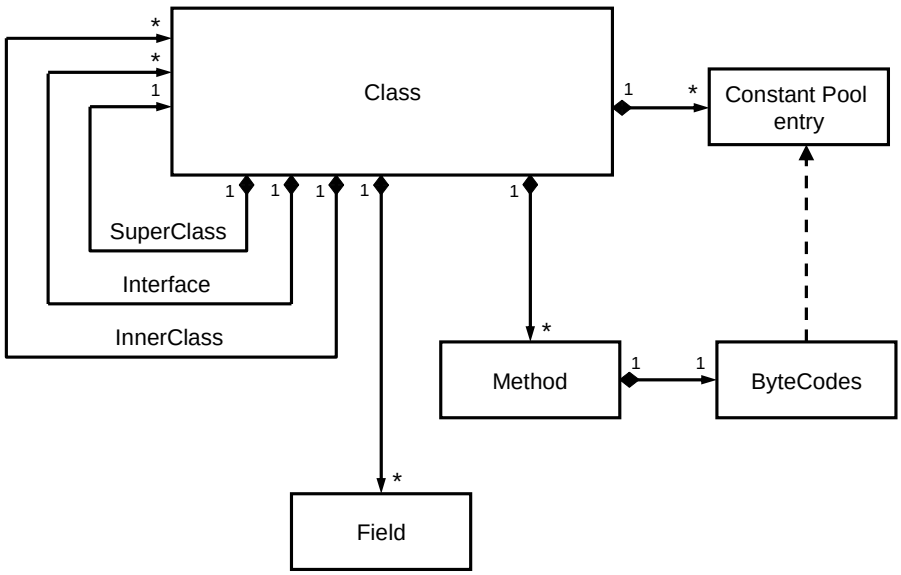


Figure 5, Metadata in the binary class format [14]

Metadata stores information about class characteristics, such as the base class, super-interfaces, method signatures and field types, as shown in . Reflection allows object instantiation, method invocation and field mutation that bypass normal language restrictions associated with the private and protected modifiers. The main drawback with reflection is the time penalty. Reflective method invocation, for example, actually involves three steps. First a *Method* object is retrieved from a *Class* object with a call to *getDeclaredMethod(String name, Class[] parameterTypes)*, or *getDeclaredMethods()*. Next any parameters need to be added to an *Object[]*, before the method can finally be invoked by calling *invoke(Object obj, Object[] args)*. The length of time for reflective calls can be reduced by caching the *Method*. Many reflection-based systems make only a single call to get a method, then allow many calls to *invoke(Object obj, Object[] args)*, avoiding some overhead after the first method invocation. In tests, the time for the complete refection process including getting a *Method* was close to 10^5 nsec [15]. The time for just boxing the arguments and calling the method was close to 10^4 nsec, this is shown in Table 2.

Table 2, Rough Estimate of Reflection Performance [15]

Operation Tested	Time (power of 10 nsec)
Increment integer field	10^0
Invoke a method	10^1
Reflectively increment integer field	10^4 - 10^5
Reflective method invocation	10^4 - 10^5
Open, write and close a file	10^6

2.3.6 Altering Running Code

Java programs, in general, require execution to be stopped and code recompiled before any alteration can take affect. This can be time consuming, particularly during development or debugging. Instead, if a program's running code could be altered

during execution then the time spent recompiling and restarting a program could be avoided.

2.3.6.1 Altering Code in the JVM Method Area

In UNIX, Windows and most other operating systems each process has its own memory area called its virtual address space [16]. The virtual address space for each process is divided into segments. Each segment is a logical division of program address space. The address space is not actually physically divided but instead conceptually divided in terms of how each segment of memory is used. The memory management unit associates a protection bit with each segment to indicate the allowed operations for that segment. A code segment, for example, has an execute protection bit. Therefore, data in a code segment can only be executed; it is not possible, for example, to write to a code segment. Whereas a data segment has a read-write protection bit, allowing both reading and writing. A segmented memory scheme adds memory protection against illegal memory access such as programs modifying their own code. Self-modifying programs went out of style in the 1950s because they were too difficult to understand and debug [16]. An attempt to write to a code segment normally results in a catastrophic Memory Protection Fault, probably halting execution, and possibly crashing your computer. In a JVM, the equivalent of the code segment is called the method area and is shared among all threads. The method area is used to store per-class information such as method code, constructor code, constants and fields [17]. The current JVM specification gives no guarantees about how the method area is handled. This is indicated by the following statement from the JVM specification “the Java virtual machine specification does not mandate the location of the method area or the policies used to manage compiled code.” [17]. This is significant because the JVM specification lists the guaranteed similarities between different versions of JVMs. If the JVM specification does not specify how to manage running code then it becomes difficult to determine the side affects of altering running code. Does this for example cause a catastrophic Memory Protection Fault? Does this crash the JVM? The answers are most probably different for different JVMs. Therefore, direct manipulation of a running program is difficult because it is hard to predict the outcome for all versions of JVM.

2.3.6.2 Reloading or Adding New Classes

Although, altering running code in a JVM can be difficult, class loading goes some way to reducing this problem. Complete classes can be reloaded in any runtime environments that support class loading; this includes not only Java but also the .NET Framework[3].

In Java, there are two types of class loading namely, explicit and implicit class loading. Implicit class loading is done automatically via the classpath whereas explicit class loading is done by specific request. To load a class explicitly, the fully qualified name is passed to a class loader, which either returns a class object that represents the loaded class or throws a *ClassNotFoundException* [18]. Implicit class loading occurs whenever an unloaded Java class is demanded at runtime usually via one of the three main non-user defined class loaders, the bootstrap class loader, the extensions class loader and the system class loader shown in Figure 6. The bootstrap class loader loads boot-classpath classes and Core API packages, such as java.lang. The bootstrap loader is different from any other class loader because the classes it loads are not verified, instead it assumes all classes it loads have well-formed byte code. The classes loaded by the bootstrap class loader are also different because they are not subject to security checks. The extensions class loader loads installed optional packages that usually come as jar files. The system class loaders loads user-defined classes found on the classpath.

When a class is loaded, say class A, by a class loader it maintains a reference to the class loader, accessible via the *getClassLoader()* method. Whenever a class A refers to another class, say class B, if class B is not already loaded then class B is loaded implicitly, using class A's class loader [19].

[3] This phase .NET Framework refers to the whole .NET Platform, which includes the Microsoft Intermediate Language (MSIL) the common language runtime and the .NET APIs

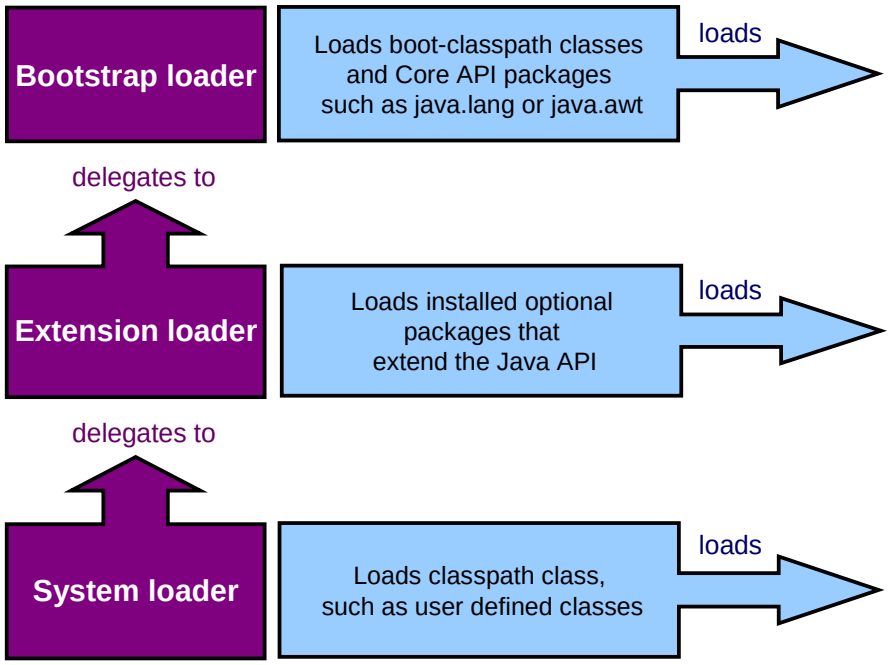


Figure 6, The three main Java class loaders [14]

Implicit class loading provides a simple automatic system for loading classes, as they are needed, whereas explicit class loading provides flexibility. However, these mechanisms must have some additional properties to deal with the tricky issue of class visibility. If there is no visibility across class loader boundaries, then there is no real benefit to dynamic class loading and separate class loaders might just as well be separate virtual machines. Alternatively, complete visibility between class loaders leads to chaos, as there is no way to hide conflicting names of different class versions from each other. The class loader architecture adopts a middle path characterized by the following three rules [20]:

1. The consistency rule: class loaders never load the same class more than once.
2. The delegation rule: class loaders always consult a parent class loader before loading a class.
3. The visibility rule: classes can only “see” other classes loaded by their class loader’s delegation, the recursive set of a class’s loader and all its parent loaders.

The consistency rule states that before a class can be reloaded it must first be unloaded. Instances of *Class* and *ClassLoader* are simple Java objects, subject to the normal rules of the Java platform. They cannot be explicitly destroyed instead they must be reclaimed like any other object, by the garbage collector. A class therefore only gets unloaded when it is unreachable by the root set of references [21]. The enforcement of this in all JVMs is a core part of the Java security infrastructure, ensuring program classes cannot be reloaded on request by external programs. If this was not the case, class reloading could be used to unload security critical classes disabling security features in Java software. This leaves the difficult problem of ensuring classes have no references. Every instance maintains a reference to its class, every class maintains a reference to its class loader and every class loader maintains a reference to every class it has ever loaded. If you have even a single reference to any part of this hierarchy of references then, it would most likely be the case that these classes cannot be reclaimed by the garbage collector and unloaded. The majority of user-defined classes are loaded by the system class loader, shown in Figure 6, and therefore cannot be reclaimed by the garbage collector. This is not the case for classes loaded by user defined class loaders. User defined class loader can be used to load multiple versions of a class from the same location at different times. This technique is called hot deployment and is useful for redeploying incremental changes to a class without having to shut down the virtual machine [22]. It is used by servlet engines such as Apache Software Foundation's Tomcat to automatically reload servets that have changed. Hot deployment works by controlling instance creation and class loading through a single factory method. This factory method has an algorithm to choose which class it uses to create the new instance. There are two main limitations to this approach. First, before a running program can be altered it needs to be redesigned to allow for hot deployment, for example, an interface type and factory method has to be created for each class that is to be updated. Second, class loading is a sophisticated technique that can often be difficult to implement. This makes it an unattractive technique for developers to use particularly during debugging.

2.3.7 Debugging

One of the most frustrating tasks a developer must face is testing and debugging software applications. A “bug” is a software defect, the nature of which may or may not have been identified. Bugs can be present in all phases of the software development life cycle. The conventional way to remove bugs is to use a debugger. A debugger can be useful because it allows the programmer the ability to step forward through a program line by line. Most debuggers can also inspect variables and run execution until a pre-specified break point, at which point, it is possible to step forwards line by line. There are many Java debuggers available. For example, Sun’s JDK includes a command line debugger called JDB [23]. Various debuggers have also been created for graphical Integrated Development Environments, such as Borland’s JBuilder [24] or NetBeans [25]. All these debuggers provide facilities for stepping forwards through compiled byte code one line at a time. Some Java debuggers can even step backwards through a program line by line [26]. Once a bug has been detected, the program’s code needs to be altered to remove the bug. In general, Java programs, like most compiled languages, have to stop execution and recompile code before any bugs can be fixed.

2.4 Java Source Code Interpreters

Although Java is generally interpreted as byte code by a JVM, it can also be interpreted as source code. There are two applications available to do this BeanShell [10] and DynamicJava [11].

2.4.1 BeanShell

BeanShell was started in 1993 by Pat Niemeyer while working at Southwestern Bell Technology Resources. After publishing one of the first books on the Java language, called *Exploring Java*, Pat Niemeyer created BeanShell by drawing motivation from Tcl [10]. BeanShell's first public release was not until 1997. BeanShell is made possible by Java's advanced reflection capabilities. Hence, it was only possible after the release of Java 1.1, which included reflection for the first time in Java. Since 1997, BeanShell has slowly grown in popularity, now being distributed with Emacs, as part of the JDE and with Sun Microsystems's NetBeans / Forte for Java IDEs. BeanShell is used for many purposes and in particular, is used as a popular educational tool for teaching Java.

BeanShell is a Java source interpreter that uses full Java statements and expression syntax. It emulates strongly typed variables, methods and arithmetic, logical or bitwise operators [27]. BeanShell begins with standard Java language and bridges it into the scripting language domain [28]. This is done partly by allowing types to be relaxed where appropriate, but also by adding scripting commands, for example:

- *source()* - Reads a BeanShell script into the current interpreter
- *run()* - Reads a BeanShell script into a new interpreter
- *frame()* - Displays a GUI component in a Frame of JFrame
- *load(), save()* – Loads or saves serializable objects to a file.
- *cd(),cat(),dir(),pwd()* – Unix shell commands
- *exec()* – Runs a native application
- *javap()* – Prints the methods and fields of an object

- setAccessibility() – Control access to private and protected components.

BeanShell can handle both single statements and statement blocks but it cannot handle complete classes, although this is on the BeanShell wish list. Instead, BeanShell allows object construction via nested methods similar to the way JavaScript and Perl 5.x define their version of classes. Although this does not provide real classes, it does provide class like functionality, for example, allowing the creation of instances [28].

```
foo() {
  int a = 42;

  bar() {
    print("the bar is open!");
  }

  bar();
  return this;
}

// Construct the foo object
fooObj = foo() ;  // prints "the bar is open!"

// Print a variable of the foo object
print (fooObj.a);  // 42

// Invoke a method on the foo object
fooObj.bar();  // prints "the bar is open!"
```

Figure 7, BeanShell scripted object [10]

BeanShell has some useful additional features. Import statements, for example, are allowed at the start of a script to improve access to a class's scope or namespace. The main advantage with BeanShell is that it is easy to use and install and comes with a complete manual. Another advantage with BeanShell is the willingness of the creator, Pat Niemeyer, to respond to email queries, ensuring most embedding problems can easily be solved.

2.4.2 DynamicJava

DynamicJava is a Java interpreter, which tries to follow the Java Language Specification as much as possible. Some extensions have been added to ease the creation of small programs. All extensions have been added with the same goal in mind: to reduce the amount of code to write and make it easy to translate a script to a valid Java program [11]. Unlike BeanShell, DynamicJava supports more features and syntax defined by the Java language, including class definition and multithreading. DynamicJava also offers some other additional features e.g. statements can be written outside classes or methods in the top-level namespace, methods can be defined outside classes and typing is dynamic as in BeanShell [28]. DynamicJava also comes with a standalone script engine, which is a simple Swing editor with the capability to interpret the content of its buffer.

Constant strings are not shared correctly in DynamicJava. For example:

"Abc" == "Abc"

or

"Abc" == "Ab" + "c"

return false with DynamicJava and true with Java.

The main drawback with DynamicJava is the minimal information available. For example, there is no manual or instructions, although there are a few pages of limited information on the DynamicJava web site.

2.5 Human Computer Interaction

RJCE contains several GUIs and user interfaces. Therefore, common modern source editor and browsers are both surveyed.

2.5.1 Graphical Source Editors

A simple study of available graphical source editors, such as Borland's JBuilder (see Figure 8) or NetBeans, shows they usually come with several basic features. Sun's *Look & Feel Design Guide* also suggests similar features [29]. These include cut, copy, paste, undo, redo, find & replace, page setup and printing. Graphical source editors often also include syntax colouring, showing keywords in deep blue, comments in green, quotes in blue and so on. Another useful feature found in certain editors is context sensitive popup menus. These can be useful as they give a list of available commands in relation to each object on the screen and therefore reduce the amount of mouse movement required.

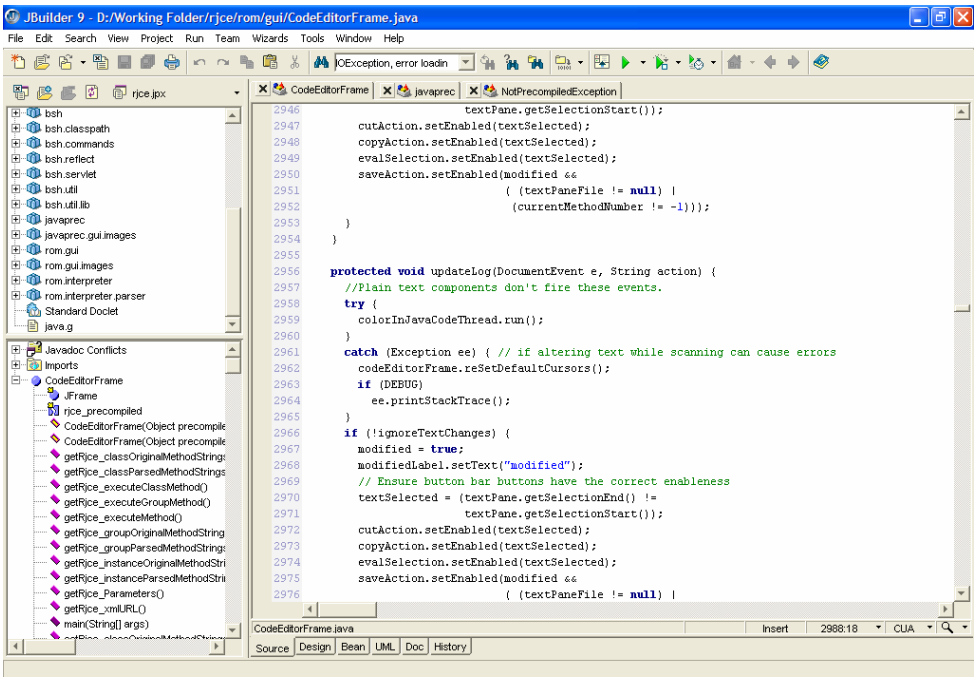


Figure 8, Borland's JBuilder is an example of a graphical source editor

2.5.2 Browsers

Probably the most commonly used browser in the world is the Windows Explorer, shown in Figure 9. This contains a directory tree structure that is easy to use for browse directories. There are also several other browsers for Linux, for example, that use similar techniques to display hierarchical information. Another feature found in both Windows Explorer and several Linux internet browsers is the ability to transverse backwards and forwards through a list of previously viewed pages. These browsers also contain several features that are also found in graphical source editors such as cut, copy, paste, undo, redo, find, page setup, printing and context sensitive popup menus.

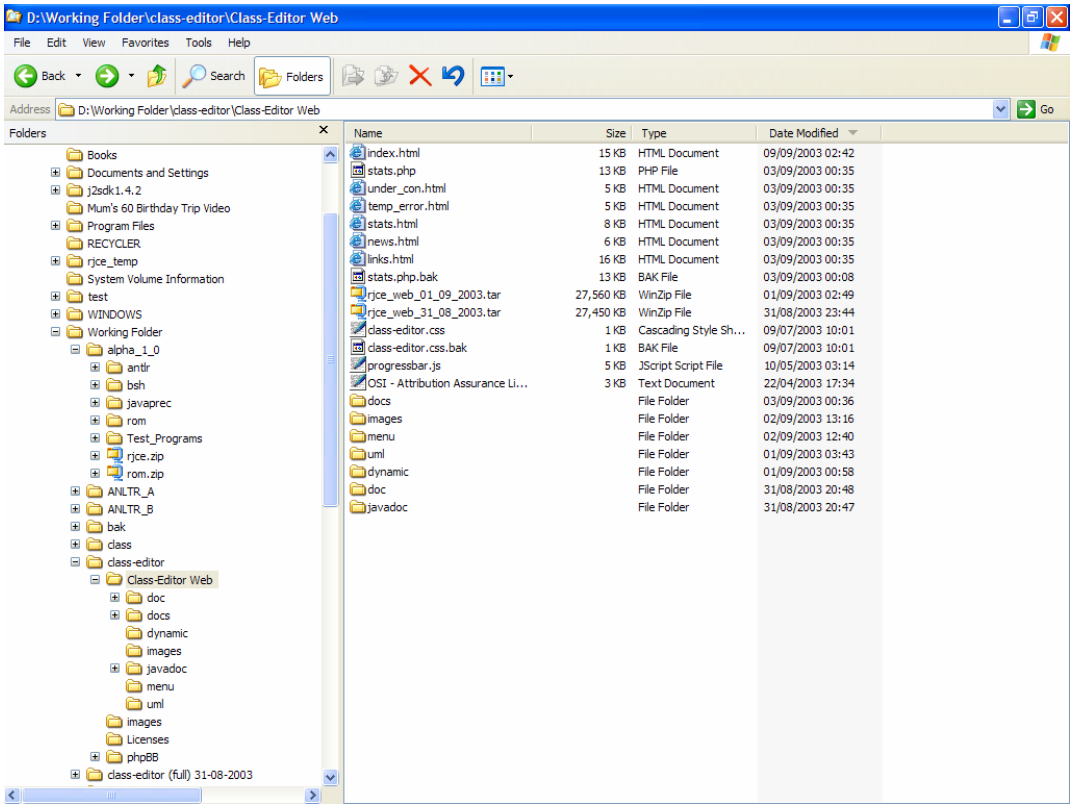


Figure 9, Microsoft's Windows Explorer, XP Version

2.6 Regular Expressions

Regular expressions are a powerful method for matching sequences of characters. They were used both in the design stage of javaprec and are used for syntax colouring in ROM

A regular expression is a pattern of characters that describes a set of strings. Regular expressions were introduced to Java 1.4 in a new package *java.util.regex*. This package handles Perl-like regular expressions, which are compiled into a *Pattern*. A *Pattern* is generated using a static factory method *Pattern.compile(String regex)* rather than a constructor and is stored in internal machine format rather than byte code. To develop a regular expression in Java, a combination of ordinary and special characters are used. Special characters serve a special purpose, for example, the **.** matches anything except a new line. A regular expression like ***r..e*** would match any four-character string that begins with r and ends with e, such as rice, race or rjce. There are many special characters used in regular expressions to find words at the beginning of lines, words that ignore case or are case-specific. Special characters can also symbolise a range, such as ***e-r***, meaning any letter from e to r, or a choice, such as ***[def]*** meaning d, e or f. Perhaps the most useful feature of regular expressions is that they can also match multiple characters using **?** to symbolize once or not at all, ***** to symbolize zero or more times and, **+** to symbolize one or more times. For more examples, see Table 3.

A *Pattern* is matched against *String* using a *Matcher*. A *Matcher* is created from a *Pattern* by invoking *Pattern*'s *matcher(CharSequence input)* method. Once created, a matcher can be used to perform three different kinds of match operations.

- Invoking the *matches()* method attempts to match an entire input sequence against a pattern.
- Invoking the *lookingAt()* method attempts to find the first occurrence of a pattern starting at the beginning of the input sequence.
- Invoking the *find()* method scans the input sequence to find the next occurrence of the pattern. This kind of match operation is the most useful when writing a

parser because, unless the matcher has been reset, the *find()* method always starts searching from the last successful match.

Using *Matcher* is a powerful and simple approach providing the location of the start and the end of every match, progressing automatically through a search sequence and failing safely when no more matches are found.

Regular expression are also used in several other parts of the Core API, including the *split(String regex)* method in the *String* class. This splits a string about each occurrence of the regular expression, returning a *String[]*.

Table 3, Examples of regular expressions

Regular expression	Meaning	Possible matches
([a-m&&[^def]])	a through to m except def	a, c, m, b, g
([abc])*([de])+	zero or more letters equal to a, b, or c followed by one or more letters equal to d, or e.	aabcbcd, ddddddd, eeddee, abcde, ce
(\\b[4](\\w[5])+\\b)	any word of any length	int, bob, verylongwordwithnospaces, SundayMorning, String
(\\b(\\w)+\\s[6](\\().(\\)).(\\{))	part of a method signature	main(String[] args) {, toString(){, Object(){, getClass(){
(\\bdouble\\b(\\[\\])*)	the world double with zero or more [] after	double, double[], double[][], double[][][], double[][][][]

[4] \\b is used to symbolize a word boundary
[5] \\w is used to symbolize a word character i.e. a-z, A-Z, _ or 0-9
[6] \\s is used to symbolize a white-space character i.e. \t, \n, \f, \r

2.7 Parser Generators

A parser generator is really a compiler. Instead of taking program source code as its input, a parser generator takes a grammar. The output of a parser generator is a parser, or compiler, that recognizes the programming language described by the grammar. In short, a parser generator is a compiler compiler, or a parser compiler.

2.7.1 Lexers & Regular Expressions

The first stage of this approach is lexical analysis (also known as scanning). The input text is fed into a scanner as a stream of characters. The scanner groups these characters into tokens, which are word-like elements, such as keywords, identifiers and punctuation. A lexical token is a sequence of characters that can be treated as a unit in the grammar of a programming language. Forming characters into tokens is much like forming characters into words in an English sentence and deciding which word is meant. A programming language classifies lexical tokens, “words”, into a finite set of token types. Tokens are matched using regular expressions, although mainly similar to the *java.util.regex* they are written in a slightly different form. For each symbol *a* in the alphabet of the language, the regular expression **a** denotes the language containing just the string *a*. Given two regular expression ***M*** and ***N***, the alternation operator written as a vertical bar **|** makes a new regular expression ***M* | *N***. A string is in the language of ***M* | *N*** if it is in the language of ***M*** or in the language of ***N***. Thus the language of ***a* | *b*** contains two strings *a* and *b*. Using two regular expression ***M*** and ***N***, in a sequence makes a new expression which is the concatenation of both ***M*** and ***N*.** If ***M*** matches *a* and ***N*** matches *b* then ***MN*** matches *ab*. Repetition can also be matched. Given a regular expression ***M*** matches *a*, ***M**** would match any sequence of the letter *a* such as *a*, *aa*, *aaa*, or *aaaa*. Tokens can be grouped into symbols, identifiers and constants. For example some of the token types identified in the javaprec parser are:

Symbols

QUESTION	*'?'*
SEMI	*';'*
LAND	*"&&"*
AND	*‘&’*

LBRACK	*'['*

Identifiers

IDENT	*('a'..'z' | 'A'..'Z' | '_' | '$') ('a'..'z' \| 'A'..'Z' \| '_' \| '0'..'9' \| '$') **

Constants

CHAR_LITERAL	*'\'' (ESC | ~'\'') '\''*
HEX_DIGIT	*('0'..'9' | 'A'..'F' | 'a'..'f')*

Keywords

IF	*“if”*
WHILE	*“while”*

2.7.2 Token Delimiters and Lookahead

The problem with scanners is that some tokens can be matched by different regular expressions. For example both *if* and *while* can be matched by *IDENT* or *IF* and *WHILE* respectively. Similarly, the string && would match both *AND* and *LAND*. To solve this issue there are two disambiguating rules used that will determine which expression to match for every case [30].

1. When a string can be either an identifier or a keyword, keyword interpretation is preferred.
2. When a string can be a single token or a sequence of several tokens, the single token is preferred.

The second rule is often referred to as the principle of longest substring [30], which is the longest string of characters that can match a regular expression. This causes a new problem to arise of exactly how to determine the end of a token. In certain examples it is obvious such as, *varOne+varTwo*, where the + delimits the end of the first token. In general, token delimiters consist of two sets. The first containing symbols such as ;, =, (or {. The second set consists of comments, end-of-line characters and white space characters. Java is as a free-form language [30] where white space is ignored, except to delimit tokens. In other non-free-form languages such as FORTRAN, lexical parsers can be more complex as white space and indentation is syntactically significant. Delimiters end token strings but are not actually part of the token; therefore, parsers

have to deal with some sort of lookahead. When a parser encounters a delimiter, it must arrange that the delimiter is not removed from the rest of the input, either by returning it to the input string known as backing-up or by looking ahead before removing the character from the input. In many cases, a single character of lookahead will be sufficient. In the above example varOne is only delimited by a single character +. This character itself is also a token but must remain in the input stream so that it can be distinguished from the increment operator ++. Sometimes a language may require more than a single-character of lookahead. In the case of Java, a scanner must be prepared to back up a possibly arbitrary number of characters. In this case, buffering of input characters and marking places for backtracking become important. The parser in javaprec uses four characters of lookahead. Such a large lookahead, standard for Java lexers, is required to distinguish between different number formats.

2.7.3 Finite Automatas

Regular expressions are a convenient tool for specifying tokens, but they cannot be implemented easily as a computer program. They are instead converted to a formalism called finite automata [31]. A finite automaton (FA) is a mathematical way of describing an algorithm, using states and transitions. This finite state machine takes an input event and the current state and uses a state transition function to determine the next state and the appropriate output event. A deterministic finite automaton (DFA), is an FA where the next state is always uniquely given by the current state. A generalization of this is a nondeterministic finite automaton (NFA). Although this type of automaton cannot be used by a scanner, regular expressions are normally converted first to an NFA. The next step is to convert the NFA to a DFA using an algorithm called subset construction that constructs a subset of all the available state transformations in the NFA [30].

2.7.4 Token Parsers and their Grammars

All that is left after tokenization are the indivisible units of the language; all the white space and comment tokens have been discarded. In free-form languages, it would unnecessarily complicate the parser to have to account for possible white space and

comments at every point. This is why lexical analysis is a separate process from parsing, making it a more reliable approach than one-stage pattern recognition.

The next stage for after tokenization is to process the stream of tokens and determine whether the syntactic structure matches its grammar. A grammar is a formal definition of the syntactic structure of a language. The Java Language Specification is written as a context free BNF grammar [32]. The grammars for most complex languages are described using Backus-Naur Form (BNF) notation or its close relative, Extended BNF (EBNF). To explain language grammars, two simple examples of BNF will be used. The first example will model a fictitious language called *myLanguage*. *myLanguage* consists of two arithmetic operators * and / that operate only on integers. The first stage of designing a grammar is to define all tokens. *myLanguage* only contains three tokens *MULTI, DIV* & *NUM*.

MULTI *'*'*
DIV *'/'*
NUM *('0' … '9') +*

Next, the grammar is constructed; each of the rules in the grammar is called a production or symbol, its name being written on the left-hand side of the rule.

myLanguage ::= NUM ((MULTI | DIV) NUM) ?

This is an example of a *terminal* production, which cannot be divided down any further. Alternatively, this grammar can be written using some *non-terminal* productions as follows:

myLanguage ::= number ((multiply | divide) number) ?

multiply ::= MULTI

divide ::= DIV

number ::= (lowNumber | highNumber) +

lowNumber ::= ('0' … '5')

highNumber ::= ('6' … '9')

This grammar can also alternatively be written in a single production with no token names as follows:

myLanguage ::= (‘0’ … ‘9’) + (('’ | '/') (‘0’ … ‘9’) +) ?*

The ability to write grammars with the same meaning in many different ways allows token handling to be very context sensitive. For example if it is decided that division by zero is not allowed this can be handled as follows:

myLanguage ::= NUM ((MULTI NUM | DIV (‘1’ … ‘9’) +)) ?

If a parser finds anything other than what is allowed in its grammar, the expression being scanned is considered invalid. One of the parser’s main jobs is to determine the validity of any expression it is passed. This is used in javaprec to ensure correct and informative error handling which will be discussed later in the report.

The second example of BNF will be used to model a comma-separated list of words. To construct this grammar will need two tokens *COMMA* and *WORD*.

COMMA ','

*WORD ('a'..'z' | 'A'..'Z' | '_' | '$') ('a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$') **

Only one production is required for this grammar as follows:

commaList ::= (WORD COMMA) WORD+*

This production defines a correct list as containing one or more comma separated words, for example: *this,is,a,comma,separated,list,of,words*.

2.7.5 Top-down Parsers

Top-down parsers, which start by recognizing productions at the leftmost derivation, come in two forms, backtracking parsers and predictive parsers. A predictive parser attempts to predict the next construction in the input string using one or more lookahead tokens, while a backtracking parser will try different possibilities for a parse of the input, backing up an arbitrary amount in the input, if one possibility fails. Although backtracking parsers are more powerful, they are not commonly used because they are much slower.

2.7.6 LL(k) Parsers

Top-down parsers can also be separated into recursive-descent parsers and LL(k) parsers. Recursive-descent parsers are normally used for handwritten parsers, whereas the most common parser generators generate LL parsers. Two common LL parser generators are for example, Java Compiler Compiler (JavaCC) and Another Tool for Language Recognition (ANLTR). The first L in the name LL(k) refers to the fact that an input string is transversed from left to right. The second L refers to the fact that a leftmost derivation is traced out for the input stream. The k refers to the amount of lookahead, for example, an LL(1) parser would uses one token of lookahead. LL parsers parse input without backtracking. Starting at the leftmost derivation, the parser replaces the leftmost non-terminal symbol with the matching definition of a grammatical rule. It repeats this process until all non-terminal symbols are replaced by terminal symbols. LL(k) parsers require k tokens of lookahead to decide which rule to apply from a given grammar.

2.7.7 Bottom-up & LR(k) Parsers

The most general form of bottom-up parser is called the LR(k) parser. The L like with a LL(k) refers to the fact that an input string is transversed from left to right. The R refers to the fact that a rightmost derivation is traced out for the input stream. Two examples of bottom-up parser generators are A Lexical Analyzer Generator (LEX) and Yet Another Compiler-Compiler (YACC). They are both early LR(k) parser generators and are still used today. LR(k) parsers have some advantages over LL(k). For example, the weakness of LL(k) parsing techniques is that they must predict which production to use, having seen only the first k tokens of the right-hand side. LR(k) is a more powerful technique. It is able to postpone a decision until it has seen input tokens corresponding to the entire right-hand side of a production and k more input tokens beyond. Although LR(k) parsers are more powerful, their grammars are more complex and very difficult to debug. Most parsers have code embedded in the productions, which performs certain tasks when a production is completed. For example, this embedded code can often be used to store field types in a symbol table. It is much more difficult to determine the

order in which the embedded code will executed with an LR(k) parser, therefore, most modern parsers are LL parsers.

2.7.8 ANLTR and JavaCC

The two parser generators considered for this project were, JavaCC and ANLTR. Theses parsers generators were chosen as they are the most modern and widely used parsers available.

JavaCC is a Java parser generator written in Java. First developed at Sun Microsystems, the Java Compiler Compiler project, formerly known as Jack, began as an effort to build a Java parser for QuickTest (now known as JavaSpec), SunTest's Java API testing tool [33].

ANLTR is a parser generator written in both Java and C++. Originally called YUCC, ANLTR was created as part of the Purdue Compiler Construction Tool Set (PCCTS). PCCTS began as a project for a graduate course at Purdue University [34].

The main difference between JavaCC and ANLTR, is that ANLTR has a linear approximate lookahead, as opposed to full LL(k) [34], where as JavaCC is both LL(1) and LL(k) at such points where a greater lookahead is required [35]. Another difference is that ANLTR has been an open source project since the start of writing RJCE. JavaCC, on the other hand, has only become an open source project, in the last two months.

2.7.9 Parse Trees

Both ANLTR and JavaCC generate a parse tree by default according to the production rules. As the parser transverses each production, a new node is added to the parse tree making a record of the rules and tokens used to match the input. The purpose of building a parse tree is to allow easy translation of the input text, such as compilation in the case of a Java source file.

2.8 XML

2.8.1 History

In the late 1960s, the United States Department of Defense sponsored an important conference. At this conference, they brought together several dozen graduate students at the University of Illinois. Shortly after they proceeded to implement what quickly became called the ARPA-Net, the grandparent of today's Internet.

Initially, use of the Internet was limited to universities and research institutions but soon the military became a big user. Eventually, the government decided to allow access to the Internet for commercial purposes. There was some resentment to this among research and military communities; it was felt that response times would become poor as the net became saturated with so many users.

In fact the exact opposite has occurred. Businesses rapidly realized that by making effective use of the Internet they could tune their operations and offer new and better services. This caused massive investment and expansion of the Internet and the development of several new programming languages and protocols.

Even though the Internet was developed more than three decades ago, the introduction of the World Wide Web was a relatively recent event in response to the Internet's rapid expansion.

In 1989, Tim Berners-Lee of CERN, the European Laboratory for Particle Physics, began to develop a technology for sharing information by using hyperlinked text documents, which are document that are linked together over the hyper net. He wrote protocols to form the backbone of his new hypertext information system, which he termed the World Wide Web.

The worldwide web enabled standardization of the increasing amounts of data being transferred between businesses. The files of the worldwide web where written in a language, called HyperText Markup Language or HTML.

HTML was originally conceived as a set of tags to mark the logical structure of a document; headings, paragraphs, links, quotes, code sections, and the like. This type of

language is called a mark-up language. As more control was demanded from the web HTML acquired more and more tags and attributes to control presentation; fonts, margins, tables, colours. The documents became more complex and it seemed the original goal of simplicity and universality were starting to slip away.

The remedy was widely seen as separation of content from presentation. Therefore, a new language was developed for the transfer of data between businesses, called Standard Generalised Markup Language (SGML). Soon it was felt that SGML was also too restrictive, so an extensible language was developed. An extensible language, by definition, is a language that can be expanded or extended to fit the requirements of programmers. Extensible Markup Language (XML) is one such extensible language. XML is designed to enable data to be easily transferred between computers in a standard manner, Figure 10 and Figure 11 shows examples of XML.

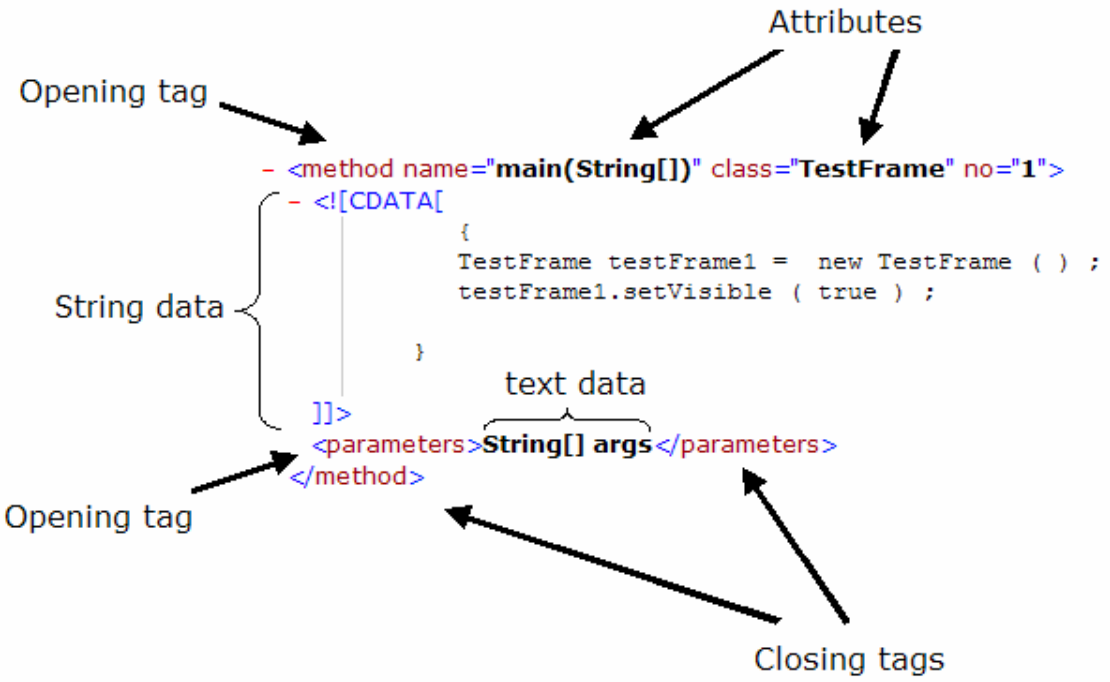


Figure 10, Example section of an XML files

```
<?xml version="1.0" encoding="UTF-8" ?>
- <file url="D:\test\iotestprogram\TestFrame.java">
  - <imports>
      <import>import java.awt.*</import>
      <import>import java.awt.event.*</import>
      <import>import java.io.*</import>
      <import>import java.util.*</import>
      <import>import javax.swing.*</import>
      <import>import javax.swing.event.*</import>
      <import>import rom.gui.*</import>
    </imports>
  - <method name="main(String[])" class="TestFrame" no="1">
    - <![CDATA[
                {
                TestFrame testFrame1 =  new TestFrame ( ) ;
                testFrame1.setVisible ( true ) ;

            }

      ]]>
      <parameters>String[] args</parameters>
    </method>
  + <method name="setUpButtons()" class="TestFrame" no="2">
  + <method name="setUpTextPanel()" class="TestFrame" no="3">
  + <method name="changeButtonColour()" class="TestFrame" no="4">
  + <method name="changeButtonFont()" class="TestFrame" no="5">
  + <method name="saveText()" class="TestFrame" no="6">
  + <method name="loadText()" class="TestFrame" no="7">
  + <method name="clearText()" class="TestFrame" no="8">
  + <method name="warnUserTextPane()" class="TestFrame" no="9">
  + <method name="warnUser(String)" class="TestFrame" no="10">
  + <method name="getConformation(String)" class="TestFrame" no="11">
  </file>
```

Figure 11, Example XML file, produced by javaprec

An XML file is divided into several elements normally starting with an opening tag and finishing with a closing tag. The opening tags can also include attributes which generally provide information that is not part of the main element's data. In Figure 10 and Figure 11, for example, the name and class attributes are not part of the main method data but are still important pieces of related information. The one important strength of XML is the hierarchical structure of data. One element can be contained inside another element or can itself contain several other elements forming a tree like structure. For example, consider a phonebook stored as an XML file. The top element

might be <*phonebook*>. This could contain an <*entry*> element each which in turn contains a <*phone_numbers*> and <*address*> element.

```
<?xml version="1.0" encoding="UTF-8" ?>
- <phonebook>
  - <entry name="tom">
    - <phone_numbers>
      - <land_lines>
          <number name="tom">0208 385 1652</number>
        </land_lines>
      </phone_numbers>
    - <address name="tom">
        <house_number>91</house_number>
        <streat>Princeston Avenue</streat>
        <postcode>E5 FS1</postcode>
      </address>
    </entry>
  - <entry name="fred">
    - <phone_numbers>
      - <mobiles>
          <number name="fred">07813 022 722</number>
        </mobiles>
      </phone_numbers>
    - <address name="fred">
        <house_number>38</house_number>
        <streat>Berkly Streat</streat>
        <postcode>N10 EJ3</postcode>
      </address>
    </entry>
  + <entry name="bob">
  </phonebook>
```

Figure 12, XML phone book example

Figure 12 shows an example of such a phone book XML file. Alternatively, this can be represented using an XML Scheme diagram that shows the structure of the XML data as in Figure 13.

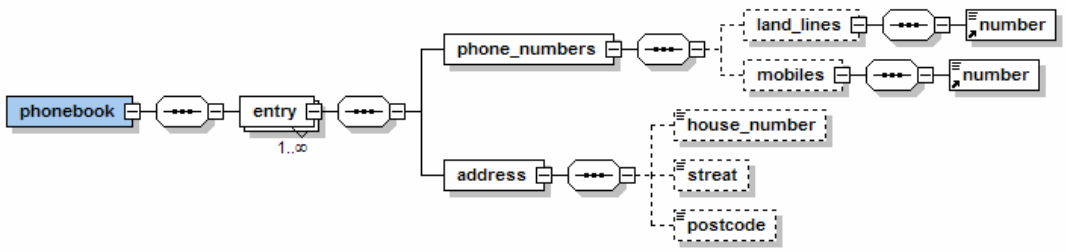


Figure 13, XML Scheme diagram showing the tree structure of the phone book data

2.8.2 Reading and Writing XML

XML files contain normal text and therefore can be read using simple text scanning. A more sophisticated approach is to use the Simple API for XML (SAX). SAX is an event-driven, serial-access mechanism for reading but not writing XML documents. An alterative approach is to load the XML file into memory as a tree structure so that the different branches can be traversed backwards and forwards extracting the required data. This type of tree structure is called a Document Object Model (DOM). There are several techniques for creating DOMs in Java these include JDOM, DOM4J and Java API for XML Processing (JAXP).

2.8.3 SAX

In SAX the event driven model works by generating an event at the start and end of each element as it scans XML data. These events are then used to trigger user-defined actions, such as reading the contents of a particular element.

SAX suffers from several problems, for example, in general it requires a lot more programming than DOM [36]. Another problem is that, SAX acts like a serial I/O stream, and therefore it is not possible to go back to an earlier position or leap ahead to a different position.

2.8.4 DOM

DOM is a standard defined for representing XML and HTML documents in memory. DOM allows more sophisticated operations than SAX such as, navigating the structure and adding, modifying or deleting elements. The main limitation with DOM is that an entire XML file is loaded into memory before any data can be extracted using DOM. If only a small amount of data had to be extracted from a large file then the complete file would have to be loaded into memory. This can be extremely inefficient for large XML files.

3. RJCE Design Considerations

3.1 Overview

Our system is called Runtime Java Class Editor (RJCE). RJCE simulates the effect of editing methods at runtime, allowing the edits to be applied to any combination of objects, such as an array, or a complete class.

Three different techniques were considered while performing the overall design for RJCE. The first technique we looked at was to alter the byte code of a running program as it resided in memory. The next possibility was to use class loaders. The final approach involved parsers and interpreters rather than class loaders. The following sections describe these approaches in turn.

3.2 Altering Byte Code

Java byte code runs from the method area on a JVM. It is therefore theoretically possible to alter a running program by altering its code in the method area. There are two main problems with this approach.

- Firstly, there are no guarantees in the JVM specification for management of the method area, as examined in the background section 2.3.2. It is, therefore, difficult to predict how a JVM might respond if an attempt was made to write code to the method area.
- The second problem is caused by the fact that programs are stored in the method area in byte code. Byte code represents a Java program it its lowest level language, after it has been compiled. Therefore, to edit a program's byte code, a good understanding of Java compilation and the production of byte code would be needed, as well as a considerable amount of translation between byte code and source code. For example, a programmer editing methods would probably want to programme in Java and not byte code, therefore some translation between the two and perhaps even decompilation would be required.

Due to the level of uncertainty and the amount of new concepts that would have to be learned, such as compilation and decompilation, this approach was not adopted.

3.3 Class Loaders

Although altering running code in a JVM can be difficult, class loading goes some way to reducing this problem. It is generally not possible to reload classes that are loaded using the system class loader, although this is not the case for user defined class loaders. As discussed in section 2.3.6.2, this is because to unload a class there must be no references from the root set of references to the class, or any of its instances. If this is the case, the class is then said to be unreachable from the root set of references. Unreachable classes are reclaimed by the garbage collector, and therefore unloaded. Every class loaders maintain a reference to each class it loads. Therefore, to unload a class the class loader that loaded it must also be unreachable from the root set of references. System class loaders are never unreachable. Therefore, they cannot be used to load classes, which are subsequently unloaded.

Unloading classes requires two steps. The first step is to load classes that will be unloaded with a user defined class loader. The next step is to remove all references from the class that is being unloaded. The first step requires all references to the user defined class loader to be removed. Once this is done, all references to instances also have to be removed. To allow for this a separate class loader can be used for each class, although this is potentially a memory intensive solution. This approach also suffers from the problem that before any method is edited all running versions of it would need to be unloaded. Therefore, a program's running code could not edited, but instead only reloaded, therefore losing field values.

3.4 Rerouting Execution

The final approach investigated avoided the use of class loaders in favour of using two alternative technologies, called parsers and source code interpreters. This design is based around two main programs, javaprec and Runtime Object Modifier (ROM). javaprec's main function is to compile a program adding extra code, to allow method editing. Each method has a hook added to the start this allows the execution to be rerouted via an interpreter. ROM's main function is to enable runtime method editing of any class that has been compiled by javaprec. ROM is also used to handle the

interpreter and all the runtime functions of RJCE. Once an application has been compiled in javaprec, then ROM can load its methods in the source editor at runtime. This is loaded from an XML files that javaprec generates during its precompilation stage. Java programs compiled with javaprec will execute normally until they reach a method that has been edited. The execution of an edited method is rerouted via a source code interpreter. This continues until its rerouting switch is turned off again. ROM is used to control rerouting switches for all methods. ROM is also used to save edited code after it has been initially loaded from the XML. RJCE allows not only methods to be edited any number of times but also the original (compiled) method to be reused at any point, by reloading the original source from the XML file. Method alterations can also be applied in three modes. Instance mode applies alterations to a single instance only. Group mode applies alterations to an entire groups such as an *Object[]*, *Collection* or *Map*. Group mode is implemented by saving the same edited method and rerouting status to every *Object* in a group. Class mode applies alterations to an entire class. Class mode is implemented is a similar way to instance mode except that static fields are used rather than instance fields. From Figure 14 which shows an overview of RJCE, javaprec and ROM can be seen

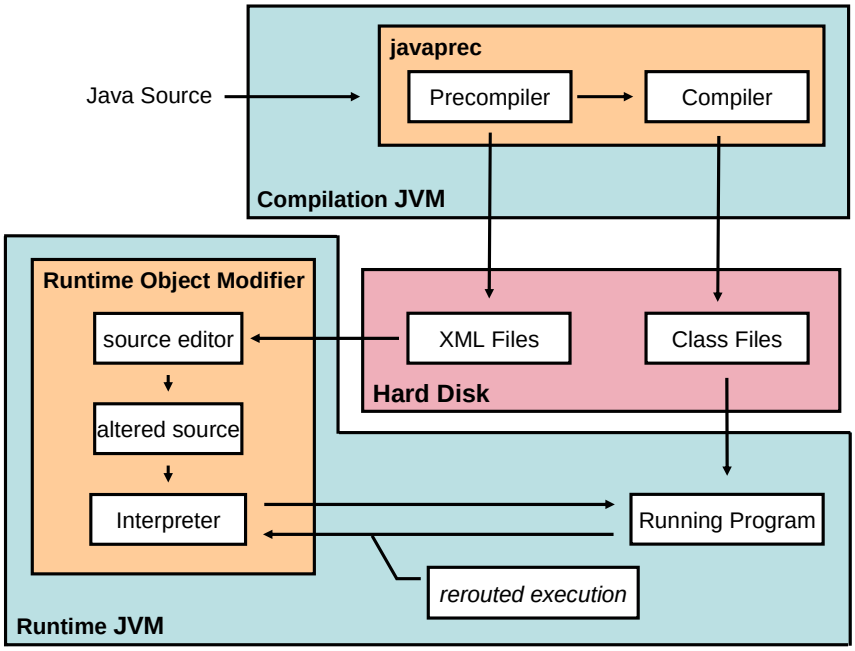


Figure 14, RJCE Overview

javaprec produces both Java byte code class files and XML files during its two stages. During the first precompilation stage, XML is used to store a copy of each method in each class. This XML is later used by ROM to load the original version of methods, prior to alteration at runtime, in the source editor. The precompilation stage is also used to add rerouting if-statements, also called hooks, to the top of each method. If the condition in the if-statement is satisfied then execution of the method is rerouted to an interpreter. The precompiler also adds methods, instance and static fields to each outer-class. Inner-classes have no variables added to them as they use the variables in their declaring (surrounding) class. Anonymous classes are ignored because it is not possible, in general, to use reflection on anonymous-classes. The fields added to outer-classes are used to store method-rerouting data, such as *boolean* arrays which are used to control the rerouting if-statements. The methods are used as accessor functions guaranteeing field access without using reflection as explained in section 4.1.8.3. To edit a method, ROM is used as an Integrated Development Environment (IDE), where methods can be retrieved and edited like a normal graphical source editor. ROM includes all basic graphical source editor features such as, syntax colouring, undo, redo, cut, copy, paste, find, replace. ROM also includes a class browser, which allows the class hierarchy including super-classes, interfaces and inner-classes to be explored. The class browser enables field values to be viewed or instead loaded in ROM enabling the field's methods to be edited. ROM is designed to be as usable as possible. For example, some potential users were asked about the look and feel of ROM. There were several conflicting opinions on the best way to layout the source editor and class browser. In an attempt to please everyone a view menu was added to the menu bar which allows customization of the layout of the *JSplitPane* containing both the source editor and the class browser.

javaprec's main functions are:

- To extract the body of each method and write this to an Extensible Markup Language (XML) file to be used later by the interpreter.
- To precompile Java source files adding rerouting if-statements, class level fields and accessor functions for the newly added class level fields.
- To compile the java source files produced by the precompiler.

ROM's main functions are:

- To allow runtime editing of methods in classes that have been compiled by javaprec.
- To handle control of method edits and rerouting switches, providing features such as persistence of method edits and the ability to revert an edited method to its original code.
- To allow three edit modes:
    a. Instance mode: apply method edits to a single object or instance.
    b. Group mode: apply method edits to all objects in a group. (Object groups being defined by arrays, collections or maps.)
    c. Class mode: apply method edits to all instances of a class.
- To allow edited methods to interpret correctly by altering the input code to take account of the change in namespace.
- To introspect on fields, inner-classes, super-classes and super-interfaces, presenting the information to the user in a tree structure.
- To allow new objects or object groupings to be loaded, for method editing, from any field in the class browser referencing an object or grouping of objects (Object[]. Collection or Map) that has been compiled by javaprec. A history is also maintained that is used to return to previously loaded objects.
- To allow the user to load ROM in several forms, including a basic form that only contains a text editor and a few buttons, and a full form with class-browser
- To allow execution of Java scripts.

4. javaprec Design & Implementation

javaprec works in two stages, as shown in Figure 14. The first stage is a precompiler, which inserts new source code, such as fields, methods and if-statements into the input source code. The second stage is the compilation of the precompiled Java source. The javac compiler, from Sun Microsystems, in the *com.sun.tools.javac* package, is used for compilation to ensure reliable and efficient performance.

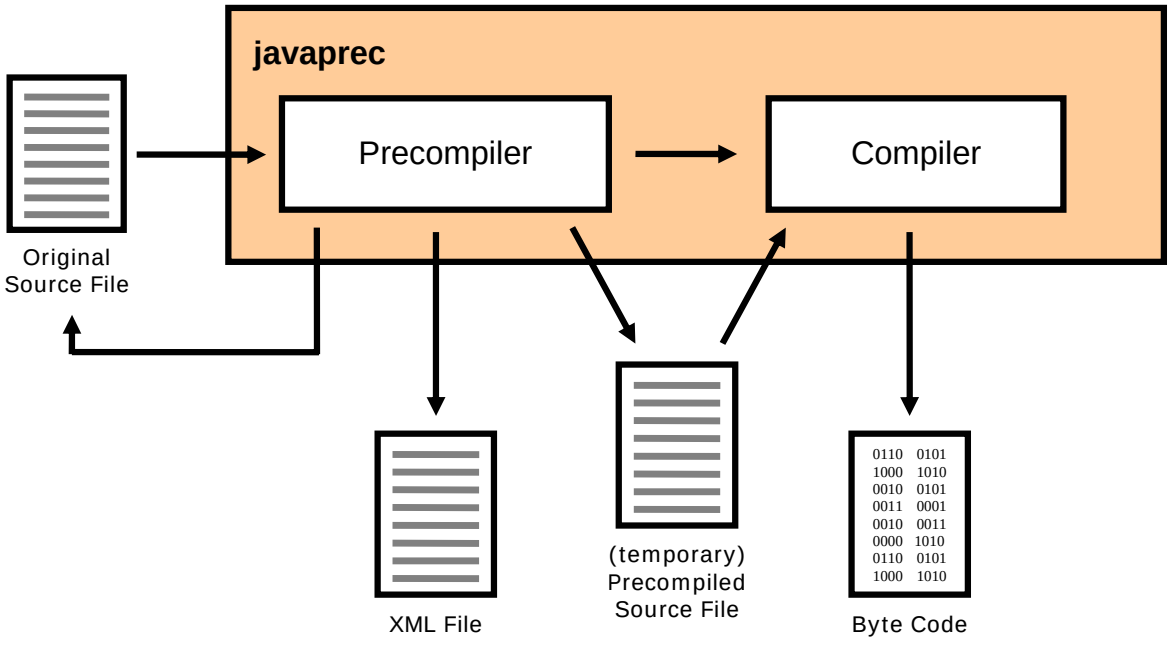


Figure 15, javaprec overview

4.1 The Precompilation Stage

The precompilation stage has two main roles, to add rerouting code and to extract method code. The code added for rerouting is as follows:

- if-statements added to the start of non-static methods,
- class level fields added to outer classes
- accessor functions added to outer classes, non-static inner-classes but not anonymous classes or classes defined in the body of a method
- each class that has accessor functions added to it is also extended to implement an interface called *rom.interpreter.rjce_precompiled*. This interface is used by ROM to guarantee access to the accessor functions, without using reflection.

The method code is extracted so that when a method is altered there is a copy of the original Java source to work from. This original copy is also used in ROM to allow users to refresh an altered method with its original code. Import statements are also extracted so that interpreted methods have access to the same namespace as the original method. For example, if a variable declaration declares a non-primitive type the class must be located by the class loader. Interpreter uses the import statements so that its class loader can correctly locate class, such as those found in the Core API.

Three different approaches were developed when designing the code parser, with each an improvement on the one before. These were character recognition, pattern matching and parsing with a parser generator's parser.

4.1.1 Character Recognition

The first approach adopted was to recognize each character one by one and compare that against the previous characters passed to determine the location of keywords within sentences. This approach was found to be successful for basic structures in a class file such as function signatures and class names. Character recognition, although extremely fast, is also very complex. The complexity is partly due to the limited context information that can easily be obtained by a simple character parser. For example, it is legal Java syntax to write a nested class within a method or as an inner-class within an enclosing class. To handle this correctly the information about what has previously parsed has to taken into account. For example, a method level class can be determined by the fact that a method signature has been already parsed but no final method body

closing brace has been parsed. Maintaining information about previous parsed code makes a character parser more complex.

```
class ActionControl {

  public ActionControl[] a = {new ActionControl( ), new ActionControl( ), new ActionControl( )};

  public ActionControl(int input) {
  }

  ActionControl( ) {
  }

  public ActionControl doSomethingPublic(int input) {
    class MyAction {
      public void actionPerformed( ActionEvent e) {
        doNothing();
      }
    }
    return this;
  }

 AbstractAction action = new AbstractAction() {
   public void actionPerformed( ActionEvent e) {
     doSomething();
   }
 };

 class MyAction {
   public void actionPerformed( ActionEvent e) {
     doSomethingElse();
   }
 }

}
```

Figure 16, Example precompiler input

Figure 16 for example, shows a class that is legal Java syntax but would be difficult to parse correctly. We see that *public ActionControl(int input)* and *ActionControl()* are both constructors and therefore should not have code added to them. Whereas public *ActionControl doSomethingPublic(int input)* is a method and should have code added to it. Rerouting code should not be added to constructors because they might contain a call to super() or this() which must always be on the first line of a constructor. This causes a conflict with the rerouting code which must also always be on the first line. Another example is *MyAction* which is declared in method *doSomethingPublic(int input).* This should not have any rerouting code added and neither should its methods. However, *MyAction* declared as an inner-class to *ActionControl* should have rerouting

code added. This example does not mention quotations, comments or multiple white spaces, which all add to the complexity of parsing. For example, although braces are used to delimit every class or method body, braces can also appear as part of a string, as a single character literal, or in both multiple line and single line comments. Comments can also appear between any two characters and end-of-line characters can be used liberally in some places but not in others.

4.1.2 Pattern Matching

Pattern matching simplifies parsing by being able to match sets of characters such as “zero or more” or “anything except”. This was, therefore, the next approach adopted in an attempt to reduce some complexity in the parser.

Pattern recognition not only considers sets of characters rather than just individual characters, but also allows any amount of white space or end-of-line characters to be ignored. Pattern matching is a much more powerful method for parsing than character recognition. This is because specific patterns or sentences can more easily recognized using the regular expressions. A regular expression is a pattern of characters that describes a set of strings. Regular expressions were introduced to Java 1.4 in a new package *java.util.regex*. Regular expressions are used to express the different combinations searched for in the parser input source code. A special set of symbols is used in regular expression that allows many different character combinations such as “one or more” or “any of the following”, this is described in detail in section 2.6.

Although pattern recognition is a powerful technique, it was often difficult to determine a constructor from a method or array field with initializer. Constructors are not only syntactically very similar to methods, but there are also a number of optional words that have to be considered. Figure 17 shows an example of a legal Java class:

```
class SimpleClass {

  public SimpleClass[] a = {new SimpleClass( ), new SimpleClass( ), new SimpleClass( )};

  public SimpleClass(int input) {
  }

  SimpleClass( ) {
  }

  public SimpleClass doSomethingPublic(int input) {
    return this;
  }

  SimpleClass doSomethingProtected( ) {
    return this;
  }
}
```

Figure 17, Example input for precompiler

The above example shows how a protected method with no *protected* qualifier can easily be confused with a constructor without the ability to backtrack and recheck ambiguous sentences. This is not available with pattern recognition. To improve the situation, context information can be maintained, such as: which class is currently being parsed, so that method names can be compared against the class name to check for a constructor. Having to maintain context information, in a similar manner to the previous character recognition approach, makes the design of this parser more complex and prone to errors. For example, if at any point the context information is incorrect, the remaining text will also be incorrectly recognized. Incorrect recognition is not a problem if it can be detected. The problem arises when an undetected error occurs. For example, it is very important to be able to detect constructors accurately because a call to *super(…)* or *this(…)* is required to be on the first line of a constructor. If there are any lines of code prior to a call to *super(…)* or *this(…)*, the compiler will fail. The rerouting if-statement must also be on the first line of a method. If a constructor with a call to *this(…)* or *super(…)* is mistaken for a method, then the source file will not compile.

Although using a pattern matcher is a good technique, it suffers from a lack of error detection. A pattern matching approach lends itself to simpler predictable parsing problems where complex error handling is of little or no importance, such as syntax colouring as used in the source editor of ROM.

4.1.3 Parser Generators (two stage pattern matching)

Error handling should be an important part of any compiler including javaprec. For this reason, the next approach we adopted was to use a parser generator, because modern parser generators generally have sophisticated error-handling systems. For example, if an unexpected token is found, an exception is thrown detailing the token found and the expected token. The exception also gives other information such as line numbers and column numbers for the start and end of the token. Both javaprec and ROM catch all exceptions thrown by their parsers and output informative error messages.

A parser generator is really a compiler. Instead of taking program source code as its input, a parser generator takes a grammar. The output of a parser generator is a parser, or compiler, that recognizes the programming language described by the grammar. In short, a parser generator is a compiler compiler, or a parser compiler. Parser generators and the parser they generate are discussed in more detail in section 2.7.

The two parser generators considered for this project were, JavaCC and ANLTR. These parser generators were chosen because they are amongst the most modern and widely used parsers available. Both JavaCC and ANLTR, also have another advantage that they both generate parsers which work approximately like LL(k) parsers. LL(k) parser are not to difficult to write grammars for, unlike some parsers such as LALR parsers. With an LALR parser, it can often be difficult to predict the order of tokens and productions being traversed. This means the order of executed for embedded code can also often be difficult to predict.

At the time this project started, it was decided that RJCE should be open source. Because JavaCC was free, but not open source, and ANLTR was fully open source ANLTR was chosen. Since the initial design decision to use ANLTR, JavaCC has become fully open source. The Abstract Factory design pattern [37] used enables the parser to be easily changed or upgraded. For example, a JavaCC parser could be added as shown by the dashed box in Figure 18.

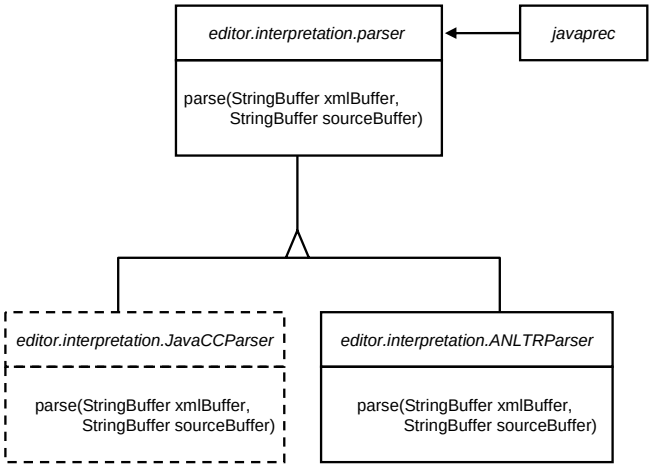


Figure 18, The Abstract Factory design pattern

4.1.4 Parse Trees

javaprec has no use for a parse tree, normally produced during a parse. To improve the speed of precompilation, this feature is disabled in the javaprec parser. Instead, the javaprec parser has two output buffers, an XML representation and a precompiled copy of the input file, as shown in Figure 19. No direct copying except identifiers, comments and literals is done between the input and output files. Normally, parse trees are generated when productions are traversed, but instead the code is sent to the two output buffers. For example, when each terminal token is matched, such as a token for *if* or *else*, the correct characters are added to the XML and source code buffers. The only exceptions being for identifiers, comments and string or character literals tokens. For these tokens, the text is directly copied from the source file to the output buffers.

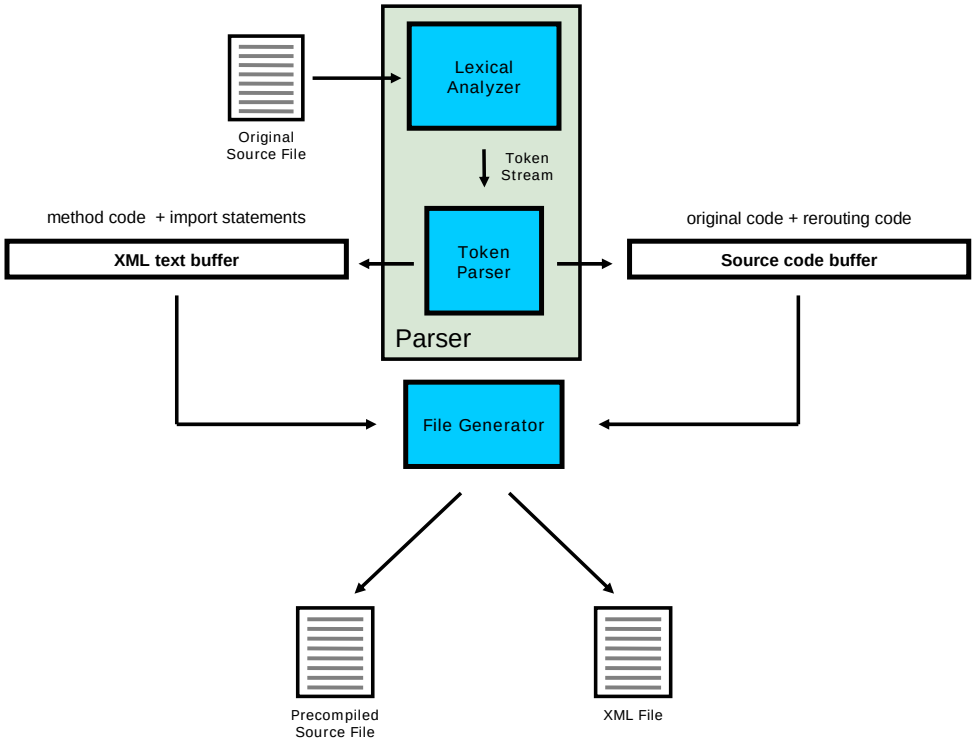


Figure 19, Precompiler overview

4.1.5 Source Beautification

The XML file produced contains the source used for editing methods in ROM. It is, therefore, essential to ensure the source stored in the XML output is human readable. For example, the each statement ending in a semicolon should be on a separate line and indented to indicate the code block it belongs to. To achieve this, syntax indentation is incorporated into the javaprec parser. The number of opening braces and closing braces are maintained in an integer called *braceNo*. Prior to each line being output, *braceNo* is used to reference a cell in an array of white space strings. Each subsequent location in the array has three spaces more than the previous location to represent a deeper indentation. It was important to check that this relatively unimportant feature does not compromise speed. Therefore, tests were run before and after adding the beautifying code. The results showed a 5% reduction in speed with an error margin of 2%. This small reduction in speed is outweighed by the advantage of having syntax indentation.

4.1.6 XML Output & IO Buffering

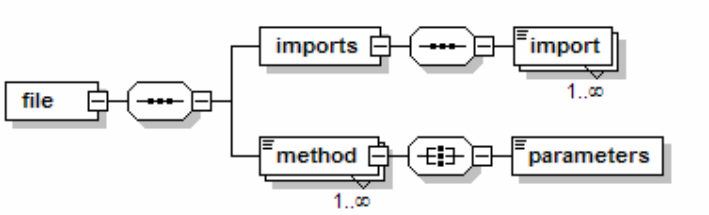


Figure 20, Scheme diagram showing structure of XML output

The structure for the XML file is kept simple with a relatively flat hierarchy no more than two elements deep, not including the root <*file*> element as can be seen in Figure 20. The purpose of this is to ensure that a simple string search can be used to read the correct information from the XML file. A simple string search was preferred to SAX. SAX is an event-driven, serial-access mechanism for reading but not writing XML documents. The event driven model works by generating an event at the start and end of each element as it scans XML data. These events are then used to trigger user-defined actions, such as reading the contents of a particular element. The javaprec XML output files are designed to be simple; SAX is unnecessarily complex for such a simple XML file. DOM was also avoided because DOM is memory intensive. DOM represents the complete tree structure of a file in memory and therefore has to load a complete XML file to memory before any data access can take place. RJCE uses, a *StringBuffer* as an IO buffer for all file reading and writing. This is done so to reduce the number of blocking IO activities to one per file. The two IO buffers, used in javaprec, are shown in Figure 19.

The naming structure designed for the XML file enables two kinds of access to method bodies stored in the <*method*> elements. A method can be uniquely identified by its class and number or by its class, name and parameters. Therefore, the XML file for javaprec was designed to allow a method body to be extracted from an XML file with either the class name and method number or the class name method name and method parameters. To achieve this, each <*method*> element has three attributes, a class name, a method name and a method number. The attributes are arranged so that a text scanner

can locate the correct <*method*> with only the class name and method number or only the class name, method name and parameters. The design of the XML file, therefore, makes it possible to relate a class name, method name and set of parameters to the method's number. This is important because the method number is an important part of tracking method switches and method edits, but cannot be determined using reflection alone. It is used to index *boolean* and *String* arrays that contain the method rerouting switches and any previous method edits.

Import statements are included in the XML file to enable the interpreter running the edited method to have access to the same namespace as the original version of the method. The import statements are passed into the interpreter importing the class namespaces, such as *java.lang.** to each method. This is essential to ensure all the correct classes are found by the interpreter. Without this feature, altered methods would potentially have problems finding classes that were available in the original method.

To extract both the method code and import statements the XML file is first completely read into a *StringBuffer*. Next, a simple algorithm using only two *String* methods, *indexOf(String str, int fromIndex)* and *split(String regex)* separates out the relevant information.

Buffering the reading and writing operations doubled the speed of the precompiler when it was introduced, proving that blocking IO was a bottleneck before introducing buffers. Although buffers greatly improve the speed, they increase the risk of running out of memory if a given file is too large. To reduce the severity of this problem when an *OutOfMemoryError* is detected, the precompiler sets all buffers to null to allow the garbage collector to free memory. Following this, the file is copied but this time without any precompilation or the use of buffers. Now the compilation stage can still proceed as all files the necessary files will be in the correct location. The only difference is that the class being scanned when the *OutOfMemoryError* occurred, would not be editable in ROM because is would have no method rerouting code, or XML file. Tests were conducted on a computer with 512MB of RAM, and 160MB available memory at the start of execution. The results showed that fabricated files of over 2MB, consisting of over 60,000 lines and larger than any Java SDK file caused an

OutOfMemoryError. In each test case, the *OutOfMemoryError* was correctly caught by javaprec, which compiled the file, but without precompilation.

4.1.7 Instance, Group & Class Mode

To make it possible for method edits to be applied to different combinations of objects, three edit modes are available. In instance mode, method edits are only applied to a single instance or object. In group mode, method edits are applied to a "group" of objects. A "group" is an easily quantifiable collection of objects. Groups are made up of all the objects contained in either: an array, a *Collection* or a *Map*. Group mode is, therefore, only available for methods of objects contained in an array, a *Collection* or a *Map.* The *java.util.Collection* interface is used because it is the super-interface to *List* and *Set*. This includes not only *Set* and *List* objects found in the *java.util* package but also collections used through the Java API, such as in both the Abstract Window Toolkit (AWT) and Swing Graphic User Interface (GUI) classes. Both *Collection* and *Map* objects are converted to an array so that all three types of object groupings can be handled in a similar manner. *Collection* objects are converted directly to an array via the *Collection* method *toArray()* and *Map* objects are converted to an array via an intermediate *Set*, using the *Map* method *entrySet()*. Group mode method edits are applied to all instances in the array containing the instance being edited. In class mode, method edits are applied to a complete class. Class mode edits rely on static fields to apply method edits to all instances both retrospectively and progressively. This, therefore, includes both instances already created and instances not yet created.

4.1.8 Method Rerouting

4.1.8.1 if-statements

```
/* START OF RJCE PRECOMPILER METHOD CODE SECTION */
if( rjce_interpretMethod[2] | rjce_interpretGroupLevelMethod[2] | (rjce_interpretClassLevelMethod != null && rjce_interpretClassLevelMethod[2]) ) {
  if(rjce_interpreter == null) rjce_interpreter = rom.interpreter.RJCEExecutionFactory.getExecuter();
  rjce_interpreter.initializeExecuter();
  rjce_interpreter.set("precompiledObject", precompiledObject);
  java.lang.Object _rjceReturnObject = rjce_interpreter.interpretMethod("startSetup(Object)", 2, this);
  return;
}
/* END OF RJCE PRECOMPILER METHOD CODE SECTION */
```

Figure 21, Typical method rerouting if-statement

There are several types of method that javaprec has to treat differently. To ensure no Java language rules are broken, a different form of the rerouting if-statement is used in each type of method, one example is shown in Figure 21.

The first, and most obvious, method classification to consider is the return type. When an executer evaluates code, it stops execution when either it runs out of statements or reaches a return statement. If the return statement includes a field, this is returned as an object by the executer. If the field has a primitive type then this is returned in the correct object wrapper, such as *Integer* for an *int*. All non-void methods, therefore, have to cast the returned *Object* to the correct type. Once this done the new value in the correct type is returned from the method allowing the rest of method body, which is the original method, to be skipped over and not executed.

The next method classification to be considered is whether the method is in an anonymous, method or static class:

- Java Reflection, in general, cannot be used with anonymous classes because it is not possible to access the correct Metadata.
- Java Reflection can usually only be used on method classes in the method where the class is defined.
- Static classes are not allowed to have non-static members. This is a problem, because the fields required for the method rerouting are, in general, not static.

javaprec, therefore, ignores anonymous, method and static classes but adds rerouting code to all other classes. Static methods are also ignored because it is not possible to access non-static instance fields from within a static method.

The rerouting if-statements test elements in three *boolean* arrays, which the method number is used to index. Using three arrays allows for the three editing modes: instance, group and class.

4.1.8.2 Class-Level Fields and Accessor Functions

Table 4, class-level rerouting fields

Number	Type	Purpose
2 local, 1 static	boolean[]	control method rerouting
2 local, 1 static	String[]	store users altered code
2 local, 1 static	String[]	store altered code, parsed for executer
1 local	String	store location of associated XML file

```
/* START OF RJCE PRECOMPILER CLASS CODE SECTION */
public rom.interpreter.RJCEReroutedExecuter rjce_reroutedExecuter = null;
public String getRjce_xmlURL() { return "D:\\test\\rjce\\rom\\gui\\CodeEditorFrame.xml"; }
public boolean[] getRjce_executeMethod() { return rjce_executeMethod; }
public boolean[] rjce_executeMethod = new boolean[26];
public boolean[] getRjce_executeClassMethod() { return rjce_executeClassMethod; }
public static boolean[] rjce_executeClassMethod = new boolean[26];
public boolean[] getRjce_executeGroupMethod() { return rjce_executeGroupMethod; }
public boolean[] rjce_executeGroupMethod = new boolean[26];
public String[] getRjce_instanceOriginalMethodStrings() { return rjce_instanceOriginalMethodStrings; }
public void setRjce_instanceOriginalMethodStrings(String[] rjce_instanceOriginalMethodStringsIn) { rjce_instanceOriginalMethodStrings = rjce_instanceOriginalMethodStringsIn; }
public String[] rjce_instanceOriginalMethodStrings = null;
public String[] getRjce_instanceParsedMethodStrings() { return rjce_instanceParsedMethodStrings; }
public void setRjce_instanceParsedMethodStrings(String[] rjce_instanceParsedMethodStringsIn) { rjce_instanceParsedMethodStrings = rjce_instanceParsedMethodStringsIn; }
public String[] rjce_instanceParsedMethodStrings = null;
public String[] getRjce_groupOriginalMethodStrings() { return rjce_groupOriginalMethodStrings; }
public void setRjce_groupOriginalMethodStrings(String[] rjce_groupOriginalMethodStringsIn) { rjce_groupOriginalMethodStrings = rjce_groupOriginalMethodStringsIn; }
public String[] rjce_groupOriginalMethodStrings = null;
public String[] getRjce_groupParsedMethodStrings() { return rjce_groupParsedMethodStrings; }
public void setRjce_groupParsedMethodStrings(String[] rjce_groupParsedMethodStringsIn) { rjce_groupParsedMethodStrings = rjce_groupParsedMethodStringsIn; }
public String[] rjce_groupParsedMethodStrings = null;
public String[] getRjce_classOriginalMethodStrings() { return rjce_classOriginalMethodStrings; }
public void setRjce_classOriginalMethodStrings(String[] rjce_classOriginalMethodStringsIn) { rjce_classOriginalMethodStrings = rjce_classOriginalMethodStringsIn; }
public static String[] rjce_classOriginalMethodStrings = null;
public String[] getRjce_classParsedMethodStrings() { return rjce_classParsedMethodStrings; }
public void setRjce_classParsedMethodStrings(String[] rjce_classParsedMethodStringsIn) { rjce_classParsedMethodStrings = rjce_classParsedMethodStringsIn; }
public static String[] rjce_classParsedMethodStrings = null;
/* END OF RJCE PRECOMPILER CLASS CODE SECTION */
```

Figure 22, Typical class-level rerouting fields from an outer class

javaprec adds several class-level rerouting fields, shown in Table 4. Each edit mode has a *boolean* array, to control method rerouting, and two *String* arrays, to store the altered method code. To reduce the memory requirement of precompiled objects, the *String* arrays only contain null values until each method has been altered. An example of the class-level code added to compiled files is shown in Figure 22.

4.1.8.3 The rom.interpreter.rjce_precompiled Interface

Multiple inheritance is not allowed in Java. Therefore, it is not possible to add the rerouting fields by class inheritance; this could only be done if all input files had just

one super-class, *java.lang.Object*, but this is generally not the case. Instead, the *rom.interpreter.rjce_precompiled* interface is used, which specifies accessor functions to allow access to rerouting fields. Using this interface also provides a mechanism to test, at runtime, whether a given class has been compiled by javaprec.

```
package rom.interpreter;

public interface rjce_precompiled {

  // Returns location of javaprec xml files that is associated with implementing class
 public String getRjce_xmlURL();

  // Returns boolean switch array for instance mode
 public boolean[] getRjce_executeMethod();

  // Returns boolean switch array for group mode
 public boolean[] getRjce_executeGroupMethod();

  // Returns boolean switch array for group mode
 public boolean[] getRjce_executeClassMethod();

  // Returns unparsed method string array for implementing class, instance mode
 public String[] getRjce_instanceOriginalMethodStrings();

  // Set unparsed method string array for implementing class, instance mode
 public void setRjce_instanceOriginalMethodStrings(String[] rjce_instanceOriginalMethodStrings);

  // Returns parsed method string array for implementing class, instance mode
 public String[] getRjce_instanceParsedMethodStrings();

  // Set parsed method string array for implementing class, instance mode
 public void setRjce_instanceParsedMethodStrings(String[] rjce_instanceParsedMethodStrings);

  // Returns unparsed method string array for implementing class, group mode
 public String[] getRjce_groupOriginalMethodStrings();

  // Set unparsed method string array for implementing class, group mode
 public void setRjce_groupOriginalMethodStrings(String[] rjce_groupOriginalMethodStrings);

  // Returns parsed method string array for implementing class, group mode
 public String[] getRjce_groupParsedMethodStrings();

  // Set parsed method string array for implementing class, group mode
 public void setRjce_groupParsedMethodStrings(String[] rjce_groupParsedMethodStrings);

  // Returns unparsed method string array for implementing class, class mode
 public String[] getRjce_classOriginalMethodStrings();

  // Set unparsed method string array for implementing class, class mode
 public void setRjce_classOriginalMethodStrings(String[] rjce_classOriginalMethodStrings);

  // Returns parsed method string array for implementing class, class mode
 public String[] getRjce_classParsedMethodStrings();

  // Set parsed method string array for implementing class, class mode
 public void setRjce_classParsedMethodStrings(String[] rjce_classParsedMethodStrings);

}
```

Figure 23, rom.interpreter.rjce_precompiled Interface

Figure 23 shows the *rom.interpreter.rjce_precompiled* interface. This interface contains three sets of methods for each edit mode and *getRjce_xmlURL()*. As the name suggests, *getRjce_xmlURL()* returns the Uniform Resource Locator (URL) location of

the associated XML file. This URL is used by ROM to locate the correct XML file containing the unaltered method code.

4.1.9 Handling Precompilation Errors

If the parser encounters an unrecognised pattern that cannot be matched against any available production, an exception is thrown. javaprec catches all exceptions thrown by the parser and outputs a suitable error message. Table 5 shows the exceptions that javaprec catches.

Table 5, Exceptions and errors thrown by the javaprec parser

Error	Cause	Action
OutOfMemoryError	Input file too large	Output message, offer to stop or continue, if continue: copy file without precompilation
FileNotFoundException	Input file URL does not exist or access is not allowed, file may have been deleted after initial directories were scanned.	Output message, offer to stop or continue, if continue: ignore this file
IOException	Error during parse, including problem writing to file, such as read-only access, or general disk failure.	Output message, offer to stop or continue, if continue: try again
RecognitionException	Error recognising source code either code is incorrect or bug in ANLTR &/or the Java grammar.	Output detailed message including file name, location and type of error i.e. filename … line … found … expected …, offer to stop or continue, if continue: copy file without precompilation
TokenStreamException	Indicates that something went wrong while generating a stream of tokens, such as an IOException, or a RecognitionException	Output message, offer to stop or continue, if continue: try file again
Exception	Indicates that something general went wrong	Output message, offer to stop or continue, if continue: try file again

4.2 The Compilation Stage

4.2.1 Overview

The compilation is done once the precompilation stage has been completed and all if-statements, class fields and accessor functions have been added. There are two ways files can be compiled with javaprec:

- Using *-re:{directories/files}* command line option causes all files and directories in the comma separated list to be recursively scanned for Java source files. These files are then both precompiled and compiled.

  -pre:{directories/files} can also be used to precompile only directories and files.

- Using normal javac commands causes normal javac compilation, but no precompilation of the specified files.

Both types of compilation and/or precompilation can all be simultaneously specified by a user. If precompilation is asked for it is always the first stage, followed by the recursive directory scanning compilation, and then finally, the normal javac compilation is performed. The javac compiler, from Sun Microsystems, in the *com.sun.tools.javac* package, is used for all compilation to ensure reliable and efficient performance. However, this introduces a small problem. The *com.sun.tools.javac* package has been updated by Sun several times recently including for Java SDK 1.4.1 and SDK 1.4.2. The parameters for several compilation methods have changed and the exact location of the package has moved. javaprec, therefore, is very sensitive to the version of SDK used. javaprec only conforms to version 1.4.2 of the Java SDK. This version was chosen because it is the latest version and is the most likely version to become the future standard for the *com.sun.tools.javac* package.

4.2.2 Handling Compilation Errors

If an error occurs during compilation, the compilation history for that particular compilation is output to the console, complete with full details of the error. Then compilation is paused, the user is informed of the error, and asked if he or she would

like to continue with the remaining files. To use javac, *Main().compile(String[] p0, PrintWriter p1)* is invoked from the *com.sun.tools.javac* package. This method returns an integer equal to one of four numbers to confirm the compilation status as shown in Table 6.

Table 6, Values returned from com.sun.tools.javac.Main().compile()

Value Returned	Cause
0	Compilation successful
1	Compilation error
2	Command interpretation error
3	System error
4	Abnormal error

For each compilation, the return value is checked if it is equal to one of the three error values, then the javac compilation status is output to the console. An error message is also displayed to the user, via either the console or a popup confirmation box.

The *NoClassDefFoundError* error is also handled. This error is, usually, caused by the Java SDK 1.4.2 not being installed or the classpath not being set correctly. The classpath has to point to the *tools.jar* contained in the *lib* subdirectory of a Java SDK 1.4.2 installation. This file contains the *com.sun.tools.javac* package used for compilation.

4.3 User Interfaces

4.3.1 Command Line

The command line user interface is designed to be similar to javac. There are five additional unique options shown in Table 7. A full list of the command line options can be seen in Figure 24.

Table 7, javaprec command line options, additional to normal javac options

Option	Outcome
-gui	Load graphical user interface
-re:{directories or files}	Specify directories or files for precompilation & compilation (comma separated list)
-pre:{directories or files}	Specify directories or files for precompilation only (comma separated list)
-tmp <directory>	Keep intermediate precompiled source files, useful for tracing error line-numbers
-quiet	Generate no precompilation info

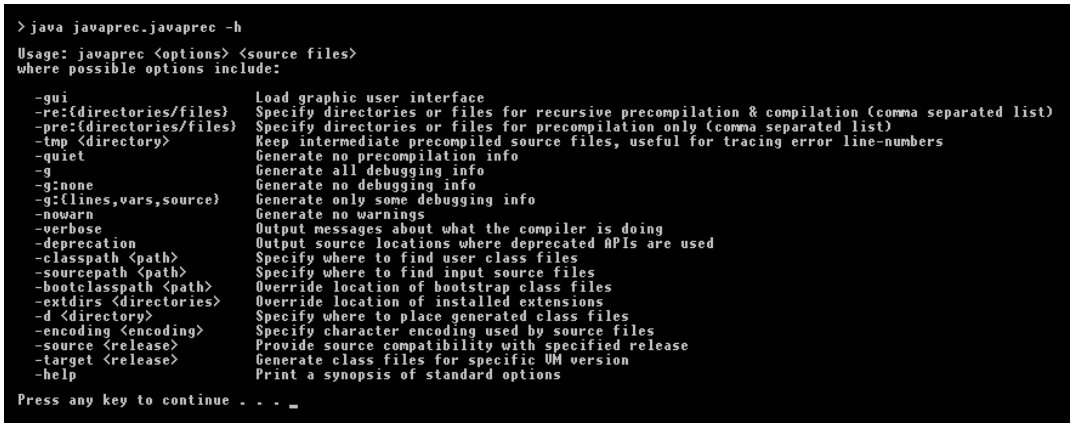


Figure 24, javaprec command line usage screen shot

4.3.2 GUI

javaprec has the same core features for both the command line and the GUI. The GUI also additionally provides a progress bar and allows the user to browse for directories rather then specifying URLs at the command line. Another feature that only the GUI provides is the ability to alter the log file name and location.

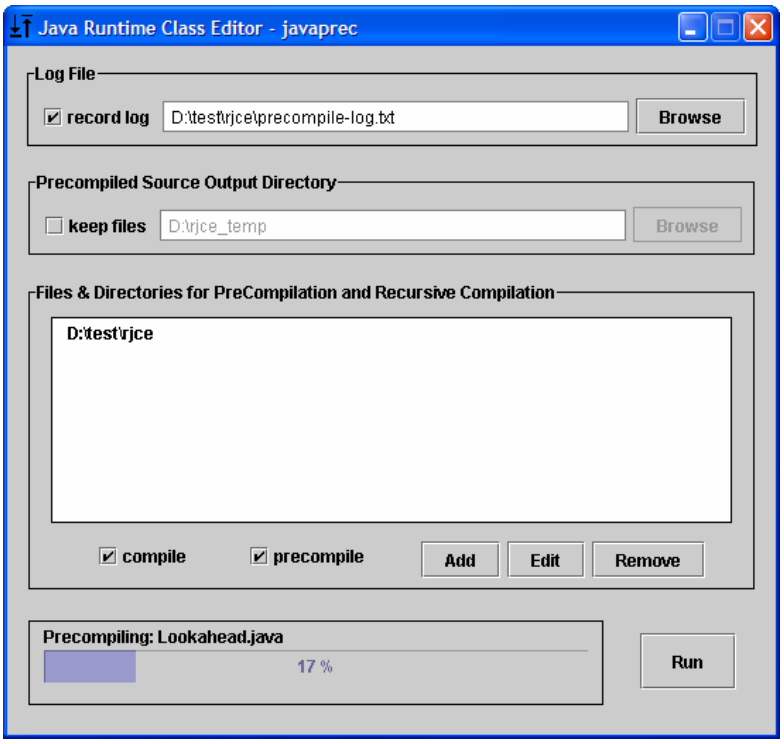


Figure 25, The javprec GUI

5. ROM Design & Implementation

When designing ROM there were several aims, as mentioned previously in section 3. These can be summarised in two main goals:

- Enable efficient interpretation of altered methods and stand-alone scripts using a source code interpreter.
- Create an easy and efficient user interface to edit methods and introspect on variables

When Java source is interpreted by the source code interpreter the execution behaves as if it was taking place in its own scope or namespace, separate from any classes already running. To allow for the change in namespace ROM uses a parsing stage to alter the code prior to execution. This is done once after each method alteration and saved. If a method is invoked repeatedly, the parsing stage is only done the first time. ROM uses two *String* arrays to save method alteration in the objects being altered. One of the *String* arrays is for original unparsed code and the other *String* array is for parsed code. Each method's number can be used to index the correct *String* elements in each array. The parsing stage to transform a *String* from the original array version to the parsed array version is implemented using the same approach as the javaprec parser, explained in section 4.1.3.

The user interface, shown in Figure 26, has two main areas, a class browser and a source editor. It is used by instantiating an instance of *rom.gui.CodeEditorFrame*. This class can be instantiated using several different forms of constructor, although each one requires an *Object* that has been compiled by javaprec as the first parameter. If an *Object* that has not been compiled by javaprec or has a null value is passed as a parameter to a *CodeEditorFrame* constructor then a *rom.gui.NotPrecompiledException* is thrown with a suitable message.

ROM contains a JSplitPane used to contain the source editor and class browser these are used as follows:

- The class browser allows the class hierarchy including super-classes, interfaces and inner-classes to be explored. The class browser also enables field values to be viewed, being displayed using the *toString()* method. Objects that have been compiled with javaprec or arrays, collections and maps of such objects can be loaded directly from the class browser. ROM maintains an ordered history *Set*. This can be transversed backwards and forwards, allowing multiple objects to be simultaneously edited. The class browser also allows methods to be loaded to the source editor. When the method is selected in the class browser if the class has been compiled with javaprec, the method code will be loaded in the source editor.
- The source editor displays Java source. The text is automatically syntax coloured 500 ms after typing stops, although this feature can be turned off. The ROM source editor also provides all the basic features of most graphical source editors, as discussed in section 2.5.1. ROM includes many features, for example: Java scripting, informative error messages, syntax colouring, cut, copy, paste, undo, redo, find & replace, view layout, page layout and printing. These features are included to increase the usability of ROM.

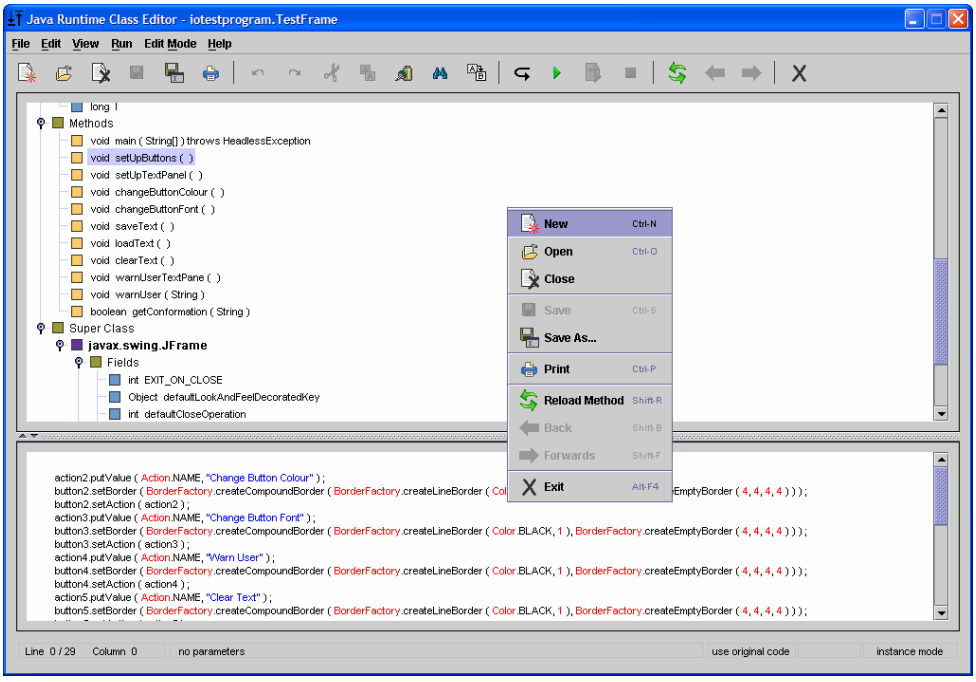


Figure 26, ROM screen shot

5.1 Source Code Interpretation

Interpretation is done in two stages. First, the source code is parsed, and then the parsed code is parsed to the interpreter.

5.1.1 Pre-Interpretation Parsing

As previously mentioned, when Java source is interpreted by the source code interpreter the execution behaves as if it was taking place in its own scope or namespace, separate from any classes already running. Therefore, before using a variable or object in any interpretation it must first be added to the interpreter's namespace. A variable or object can been added to the interpreter's namespace by using one of the primitive *set* methods such as *set(String name, int variable)* or the *Object set* method *set(String name, Object object)*. Although, this approach is sensible for setting method parameters prior to rerouted execution in the interpreter, it would not be sensible to use for all fields in a class. Not only can each class potentially have hundreds of public, private or protected fields - and several super-classes - but also it is not possible to *set* methods. Therefore, an alternative approach is adopted for class fields and methods.

First using Java Reflection a comprehensive list of class fields and methods is generated. The code is then scanned using a parser, generated from ANLTR. The parser adds "*CALLING_CONTEXT.*" to the front of each method or field. Inner-class types also have their package and outer-class name added. If a new field is declared having the same name as a field in its class then this field is ignored for the remainder of the parse. Every use of *this* and *super* is also replaced by "*CALLING_CONTEXT*". Prior to interpretation the set method is called as follows, *set("CALLING_CONTEXT", this)*. This passes the current object into the interpreter as a variable called *CALLING_CONTEXT*, so that class fields and methods references work correctly. The interpreted solution used in RJCE is particularly effective because the interpreter uses reflection, for method and field references. Reflection can easily be setup to ignore private and protected modifiers, as discussed further in section 2.3.5.

There is one problem using the *CALLING_CONTEXT* approach. There is no way to reference a variable in the super-class of *CALLING_CONTEXT* as well as in *CALLING_CONTEXT* itself. The normal approach to do this is to use *super*, but use of *super* presents a problem when *this* could identically be used, as they are both replaced with *CALLING_CONTEXT* in the parser. For example if a *Tree* class was a sub-class to a *Plant* super-class, they might both have *name* fields as shown in Figure 27. The *Plant* name field might be "Tree" whereas the *Tree* name field might be "Eucalyptus". If *super* was used in a *Plant* method, and it was executed correctly, it would refer to the *Tree* name field. Instead, the value for the Plant field name would be used during execution in the interpreter.

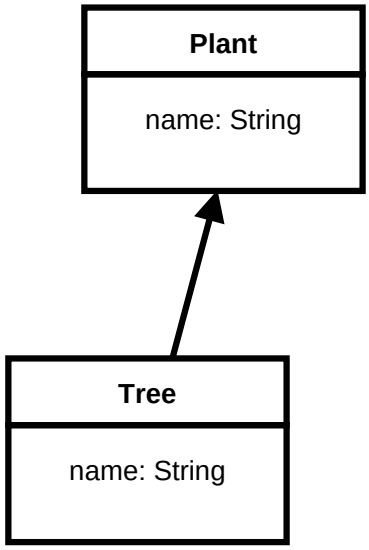


Figure 27, Inheritance example

For the following statement, for example,

System.out.println(" I am a " + this.name + super.name);

the output would be as follows:

un-altered method "I am a Eucalyptus Tree"

altered method "I am a Eucalyptus Eucalyptus"

Once a method has been parsed, it is stored in the correct code array, for parsed code. Prior to execution, the parsed code array is checked for *null* to determine whether the altered method code needs to be parsed. If the array or the correct element is null then the method is parsed and this is loaded into the array ready for a subsequent invocation of the same method. If a method is altered, by a user, the parsed code array element for that method is set to *null*. This ensures methods are only parsed once after each alteration.

5.1.2 Handling Parser Errors

If the parser encounters an unrecognised pattern that cannot be matched against any available production, an exception is thrown. ROM catches all exceptions thrown by the parser, shown in Table 8, and outputs a suitable error message.

Table 8, Exceptions and errors thrown by the ROM parser

Error	Cause	Action
OutOfMemoryError	Input file too large	Output message, offer to stop or continue, if continue: copy file without precompilation
FileNotFoundException	Input file URL does not exist or access is not allowed, file may have been deleted after initial directories were scanned.	Output message, offer to stop or continue, if continue: ignore this file
IOException	Error during parse, including problem writing to file, such as read-only access, or general disk failure.	Output message, offer to stop or continue, if continue: try again
RecognitionException	Error recognising source code either code is incorrect or bug in ANLTR &/or the Java grammar.	Output detailed message including file name, location and type of error i.e. filename … line … found … expected …, offer to stop or continue, if continue: copy file without precompilation
TokenStreamException	Indicates that something went wrong while generating a stream of tokens, such as an IOException, or a RecognitionException	Output message, offer to stop or continue, if continue: try file again
Exception	Indicates that something general went wrong	Output message, offer to stop or continue, if continue: try file again

5.1.3 The Interpreter

Two interpreters were considered during the design of ROM. The first interpreter considered, BeanShell, provides all the features required by ROM. The second interpreter, DynamicJava, provides more features in certain areas such as the ability to interpret complete classes. Under tests, DynamicJava performed at twice the speed when compared to BeanShell although both performed slowly when compared to

normal execution. The first test involved running a for-loop and a while-loop for 1000 revolutions each time outputting a message to the console. The second test involved reading text from a file reversing it then writing it back to the file. Both tests were done 10 times each to get an average and a margin of error. The timings were calculated using the *Calendar* class by outputting a message before and after execution, with the time since UNIX zero, in milliseconds.

Table 9, Execution times for two test cases, in BeanShell and DynamicJava

Test Case	BeanShell	DynamicJava
One	13s (±1s)	6s (±2s)
Two	133s (±3s)	65s (±2s)

Based on the information shown in Table 9, DynamicJava should be the obvious choice, but it has no manual or instructions and little information on its web site. It was possible to get DynamicJava to execute scripts, although, it provided difficult to use with altered methods. This was mainly due to problems with both the return type, often the null value was returned when an Object was expected. Therefore, after several unsuccessful attempts to get DynamicJava to work, BeanShell was instead used.

5.1.4 Abstract Factory design pattern

An Abstract Factory design pattern [37] was used to enable the interpreter to be easily changed. This was done so that if the current problems with DynamicJava were fixed it could be used instead of BeanShell. This approach also allows for the possible use of new interpreters or executers not yet considered in the design, such as Java Native Interface (JNI). JNI is a cross-platform standard provided by the JVM. It allows a JVM to share a virtual address space with platform native compiled code [38].

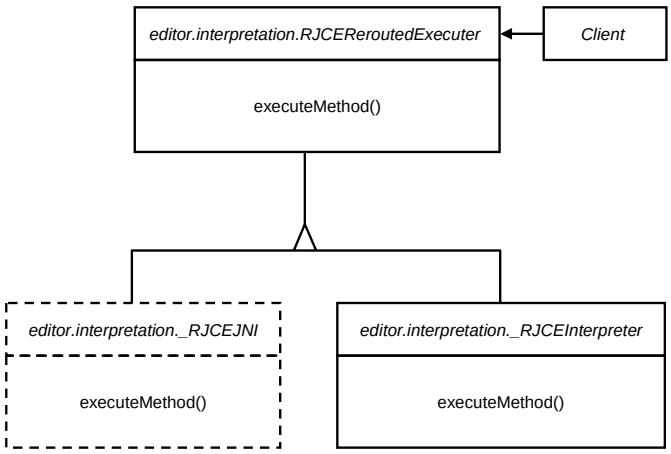


Figure 28, The Abstract Factory design pattern

RJCEExecutionFactory implements the Abstract Factory design pattern by providing a factory method that loads the correct class to handle the rerouted execution. Figure 28 shows the possibility of added a JNI rerouted method executer. This design makes it theoretically possible to optimise the rerouted execution by passing methods, which contain no private or protected references to a faster JNI executer, while passing all other methods to an interpreter. The rerouted execution of each method is also handled using the Factory Method design pattern [37], see Figure 29.

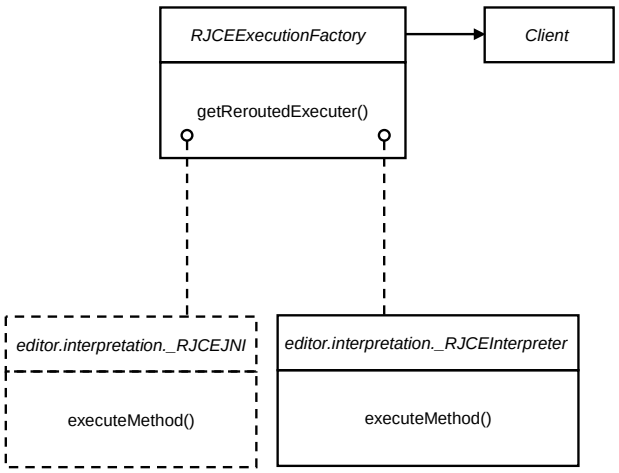


Figure 29, The Factory Method design pattern

Each class that can potentially handle rerouted execution must implement the *editor.interpretation.RJCERoutedExecuter* interface, providing all the methods required for execution.

5.1.5 Handling Interpreter Errors

If the interpreter encounters a problem including, for example, unrecognised code or incorrect use of a null object, an exception is thrown. ROM catches 50 different exceptions thrown by the interpreter as shown in Table 10.

java.lang.Throwable
java.lang.Exception
java.lang.Error
koala.dynamicjava.parser.wrapper.ParseError
antlr.ANTLRException
antlr.RecognitionException
antlr.MismatchedCharException
antlr.MismatchedTokenException
antlr.NoViableAltException
antlr.NoViableAltForCharException
antlr.SemanticException
antlr.TokenStreamException
antlr.TokenStreamIOException
antlr.TokenStreamRecognitionException
antlr.TokenStreamRetryException
java.lang.ArithmeticException
java.lang.ArrayStoreException
java.nio.BufferOverflowException
java.nio.BufferUnderflowException
javax.swing.undo.CannotRedoException
javax.swing.undo.CannotUndoException
java.lang.ClassCastException
java.lang.CMMException
java.lang.ConcurrentModificationException
java.lang.DOMException

java.lang.EmptyStackException
java.lang.IllegalArgumentException
java.lang.IllegalMonitorStateException
java.lang.IllegalPathStateException
java.lang.IllegalStateException
java.lang.ImagingOpException
java.lang.IndexOutOfBoundsException
java.lang.MissingResourceException
java.lang.NegativeArraySizeException
java.lang.NoSuchElementException
java.lang.NullPointerException
java.lang.ProfileDataException
java.lang.ProviderException
java.lang.RasterFormatException
java.lang.SecurityException
java.lang.SystemException
java.lang.UndeclaredThrowableException
java.lang.UnmodifiableSetException
java.lang.UnsupportedOperationException
java.lang.LinkageError
java.lang.ExceptionInInitializerError
java.lang.NoClassDefFoundError
java.lang.NoSuchFieldException
java.lang.NoSuchMethodException
editor.gui.EvaluationException

Table 10, List of exceptions handled by ROM

Each exception is handled by displaying an appropriate error message, including line and column numbers for relevant exceptions. If an exception is thrown, execution of the current method stops. If there is an *Object* return type then null is returned. If there is a primitive return type the default value is returned, such as, false for *boolean*. Otherwise, if there is no return type then nothing is returned.

5.2 User Interface

Figure 30 shows the GUI for ROM. An important consideration throughout the design and implementation of the ROM GUI was usability and look & feel. Therefore, during the implementation of ROM, several potential users were used to test the look & feel.

ROM was designed to look familiar but distinctive. ROM was designed to look familiar in that it used easily recognizable symbols and methods of operations in order to make the interface easier to use.

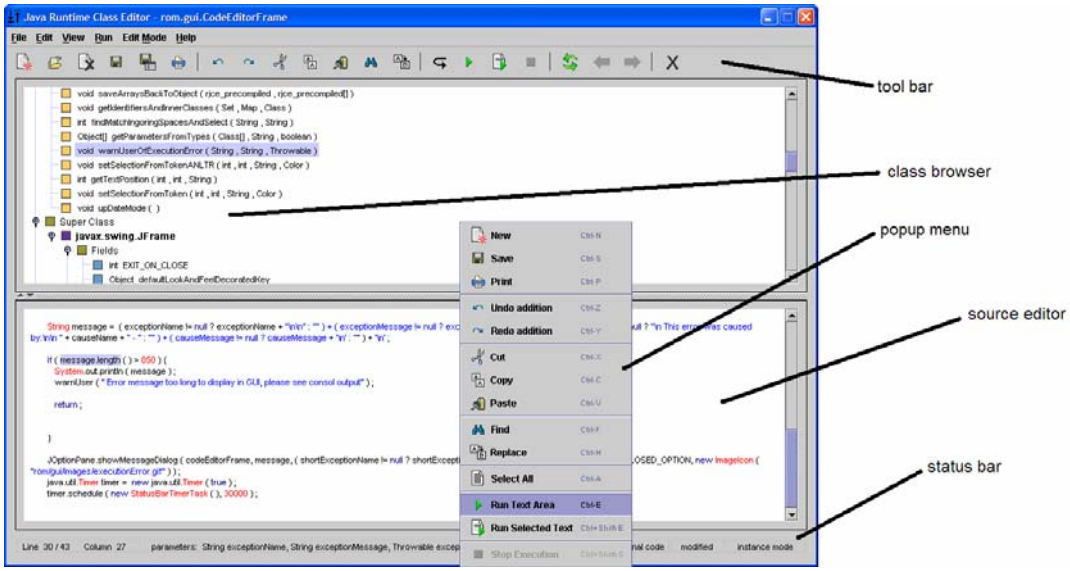


Figure 30, RJCE screen shot

5.2.1 Human Computer Interaction

To implement an effective GUI several sources of information were simultaneously considered. First, ROM was designed to look similar to current popular source editors such as Borland's JBuilder or NetBeans. Borland has had many years experience in designing and implementing GUIs. Therefore, it was felt this would be a suitable product to look at for inspiration. The next source of information was the *Java Look and Feel Design Guidelines* [29]. These guidelines were used to choose the wording and order of the menu and toolbar items. The last source of information used was to ask potential users what they though of the GUI.

The features that were added because of the comparison with other source editors, are cut, copy, paste, find, replace, page setup, printing, status bar, syntax colouring, syntax indenting (added to original XML in javaprec).

There were several features also added because of look & feel testing. The main initial requests were to add more colours; previously the whole GUI was monochrome. Due to the conflicting opinions between potential users about whether the source editor should go on the right, left, top or bottom, the view menu was added. This enables, for example, the split panes to swap around, shown in Figure 31.

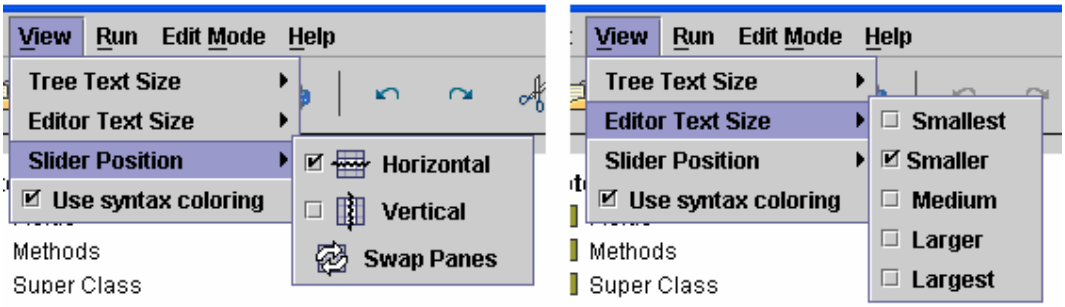


Figure 31, ROM's view menu

5.2.2 Constructors

One of the aims of RJCE was to allow the user to load ROM in several forms, including a basic form that only contains a text editor and a few buttons, and a full form with class-browser. It was intended that this be implemented by providing several different constructors for the *CodeEditorFrame* class that is used to open an editor window. Unfortunately, due to lack of time, this feature was not fully implemented. Only two non-private constructors are provided as shown in Figure 32.

Constructor Summary	
private	CodeEditorFrame()
	CodeEditorFrame(java.lang.Object precompiledObject) Constructors a text view for displaying and editing a text model
	CodeEditorFrame(java.lang.Object precompiledObject, boolean slow_cpu, boolean low_memory, boolean debug, boolean setAccessible, boolean syntaxColouring)

Figure 32, Constructors for *rom.gui.CodeEditorFrame* the main ROM GUI class

5.2.3 Source Editor

5.2.3.1 Syntax colouring

Syntax colouring is achieved in ROM using a dedicated pattern matching *Thread*. A pattern matcher scans the text for each pattern that has a specified colour. The regular expression *(\\bprotected\\b)* is used, for example, for the word *protected*. The syntax colouring thread is triggered after every text alteration. To reduce the effect syntax colouring has on maximum typing speed a timer is used. This enabled syntax colouring to start only 500 ms after the last text alteration; although if alterations are made more than 500 ms apart, syntax colouring occurs for each one. Before this timer was introduced, syntax colouring could often be the cause of the source editor freezing for a short period when many text alterations were done in quick succession. The automatic syntax colouring can be turned on or off, using the toolbar, view menu in Figure 31 or the third constructor in Figure 32.

5.2.3.2 Line and Column Number

Each time the syntax colouring is performed, the line and column numbers on the status bar are updated. This means that the calculation of the line and status numbers also only takes place 500 ms after the last text alteration. The line number is calculated by counting the number of '\n' characters. One of the simple but useful features of *JTextPane* is that a *JTextPane* instance always uses the '\n' character to represent a new line regardless of the operating system. In Java there are two properties that deal with new lines. The system property, line.separator, is defined to be platform-dependent, either "\n", "\r", or "\r\n". There is also a property defined in *DefaultEditorKit*, used in *JTextPane*, called *EndOfLineStringProperty*, which is defined automatically when a document is loaded, to be the first occurrence of any of the new line characters. When a document is loaded, *EndOfLineStringProperty* is set appropriately, and when the document is written back out, the *EndOfLineStringProperty* is used. While the document is in memory, the "\n" character is used to define a new line, regardless of how the new line is defined when the document is on disk. Therefore, for searching purposes, "\n" is always be used, which simplifies the calculation of line and column numbers.

5.2.3.3 Cut, Copy, Paste

Cut, copy & paste, are automatically implemented with *JTextPane* in the methods *cut()*, *copy()* and *paste()*. All that was required in ROM, therefore, was to connect these features to the menu buttons.

5.2.3.4 Undo & Redo

At first, a normal *UndoManager* was used to control undo and redo. *UndoManger*s work well with *Document* objects. All that is required to use undo and redo is to invoke the *addUndoableEditListener(UndoableEditListener listener)* method on a *Document.* Each time the syntax colouring Thread runs, the undo and redo buttons are updated and the edit menu, which displays the undo or redo presentation name is updated. The undo and redo presentation name gives the user an indication of the type of action available. A presentation name might be, for example, “undo addition”, “redo subtraction” or “undo insertion”. Therefore, when a user has deleted some text the undo presentation message would be “undo deletion”. One problem found with the normal Core API *UndoManager* class is that it stored syntax colouring changes. The syntax colouring edits text to change its colour. When using the *UndoManager* it adds syntax, colouring edits to the undo and redo lists. To solve this problem a new undo handler was written as a subclass of *UndoManger* but with an overloaded *addEdit()* method. The new addEdit() tested an edit for its type and ignored style changes.

5.2.3.5 Find & Replace

Find and replace uses a simple recursive text scanning algorithm. The features implemented include: whole words only, match case, upwards search direction, downwards search direction, find, replace and replace all. When a match is found the characters are highlighted and the scroll bar is moved so that a user can easily see the location of the identical character sequence.

5.2.3.6 Saving & Loading

Any normal text file can be loaded into the ROM source editor. This is done using *JFileChooser*. *JFileChooser* allows users to browse the directory structure or manually type a URL. JFileChooser also enables control of the files displayed to users. ROM, by

default, only displays Java source files, although this can be easily changed. The source editor can also be saved to any valid URL. If no extension is specified by the user the "java" extension is added.

5.2.3.7 Page Setup & Printing

To enable printing, *JTextPane* was extended to create a new class *PrintableJTextPane*. *PrintableJTextPane* implements the *Printable* interface by providing the *print()* function. This function is used to scale the text in the source editor ensuring it fits correctly on a printed page. The scaling formula includes terms for font size, page width and longest line. The longest line stored is recorded as the line and column numbers are calculated. The longest line is used to allow scaling to be based on the width of actual text rather than the width of the text area, which also includes the empty space at the end of a line. Figure 33, shows two example printouts. Example one has the longer lines and therefore is scaled so that the text is smaller than in example two.

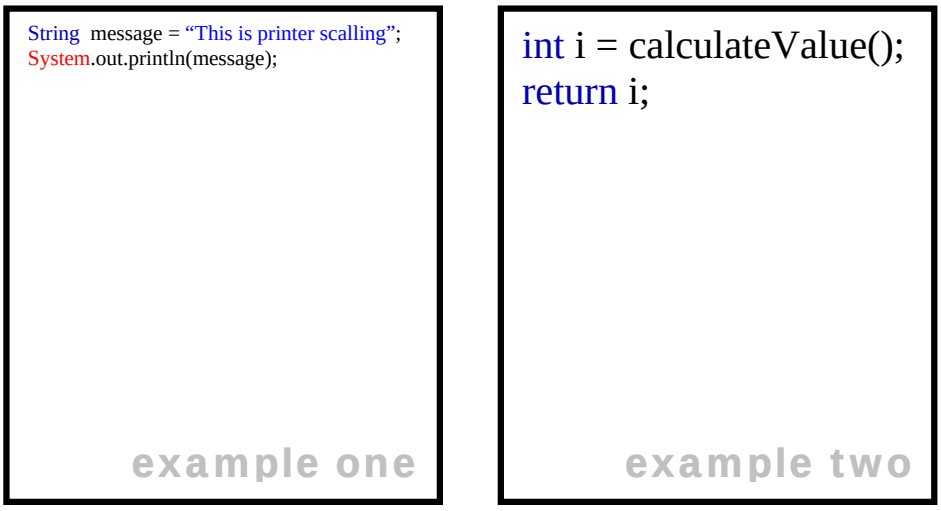


Figure 33, Two example printouts from ROM, showing printer scaling

The scaling used also considers the font size so that a larger font size results in larger text and page size, so that text on horizontally aligned pages will appear larger then on vertically aligned pages. This approach cannot handle multiple pages; instead only the first page is printed. Although this is a limitation, this design decision was a trade-off

against taking longer to implement a multi-paged scaling system that would allow multi-page printing.

To handle interactions with printers *PrinterJob* is used. *PrinterJob* encapsulates and manages the printing process for a given printer that is external to the JVM. Therefore, *PrinterJob* cannot be instantiated using a constructor. Instead a *PrinterJob* can only be obtained from the operating system by using the static method *getPrinterJob()* that is defined in the *PrinterJob* class, for example:

PrinterJob printJob = PrinterJob.getPrinterJob();

Once a PrinterJob has been setup, any *Object* that implements the Printable interface can be passed to the PrinterJob by calling *setPrintable(Printable painter)*. This method renders the *painter* parameter correctly using the *print()* function, which in the case of ROM scale the text to fit the printer's page size.

5.2.4 Class Browser

The class browser is an important part of the ROM GUI, increasing the usability by showing a class view of the running program. The *JTree* used to display the class hierarchy is dynamic, only loading cells when they are viewed for the first time. Any branches that contain no leaves or other sub-branches are not displayed in order to make the tree structure clearer. There are five different types of class node in the class browser as follows:

5.2.4.1 Fields

The field's values are retrieved using reflection, the value returned in an *Object*

- If the *Object* retrieved has not been compiled by javaprec then its value is displayed using the *toString()* method for 4500 ms.
- If the *Object* retrieved has been compiled by javaprec then this Object is loaded in ROM, and the previous Object is added to the history. Once there is more than one *Object* in the history it is possible to use the backwards and forwards button on the toolbar to move through the history.

5.2.4.2 Methods

Both outer-class and inner-class methods can be edited at runtime. Methods that cannot be edited at runtime are greyed out and it is not possible to select them. When a method that can be edited, is selected in the class browser its code is loaded in the source editor. The parameter's names and types are also displayed in the status bar. This is to inform users of parameter names. These are useful when referring to a parameter in an altered method. The parameter names are not displayed in the class browser because of the amount of rendering required for a *JTree*. When expanding a tree branch, scrolling the class browser or moving the ROM *JFrame*, all the visible tree cells get re-rendered. There can often be over 100 visible cells. Therefore, rendering code has to be relatively efficient. XML file produced by javaprec includes the parameter names and their types. Therefore one way to obtain parameter names and their types would be to use the XML file. When this approach was implemented, ROM repeatedly froze for long periods. To solve this problem, only parameter types are displayed because these can be obtained using reflection. Reflection is much faster than IO in this case, and has little or no detrimental effect on the responsiveness of ROM.

To load a method, the XML file associated with the method's class is scanned for the correct method element. Once the correct method start tag has been found the section between this point and the next method close tag is both copied to the source editor and saved in one of the object's *String* arrays.

5.2.4.3 Super-Classes

The super-class structure can be followed back to *java.lang.Object*. The top-level super-class branches allow field access because super-class fields can be accessed via reflection.

5.2.4.4 Inner-Classes

All inner-classes can be viewed. Top-level inner-class methods can be edited in ROM but their fields cannot be viewed or loaded. This is because to access a field in the inner-class an instance of the inner-class is needed - this is not available.

5.2.4.5 Interfaces

All interfaces are displayed, including super-interfaces. Top-level interfaces can have their fields values viewed or loaded.

Figure 34 shows an example of the ROM class browser.

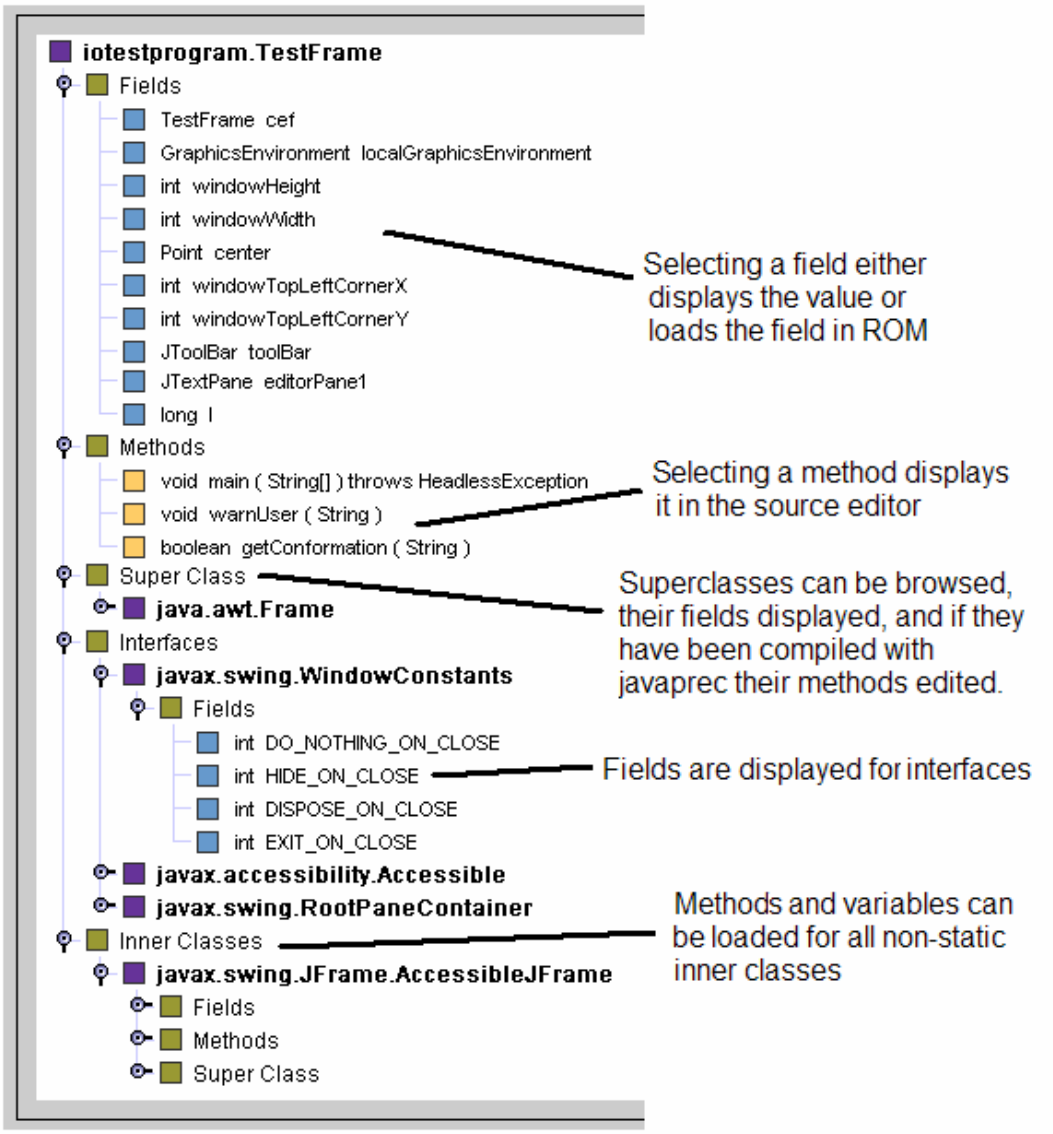


Figure 34, ROM's class browser

5.2.5 Menus & Buttons

Figure 35, shows the menu bar options and Figure 36 shows the toolbar options available in ROM. Each menu bar, tool bar or popup option is implemented as a subclass of *AbstractAction*. *AbstractAction* objects are extremely flexible and powerful. *AbstractAction* implements the *Action* interface. This interface declares methods required for an action, such as *actionPerformed(ActionEvent e)*. The *Action* interface also extends the *ActionListener* interface making all *Actions* also listeners. Some GUI components, such as *JMenu* and *JToolBar*, have an *add()* method that accepts an argument of type *Action*. When an *AbstractAction* is added to a *JMenu* the *add()* function returns a *JMenuItem;* but when an *AbstractAction* is added to a *JToolBar* the *add()* function returns a *JButton*. This means each action class only has to be written once, but can be used simultaneously in a *JMenuBar*, *JToolBar* and *JPopupMenu.*

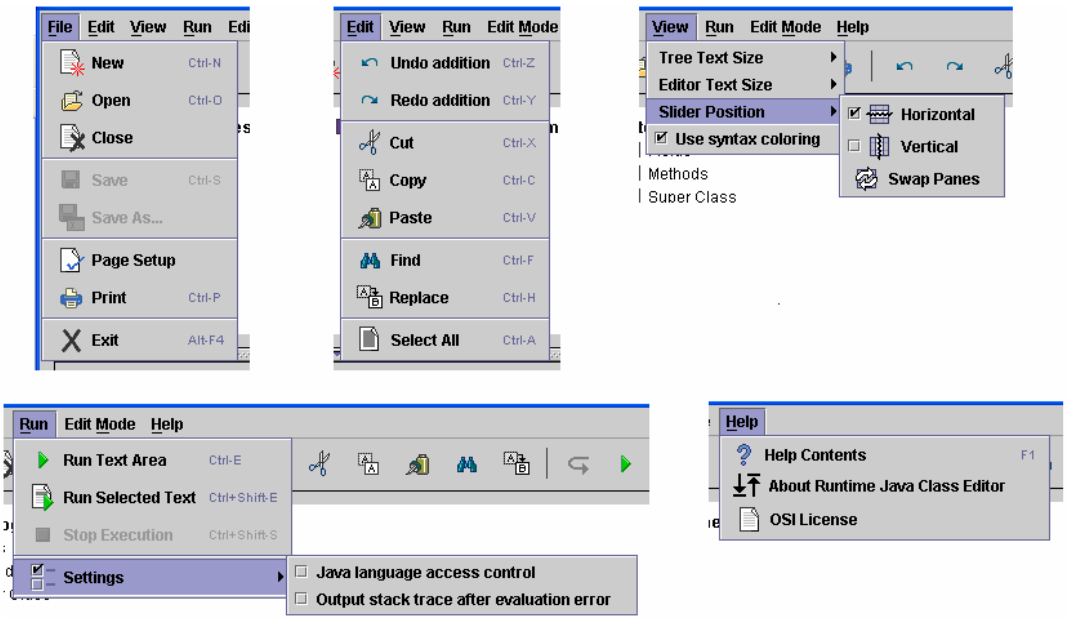


Figure 35, ROM's toolbar and menus

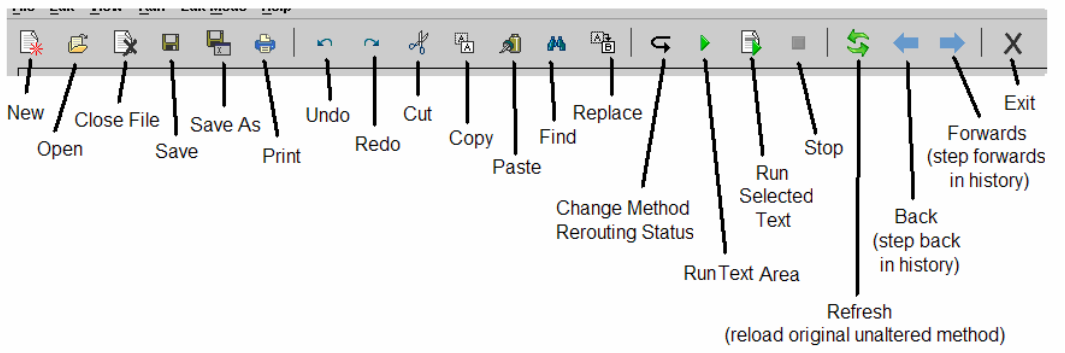


Figure 36, ROM's toolbar

5.2.6 Status Bar

The status bar displays useful information to users, as shown in Figure 37. The status bar displays the line number, column number, status message, rerouted status, modified status and edit mode. The line number and column numbers on the left hand side of the status bar are calculated as part of the syntax colouring thread. The rerouted method status displays two messages. "Use original code" is displayed if method rerouting is not switch on, whereas "method rerouted" is displayed if method rerouting has be switched on for a particular method. The edit mode displays one of: "instance", "group" or "class". The status message displays method parameter names. It is also used to echo error and confirmation messages.

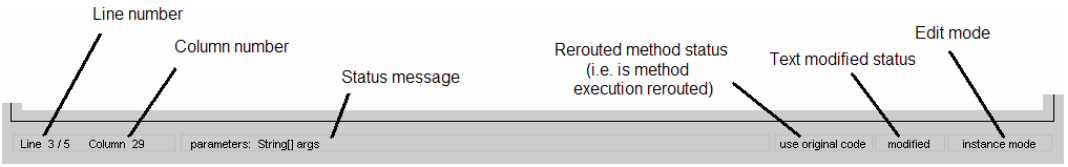


Figure 37, ROM's status bar

5.2.7 Running Scripts

Java source can be run from ROM's source editor by using either the keyboard short cuts Ctrl-E and Shift-Ctrl-E or by selecting one the run buttons. Source code is interpreted in an identical manner to altered methods. For example, when scripts are

interpreted, there is a pre-parser stage identical to the pre-parser stage used for interpreting altered methods.

5.2.7.1 Error Handling Scripts

The error handling for running scripts uses the same classes as the error handling for altered methods and therefore works identically to the description in section 5.1.5.

6. Experiments

An important aim of this project was building a stable, reliable and complete system. To test the RJCE system, some hypotheses were considered against the aims of the project. Experiments were then performed with the aim of proving or disproving each hypothesis.

6.1 Hypotheses

The key aims of the RJCE project are:

- To simulate alteration of the underlying code in a running program, by allowing alteration of methods in classes.
- To do this in a complete, stable and reliable way, with no limitation on Java code used.

There were other subsidiary aims, but the hypotheses focus entirely on the key ones. In order to test that the key aims have been achieved the following hypotheses were considered:

1. The RJCE system does not allow the simulation of the alteration of the underlying code, by allowing method edits.
2. The RJCE system is not complete, nor stable or reliable.
3. The RJCE system does meet its key aims.

Obviously, it is necessary to disprove the first two hypotheses and prove the third.

6.2 Experiments & Test Programs

To test the hypotheses six test programs were used. These test programs were chosen in part because each has been written by a different author. This approach was chosen because each programmer has his own style, increasing the probability that the different test programmes will cover the widest area of the Java language. The test programs used are summarized in Table 11.

Table 11, Test programmes for RJCE

Program Name	Size (characters)	Main Activities	Author
IOTestProgram	9952	Read and write and edit text files	James Bloom
HTMLProcessor	9976	Loads HTML files	Sun Tutorial
TransformArt	10494	Transforms simple shapes using Java2D	Sun Tutorial
AutoArt	364896	Generates sophisticated drawings	Simon Colton
BeanShell	957600	Interprets Java Source Code	Pat Niemeyer
ANLTR	1169172	Generate parsers	Terence Parr

The test programmes showed that the main activities in Table 11 all work satisfactorily after being compiled by the RJCE compiler. In addition, each test program independently proves that RJCE can alter methods at runtime. The test programmes show that the first hypothesis is satisfied. The third hypothesis is partly satisfied – as is the second.

Two experiments were performed to test the hypotheses. The first experiment involves comparing javac with javaprec. This was done by compiling each of the six test programs with both compilers and comparing the results. Both the speed of compilation and the size of files produced were compared.

The second experiment involved testing the interpreted execution of four different methods:

- for-loop: this was contained a simple for-loop with some basic integer manipulations
- while-loop: this was similar to the for-loop except a message was output to the consol rather than performing integer manipulations
- recursion: this method recursed 20 times
- GUI creation: a simple GUI was created with two *JButton* objects

At the start and finish of each method, the time in milliseconds was recorded using an instance of *Calendar*. This allowed comparisons to be made between the time taken to execute each method. The results for these experiments are shown in section 7.

The experiments showed that execution is possible in a reasonable period for an interpreted method. With refinement, the RJCE project could be improved so that the time for execution would be reduced. However, the experiments were sufficient to show that the first two hypothesis were disproved and the third proven.

7. Results & Conclusions

The speed to compile each program was recorded as follows in Table 12:

Table 12, Average results from javaprec - javac comparison

	javac		javaprec	
Test Program	**Duration (ms)**	**Speed (char/ms)**	**Duration (ms)**	**Speed (char/ms)**
AutoArt	1181	309	4697	78
HtmlProcessor	581	17	2975	4
IOTestProgram	566	18	3765	3
TransformArt	525	20	2647	4
ANLTR	3597	338	15309	77
BeanShell	2635	372	11118	87
Average	1514	179	6752	42

These results clearly show that javaprec is on average just over four times slower than javac. Although these figures are not ideal, the absolutes times are still very low and therefore do not represent a significant problem. javac is an application which has had many years of optimisation whereas javaprec is a new product and therefore should have a much greater potential for optimisation. This should help to reduce the gap between javac and javaprec in due course.

Table 13, Comparions between interpreted and normal execution

Average	Normal Execution Time (ms)	Interpreted Execution Time (ms)	"Slow-down" factor
for-loop	5388	6208	1.2
while-loop	22839	24109	1.1
Recursion	31	1208	38.0
GUI creation	2432	2479	1.0

Table 13 shows that with the exception of recursion the interpreter appears to compare surprisingly well with normal execution. It would be desirable to test with a wider set of examples – not least to ensure that the good results so far are correct.

8. Further Work

My wish list for the most important further work is to:

- investigate RJCE's potential for Aspect Oriented programming
- investigate the potential of a JNI executer
- improve the installation process by building an installer
- enable persistence of editing by saving method edits to their XML file
- enable the saving of method edits to their source file
- to incorporate tabbed panes which will allow multiple method to be simultaneously edited
- test RJCE on more platforms as it has only been tested so far on Windows 2000/XP
- implement multi-page printing

13. Bibliography

[0] Andrew S. Tanenbuam *Structured Computer Organization,* Fourth Edition, Prentice Hall, p 34-36

[1] Andrew S. Tanenbuam *Modern Operating Systems,* Second Edition, Prentice Hall, p 6

[2] Kenneth C. Louden, *Compiler Construction Principles and Practice,* PWS Publishing Company, 1997, p4-8

[N2] Laurence Vanhelsuwé, *Mastering JavaBeans*, SYBEX, 1997, p2, 3

[3] Phillip Farrel, *Interpreted Languages*, Stanford University, http://pangea.stanford.edu/computerinfo/unix/programming/interpreted.shtml, 1992

[4] Jon Byous, *Java Technology: An Early History*, http://www.comedition.com/Computers/Java/JavaHistory.htm

[5] Patrick Naughton, *Java History*, University of North Carolina, http://www.ils.unc.edu/blaze/java/javahist.html

[6] William Wagers, *Java Timeline (Java Milestones)*, Focus on Java, http://java.about.com/cs/articles/a/javahistory.htm

[7] *Java History*, Physics Simulation and Java, http://java.about.com/cs/articles/a/javahistory.htm

[8] Tim Lindholm & Frank Yellin, *The Java™ Virtual Machine Specification*¸ 2nd Ed., Addison-Wesley, 1999

[10] Pat Niemeyer, *Lightweight Scripting for Java*, http://www.beanshell.org/

[11] Stéphane Hillion, DynamicJava homepage http://koala.ilog.fr/djava/

[12] NativeJ homepage http://www.dobysoft.com/products/nativej/

[13] Monica Pawian, *Reference Objects and Garbage Collection,* java.sun.com, http://developer.java.sun.com/developer/technicalArticles/ALT/RefObj/

[14] Stuart Dabbs Halloway, *Component Development for the Java Platform*, Addison-Wesley, 2002, p 17-83

[15] Stuart Dabbs Halloway, *Component Development for the Java Platform*, Addison-Wesley, 2002, p 93

[16] Andrew S. Tanenbuam *Modern Operating Systems, Second Edition*, Prentice Hall, p 710-715, 811-817

[17] J. Gosling, B. Joy, G. Steele & G. Bracha, *The Java Language Specification*, Sun Microsystems, 2000,

[18] Ken McCrary, *Create a custom Java 1.2-style ClassLoader*, Java World, http://www.javaworld.com/javaworld/jw-03-2000/jw-03-classload.html

[19] Chuck McManis *The basics of Java class loaders*, Java World, http://www.javaworld.com/javaworld/jw-10-1996/jw-10-indepth.html

[20] Stuart Dabbs Halloway, *Component Development for the Java Platform*, Addison-Wesley, 2002, p 17-83

[21] Monica Pawlan, *Reference Objects and Garbage Collection*, java.sun.com, http://developer.java.sun.com/developer/technicalArticles/ALT/RefObj/

[22] Stuart Dabbs Halloway, *Using Class Loader for Hot Deployment*, Java Developer Connection Tech Tips, October 31, 2000,

[23] Thomas Ball, *Java Language Debugging*, http://java.sun.com/products/jdk/1.2/debugging/index.html

[24] JBuilder homepage http://www.borland.com/jbuilder/

[25] NetBeans homepage http://www.netbeans.org/

[26] J. J. Cook, *Reverse Execution of Java Bytecode*, The Computer Journal, Vol. 45, No. 6, 2002

[27] David Kearns, *Java scripting languages: Which is right for you?*, Java World, www.javaworld.com/javaworl/jw-04-2002/jw-0405-scripts.html

[28] Rick Hightower, *BeanShell & DynamicJava: Java Scripting with Java*, Java Developers Journal, http://www.sys-con.com/java/article.cfm?id=881

[29] *Java Look and Feel Design Guidelines: Advanced Topics*, Sun Microsystems, 2001, http://java.sun.com/products/jlf/at/book/Titlepage.html

[30] Kenneth C. Louden, *Compiler Construction Principles and Practice*, PWS Publishing Company, 1997, p45, 64, 69

[31] Andrew W. Appel, M*odern Compiler Implementation in Java*, Cambridge University Press, 2002, p 21

[32] J. Gosling, B. Joy, G. Steele & G. Bracha, *The Java Language Specification*, Sun Microsystems, 2000, http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html

[33] *Java Compiler Compiler[tm] (JavaCC[tm]) – The Java Parser Generator*, https://javacc.dev.java.net/

[34] Terence Parr, *ANLTR Reference Manual*, http://www.antlr.org/doc/

[35] *JavaCC [tm]: Features*, https://javacc.dev.java.net/doc/features.html

[36] Eric Armstrong *Simple API for XML*, The Java Web Services Tutorial http://java.sun.com/webservices/docs/1.0/tutorial/doc/JAXPSAX.html

[37] E. Gamma, R. Helm, R. Johnson & J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, Addison-Wesley, 1997, p 87-97, 107 -117

[38] Tal Liron, *Enhance your Java application with Java Native Interface (JNI)*, Java World, http://www.javaworld.com/javaworld/jw-10-1999/jw-10-jni-p2.html

[39] Steve Wilson & Jeff Kesselman, *Java Platform Performance – Strategies and Tactics*, Addison-Wesley, 2000, p 57-63