A DISTRIBUTED JAVA VERSION CONTROL SYSTEM

Shakil A. Shaikh
shak@doc.ic.ac.uk

Supervisor: Susan Eisenbach

A project submitted in partial fulfilment of the requirements for the degree of MEng Computing

Project Directory: ~sas97/project/
ABSTRACT

A Distributed Java Version Control System

By Shakil A. Shaikh
Supervisor: Susan Eisenbach
Department of Computing

Distributed Object Orientated Maintenance seeks to find a means to evolve software that already has a distributed client base. Sub-classing (or the sub-interfacing) of existing classes (interfaces) is an example of this evolution. The standard way to keep track of these modifications is to make new binaries by compiling all the sources together or using a ‘makefile’ system to track dependencies and re-compile dependant sources.

However, this solution does not work in a distributed environment. Discovering the dependencies across the distributed clients and suppliers is an immense task, and can be seen as being against the spirit of client-server computing.

This report follows the design and implementation of a tool that attempts to alleviate this problem. It separates and tackles the two sub problems of Binary Compatibility detection, and class distribution. It then goes on to evaluate the tool, and present the main conclusions of the project.
# TABLE OF CONTENTS

ABSTRACT ............................................................................................................. I

TABLE OF CONTENTS ....................................................................................... II

LIST OF FIGURES ............................................................................................ VI

LIST OF TABLES .............................................................................................. VIII

ACKNOWLEDGMENTS ...................................................................................... IX

CHAPTER 1

INTRODUCTION 1

Overview........................................................................................................... 1
Scenario ............................................................................................................ 2
Proposal .......................................................................................................... 4
Report Structure ........................................................................................... 4

CHAPTER 2

BACKGROUND 6

Static Linking and Loading ............................................................................ 6
Dynamic Linking and Loading ...................................................................... 8
Binary Compatibility .................................................................................... 8
Custom Classloaders in Java ...................................................................... 10
The Java Reflection API ............................................................................. 10
Serializable Classes ..................................................................................... 11
Related Work ............................................................................................... 11

CHAPTER 3

PROJECT REQUIREMENTS AND ASSUMPTIONS .................................. 16

Assumptions ................................................................................................. 16
Preliminary Requirements ......................................................................... 16
Detailed Requirements ................................................................................ 17
CHAPTER 4

EXPERIMENTATION AND DESIGN .......................................................... 21

Library Developer Actions ................................................................. 21
Application Developer Actions ............................................................. 22
Choosing a Communication Protocol ............................................... 23
Rules to Implement ........................................................................... 26
Component Design: Rule Engine ...................................................... 27
Component Design: Rule Server ......................................................... 28
Persistent Information ....................................................................... 33
Component Design: Library Developer ............................................ 33
Component Design: Library User ....................................................... 34
Requirements Analysis ..................................................................... 40
Visual Representation of the Design ............................................... 40

CHAPTER 5

IMPLEMENTATION AND TESTING .......................................................... 42

RMI File Transfers ............................................................................. 42
Component Testing: RMI File Transfer ............................................. 44
Disabling Default Classloading ........................................................ 44
The RuleClassLoader ......................................................................... 45
Component Testing: Custom Classloader ......................................... 45
The Abstract Rule Class ................................................................... 46
Component Testing: Rule Engine and Rules ..................................... 48
Persistent Settings Files ..................................................................... 50
Component Testing: Persistent Settings .......................................... 51
RuleController and Slave Objects ..................................................... 52
Library Developer Client .................................................................. 54
Application Developer Client .............................................................. 55
Fault Tolerance and Usability ............................................................. 56
Packages .......................................................................................... 57
Integration Testing ............................................................................ 58

CHAPTER 6

EVALUATION ...................................................................................... 62

Behaviour of Interest ......................................................................... 62
Weaknesses and Limitations .............................................................. 65
Quantitative Analysis ........................................................................ 67
Contrast With Existing Technologies .............................................. 71
Requirements Analysis ...................................................................... 73

CHAPTER 7

CONCLUSIONS ...................................................................................... 75
Lessons Learned ............................................................................... 75
Improvements and Future Work........................................................ 76

BIBLIOGRAPHY ..................................................................................... 80

APPENDIX A

BINARY COMPATIBILITY RULES .......................................................... A1
Evolution of Classes ........................................................................ A1
Evolution of Interfaces ................................................................... A8
Evolution of Packages .................................................................... A11

APPENDIX B

MAIN LITERATURE REVIEWS ............................................................... B1
Client Specification: The Doom Scenario ........................................ B1
The Java Language Specification, Chapter 13 ................................ B1
Java Product Versioning Specification ............................................ B2
Java Version Control System ........................................................ B2
Developing Distributed Systems .................................................... B2
Binary Component Adapation ......................................................... B3
Automatic Program Transformation with JOIE .............................. B4
Adaptable Binary Programs .......................................................... B4
Distributed Events in Jini Technology ............................................ B5
The Jini Architecture Specification ............................................... B5
Why Jini Technology Now? ......................................................... B6

APPENDIX C

RULE ALGORITHMS ............................................................................ C1

APPENDIX D

CLASS DIAGRAMS .............................................................................. D1
APPENDIX E

JAVADOCS ........................................................................................... E1

APPENDIX F

USER GUIDE ........................................................................................ F1
  For Tool Administrators ................................................................. F1
  For Library Developers ................................................................. F2
  For Application Developers and Users ......................................... F3

APPENDIX G

DEVELOPMENT AND TESTING ENVIRONMENT .............................. G1
  Hardware ..................................................................................... G1
  Software ..................................................................................... G1

APPENDIX H

PROJECT LOG ..................................................................................... H1
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIGURE 1</td>
<td>LINKING AND LOADING</td>
<td>7</td>
</tr>
<tr>
<td>FIGURE 2</td>
<td>A GENERAL BCA SYSTEM</td>
<td>12</td>
</tr>
<tr>
<td>FIGURE 3</td>
<td>LIBRARY DEVELOPER USE CASE</td>
<td>22</td>
</tr>
<tr>
<td>FIGURE 4</td>
<td>APPLICATION DEVELOPER USE CASE</td>
<td>23</td>
</tr>
<tr>
<td>FIGURE 5</td>
<td>SOCKET PSEUDOCODE</td>
<td>24</td>
</tr>
<tr>
<td>FIGURE 6</td>
<td>RMI ARCHITECTURE</td>
<td>24</td>
</tr>
<tr>
<td>FIGURE 7</td>
<td>RMI PSEUDOCODE</td>
<td>25</td>
</tr>
<tr>
<td>FIGURE 8</td>
<td>PSEUDOCODE OF VERSIONING ALGORITHM</td>
<td>30</td>
</tr>
<tr>
<td>FIGURE 9</td>
<td>PROOF OF CORRECTNESS OF VERSIONING ALGORITHM</td>
<td>31</td>
</tr>
<tr>
<td>FIGURE 10</td>
<td>DISK BASED REPOSITORY STRUCTURE</td>
<td>32</td>
</tr>
<tr>
<td>FIGURE 11</td>
<td>AN RMI CLASSLOADER</td>
<td>36</td>
</tr>
<tr>
<td>FIGURE 12</td>
<td>AN HTTP CLASSLOADER</td>
<td>37</td>
</tr>
<tr>
<td>FIGURE 13</td>
<td>UPDATING A CLASSCACHE</td>
<td>38</td>
</tr>
<tr>
<td>FIGURE 14</td>
<td>ARCHITECTURE OF THE TOOL</td>
<td>41</td>
</tr>
<tr>
<td>FIGURE 15</td>
<td>NAIVE IMPLEMENTATION OF A REMOTEINPUTSTREAM</td>
<td>43</td>
</tr>
<tr>
<td>FIGURE 16</td>
<td>CORRECT IMPLEMENTATION OF A REMOTEINPUTSTREAM</td>
<td>44</td>
</tr>
<tr>
<td>FIGURE 17</td>
<td>THE RULE CLASS HIERARCHY</td>
<td>48</td>
</tr>
<tr>
<td>FIGURE 18</td>
<td>TESTCLASS USED FOR RULE TESTING</td>
<td>49</td>
</tr>
<tr>
<td>FIGURE 19</td>
<td>BREAKING RULE C</td>
<td>50</td>
</tr>
<tr>
<td>FIGURE 20</td>
<td>BREAKING RULE I</td>
<td>50</td>
</tr>
<tr>
<td>FIGURE 21</td>
<td>SAMPLE SETTING.TXT AND VER1.0.TXT FILES</td>
<td>51</td>
</tr>
<tr>
<td>FIGURE 22</td>
<td>SAMPLE LCLTC.TXT FILE</td>
<td>51</td>
</tr>
<tr>
<td>FIGURE 23</td>
<td>RULECONTROLLER AND SLAVE OBJECTS</td>
<td>53</td>
</tr>
<tr>
<td>FIGURE 24</td>
<td>RULESUBMIT SEQUENCE DIAGRAM</td>
<td>54</td>
</tr>
<tr>
<td>FIGURE 25</td>
<td>CLASSSYNCR SEQUENCE DIAGRAM</td>
<td>55</td>
</tr>
<tr>
<td>FIGURE 26</td>
<td>UP TO DATE LIBRARY OUTPUT</td>
<td>60</td>
</tr>
<tr>
<td>FIGURE 27</td>
<td>LATEST LIBRARY DOWNLOADED</td>
<td>60</td>
</tr>
<tr>
<td>FIGURE 28</td>
<td>LATEST LIBRARY NO DOWNLOADED</td>
<td>61</td>
</tr>
<tr>
<td>FIGURE 29</td>
<td>SUBCLASSING LIBRARIES (1)</td>
<td>63</td>
</tr>
<tr>
<td>FIGURE 30</td>
<td>SUBCLASSING LIBRARIES (2)</td>
<td>63</td>
</tr>
<tr>
<td>FIGURE 31</td>
<td>REPOSITORY HIERARCHY</td>
<td>77</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Number</th>
<th>Table Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CLASSIFICATION OF LIBRARY MODIFICATIONS</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>LIST OF SUPPORTED BC RULES</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>FIELDS IN THE ABSTRACT RULE CLASS</td>
<td>46</td>
</tr>
<tr>
<td></td>
<td>IMPORTANT METHODS IN THE ABSTRACT RULE CLASS</td>
<td>47</td>
</tr>
<tr>
<td></td>
<td>RESPONSIBILITIES OF AND FACILITIES OFFERED BY SLAVE OBJECTS</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>COMMAND LINE ARGUMENTS</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>PACKAGES</td>
<td>57</td>
</tr>
<tr>
<td></td>
<td>TESTED ITEMS BETWEEN SUBMISSIONS</td>
<td>58</td>
</tr>
<tr>
<td></td>
<td>DESIRED BEHAVIOUR OF THE TEST APPLICATION</td>
<td>59</td>
</tr>
<tr>
<td></td>
<td>FILE UPLOAD TIMES MS (SPEED KB/SEC)</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>FILE DOWNLOAD TIMES MS (SPEED KB/SEC)</td>
<td>68</td>
</tr>
<tr>
<td></td>
<td>ENGINE PERFORMANCE (ABSOLUTE NUMBER OF CLASSES) MS</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>ENGINE PERFORMANCE (ABSOLUTE NUMBER OF MEMBERS) MS</td>
<td>69</td>
</tr>
<tr>
<td></td>
<td>ENGINE PERFORMANCE (RELATIVE NUMBER OF CLASSES) MS</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>ENGINE PERFORMANCE (RELATIVE NUMBER OF MEMBERS) MS</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>ENGINE PERFORMANCE (BROKEN BC) MS</td>
<td>70</td>
</tr>
<tr>
<td></td>
<td>EVOLUTION OF CLASSES A1</td>
<td>A8</td>
</tr>
<tr>
<td></td>
<td>EVOLUTION OF INTERFACES A11</td>
<td>A11</td>
</tr>
<tr>
<td></td>
<td>REQUIRED PROPERTIES FOR THE RULE SERVER</td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>PROJECT LOG</td>
<td>H4</td>
</tr>
</tbody>
</table>
ACKNOWLEDGMENTS

I wish to thank Susan Eisenbach for the great help and guidance offered by her in the completion of this project. She went beyond what I expected of a supervisor of a final year project, both in terms of the time (at least one meeting a week!) and effort expended. It is clear that this project would not have evolved as it did without her insights and influence.

Thanks also to all the geezers who endured both the project and course with me over the last four years. It was good to know that we were not alone. It goes without saying it would not have been the same without them: Achmed, Rohit, Joey, Nawaf, Haroon, Jessel, George, James, Steve and the rest…

Finally, thanks to my family for bearing with my stress over the last few weeks – at last it is over, and you can now use the computer without fear of destroying my project!

Shakil A. Shaikh
INTRODUCTION

OVERVIEW

This project is to design and implement a tool to help software developers maintain their distributed code.

Distributed Object Orientated Maintenance (DOOM) faces two problems, which are out of reach of the current software maintenance practices. The first regards the OO part. OO software systems, such as C++ and Java are designed to promote the re-use of software and design, through facilities allowing the sub-classing (and sub-interfacing) of base classes. Clients can tailor third party supplied classes to their own needs, by either adding new members (fields or methods), or overriding existing ones. Interfaces may be provided (in the Java language) as contracts to which members an implementing class should provide.

The system evolves whilst gaining more clients and each client relies on the class version it was developed from. This implies that each change to a base class may split the client population into two or more different types that may respond differently to each version of the base class (before and after the modification). This problem is currently handled in two ways – the first is to make the whole source base available to the compiler (to enable it to statically type check and link binaries correctly). The alternative is to create and maintain a ‘makefile’ system that tracks dependencies between binaries. This can therefore be used to identify which dependant sources need to be recompiled.
This solution cannot be applied in a distributed environment. Not only may there be many clients (and clients of these clients), these clients will be running on different machines in different geographical locations. Implementing the makefile solution in such an environment requires finding a set of ‘distributed dependencies’ (mathematically or otherwise), a task considered to be very difficult due to problems such as client anonymity and network availability. However this remains a more viable solution than compiling with source – there is little hope of obtaining a complete set of remotely distributed sources.

The main difficulty with DOOM is the difficulty in recognising which changes would affect which clients, and how.

The aim of this project is to develop a tool that will help alleviate this problem. It should help developers of the base classes to keep track of their modifications and predict how they will affect clients. The information it provides should help developers to avoid client difficulties. Clients should be able to use the tool to decide which version of a supplied class they should use.

**Scenario**

To help understand the problem, and to explain some concepts, the following scenario in which the tool can be used has been contrived.

Take an everyday investment bank, The Bank of Money. BOM has a main IT division (Global IT, or GIT) for company wide projects, taking care of relevant issues, like infrastructure, security, and information sharing. BOM also has smaller IT departments for each of the other financial divisions, Foreign Exchange, Asset Management and Equities, for example. These smaller IT departments develop applications to be used mainly by their
relevant divisions (as an aside, these applications may also be used in other departments, developed jointly etc).

The ‘Library Developers’ at GIT develops an API (version 1.0) to access a new communications middleware initiative. This will be used to create and connect to communication channels, and send and receive data through them. The ‘Application Developers’ in the FXIT department decided to use this API to create an application as a front end to the transaction database where all FX deals are stored.

GIT then publishes a new version of the API, v1.5. These have minor changes to do with ‘tuning’ parameters for efficiency. As such, client applications still work, and since the FXIT department have a policy to use classes distributed by a web store, they automatically use the new versions and their applications still work, automatically. Other departments are not so fortunate – Equities decided that libraries should be distributed with the client application (since network communications are very unreliable, and class files may not be available via the web – there is no point in using the latest version of a library if it is not available!).

GIT now publishes version 2 of their API. This has enhanced security and authentication features, which will make the whole communications channel secure and less prone to attack. Unfortunately, in adding these features, they had to redefine some aspects of the API, so much so that existing clients would not be able to use them without some modification. GIT, however, do not know how or who the changes will affect – indeed, they do not even realise there is a problem.

All of a sudden, FXIT’s database application ceases to work. The web store contains version 2 of the API, and so numerous linking errors occur when the Java system executing the front end tries to access the members it was able to before. More destructively, users find this out before even FXIT, resulting in
a great loss in money, time and confidence in the IT department. Equities’ application still runs fine, albeit without the latest security enhancements.

The developed tool should help avoid situations like these. It should notify GIT if and when changes to their public libraries will affect clients. For the client, facilities will be provided to enable them to use the latest version of a library, provided that it will still work. If the latest version would not work, the tool would provide reasons why it would not, and how to change the client application to use the latest version. In short, the versioning and distribution of a library will be controlled in such a way as to minimise disruption to all parties concerned.

**PROPOSAL**

The project is to design a tool to provide a solution to this scenario. As far as I am aware, there is no first, second or third party technology available that would easily address this problem of compatible binary distribution. As well as resolving the problems in the above scenario, the tool should also fulfil the list of requirements defined later.

If successful, it is expected that the tool would serve as a prototype or reference for a more ambitious enterprise-level system, if not used in its own right.

**REPORT STRUCTURE**

This report is split into 8 chapters. First, a concise background covering the technical aspects of the project is presented, including Binary Compatibility, dynamic linking and loading mechanisms and some workings of the Java Virtual Machine. The main requirements of the tool are listed, followed by a chapter on the design decisions made, and then by the implementation details and testing of the tool.
The report ends with an evaluation of the project, including a reanalysis of the requirements, and a conclusion of the main findings of the project including further extensions to the tool.
Chapter 2

BACKGROUND

Following is an account of all the potential technical issues that may have to be explored in the design of this tool. Some further specifics could not be inferred from the research material. Further experimentation was required determining these – they will be explained in the design decisions section.

To end, a summary and brief evaluation of current related projects is given.

STATIC LINKING AND LOADING

In most programming languages, data and programs (functions and procedures) are compiled to a binary code. One of the first forms of this code was the bit representation of the instructions given to a CPU – each particular function, procedure or data item (all three can be referred to as members) started or lived at a certain address, and was referenced by this address by other functions and procedures (the situation became slightly different for user variables).

The extension of this was the ability to have discrete, separate binaries; each perhaps performing isolated associated functions. References to external members now included a file (for the whole binary) as well as an offset into the file (for the specific member). This was useful, as individual binary files could be anywhere in memory, with the offset being enough information to locate the code for a member. This occurs during the loading phase. The resolution of these references is referred to as linking. 
Generally, these references were calculated at compile time, and then stored within the clients that referenced the external members. The obvious consequence of this is that if the positions of the members in the library (or the binary being used by others) were moved around (by a recompilation, for example), any external references to them would have probably been invalidated. This could sometimes be detected by a client program – in less safer systems, random code may have been loaded instead. These ‘out of date’ clients would operate either with the older (correctly organised) binaries, or if they themselves are recompiled with access to the new references.

A less subtle problem could be introduced when modifying libraries. An example is if the signature (or type, based on its name and possible arguments or results) of a member is changed, then the compiler itself will not be able to locate a suitable external reference to supply to the client. In this case, the client will not even compile successfully. These changes can be distinguished by the term **consequentiality**: a consequential change requires changes to the
DYNAMIC LINKING AND LOADING

Modern programming languages and compilers offer the ability to dynamically link the binary files that they produce. This means that references are discovered at runtime. The usual implementation of such a scheme is to provide an index or directory of all the members of a library class with its binary. A linker and a loader then use this information to resolve references external to a client class.

To check that any calls to library members by clients are legal (e.g. type safe, correctly invoked, etc), the library has to be available to the compiler at compile time. In fact, it even turns out that code is not needed to pass this ‘symbolic’ check – a ‘contract’ of member signatures is enough. In the Java language, these are referred to as interfaces. Member names are then resolved via these interfaces at runtime.

Hence, libraries may be recompiled without fear of invalidating references in clients. Of course, the problem of consequentiality remains, although now it may occur at runtime, as opposed to compile time (although most compilers of dynamically linked languages also check for this at compile time).

BINARY COMPATIBILITY

Fortunately, Chapter 13 of the Java Language Specification lists modifications possible to libraries that keep them Binary Compatible (BC) with respect to any client files. A modified binary is BC with pre-existing binaries (clients) if the pre-existing binaries that previously (i.e. before the modification) linked without error will continue to link without error. [JLS] comprehensively lists various rules, which if adhered to, will guarantee BC. The tables in Appendix A summarises these rules.
It is clear that a BC change to a library should not affect the dynamic linking of a client to a library (providing of course, the change was inconsequential). However there is more to it than this. Language specific nuances must also be considered, which may cause problems when adapting libraries. For example, in the Java language final fields (whether inherited or not) are defined at compile time. The effect of this is that clients of libraries in which a final field has a new value may continue to use the value of the field it was linked with. A fresh compilation will fix this. The term palpability is introduced to distinguish between these modifications. A palpable modification is seen straight away. A client would have to be rebuilt to see an impalpable modification. In both cases, however, the client will still run (although maybe without the intended behaviour).

Say a library is at version 1.0. A BC change is made to it, to bring it to v1.5. Another BC (with respect to clients compiled with v1.5) change is made – library v2.0. By virtue of BC, any clients that link with v1.0 can also link with v1.5, and therefore with v2.0. What this demonstrates is that BC is transitive. Allowing xBCy to mean library x is BC with clients successfully linking with library y, this property can be written:

\[
\text{If } (1.2 \text{BC} 1.0 \&\& 1.4 \text{BC} 1.2) \text{, then } 1.4 \text{BC} 1.0
\]

And so taking the contra positive:

\[
\text{If not } (1.4 \text{BC} 1.0) \text{, then } (\text{if } 1.2 \text{BC} 1.0 \text{ then not } 1.4 \text{BC} 1.2)
\]

where the second implication is also valid under contra positive. [TDS] also mentions that BC modifications ‘scale up’ – that if a change is BC for a small known set of classes, then they will be BC for an arbitrarily larger system.
<table>
<thead>
<tr>
<th>Modification Type</th>
<th>Palpable</th>
<th>Inconsequential</th>
<th>Compatible</th>
<th>Remark</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>No problem</td>
</tr>
<tr>
<td>1</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No problem, until rebuild</td>
</tr>
<tr>
<td>2</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Rebuild needed for required operation</td>
</tr>
<tr>
<td>3</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Problems occur after recompilation</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>No</td>
<td>Binary incompatible, try new modification</td>
</tr>
</tbody>
</table>

Table 1 - Classification of Library Modifications

**CUSTOM CLASSLOADERS IN JAVA**

Java has provided a way for programmers to build their own classloaders. A classloader does just that – loads classes to be used in the executing JVM. A custom classloader allows developers to load classes from non-standard locations – a SQL database, an online store or a location not in the current classpath. A side effect of using a classloader to load classes is that it allows the partitioning of user code in a JVM[CCL]. Since classes are identified in a JVM by their names and which classloader loaded them, classes loaded by separate classloaders are effectively separated.

**THE JAVA REFLECTION API**

Java provides the developer with a way to interrogate and query loaded classes. They can then find out information about the members of these classes and can even use the ‘reflected’ members on associated objects.

The class that make this possible is java.lang.Class. Instances of this class represent the actual classes and interfaces running in a Java application.
Methods to retrieve and act on the members of the represented class are provided by the standard `java.lang.reflect` package. This package provides further classes for obtaining specific reflective information about classes and objects. Additional features are the ability to compare classes and their members for equality and to instantiate new instances of a `class`. Due to the type system of Java, the developer is unable to reflect private members of a class.

**SERIALIZABLE CLASSES**

The class `Class` also implements the `java.io.Serializable` interface. This is a special interface that allows any instance of a class that implements it to be expressed, in whole, as a stream of bytes. This enables the developer to write serializable objects to a socket or file. The Java Remote Method Invocation technology also relies on serialization to pass objects between remote JVMs (RMI also supports the different passing of remote references to objects still in a remote location).

As `Class` is serializable, the developer can transport `Class` instances to remote locations. It remains to be seen whether serialized classes can be used as normal – simple experiments should show how useful this technique is.

**RELATED WORK**

Here, the existing state of the art that may be relevant to the project is summarised.

**BINARY COMPONENT ADAPTATION, Ralph Keller and Urs Holze [BCA]**

This is an implementation of how to adapt Java binaries (in the class file format) after they have been compiled (referred to as Binary Component Adaptation), without necessarily having access to their sources. This enables
Application Developers to adapt libraries themselves, to fix bugs, conform to interfaces or anything else in order to evolve or integrate them.

The binary is passed through a BCA system that applies a ‘delta’ transform on it, before passing it as a valid classfile to a standard JVM classloader. The BCA system uses the delta file to effectively search and replace any code in the binary.

BCA tackles a different problem, in that it is the Application Developers making the changes to libraries, rather than the Library Developers. The method in which it analyses existing binaries, however, may be relevant to this project.

![Diagram of BCA System]

Figure 2 - A General BCA System
JOIE, Geoff A Cohen and Jeffery S Chase [JOIE]
The Java Object Instrumentation Engine is a toolkit for constructing transformations of Java classes. Here, delta files are used to transform classes at loading time via a custom classloader – on disk, class files remain as they were originally. Practically, Library Developers can distribute these delta files instead of whole new classes. If the JOIE classloader were configured to use the latest available delta file, then an updated version of the class would be used instead. JOIE, however, does not seem to make any attempt of notifying the creator of the delta file of how the changed class will affect classes. Indeed, if the client was to indiscriminately load new classes, link errors may occur.

ADAPTABLE BINARY PROGRAMS, Robert Wahbe et al. [ABP]
ABP is a specification of how adaptable binaries should be implemented to allow other systems to reliably control and edit them. It is not concerned with whether Application Developers or Library Developers will make these changes, or how the transformation would be made.

JAVA PRODUCT VERSIONING SPECIFICATION, Sun Microsystems [JPVS]
In 1998, Sun published a specification regarding the versioning of a Java system. Basically, it defines the fields and format of versioning information, to be kept in a package manifest. As this manifest can be read by a Java application, a classloader can detect and act on the version of a class. It is the responsibility of the Library Developer to keep this versioning up to date and accurate. Thus, there is no ‘intelligence’ in the versioning, and it can be set and reset arbitrarily. Although the specification outlines why, when and how developers should be concerned with versioning, it provides no automatic assistance in doing so.
THE JAVA VERSION CONTROL SYSTEM, Yu Kor Lam et al. [JVCS]

A 2000 Computing Group Project, the JVCS is a tool that allows a developer to check if any changes made to a “project” class are binary compatible with any other in the same project. Although it works, only a limited number of rules are supported. Also, it only keeps track of source files in a given directory – no facilities for distributed environments are present.

JINI, Sun Microsystems [DEJT]

From Sun: “Jini technology provides simple mechanisms that enable devices to plug together to form an impromptu community – a community put together without any planning, installation, or human intervention. Each device provides services that other devices in the community may use. These devices provide their own interfaces, which ensures reliability and compatibility.”

The main emphasis is on the ad-hoc, impromptu nature of Jini networks. To this end, Jini provides discovery and join protocols, and a lookup service. It should also be noted that Jini uses Remote Method Invocation to tie these services together.

For the tool in development, it is assumed that all distributed services are known and available. To automatically infer these services at runtime is beyond the scope of this project. Although it may be a useful facility to have, it does not solve the main problems of versioning or binary compatibility.

Another point is that the use of Jini may have required the application developer to declare and use application fields in a non standard manner (perhaps through an API, provided by the tool).
ENTERPRISE JAVA BEANS, Sun Microsystems

The main cause for components was to package related classes and software modules into larger components. These are then reused as a whole – clients interact with them through a common component interface, and so with the component as a whole, as opposed to each individual class/object within the component. Sun’s specification of a component architecture resulted in JavaBeans. 

Distributed components primarily take components to the network. They also extend distributed object technology (e.g., RMI, CORBA) by providing persistence, transaction semantics, and security. The distributed version of JavaBeans is Enterprise JavaBeans, or EJB. Like Jini, EJB operates over RMI.

Like other distributed technologies, EJB focuses on the availability of remote objects. This is preferable for various reasons – security and resources may be greater on the server. Thus any objects in use remain on the server, with function calls made over the remote link.

A review of relevant publications relevant to the above technologies is given in Appendix B.
Chapter 4

PROJECT REQUIREMENTS AND ASSUMPTIONS

ASSUMPTIONS

Listed are the assumptions that are being made in the development of his project.

A1. **The tool is for the Java programming language.** All libraries and applications are written in 100% pure java.

A2. **All modifications are intentional and effective.** Only Library Developers make changes to libraries and Application Developers use the libraries normally. The changes made are planned and relevant to the libraries’ development.

A3. **All modifications to class files are via source only.** The Library Developers do not adapt class binaries directly. Each change to source is followed by a standard recompilation by a standard compiler (e.g. javac).

A4. **Standard libraries and facilities are available to the client.** This includes access to the standard Java API, an unlimited TCP/IP network connection, sufficient disk space, etc.

PRELIMINARY REQUIREMENTS

Following are the requirements as presented in the project brief by the supervisor. Here, the ‘tool’ refers to the tool in development.
P1. **Cover more rules.** In relation to the JVCS project, the tool should cover at least twice as many rules. These rules must have an association with, or be derived from those in [JLS], although any of these may be chosen to implement.

P2. **Easy to expand.** With regard to future development, it should be relatively easy for further rules to be added to the system. These may or may not be derived from [JLS], and may not even be to check for BC breakage.

P3. **No ‘bit-poking’ allowed.** If and when the tool analyses and changes class files, it is not allowed to delve into the actual file on disk. All members, symbols, etc. should be retrieved in an abstract and controlled manner. Class binaries should be unaffected by the tool.

P4. **Library Developers know when BC has been affected.** The tool should check and diagnose any new classes submitted to it. If BC has been broken, adequate information should be returned, correctly and accurately specifying what change broke BC, and how it did so.

P5. **Clients should remain anonymous.** They should be free from the necessity of identifying themselves to any server for any purpose beyond the immediate transaction. This consequently frees the server from any responsibilities beyond those of the immediate transaction.

P6. **Java and it’s facilities should be unaffected by the tool.** For example, sub-classing of libraries and polymorphism should still work correctly.

**Detailed Requirements**

These further requirements have been added. They result from further introspection of what the tool is supposed to do.

**D1. Users can only use standard network connections available to them.** These include HTTP, FTP and other TCP/IP services. They
can connect to standard Java services, for example an RMI registry, and published remote objects.

D2. **Use of the tool can be at any time.** Users may run their applications at any time. Library and Application Developers may use the tool at any time.

D3. **Any information stored should be persistent.** Rebooting of a server, or running of another server should not affect the tool’s operation.

D4. **No radical change in programming is required by the Library Developers.** Libraries subjected to the tool should very similar in style to existing classes. No unusual programming practices to use the tool should be required by the Library Developer.

D5. **No radical change in programming is required by the Application Developers.** The use of ‘controlled’ libraries by an application should be as close to normal as possible. Library objects should be declared and used as normal.

D6. **Application Developers should have access to BC information.** They have to have access to enough accurate and correct information as required to fix any BC problems they may have.

D7. **There should be little restriction on the distribution of libraries.** The tool should handle any non-initial distribution of all controlled libraries. There should be no need for any distributed classes outside the control of the tool.

D8. **The operation of the tool should be transparent to any user.** The use of the tool should not affect users of the application, both in normal and abnormal situations.

D9. **The tool should require little change in the user’s environment.** This is both in terms of equipment and the installation and use of any application developed with the tool.

D10. **Applications should be safe.** User applications should always run – at least they should not cease to run because of the tool.
D11. **There should be no change in standard Java technical standards.** The tool should not require a change in the class file format, for example.

D12. **Libraries and applications subject to the tool should be backward and forward compatible.** Libraries and applications already developed should easily be used with the tool. Libraries and applications created with assistance of the tool should function without it.

D13. **The tool should use, and only require the use of, standard Java libraries, APIs, runtime environments, editors and compilers.** Library and Application Developers should be able to use their own environments to develop classes. The use of the tool should not require any unusual first or third party technology.

**Requirements Analysis**

P1 says that more rules must be covered. Of course those covered by JVCS can be re-implemented. For the others, the most straightforward and easily testable ones will be implemented, to aid development of the rest of the tool.

P3 is broken by the technology that use delta files to adapt a binary at the bit level. P6 may also be broken depending on the implementation. As custom classloaders are sometime required to apply deltas to binaries, D4/5 may be broken. It is not known what (if any) fault tolerant measures are available in these tools – the deletion or corruption of a delta file may crash an application, affecting D10.

In the tools, binaries are not changed themselves, although they are transformed at runtime, so requirement D11 is broken then. This happens internally, so may not be as relevant as if the actual binary itself was changed. It is not known if D12 is fulfilled – the transformation engine applying the
delta seems to be tied well to the application, and it may be difficult to separate them.

Finally, as standard Java classloaders are extended to implement the transformation engines, it seems that D13 is satisfied.

P4 and the related D6 are only tackled by [JVCS]. It is an example of providing a way to avoid breakages rather than fixing them after detection. In fact, these BC detection requirements play a fundamental role in the tool being developed.

It appears that no requirements are in conflict.
EXPERIMENTATION AND DESIGN

This chapter has many sub parts. First a story is given of how the tool will be used. It will be specific enough to cover many of the requirements. To keep it abstract, there will be no commitment to any specific design. From this story, the possible implementations that may be used for the different parts of the tool will be inferred. After these are evaluated and chosen, the main components will be described and designed in detail, including which specific technologies are to be used (if any).

In the situations where deep knowledge about the techniques were not known, experiments to resolve issues are described. The main conclusions of these are also presented.

LIBRARY DEVELOPER ACTIONS

A typical Library Developer will first develop and compile their classes as normal. On successful compilation, they will submit the group of classes to the tool. If this is a first time registration of a library, this procedure may be slightly different.

The system will check with the last published library (if any) to see if BC has been broken. Whether it has or not, the tool will ask the Library Developer to confirm the publication of the library. Confirmations of version number, etc. are returned to the Library Developer.
APPLICATION DEVELOPER ACTIONS

The Application Developer will develop their application as normal. The tool will provide an API or class to access the features of the tool. Details of this will be discussed later. When the application is started, it will first check with the tool for new versions of any libraries it is using. If there are none, it will continue as normal, using the current version of the library. If there is a new, compatible version, it will use that instead.

To allow Application Developers to receive diagnostic information, whilst keeping things transparent to users, there should be a ‘diagnostic’ mode available to the Application Developer that provides the status of the tool. This will include information concerning versions used and available, and how to adapt their application to use any latest versions. This information should be invisible to the user.


**CHOOSING A COMMUNICATION PROTOCOL**

The distributed components of the tool can communicate in many ways. The main methods are by using TCP Sockets, Java’s Remote Method Invocation or Jini.

Sockets are probably the easiest way for the components to communicate. To use them, just the server and client are required to be running. Both are compiled normally, and no third party software is required.

The classes provided by the Java API to use sockets are `java.net.Socket` and `java.net.ServerSocket`. The client uses the former, the server the latter. They are relatively simple to use. A basic pseudocode implementation is given.

All naming conventions follow normal TCP/IP rules. A bonus of this method is that the server or client can be written in another programming language, as socket technology is usually implemented at the operating system level.
Server:
initialise;
wait for connection from client;
read/write to socket;
clean up and exit;

Client:
initialise;
create socket, and attempt to connect to server;
read/write to socket;
clean up and exit;

Remote Method Invocation is a distributed object technology included with Java that allows client objects to access the methods of remote objects. As well as server and client programs, it requires an ‘RMI-Registry’ program to be running.
Compilation of an object that wishes to register itself in the registry is slightly more complex. It has to be compiled with javac and then rmic. rmic creates the stub and skeleton files required by RMI to operate. The stub has to be accessible from the client side either directly, or after being downloaded from a web server.

Various packages are provided by Java to use RMI. A basic pseudocode implementation would be:

```
Server:
 initialise;
 provide implementations of remote methods;
 create object to register as a remote object;
 register this object in the RMI-Registry;

Client:
 initialise;
 attempt to lookup remote object in registry;
 use object as normal;
 clean up and exit;
```

Here, there is no concern with any communication protocol to be developed, or specific messages to be sent between the client and server. After the client has looked up the remote object, it can use it as normal. The downside is that both client and server have to be written in Java, or at least a programming language that supports Java’s RMI.

Jini is a collection of technologies and protocols that allow the creation of an ad-hoc network of services. This uses RMI as the underlying communication method, but then provides mechanisms on top of this for discovery, join and lookup.
It seems that, although all three could be used, RMI would be the best suited for this project. Using sockets would require the design of a communications protocol, which may introduce further problems. As all components of the software will be written in Java, the multi-language support that sockets provide is not required.

As Jini is primarily aimed at numerous arbitrary connections to the network, it seems that this technology may be overkill for this project. Any network connections to the tool would be handled in a controlled manner. Any small advantage brought by its use may be costly in terms of performance and complexity. Any of Jini’s facilities required could be simulated using RMI or another technology.

RMI seems just right in terms of how much the Java platform does for you. As well as being simple and robust, it deals most of the communication issues present in a distributed system.

**RULES TO IMPLEMENT**

It seems sensible to expand on the rules supported by [JVCS]. The extra rules chosen seem to reflect the most common changes that would be made in a normal evolution of a library. This would allow the resulting tool to have some real-life use. Labelling each rule by a letter, the following table lists and describes what they check for.
Rule A  Checks for any deleted public classes.
Rule B  Checks for any change in the inheritance chain.
Rule C  Checks for any deleted public fields.
Rule D  Checks for any deleted public methods.
Rule E  Checks for classes that have become abstract.
Rule F  Checks for classes that have become final.
Rule G  Checks for classes that are no longer public.
Rule H  Checks for any deleted constructors.
Rule I  Checks for a breaking change of access of fields (should run after Rule C).
Rule J  Checks for a breaking change of access of methods (should be run after Rule D).

| Table 2 - List of Supported BC Rules |

Algorithms to decide these rules are in Appendix C.

**COMPONENT DESIGN: RULE ENGINE**

This component will be in charge of checking two arbitrary sets of classes for BC breakages. It will not be concerned with versions or distribution, and so a fixed way for the external Rule Server to call on the engine will be provided.

Requirement P2 says that more rules may be added in the future. To allow for this, the Rule Engine will have an open structure – ‘rules’ may be installed and uninstalled as required. The abstract class Rule representing a general rule will be created – concrete classes subclassing this can successfully be installed in the Rule Engine. An abstract class was chosen over an interface, as the former allows common code to be shared between the rules. Also, static variables can be shared amongst the subclasses (e.g. the library classes being checked).
Component Design: Rule Server

The Rule Server has two jobs in the tool. It will first act as a repository for class binaries, allowing clients to ask for and access a certain version of a library. It will also provide a way for clients to check relevant library versions for any BC breakages, via a Rule Engine.

When compiling applications, the compiler needs a list of public members of any client classes used. This is usually provided by the class file itself, or an implemented interface. The compiler uses this to make sure all references are well formed and type safe. Thus, for Application Developers to compile their programs, the Rule Server needs to make available the classes being compiled against, in a standard class file format. Version information regarding this library should also be provided. The easiest way to do this is via an HTTP server — Java compilers can use these as a code base, or alternatively Application Developers can download classes from here directly.

Methods must be provided for an application to request the version number of the latest library compatible with it. There are two ways to infer this:

1. When new versions are made available, calculate this value for each currently available library. Checking the new library for BC against each old version, and updating a ‘LatestBCVersion’ field for each version does this. Assuming a BC check is costly, this may be inefficient as values for versions no longer in use may be calculated. As applications may be using any version of a library, all of them have to be checked. The algorithm is straightforward, and easy to implement.

2. Calculate the value each time an application requests it. Each library is stored with the LatestBCVersion, as well as the latest version is had been checked with. If the latest version checked with is the latest
version published, then the LatestBCVersion is accurate. If not, the value must be found by brute force – i.e. do a BC check for each version between the last checked and the latest version. Exploiting the transitive property of BC can reduce this ‘gap’. Advantages of this solution is that only libraries in use are subject to this procedure, and then only for the first time an application asks for it (between new libraries). However, in certain situations (e.g. when numerous libraries are published), the actual work done by this algorithm may be more than the first.

The first algorithm is more efficient than the second only in the case when libraries are always changed in a way that breaks BC. Since it is envisaged that this is not the case (that in fact, concurrent versions will usually keep BC), the second algorithm will be implemented for the tool.

Pseudocode for the second algorithm follows. ‘Ver’ represents the version that the application is using. It has two fields, LastCheckedWith and LatestBCVer. LatestVersion is the last published version.

---

1As this is when the second algorithm duplicates work the most.
if (LatestVersion == Ver.LastCheckedWith)
    return Ver.LatestBCVer; // up to date
else {
    // easy check – BC is transitive
    while (true){
        if (Ver.LatestBCVer == Ver.LatestBCVer.LatestBCVer)
            end while;
        Ver.LatestBCVer := Ver.LatestBCVer.LatestBCVer;
    }
    // mark bottom of ‘gap’
    float CheckTill;
    CheckTill := max(Ver.LatestBCVer, Ver.LastCheckedWith);

    // Hard check – use brute force.
    for (CheckVer: all Vers from LatestVersion to CheckTill){
        if (BCCheck(CheckVer, Ver)){
            Ver.LatestBCVer := CheckVer;
        end for;
    }
}
Ver.LastCheckedWith := LatestVersion // by above
return Ver.LatestBCVer;

Figure 8 - Pseudocode of Versioning Algorithm

An informal inductive proof follows, as proof that the above algorithm is correct. Correct in this context means that at all times in which the algorithm is not running, all library versions and associated information are accurate. This means that for Ver, LatestBCVer is the highest published version up to LastVersionChecked such that (Ver)BC(LatestBCVer) holds true, and that LastVersionChecked == LatestVersion after running the algorithm for Ver.
Base Case:

When libraries are published, LatestBCVer := LastVersionChecked := Ver. This is correct, trivially, as all versions are BC with themselves.

Inductive Case:

Assume that at a start of a call to the algorithm, all information regarding Ver is correct, and LCV <= LV.

If the first if statement is true, all information is already correct, and LVC and LV are equal.

If not, before entry into the while loop, LCV is strictly less than LV. LBC is correct, however. During the while loop, LBCV is set to LBCV's LBCV, until no change in LBCV's value. By transitivity of BC, (LBCV)BC(Ver) holds. The loop exits without changing any variables. Now, LBCV >= LBCV_o and LBCV <> LCV.

CheckTill is set to the larger of LBCV and LCV. All versions between CT and LBCV are guaranteed not to be BC with Ver. If CT == LBCV, then this is vacuously true. If CT == LCV, then this is true by the Inductive Property.

The for loop sets LBCV to the highest BC version between LV and CT. Due to the guarantee given above, the value of LBCV after this loop will be the highest such that (LBCV)BC(Ver) holds. In the extreme case, LBCV does not change, thus there are no versions between LBCV and LV that are BC with Ver, and so LBCV has the correct value.

LCV is set to LV. This is correct, as there are no versions between LBCV and LV that are BC with Ver (by the for loop above). Hence all information is now correct, and LCV == LV.

Hence the algorithm is correct. □

Figure 9 - Proof of Correctness of Versioning Algorithm

The repository where libraries will be stored can now be designed. This should be designed as abstractly as possible to enable the libraries to be stored on different media (e.g. on disk, or in a database). For this particular implementation, a disk will be used, but the classes that deal with storage may be rewritten to use a database instead.

The structure of the (disk) repository will be as follows. Each library will have its own directory in the root repository directory. Each library directory has a unique name chosen by the Library Developer. Each library version will have its own directory in this library directory, named x.x, the version that it represents. Each version directory will have all the classes belonging to a library. These are unaltered Java classfiles, output from a compiler. They will
be named <classname>.v<x.x>, where classname is the original name of the class, and x.x is the version of the library (same as the directory it is in).

The library directory will also have a ‘Latest’ directory. This directory contains the latest library classfiles (with their original .class names). As the full path to this directory has an unchanging and known name, it is easy to make it available via an HTTP or FTP server, providing access to Application Developers. The directory will also include a version.txt file, containing the version of the library and contents of the directory.

This repository structure seems to be the most flexible and robust, containing persistent and redundant information. A custom classloader will be required to retrieve classes of a certain version, as classes are not named conventionally. In any case, some control over which class to load is needed – there will be many versions of the same class in the repository, so care must be taken not to reload misidentified classes.
PERSISTENT INFORMATION

Some information about known libraries, versions, etc. is encoded in the directory structure of the repository. The algorithm needs more detailed information, like the LV for each library, and LCV and LBCV for each version in a library.

To allow this information to be persistent, it will be held on disk. There are many ways to do this – objects representing this information can be serialised, or a human readable text file can be built. The latter is a better solution, as administrators can read and edit these settings files directly.

A settings file (settings.txt) will be included in each library directory, containing information regarding the versions published. Each published version will have to be listed, as the versioning algorithm needs to know which libraries were made available to Application Developers to avoid checking against an unreleased version.

Each published version will also have a settings file (ver<x.x>.txt), containing information required by the versioning algorithm (Ver, LatestBCVersion, and LastComparedWith). The exact format of these files will be described in next chapter.

Again, access to these files should be as abstract as possible, allowing future implementations to use (for example) a database instead.

COMPONENT DESIGN: LIBRARY DEVELOPER

The Library Developer’s use of the system will be quite straightforward. Methods must be provided to upload a set of classes to the server under a version number, check for BC of this version against the last published library, and to finally publish the just uploaded version. Publishing makes the library publicly available via the Latest directory.
This allows Library Developers to upload libraries, but not release them. Unpublished libraries will not be available at any time to Application Developers, and will not take part in any future BC checks (including those in the versioning algorithm).

**COMPONENT DESIGN: LIBRARY USER**

There are three ways that classes can be distributed to clients.

1. Creating a custom classloader that can access the libraries, and accessing instances of classes via this classloader.

This would work by providing a general ‘tool classloader’ to application developers. This would override a standard `java.lang.ClassLoader`, and would be in charge of connecting to the Rule Server, providing adequate version information, and to transfer classes to the local JVM. The library binaries themselves are not stored locally on disk – they will be stored as instances of `Class` in the local JVM.

The details are as follows. The `loadClass(String s)` method of the `ClassLoader` returns an instance of a `Class`. This is an abstract representation of a library. The `newInstance()` method of `Class` returns an object (of type `Object`) as an instance of the loaded class.

Due to the way that a normal JVM links and loads classes, in order to use this object safely and effectively Application Developers can use two methods. First they can probe the `Class` object representing the object (via the reflection API), to find out about the members belonging to the object. These can then be ‘invoked’. As all these ‘meta-method’ calls are available in the standard library, this would compile, link and load successfully. This seems to break requirement D5, as Application Developers would have to change their programming style to accommodate this restriction (even if an API to ‘wrap’ this process is provided by the tool).
The second method is to cast the object returned from the newInstance() method to a known type (in this case, an interface). The Application Developer can then use members of the (casted to) interface to invoke methods of the object. This should link and run correctly, provided that the cast is correct. This still requires Application Developers to pay more attention when instantiating objects, although the general use of the object is as normal, complying with D5.

The use of RMI presents more difficulties. It seems that Class instances cannot be successfully serialized and deserialized without the actual classfile (that the Class instance represents) is available. This is the case whether the RMI system is allowed to marshal and unmarshal Class instances\(^2\), or if serialization and deserialization of Class instances is done manually\(^3\) (using java.io.ObjectOutputStream and ObjectInputStream, respectively).

A slight variation of this technique is to transfer the class as a binary, i.e. before the classloader loads and defines it. This can be done by transferring the bytes composing the class file to the client via RMI, and then loading the class by using these bytes instead of those from a file on disk. As the class is no longer being serialized, there are no deserialization problems, and the class is correctly defined and loaded.

\(^2\) At runtime, this resulted in a java.rmi.UnmarshalException: ClassNotFoundException when a call to newInstance() was made. The exception was not thrown when the class was available in the classpath. Of course, the version of the class made available was then used.

\(^3\) At runtime, this resulted in a ClassNotFoundException when a call to readObject() was made. It is assumed that this was the same exception thrown as above. The exception was not thrown when the class was available in the classpath.
As explained above, custom classloaders require defined interfaces (to cast to) to comply with D5. Interfaces would have to be both defined and made available by Library Developers, outside of the control of the tool. This restriction is in breach of requirement D7, as these interfaces could then not be changed at a later stage via the tool.

2. To use an HTTP server, in conjunction with CGI, to dynamically provide classes via normal HTTP delivery methods.

This solution requires the least amount of extra work for the Application Developer. Little, if any, change is required to application sources to use the tool. Instead, client information will have to be encoded in the classpath passed to the JVM. When this classpath is searched for libraries by the
runtime, the HTTP server will automatically provide the correct classes, depending on the directory requested of it.

It would be difficult for the tool to change the runtime classpath of the client JVM. This means that an arbitrary library version cannot be included in the classpath. The Rule Server would have to keep persistent information regarding what library version each client is using. So, a unique constant login ID for separate users (potentially many more than the actual applications developed) would have to be provided. This login would be included in the classpath, and then in conjunction with the CGI protocol, passed on to the Rule Server:

```
java -cp http://rules.bom.com/app/lib/id/ app
```

Because of these identification issues, this solution breaks requirement P5. Even without this requirement, the potential number of clients would increase the administrative cost of using the system. Requirement D6 may
also be affected, as it would be very difficult for the tool to let developers know what library an application is using. A separate application to log on and query the Rule Server for this information would be necessary, as the application’s access to the server is only one way (downstream HTTP).

3. To make use of a local ‘classcache’ that would be managed by the tool.

Here, local copies of the classfiles required are made available to the JVM on a local known classpath. The JVM will find them at runtime, as normal. This ‘classcache’ would have to be maintained by a ‘library synchroniser’, provided by the tool, and invoked by the application. Although it will be slower than other solutions when updating the cache, it would be the quickest when no updating is required.

---

Figure 13 - Updating a Classcache
This solution requires no change to object declarations in the source of the application, just a call to the synchroniser. Also, this is the only solution that will work if the network or servers become unavailable.

Solutions 1 and 3 allow the application to hold information about which library versions are being used by it. This can be passed to the Rule Server to decide further action. Like persistent information in the Rule Server, this information could be held in a text file, managed by the classloader or library synchroniser. Thus, no identification of the client is required, and an arbitrary number of them can run, fulfilling Requirement P5. The file will be named lcl<name>.txt, where <name> is the descriptive name of the library being used.

It seems that solution 3 seems to fit the requirements more than the others. Choosing this to implement, it can be discussed in more detail. An initial classcache should be distributed with the application. This ensures availability of library classes and a well-formed and accurate local settings file. The application should invoke the synchroniser as soon as possible, to update any classes in the cache. The synchroniser connects and passes version information to the Rule Server, which checks for any latest libraries and sets up a file transfer if there are any. The application then continues as normal.

The Synchroniser will have two modes – Diagnostic and Normal. Normal mode would download the latest files that are BC. This should be transparent to any user, with no input required, and no output returned. If this process fails at any step, the application should continue as normal, unaffected.

The Application Developers will use diagnostic mode. It will notify them of the highest library version that can be used by their application. If this is not the latest version available, details of why and how the new version breaks BC will be provided, to help the Application Developer compile the application.
with the library available in the Latest directory. The application is then redistributed, with an updated classcache and version file.

**Requirements Analysis**

It seems that this design conforms to most of the requirements specified in Chapter 4. However, some by nature cannot be checked until the tool has been implemented. These include:

P2. A bit of a subjective requirement, it will be clear whether it is fulfilled or not when the extra rules specified are added.

P6. It remains to be seen how the tool affects standard Java facilities. Unanticipated problems may occur during testing, that cannot be verified by a design alone.

D5. Although only one method call is required to use the tool, it is not apparent how the programming process in use by the Application Developer will be affected. They may already have versioning processes or systems in place that conflict with the workings of the tool.

D8. Delays in execution may be noticeable by users. Whether this breaks this requirement depends on the length of any delays.

D9. This is fulfilled provided standard libraries are available to the user. Assumption A4 assures us that this is the case.

D10. Testing will confirm whether sample applications run as intended.

D11. Testing will confirm whether the JVM correctly loads classes from the cache.

**Visual Representation of the Design**

The following diagram illustrates the informal architecture of the tool. Finalised class diagrams are included in Appendix D.
Figure 14 - Architecture of the Tool
Chapter 6

IMPLEMENTATION AND TESTING

Here, the issues involved whilst implementing the design of the previous chapter is described. The major ‘brick walls’ encountered are recorded and solved. Some further design decisions are made and justified. When required, component tests are described and results from these are given.

RMI FILE TRANSFERS

Although RMI provides automatic transfer of primitive types and (serializable) classes, there does not seem to be any facility to transfer files. This is not surprising, as files are accessed abstractly via handles, and if anything were to be transferred it would be those.

So a method to transfer files easily over RMI was devised. This was developed in separation of the tool in general, keeping the task of file transfers isolated from the main problem. This was to make sure it worked correctly. The method is an adaptation of a technique used by Kang Yuan, who used RMI to transfer files for another project. His implementation was highly tied to his project – the aim here is to provide a protocol that may be reused in future applications. The following describes a RemoteOutputStream, used for uploading files. A RemoteInputStream was developed similarly.

There are two classes involved in the transfer (upload) of a file. One is a RemoteOutputHandler, a remote wrapper of an OutputStream residing on the server. It has remote methods calling each available method of
an OutputStream. A reference to this remote wrapper is passed to the client who builds a RemoteOutputStream around it. This class extends the Java class OutputStream. This overrides all methods presented in the OutputStream class, and can safely be used where methods require an OutputStream class (other OutputStreams, for example). Each overridden method calls the corresponding method of the passed RemoteOutputHandler, passing the parameters to and returning any result from them.

This works well when parameters and return types are serializable or primitive. However, the read methods of the InputStream are slightly different in that their parameters include a reference to an array of bytes. As this is a reference local to the remote method, any bytes copied to the array are lost when the remote method returns. The initial naive implementation of one of the methods is given below. The remote method is on the left, the local on the right. For clarity, exception handling has been omitted.

```java
public int read(byte[] b){
    return ins.read(b);
}

public int read(byte[] b){
    return rih.read(b);
}
```

Instead, a copy of the altered array is returned, as well as the int as required by the overridden method. To do this, an auxiliary field and method are used to store/read the ‘last int returned’ by a read method, and the required array is returned by the remote method instead (along with its contents). Both of these are used to implement the local read method.
public byte[] read(byte[] b)
{
    lastInt = ins.read(b);
    return b;
}

public int getLastInt()
{
    return lastInt;
}

public int read (byte[] b)
{
    byte[] temp = rih.read(b);
    int length=rih.getLastInt();
    for(int i=0; i<length; i++)
        b[i] = temp[i];
    return length;
}

Figure 16 - Correct Implementation of a RemoteInputStream

Other methods are straightforward to implement, and follow the structure of the naïve read method above. This second implementation works as intended.

**Component Testing: RMI File Transfer**

A toy application was set up to test both remote streams, and files were uploaded and downloaded correctly over RMI. As executable binaries and compressed files were successfully transferred, the integrity of the transfer is assured for classfiles as well.

**Disabling Default ClassLoading**

A major concern was the inability to disable the default JVM classloader. Due to the delegation model\(^4\) of classloaders, the default classloader is always used to load a class before any custom classloader. As library classes were present with correct names in the Latest directory of a repository, any class requested to be loaded would be loaded from this directory by the default classloader. To stop this from occurring, the Latest directory must be made unreachable from the JVM running on the server, by not allowing it to be in the classpath. This does not seem to break any requirement, so is a satisfactory solution to the problem. To reduce further loading errors, whenever the latest classes are

\(^4\) Sometimes referred to as a Chain of Responsibility pattern
required (for BC checks) they will be retrieved from the respective version directory, and not the Latest directory.

**THE **\textbf{RuleClassLoader}**

The tool required a custom classloader that could be directed to an arbitrary directory (not in the current classpath) to load classfiles with arbitrary extensions. The \texttt{RuleClassLoader} was developed for this role. It is an adaptation of the example \texttt{Simple1_2ClassLoader} class\footnote{The \texttt{Simple1_2ClassLoader} is free to use for non-commercial applications. Please refer to full license in \texttt{RuleClassLoader} source.} written by Ken McCrary\footnote{The Simple1_2ClassLoader is free to use for non-commercial applications. Please refer to full license in RuleClassLoader source.}. This class loader loaded classfiles with the .impl extension, from a fixed directory. This extension and directory can now be set using the new \texttt{RuleClassLoader}'s constructors. A public static method was also added to obtain a vector of classes, representing the version of a library. This was used to feed a Rule Engine.

**COMPONENT TESTING: CUSTOM CLASSLOADER**

First, a test interface was developed. This defined a single print method. A test class was written implementing this interface, with along with a private integer field that the print method outputted to the screen. Two versions of this class, with different integer fields were compiled, renamed and placed in separate directories. The classloader was then invoked in another test application, which had access to the interface (to cast objects to), but not the classes. The classloader was used to load each version of the test class, and the output checked to see if the correct integers were being printed. As the correct versions, in the correct order, were loaded it is certain that the classloader works as intended.
THE ABSTRACT RULE CLASS

The abstract Rule class contains fields and (abstract and non-abstract) methods that will be used by concrete rules. The abstract methods are used by the Rule Engine to check for BC breakage for an arbitrary rule. The following figures describe the roles of the important fields and methods. Notice that the rules do not specifically have to check for BC breakages, and may check for other properties. The main use for fields of type String is to build a coherent diagnostic string, obtained via the diagnose() method.

<table>
<thead>
<tr>
<th>Protected Field</th>
<th>Purpose</th>
<th>Example from Rule A</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector newClasses</td>
<td>Vector containing all classes from the new class set.</td>
<td>Lib v1.5</td>
</tr>
<tr>
<td>Vector oldClasses</td>
<td>Vector containing all classes from the old class set.</td>
<td>Lib v1.0</td>
</tr>
<tr>
<td>String rule</td>
<td>String describing the rule's operation.</td>
<td>“Checks for deleted Classes”</td>
</tr>
<tr>
<td>String ruleName</td>
<td>String giving the rule a short name.</td>
<td>“Rule A”</td>
</tr>
<tr>
<td>String ruleExp</td>
<td>String used for diagnosis - Explaining how BC has been broken.</td>
<td>“The following Classes are missing:”</td>
</tr>
<tr>
<td>Vector ruleBreakingClasses</td>
<td>Vector of classes in the new set that are responsible for BC breakage.</td>
<td>Classes missing in the new library.</td>
</tr>
</tbody>
</table>

Table 3 - Fields in the Abstract Rule Class
<table>
<thead>
<tr>
<th>Method</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract boolean</td>
<td>Applies the rule to the current class sets. Responsibility of concrete subclass.</td>
</tr>
<tr>
<td>checkRule()</td>
<td></td>
</tr>
<tr>
<td>abstract string</td>
<td>Returns a coherent diagnostic string describing if and how the rule was broken. Responsibility of concrete subclass.</td>
</tr>
<tr>
<td>diagnose()</td>
<td></td>
</tr>
<tr>
<td>abstract vector</td>
<td>Returns the Vector of classes responsible for BC breakage. Responsibility of concrete subclass, and specific to each rule.</td>
</tr>
<tr>
<td>getDetails()</td>
<td></td>
</tr>
<tr>
<td>void setClasses(Vector nc, Vector oc)</td>
<td>Sets the classes to be compared by this rule. Overwrites any currently held class sets.</td>
</tr>
<tr>
<td>static Class extract(Class c, Vector v)</td>
<td>Returns the Class with the same name as c in the Vector of Classes, v. Used to extract corresponding classes in the different class sets.</td>
</tr>
</tbody>
</table>

Table 4 - Important Methods in the Abstract Rule Class

For further ease of development, two further abstract classes (each subclassing Rule) were created. One was for checking BC rules concerning members of a class (Rules C, D, H, I and J), and one was to check BC rules concerning classes as a whole (Rules A, B, E, F and G). These contain various utility methods relevant to their rule type and concrete implementations of some of Rule’s abstract methods. For example, a concrete implementation of checkRule() is given in both, utilising a new abstract method, applyRule(). In RuleClass, applyRule() is applied to each pair of classes (each correspondingly from each library) and returns a boolean; in RuleMember, it is again applied to a pair of classes but returns a Vector of the ‘problem’ members. The diagnose() method is also implemented for both, each implementation building the diagnostic string in a way that makes sense for each type of rule.
Due to this Rule class hierarchy, the concrete rule classes have very little code – a constructor setting the base strings and a concrete implementation for `applyRule()` method. It is envisaged that further added rules will be created just as easily.

The RuleEngine then iterates through each installed rule (held internally in a `Vector`), calling the `checkRule()` method of each. Depending on the result of each of these, the diagnostic string is constructed.

**COMPONENT TESTING: RULE ENGINE AND RULES**

The Rule Engine and Rules were tested in isolation. This was to rule out any errors due to RMI or file transfers, etc. A test class was developed that could be ‘adapted’ and checked for BC breakage. Different versions (with missing members, for example) of this test class were then compiled into two separate directories, and renamed so that any default classloader would not load them. The RuleClassLoader was then used to load these two different versions into a new Rule Engine. The Rule Engine was initialised and started, and it’s diagnostic string returned to the system output. This was then checked against the changes made. All rules and combinations of rules tested resulted in the predicted diagnostic string.
public class TestClass extends TestClassSuper implements TestClassIF{
    //Fields
    static String ver = "1.3";
    private int one = 1;
    int two = 2;
    protected int three = 3;
    public int four = 4;
    //Constructors
    static {
        System.out.println("tc class loaded" + ver);
    }
    public TestClass() {
        System.out.println("tc constructor called " + ver);
    }
    public TestClass(int i){
        System.out.println("constructor called + i + ver");
    }
    //Methods
    private void print(String s){
        System.out.println(s + ver);
    }
    void print(int i){
        System.out.println(i + ver);
    }
    protected void print(char c){
        System.out.println(c + ver);
    }
    public void print() {
        System.out.println("TestClass, version: " + ver);
    }
}

Figure 18 - TestClass used for Rule Testing

For example, deleting the public field four resulted in the following correct output:
Results of Rule C
(Checks for any deleted fields!):
Rule C Failed!
The following classes have missing fields:
TestClass

    public int TestClass.four

Figure 19 - Breaking Rule C

While reducing the field’s access modifier resulted in:

Results of Rule I
(Checks for a breaking change of access of fields):
Rule I Failed!
The following classes have fields with less access then they had before:
TestClass

    private int TestClass.four

Figure 20 - Breaking Rule I

All ten rules were tested in this way and seen to be correct.

**PERSISTENT SETTINGS FILES**

Three settings files were designed in the previous chapter. These were implemented as RuleSettings (representing the settings.txt file), ServerVersionSettings (ver<x.x>.txt) and LocalVersionSettings (lcl<lib>.txt). As well as the information required by the design, extra redundant information was included to provide some fault tolerance. These include a path of where the library resides, and the ID of the library associated with the file.

The general structure of these classes were the same – private fields representing each unit of information, public methods to set and get these values, and save and read methods to write and read the text files. It would be
these last two methods that would be re-implemented if the data were to be stored on a database. For this implementation, they open a file on disk, read/write data from/to the file, and then close them. The relevant IOExceptions are thrown if any errors occur.

<table>
<thead>
<tr>
<th>classId</th>
<th>tc</th>
<th>dirDescrip</th>
<th>version</th>
<th>latest compatible version</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>tc</td>
<td>repository/tc</td>
<td>1.0</td>
<td>1.2</td>
</tr>
<tr>
<td>number of published sets</td>
<td>dirDescrip = repository/tc</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>currently published sets = latest compatible version =</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.0</td>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.1</td>
<td>last compared with = 1.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 21 - Sample setting.txt and ver1.0.txt Files

<table>
<thead>
<tr>
<th>classId</th>
<th>tc</th>
<th>dirDescrip</th>
<th>version used</th>
</tr>
</thead>
<tbody>
<tr>
<td>tc</td>
<td>classcache</td>
<td>version used =</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 22 - Sample lcltc.txt File

**COMPONENT TESTING: PERSISTENT SETTINGS**

All three settings files were tested similarly. Objects were created and set, and saved to disk. These were checked by an ASCII viewer for consistency, and then read back into a fresh object. Data from this was outputted to the screen.
and again checked for consistency. Both the file and runtime objects were consistent at all times.

**RULE CONTROLLER AND SLAVE OBJECTS**

The server is implemented by publishing a single ‘shell’ RuleController class. This accepts connections from both submitters and synchronisers, and then creates ‘slave’ objects to handle the specific job depending on the client (either a RuleRemoteObj for a submitter, or a RuleRemoteCL for a synchroniser). References to these slaves are returned to the client. No further communication is made to or from the controller class.

<table>
<thead>
<tr>
<th>RuleRemoteObj</th>
<th>RuleRemoteCL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Retrieve up to date version information, including last uploaded and published versions.</td>
<td>Retrieve the latest compatible version of a library with respect to the version being used by the client (using the version algorithm). Update SVS files, if necessary.</td>
</tr>
<tr>
<td>Prepare uploads of class files.</td>
<td>Prepare download of class files.</td>
</tr>
<tr>
<td>Check BC of arbitrary classes.</td>
<td>Check BC of arbitrary classes.</td>
</tr>
<tr>
<td>Publish a library.</td>
<td>Retrieve file lists of members of a published library.</td>
</tr>
<tr>
<td>Save and close all persistent settings files. Ensure these files and repository structure are well formed.</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 5 - Responsibilities of and Facilities offered by Slave Objects

See the JavaDoc documentation in Appendix E for the specific methods providing these facilities to clients.

This controller-slave pattern was used to make the server object as robust as possible. As it is simple, less can go wrong with it. The slave objects do any heavy processing. As the slave objects are also remote objects, they could even run on another machine, allowing the possibility of one server for multiple repositories, mirrors, etc. Finally, as the slave objects deal with the
specific storage of classes, they may be re-implemented to use a database store instead. The remote interface would ensure that other components would still work with any new implementations.

Code not relevant to class storage was placed in the controller class, so that new implementations would not have to re-implement them. This had another advantage, as this code concerned the loading and initialisation of RuleEngines to check for BC, and was shared between slave objects anyway. An installRules() method to be used when initialising Rule
Engines is provided in the main server executing class, to ensure that all BC checks use the same number and types of Rules.

**LIBRARY DEVELOPER CLIENT**

The RuleSubmit component acts as a front end to a newly created RuleRemoteObj passed from the controller. It connects and retrieves a reference to this remote object, obtains up to date version information from the server and allows the Library Developer to upload, check for BC against the last version and publish a library.

---

Figure 24 - RuleSubmit Sequence Diagram
APPLICATION DEVELOPER CLIENT

The `ClassSyncr` class has a single `syncClasses()` method that manages the local classcache. When a host application invokes this method, it connects and retrieves a reference to a new `RuleRemoteCL` from the controller. It then checks for the latest BC library version. If there is a new version, it clears the current cache (taking care not to delete any non library files from it by using a file list of the old library). It then downloads the new library to the classcache. If necessary, a diagnosis is returned to the Application Developer.

Figure 25 - ClassSyncr Sequence Diagram
FAULT TOLERANCE AND USABILITY

To increase usability, and to fulfil requirement D8, all components were written using the exception mechanism provided by Java. All exceptions thrown by the Java API are eventually caught. A new RuntimeException is then thrown describing when and how the exception was thrown. If necessary, these exceptions are thrown across the RMI connection to be caught by the clients (both submitters and synchronisers).

If an error does occur, the current transaction ends, the clients are notified, and the remote slave objects are discarded. The controller is not affected, and is available to provide further slave objects to new clients. Thus the server does not need to be shut down or restarted in erroneous circumstances.

Integrity of the repository is also checked during runtime. If the directory structure is incorrect or incomplete, the server will create new default directories and settings files containing safe default values. Thus clients will continue to work if the repository is deleted or moved. In the event of no current published library, the last version uploaded is published as the latest version.

If the application’s synchroniser catches any exception while it is running, it will end the current transaction and allow the application to continue running, using the last version of the library in the classcache. Errors include losing a network connection, or the crashing/unavailability of the Rule Server. The Application Developer is also notified of the exception, provided the debug mode is true.

All main executing methods were also written to take as many runtime parameters from the command line as possible. As the class synchroniser is
called by a host application, options are set via its constructor only. Refer to the user guide in Appendix F for further details.

<table>
<thead>
<tr>
<th>Class</th>
<th>Arg 0</th>
<th>Arg 1</th>
<th>Arg 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>RuleServer</td>
<td>Path to repository</td>
<td>Location of RMI Registry</td>
<td>-</td>
</tr>
<tr>
<td>RuleSubmit</td>
<td>ID of lib to be submitted</td>
<td>Path to transfer directory</td>
<td>Location of RMI Registry</td>
</tr>
</tbody>
</table>

Table 6 - Command Line Arguments

**PACKAGES**

The tool is distributed in four packages. These are summarised below.

<table>
<thead>
<tr>
<th>Package Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bcrules</td>
<td>Contains RuleEngine and all supplied Rules, including Rule, RuleClass and RuleMember</td>
</tr>
<tr>
<td>rmifiletransfer</td>
<td>Contains RemoteOutputStream, its remote interface, and RemoteOutputStream. Also contains the corresponding InputStream classes.</td>
</tr>
<tr>
<td>ruleserver</td>
<td>Contains classes relevant to the Rule Server, including the slave classes, custom classloader, and FileFilter implementations. Relevant remote interfaces are in this package, as well as classes to handle configuration files.</td>
</tr>
<tr>
<td>rulesubmit</td>
<td>Contains single class, to be run by Library Developers to submit classes to a Rule Server.</td>
</tr>
<tr>
<td>classsync</td>
<td>Contains class to manage a local classcache and a class to handle the local version file.</td>
</tr>
</tbody>
</table>

Table 7 – Packages

Packages allowed the isolated development of the components. They allow the components to be used separately (e.g. package rmifiletransfer may be used in another project, similar for bcrules).
INTEGRATION TESTING

At this point, it was assumed that RMI file transfer streams, custom classloaders, the Rule Engine, and the settings classes were working as intended (see the relevant component tests for details). As the versioning algorithm was proven for correctness, it was assumed that any later versions identified by the RuleRemoteCL class were correct\(^6\). So only the interactions between the three locations (Rule Server, Rule Submitter and the classcache synchroniser) were tested during integration. Interesting behaviour of test client applications and libraries using the tool are deferred until the evaluation.

The test class from the testing of the Rule Engine was used here as well. Different versions were compiled, checked for BC, and uploaded to the server. The server’s repository and associated settings files were checked to see if they reflected the correct information between each submission. The data checked is listed below. Contents of the classes (i.e. their programmed behaviour) were not checked at this point.

<table>
<thead>
<tr>
<th>Tested Item</th>
<th>Desired result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Settings.txt file</td>
<td>Correct ID, path, number of published files. The version at the bottom of the list should be the version just published.</td>
</tr>
<tr>
<td>Repository structure</td>
<td>Correct number and values for directories. Names should reflect all versions uploaded (not just those published).</td>
</tr>
<tr>
<td>Individual SVS files</td>
<td>Only present in published libraries. Should contain correct ID, path. LCV and LBCV should equal the version the file belongs to.</td>
</tr>
<tr>
<td>Version directories</td>
<td>Contains correct number of classfiles. Each classfile is named correctly.</td>
</tr>
<tr>
<td>Server</td>
<td>Still running, and able to accept further connections.</td>
</tr>
<tr>
<td>BC Output</td>
<td>A correct diagnostic string reflecting the changes made between submitted versions.</td>
</tr>
</tbody>
</table>

Table 8 - Tested Items Between Submissions

\(^6\) The java implementation of the algorithm was a relatively straightforward translation of the pseudocode given in chapter 5.
A test application was then created to use the published test class. It used one of the public print methods, to print a given integer as well as the version of the library being used. The application also makes use of a classcache and synchroniser, with the debug option activated. The class cache had an unaltered test class published as the first version, and the LVS file was written to reflect this.

Various new versions of the library were then published, with the application run between each publication. The table summarises the intended behaviour of the application at the various times it was run.

<table>
<thead>
<tr>
<th>Library Status</th>
<th>Application version</th>
<th>Desired Behaviour</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.0</td>
<td>v1.0</td>
<td>No change, application runs with v1.0</td>
</tr>
<tr>
<td>v1.1, 1.1BC1.0</td>
<td>v1.0</td>
<td>Download v1.1, run using v1.1. LVS reflects change.</td>
</tr>
<tr>
<td>v1.1, not(1.1BC1.0)</td>
<td>v1.0</td>
<td>No change, application runs with v1.0</td>
</tr>
<tr>
<td>v1.2, 1.2BC1.1, 1.2BC1.0, 1.1BC1.0</td>
<td>v1.0 or v1.1</td>
<td>Download v1.2, run using v1.2. LVS reflects change.</td>
</tr>
<tr>
<td>v1.2, 1.2BC1.0, not(1.1BC1.0)</td>
<td>v1.0</td>
<td>Download v1.2, run using v1.2. LVS reflects change.</td>
</tr>
<tr>
<td>v1.3, not(1.3BC1.2), 1.2BC1.1, 1.1BC1.0</td>
<td>v1.0, v1.1 or v1.2</td>
<td>Download v1.2, run using v1.2. LVS reflects change. No change if using v1.2.</td>
</tr>
</tbody>
</table>

The desired behaviour was confirmed when the application outputted the library version on the screen. The static constructor of the test class was also called, as was expected. Any adjusted SVS files are assumed to be correct, as was proven by induction. As these cases can be combined to form any general situation, the synchroniser is working as required.
The following is the sample output from various cases. The first is if the application is already using the latest version. Debug output is in bold.

```
main started, debug = true
found: tc ver: 1.0
got rrcl obj!
no new version available!!
You are currently using tc version 1.0
This is the latest version
tc class loaded 1.0
tc constructor called 1.0
```

Figure 26 - Up To Date Library Output

The next case was when the application was able to download a new library, and that version was the latest.

```
main started, debug = true
found: tc ver: 1.0
got rrcl obj!
new version available on server...
new version number: 1.4
attempting to delete: TestClass.class
file deleted
copied 1255 bytes...
transferred file: TestClass.v1.4
You are currently using tc version 1.4
This is the latest version
tc class loaded 1.4
tc constructor called ver 1.4
```

Figure 27 - Latest Library Downloaded

The final case is if there is a later version available on the server, but cannot be used because of BC restrictions (in this case, the deletion of a public field). The behaviour of interest is after any other libraries are downloaded.
main started, debug = true
found: tc ver: 1.0
got rrcl obj!
no new version available!!
You are currently using tc version 1.0
Latest version on Server is 1.5

This latest version breaks binary compatibility with you in the following way(s):
Results of Rule C
   (Checks for any deleted fields!):
The following classes have missing fields:
TestClass
   public int TestClass.four

tc class loaded 1.0
tc constructor called 1.0
Chapter 7

EVALUATION

This chapter presents an evaluation of the developed tool. First, arbitrary ‘everyday’ runs of the tool are applied. Any interesting results are presented. Then the drawbacks of using the tool are considered, and further work or enhancements that would increase the usefulness of the tool. Finally, the assumptions and requirements given in Chapter 3 are analysed to see if all are fulfilled.

BEHAVIOUR OF INTEREST

Since after classes are downloaded they are accessed normally, it is possible for Library Developers to also make use of the system as clients. As long as the use and location of the classcache is fixed for each library, the library should automatically manage new versions. This ‘library cache’ could even be located in the classcache, as other files in the cache are not affected when a library is being updated. Each library has its own LVS file, and only files from a library are deleted on an update. Of course, the library developers may not know the classcache name, so a different one would usually be used. This would have to be documented and told to other Library and Application Developers, and distributed as required.

Interesting behaviour is also demonstrated when applications extend library classes. In this case, the following is output:
As shown, it appears that after downloading a new library the previous version is still being used. This is odd, as the v1.0 classfile has been deleted, with the new version in place instead. Running the application again:

```
main started, debug = true
found: tc ver: 1.4
got rrcl obj!
no new version available!!
You are currently using tc version 1.4
This is the latest version
Print called 1.4
```

So this time the cache is unchanged, and the application works as intended. Somehow the JVM must load classes before the synchroniser can update them – even if the synchroniser is invoked before any declaration of library classes. As this behaviour only occurs when the Application Developer subclasses libraries, and not when the library is used directly, the subclassing mechanism must load superclasses at the start of the application’s execution. As the application still runs safely for the user (albeit with the previous library
version) requirement D10 holds. Also, as the correct behaviour is observed in further executions, P6 also holds.

A nice feature of the tool is that libraries do not have to just consist of classes. As the custom classloader only searches for classes in the version directories, any other files may exist in them without interfering with the BC checks or distribution of the classes. Hence, readme files, API documentation or data files may also be distributed via the tool. For example, the tool could be used to distribute a (continuously updated) list of user IDs.

As mentioned before, the Rule Engine can be used to check for other properties of a class. This can be done by creating new installable subclasses of the Rule class, and implementing the checkRule() method as necessary. For example, rules can be created that check a class for conformity to an unimplemented interface, or that it has no more than a certain number of public members. A particularly useful set of rules could detect palpable changes. The Engine itself would not have to be adapted.

The tool does not address palpability until now. Although the tool would benefit from palpability detection, the current implementation can be used to redistribute classes that may need recompiling to see the effects of a new library. Application Developers could write ‘shell’ programs that wrap the true functionality of their programs held in their own ‘pseudo libraries’. These could be distributed as a normal library by the tool (see the first point made in this section regarding library caches). If any of these pseudo libraries require recompilation due to a new library being published, they may be recompiled (using the unchanged source) by the Application Developers, and submitted to the tool. As the source is unchanged, BC compatibility is guaranteed, and the new pseudo library will be downloaded by all existing applications. This affects requirement D5, as applications would have to be written with this model in mind.
WEAKNESSES AND LIMITATIONS

Of course, the current form of the tool incomplete, as it cannot detect all changes that break BC. It can be argued that the ten it does detect are the most common, although this fact does not help if an application does not link to a library marked as compatible. The Rule Engine helps with this problem by allowing further rules to be developed.

Packaging is currently unsupported by the tool. All components currently assume that library classes belong to the default package. If libraries are added to a specified package, then both the RuleClassLoader and the default classloader belonging to the user application will fail to find the library class. Enforcing a correspondence between package names and library IDs, and adapting the custom classloader slightly to look for packaged classes can fix this. The classcache may also have to be adapted by adding another level of directories, each containing it’s own package.

The Java 1.2 security model requires a policy file to be passed to a JVM as permission to access the potentially insecure facilities offered by its API and classes (including RMI, Reflection and Classloading). During development, the policy file used disabled all security features. Although sufficient policy files can be distributed with the submitter and Rule Server, the Application Developer would have to supply their own to the users. Although examples may be provided by the tool, these would have to be checked and adapted for the application’s (and environment’s) own purposes.

Depending on the application, the characteristics of a cache mechanism could be fatal. Libraries may have been adapted to conform to a third party system, and any older versions may cause problems, especially if they cannot automatically be updated (due to BC or network unavailability). There are simple workarounds to the cases when the cache is not up to date (redistribution or even a simple restart of the application may be enough),
although what may be needed is a mechanism to halt the application from running if anything but the latest library is being used.

Although it is not specified how the latest library is distributed to the Application Developers (they are only in a known location), it is clear that they would have to obtain them manually. It would have been preferred if these classes were somehow distributed automatically, even though the requirements do not specify this. Still, it is likely that developers have experience with the download or remote access of library classes, so this is not that much of a grave issue.

A related problem is the existence of many copies of different libraries. Although these are managed automatically, the initial classcache has to be provided and set up by the Application Developers. Again, placing this responsibility with the Application Developers does not break any requirement, but automating the creation of the initial classcache would assist with the use of the tool.

A consequence of this is the (manual) maintenance work required by Application Developers on the local classcaches. For example, let a new library have a public method missing, thus breaking BC. This new library would not be downloaded by any application, even one that does not call this public method at any time (and so would link correctly with the new library anyway). Once Application Developers realise this, the only way they can use the new library is to manually update all caches, or (a potentially easier task) to redistribute the applications.

Although BC breakage information is supplied to an Application Developer whilst in debug mode, reasons why these changes were made are not available. This may be important, as Application Developers can decide whether or not it is worth redistributing a new application to users (If the change is safety critical then they may want to stop the application from
If the change is merely cosmetic, they may not want to notify the users at all.

**QUANTITATIVE ANALYSIS**

As was mentioned in the introduction, no existing technology is available that provides the feature set of the tool. This reduces the amount learned by any benchmarking made of the tool, although the results can be considered in absolute terms.

All output was suppressed until all timings were measured. The time was taken before and after a method call, and the average over many runs of the difference in these times was taken as the result. The same equipment was used for all tests, including server and client PCs, and the network. Details of these are in Appendix G.

The main results sought were those of class submission and retrieval and BC checking. These are the main processor heavy elements of the tool. File transfer is very network dependant, and the Rule Engine, by nature, involves a lot of inner loops. The following tables summarise the main results sought.

To analyse file uploads, results from the RMI File Transfer tests were used. Files of various sizes were opened, transferred and closed using the Remote Output Stream with different buffer sizes. The timings by using raw TCP/IP were estimated by transferring the file via FTP\(^7\). Overheads of using the FTP protocol are included in the timings.

---

\(^7\) The laptop acted as the FTP server, using Serv-u v.3.0 with default settings. The Desktop was the FTP client, using Leech-FTP v1.3 with default settings.
This table shows the trends that were expected. Larger files take longer to transfer. It is also noticed that the least time is taken when the buffer size is set similar to the file size. This was when there was the least amount of wastage per buffer, and the least amount of buffers in total was needed to transfer the file. Because of this, the least amount of RMI calls were made, reducing the overheads associated with each call.

The test was repeated for the Remote Input Stream:

This table shows similar relative trends to when files were uploaded. The absolute values are much greater (up to three times faster) than when uploading, showing an asymmetry between the two remote streams. It is not clear why this is the case, although it is probably quicker for RMI to package and send method arguments (used when uploading) than it is to package and send method return values (used when downloading).
It is also apparent how inefficient RMI is with downloading files in relation to raw TCP/IP. This was probably due to the overheads of marshalling and unpacking each call. It is interesting to see that upload speeds are comparable in both protocols. It seems that for small files, any speed advantage that FTP has is reduced by the overheads of using the protocol (during preparation of the transfer, for example). Since classfiles are usually no larger than a few Kb, this is an encouraging result for the RemoteOutputStream. However, since there will be potentially many more downloads than uploads, it may be worth replacing the RemoteInputStream with a lower level TCP/IP transfer mechanism.

The Rule Engine was analysed using various ‘dummy’ libraries. The time taken for the Engine to stop processing both sets of libraries was measured. Differences in the sample library were: the absolute and relative number of classes and members, and whether BC was broken or not. The same set of ten rules was used for each test, and members were shared among fields, constructors and methods. The following tables summarise the results.

<table>
<thead>
<tr>
<th>Number of Classes (Ten Members per Class)</th>
<th>New Lib: 1</th>
<th>New Lib: 5</th>
<th>New Lib: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Old Lib: 1</td>
<td>Old Lib: 5</td>
<td>Old Lib: 10</td>
</tr>
<tr>
<td>Time Taken</td>
<td>10</td>
<td>15</td>
<td>52</td>
</tr>
</tbody>
</table>

Table 12 - Engine Performance (Absolute Number of Classes) ms

<table>
<thead>
<tr>
<th>Number of Members (Ten Classes per Library)</th>
<th>New Lib: 1</th>
<th>New Lib: 5</th>
<th>New Lib: 10</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Old Lib: 1</td>
<td>Old Lib: 5</td>
<td>Old Lib: 10</td>
</tr>
<tr>
<td>Time Taken</td>
<td>Neg.</td>
<td>30</td>
<td>52</td>
</tr>
</tbody>
</table>

Table 13 - Engine Performance (Absolute Number of Members) ms
<table>
<thead>
<tr>
<th>Number of Classes (Ten Members per Class)</th>
<th>New Lib: 5</th>
<th>New Lib: 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Old Lib: 5</td>
<td>Old Lib: 1</td>
</tr>
<tr>
<td>Time Taken</td>
<td>15</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 14 - Engine Performance (Relative Number of Classes) ms

<table>
<thead>
<tr>
<th>Number of Members (Ten Classes per Library)</th>
<th>New Lib: 5</th>
<th>New Lib: 5</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Old Lib: 5</td>
<td>Old Lib: 1</td>
</tr>
<tr>
<td>Time Taken</td>
<td>30</td>
<td>Neg.</td>
</tr>
</tbody>
</table>

Table 15 - Engine Performance (Relative Number of Members) ms

<table>
<thead>
<tr>
<th>Time Taken for Ten Classes, each with Ten Public Members (Libraries BC)</th>
<th>Time for Ten Classes, new with Ten Protected Members (Libraries not BC)</th>
</tr>
</thead>
<tbody>
<tr>
<td>52</td>
<td>50</td>
</tr>
</tbody>
</table>

Table 16 - Engine Performance (Broken BC) ms

As expected, these results show that the more classes in a library, and the more members in a class, the longer the Engine takes to process them. Another expected result was that if BC breakage is detected, the time taken to check is not affected, as the rest of the classes/members would have to be checked anyway. The interesting result lies in the relative number of classes or members of a library. Although increasing the number of classes in a library does not affect the processing time much, adding new members does. This is unexpected – the prediction was that adding more classes would increase the time more than when adding more members. It is not clear why this happened. All the same, Library Developers may use this fact to make the distribution of new classes more efficient.
It was extremely difficult to make accurate measurements of time with the standard Java API\(^8\). They always read in multiples of ten, and the smallest non-zero value was 50 ms. Values reading as zero have been marked as negligible in the tables (meaning that they were under 50ms). Where averages returned values between zero and 50, the average value was used. It should also be noted that other aspects of the Java language could have affected these results. Although care was taken to keep constant the environments that they were taken in, other elements (e.g. the operating system ‘housekeeping’) may have affected the results.

**CONTRAST WITH EXISTING TECHNOLOGIES**

Here, the related technologies presented in the background are revisited. The functionality of the final tool is compared with theirs, and the respective strengths and weaknesses highlighted.

As EJB operates over RMI, it shares the same object reference semantics. Thus any implementation of a distributed version system would hit the same problems as the tool did using RMI. Specifically, these include the need for local (i.e. available) class definitions to refer to when creating local (typed) objects. The extra features presented by EJB, and componentisation in general, do not provide solutions to these problems, and instead affect communication and design issues not in the scope of the tool.

The common facility that Jini and EJB provide is controlled access to remote resources through remote objects. The tool developed is more concerned with the classes in general, whether loaded into the local JVM or not, and in particular their versions. Hence, the objects in question may not be subject to remote calls in the first place.

---

\(^8\) Classes used include Calendar and Date, members of the java.util package.
The ‘distributive’ aspect in the tool is thus restricted to the migration of these objects from the server to the client – either before instantiation via transfer of class files, or after, via object serialisation. As discovered, serialisation of objects of an unknown type brings about many problems, some of which cannot be solved with the current generation of virtual machines and compilers (see Chapter 4 – Component Design: Library User). So it seems that any distribution technology can only be applied to the transport of objects (instantiated or not). Since RMI is more than sufficient to handle this, the conclusion is that the use of any of the distributed technologies above would not contribute any (relevant) functionality to the tool.

Of course, the controlled libraries and applications themselves could use any of the above technologies. This use of distributed technology has nothing to do with the tool itself, and is unaffected.

With regard to the binary adaptation toolkits, the main differences between these three projects and the one developed here are:

1. They make no attempt at detecting what effect changes to a class will have on client applications
2. They are not concerned with which environment the class files are used in, specifically a distributed one.
3. They concentrate on how to fix any problems with (maybe incompatible) class files, rather than detecting and possibly avoiding these incompatibilities in the first place.

So it seems that these tools allow Application Developers (or even users) to change libraries themselves, locally. The aim is to cure any (BC or otherwise) linking problems, while the developed tool aims to prevent these problems in the first place.
There is a scope for a hybrid tool. Instead of libraries in whole, delta files could be distributed instead and applied at runtime. As the delta file is potentially smaller than the classfile, this tool would be more efficient when transferring updates. Also, delta files may be automatically generated and distributed by the tool itself to provide ‘quick fixes’ if and when BC is broken.

**Requirements Analysis**

As mentioned in the design chapter, most of the requirements were fulfilled before implementation. The requirements that were not covered by the design were listed. They are now analysed again to see if they have been covered by the implementation. Some interesting notes regarding other requirements are also listed:

P2. As all rules were added with relative ease, it is expected that further rules would also be just as easy, whether they are from [JLS] or not.

P5. Clients remain anonymous, with only the (non unique) class ID and library versions submitted to the server.

P6. All tested Java facilities worked as intended, with the exception of subclassing. As mentioned above, a restart is required for the class to inherit correctly. So the requirement holds with respect to the eventual behaviour of applications.

D3. The Rule Server can be restarted between transactions, and infer all necessary version data from the persistent information files.

D4. Libraries are developed as normal, unless they require further libraries under the control of the tool. Then a similar argument regarding D5, below, holds.

D5. No difficult change in programming is required by any Application Developer, all that is required is an import of a package, an instantiation of an object, and an invocation of a method.
D8. The running of the tool is totally transparent to the user, provided the debug option is turned off. Delays when communicating with the Rule Server are negligible.

D9. Any libraries required for the operation of the application and tool should be packaged with the application distribution. For the updating of the cache and LVS file, write permission to the disk is assumed.

D10. The synchroniser catches all exceptions thrown as a result of the tool’s operation. In the event of an error, the application will run with a guaranteed BC library present in the classcache.

D11. All class file formats are unchanged. Classes in the repository are not to be used directly by an application, and are named differently.

D12. Applications can be removed from the control of the tool by removing the call to the synchroniser. Libraries are already fit to be used outside of the tool.

D13. Normal tools complying with Java technical standards can create all relevant classes and sources to be used with the tool. In all of the tests, the normal javac compiler was used to compile all test classes and applications. It is safe to assume that an arbitrary second party class can be loaded and processed by the Rule Engine, as it uses standard Java APIs.

Hence all requirements are fulfilled to a satisfactory level.
CONCLUSIONS

To conclude, lessons learned as a result of the work done on this project are listed. A list of possible improvements and future work is presented at the end.

LESSONS LEARNED

A project of this nature presents difficulty due to the need to examine already compiled programs. This would be very difficult in other programming languages (and in Java if a different approach was made), requiring extensive ‘bit poking’ and interpretation of the binary files.

Java’s reflection API provides an abstract method of accessing this information and more. Any superclass of a class is automatically determined by the API, and can in turn be loaded and examined. The same applies for classes as return types of a method. It is safe to say this project would have been a lot more difficult, and perhaps infeasible, if this API did not exist.

The same could be said for the other standard classes used. The notion of a custom classloader is new in Java, and presents a flexibility in the running of an arbitrary program not seen in other languages. Even simple concepts like input and output streams made creating an RMI extension relatively painless.

These characteristics of the Java language (extension of supplied libraries, program examination and manipulation, ease of using remote objects) allow tools like that developed to exist. It remains to be seen how easy it would be to develop the same tool in another OO language like C++.
Other non-project specific skills were also learned. This was my first large-scale project, totally in my control. Concentrating on developing the design to fulfil the requirements made implementation of the tool less problematic and time consuming. Adding flexibility to the design allowed smaller details of the implementation be decided during the later stages. It was important to componentise and integrate the separate parts. This aided in testing, and allows the individual parts to be used in further unrelated applications.

**IMPROVEMENTS AND FUTURE WORK**

The first thing to address is the limited number of rules supported. Although the way is clear to develop further subclasses of the Rule class, they should be designed and implemented very carefully so as to not misidentify any BC libraries (or less seriously any non BC libraries). As mentioned, other rules may also be developed.

As mentioned in Chapter 2, Sun released a Java versioning specification to be included with packages. Although the tool was developed without using this specification, it may benefit from doing so. In this way less information could be kept in the SVS files, if any at all. The Library Developers would have to start packaging their classes, although it is highly likely that they already do this. This is assumes that the support for packages in general is provided.

Along with this, support for Java jar files can also be added. Packaging classes in a compressed archive would help with both storage in the repository and transfer of libraries to and from clients. Another solution would be to define a new compressed library format, to be used internally by the tool for storage and transfer, although the previous suggestion is preferred as the jar files can easily be extracted from the repository.
As the quantitative tests show, raw TCP/IP is much faster in transferring files than RMI. Although RMI should be kept when managing remote objects, there is no reason why sockets cannot be used for transfer responsibilities. This would improve the transparency of the application to the user by reducing any delays whilst updating the classcache.

Another way to make library updates more efficient is to download only files that have been changed by the Library Developer. Checksumming and date stamping could be used to determine this. Of course, to ensure integrity of libraries, a full cache renewal could occur at periodic intervals – the details of the update policy could be left to the Application Developer.

If there are a very large number of users of an application, the tool may benefit from a ‘repository hierarchy’. This would be made of master and slave
caches, each with a varying degree of freshness. Updates would filter down from masters to slaves, and mirror repositories would ease the demand on the main repository. The tool could also be instructed to distribute slightly different versions of a library to different locations (e.g. libraries with different base currencies to different geographical locations).

It seems a lot of administrative duties of an Application Developer would be resolved if the application had a way of identifying itself. In this way, Application Developers can exert control over all of their distributed applications (although not one user in particular) via the tool. They can then force a certain application to use a specified library, to clear a cache, or to no longer execute until further notice (after a later library is made available, for example). This process could even be automated, by providing the tool with detailed information about the dependency between an application and library. The tool can then decide itself what libraries would cause linking problems and should not be downloaded. This issue of inter class dependency is a very deep topic – it has a much larger scope than the safer approach (‘always assume BC is broken’) made by the tool.

Very little is known about the dynamic downloading of classes. It seems that this is an issue internal to a virtual machine, and cannot be adapted by the programming language itself. A lot of the cache management issues presented above would be resolved if the first classloader design (see Chapter 4) was feasible to implement, without the restriction of having to use interface classes. The tool would have more control over the distribution of libraries, and what to allow applications access to. Clearly more study is needed into acceptable ways to make this happen, and if a new implementation of the tool using this mechanism would work. Any results of these would have large consequences, both to the functionality of the developed tool and other technologies.
C# is a new OOP language that provides all of Java’s features (including a port of Java’s API, including reflection and custom classloaders). It is imagined that it would be a simple task to port this tool to the new language. Initial reports of the C# class specification mention that versioning is included for each class as well as for each package. Classloaders can then be instructed to load classes depending on this version number. It would be interesting to see what new avenues this facility would present to the tool.

Many of these extra facilities break the initial requirements of this project. It remains to be investigated if the cost of this is worth the extra functionality offered. Indeed, the tool would evolve beyond its original role and responsibility. What is certain is that all of them fit perfectly with the tool’s reach and level of control over libraries and applications.
BIBLIOGRAPHY

[TD] S Eisenbach, C Sadler, Y Lam, Client Speciation: The DOOM scenario. (http://www-dse.doc.ic.ac.uk/research/slurp/pubs.html#jvcs)


[B] B Cornelius, Developing Distributed Systems. (http://www.dur.ac.uk/~dcl0bjc/Java/distributed/onefile/index.htm)


[DEJT] S Roberts and J Byous, Distributed Events in Jini Technology.  
(http://developer.java.sun.com/developer/technicalArticles/jini/JiniEvents/)

(http://www.sun.com/jini/whitepapers/architecture.html)

[JTN] Sun Microsystems, Why Jini Technology Now?  
(http://www.sun.com/jini/whitepapers/whyjininow.html)

(http://www.sun.com/jini/whitepapers/technologies.html)

[JSS] Sun Microsystems, JavaSpaces Service Specification.  

[JTCP] Sun Microsystems, Jini Technology Core Platform Specification.  

(http://atlsysguild.com/GuildSite/Robs/Template.html)

[CCL] K McCrary, Create a custom Java 1.2-style ClassLoader.  

[DK] Sun Microsystems (WinHelp version by F. Allimant), The JDK 1.3 Documentation.  
(http://java.sun.com)

[JTUT] Sun Microsystems (WinHelp version by F. Allimant), The Java 2 Tutorial.  
(http://java.sun.com)

(http://courses.cs.vt.edu/~cs5204/fall99/ProjectFiles/KangYuan.zip)


## BINARY COMPATIBILITY RULES

The following tables summarise the rules as given in [JLS]. References refer to relevant subsections in [JLS].

### EVOLUTION OF CLASSES

<table>
<thead>
<tr>
<th>Rule</th>
<th>Change</th>
<th>Affected</th>
<th>Error/Notice</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>abstract</td>
<td>Non-abstract class changed to abstract.</td>
<td>Pre-existing binaries that attempt to create new instances of that class.</td>
<td>InstantiationError at link time, or InstantiationException at run time.</td>
<td>Changing abstract classes to non-abstract does not break BC.</td>
</tr>
<tr>
<td>(Rule E)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>final</td>
<td>Non-final class changed to final.</td>
<td>Subclasses of that class.</td>
<td>VerifyError thrown at compile time.</td>
<td>Changing final to non-final does not break BC.</td>
</tr>
<tr>
<td>classes (Rule F)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rule</td>
<td>Change</td>
<td>Affected</td>
<td>Error/ Notice</td>
<td>Other</td>
</tr>
<tr>
<td>----------------------</td>
<td>---------------------------------------------</td>
<td>----------------------------------------------------</td>
<td>------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>public classes</td>
<td>public class changed to non-public.</td>
<td>Pre-existing binary that needs but no longer has access to the class.</td>
<td>IllegalAccessError thrown at link time.</td>
<td>Changing non-public to public does not break BC.</td>
</tr>
<tr>
<td>Superclasses and superinterfaces</td>
<td>Class made a superclass of itself.</td>
<td>Itself.</td>
<td>ClassCircularityError thrown at load time.</td>
<td>Changing direct superclasses does not break BC, provided the superclass loses no members.</td>
</tr>
<tr>
<td>Class body and member declarations</td>
<td>Deleting a class member or constructor that is not declared private.</td>
<td>Binary using the member or constructor.</td>
<td>Linkage errors.</td>
<td>Adding an instance/static member that has same name, access, signature and return type as a member of a superclass or subclass. See 13.4.5 for further discussion.</td>
</tr>
<tr>
<td>Rule</td>
<td>Change</td>
<td>Affected</td>
<td>Error/Notice</td>
<td>Other</td>
</tr>
<tr>
<td>-------------------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>---------------------------------------</td>
<td>------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>Access to members and constructors</td>
<td>Changing the declared access of a member to permit less access.</td>
<td>Pre-existing binaries requiring access to this member.</td>
<td>Linkage errors when the binary is resolved.</td>
<td>Access order – private &lt; default &lt; protected &lt; public</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Also, changing member to be more accessible causes no linkage error when a subclass already defines a method to have less access.</td>
</tr>
<tr>
<td>Field declarations</td>
<td>Adding a field to a class.</td>
<td>Pre-existing binaries that are not recompiled.</td>
<td>Linkage errors.</td>
<td>See 13.4.7 for further discussion.</td>
</tr>
<tr>
<td>Rule</td>
<td>Change</td>
<td>Affected</td>
<td>Error/ Notice</td>
<td>Other</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>----------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>final fields</td>
<td>Non-final fields changed to final.</td>
<td>Pre-existing binaries that attempt to assign values to that field.</td>
<td>IllegalAccessError thrown at runtime.</td>
<td>Changing final to non-final or changing the initial value does not break BC. However, pre-existing binaries will have to be recompiled to see the new value. See 13.4.8 for further discussion.</td>
</tr>
<tr>
<td>static fields</td>
<td>Non-private, non-static fields changed to static.</td>
<td>Pre-existing binary expecting a non-static field.</td>
<td>IncompatibleClassChangeError thrown at link time.</td>
<td>None.</td>
</tr>
<tr>
<td>transient fields</td>
<td>Non-transient changed to transient, or vice versa.</td>
<td>None.</td>
<td>None.</td>
<td>None.</td>
</tr>
<tr>
<td>Rule</td>
<td>Change</td>
<td>Affected</td>
<td>Error/Notice</td>
<td>Other</td>
</tr>
<tr>
<td>------</td>
<td>--------</td>
<td>----------</td>
<td>--------------</td>
<td>-------</td>
</tr>
<tr>
<td>Method and Constructor Declarations</td>
<td>Adding a method to a subclass of a superclass that has a method referenced by a further subclass, where the new method is either less accessible than the old one or static (when the old one was not).</td>
<td>The lower subclass, making the reference.</td>
<td>Linkage errors.</td>
<td></td>
</tr>
<tr>
<td>Deleting a method from a class.</td>
<td>Any pre-existing binary which referenced the method.</td>
<td>NoSuchMethodError thrown at linking.</td>
<td></td>
<td>Only thrown if no method with the deleted methods signature is found in a superclass.</td>
</tr>
<tr>
<td>Creating a constructor with no parameters, with less accessibility as the class of its declaration.</td>
<td>Pre-existing binaries requesting the constructor.</td>
<td>IllegalAccessErrors thrown.</td>
<td></td>
<td>The new constructor replaces the default one provided by the java compiler. This constructor had the same access as its enclosing class.</td>
</tr>
<tr>
<td>Rule</td>
<td>Change</td>
<td>Affected</td>
<td>Error/ Notice</td>
<td>Other</td>
</tr>
<tr>
<td>------------------------------</td>
<td>------------------------------------------------------------------------</td>
<td>---------------------------------------</td>
<td>-----------------------------</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Method and constructor</td>
<td>Changing the name of a method, the type of a formal parameter to that method, or adding or deleting a parameter.</td>
<td>Pre-existing binaries requiring the method.</td>
<td>NoSuchMethodErrors thrown.</td>
<td>Same as deleting the method and creating a new one.</td>
</tr>
<tr>
<td>Method result type</td>
<td>Changing the type of a result, or changing a void to a type, or vice versa.</td>
<td>Pre-existing binaries requiring the method.</td>
<td>NoSuchMethodErrors thrown.</td>
<td>Same as deleting the method and creating a new one.</td>
</tr>
<tr>
<td>abstract methods</td>
<td>Non-abstract method changed to abstract.</td>
<td>Pre-existing binaries invoking the method.</td>
<td>AbstractMethodErrors thrown.</td>
<td>Changing an abstract method to a non-abstract one does not break BC.</td>
</tr>
<tr>
<td>final methods</td>
<td>Non-final method changed to final.</td>
<td>Existing binaries that depend on the ability to override this method.</td>
<td>VerifyError at runtime.</td>
<td>Changing a static method from non-final to final does not break BC. Removing a final modifier does not break BC.</td>
</tr>
<tr>
<td>native methods</td>
<td>Adding or deleting a native modifier.</td>
<td>None.</td>
<td>None.</td>
<td>None.</td>
</tr>
<tr>
<td>Rule</td>
<td>Change</td>
<td>Affected</td>
<td>Error/ Notice</td>
<td>Other</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>---------------------------------------------</td>
<td>---------------------------------------------</td>
<td>----------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>static methods</td>
<td>Non-private static method changed to non-static.</td>
<td>Pre-existing binaries that use this method.</td>
<td>IncompatibleClassChangeError thrown at link time.</td>
<td></td>
</tr>
<tr>
<td>synchronized methods</td>
<td>Adding or deleting a synchronized modifier.</td>
<td>None.</td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td>Method and constructor throws</td>
<td>Changes to the throw clause of methods/constructors.</td>
<td>None.</td>
<td>None.</td>
<td>These clauses are checked at compile time.</td>
</tr>
<tr>
<td>Method and constructor body</td>
<td>Changes to a body of a method/constructor.</td>
<td>None.</td>
<td>None.</td>
<td></td>
</tr>
<tr>
<td>Method and constructor overloading</td>
<td>Adding new methods or constructors that overload existing methods.</td>
<td>None.</td>
<td>None.</td>
<td>Does not break BC, as the signature of the method to be used was determined at compile time – the new methods will not be used.</td>
</tr>
</tbody>
</table>
### Method overriding

<table>
<thead>
<tr>
<th>Rule</th>
<th>Change</th>
<th>Affected</th>
<th>Error/ Notice</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method overriding</td>
<td>Class method added to a subclass, which overrides an existing method in a superclass.</td>
<td>Pre-existing binaries that call the method.</td>
<td>The pre-existing binary will not find the new method.</td>
<td>The binary will find the method if the qualifying type is of the subclass type. Also a static method that overrides a method will be found, and the binary will be unaffected.</td>
</tr>
</tbody>
</table>

**static initialisers**

- Adding, deleting or changing a static initialiser.

<table>
<thead>
<tr>
<th>static initialisers</th>
<th>None.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>None.</td>
</tr>
</tbody>
</table>

Table A1 - Evolution of Classes

### Evolution of Interfaces

<table>
<thead>
<tr>
<th>Rule</th>
<th>Change</th>
<th>Affected</th>
<th>Error/ Notice</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>public Interfaces</td>
<td>public interface changed to a non-public one.</td>
<td>Pre-existing binary that needs the interface.</td>
<td>IllegalAccessException thrown at link time.</td>
<td>Changing a non-public interface to a public one does not break BC.</td>
</tr>
<tr>
<td>Rule</td>
<td>Change</td>
<td>Affected</td>
<td>Error/ Notice</td>
<td>Other</td>
</tr>
<tr>
<td>---------------------</td>
<td>---------------------------------------------</td>
<td>-------------------------------</td>
<td>------------------------------</td>
<td>--------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Superinterfaces</td>
<td>Interface made superinterface of itself.</td>
<td>Itself.</td>
<td>CircularityError thrown.</td>
<td>Changing direct superinterfaces does not break BC, provided the superinterface loses no members.</td>
</tr>
<tr>
<td></td>
<td>To a direct superinterface, resulting in any other interface (including this one) to no longer being a superinterface.</td>
<td>Various.</td>
<td>Various link-time errors.</td>
<td>See related material in 13.4.4 “Superclasses and Superinterfaces”.</td>
</tr>
<tr>
<td>The interface members</td>
<td>Adding a field to a superinterface.</td>
<td>Any superclass used by the class implementing a superinterface.</td>
<td>IncompatibleClassChangeError if originally a static field, IllegalAccessError if originally an assignment.</td>
<td>Added methods to an interface does not break BC.</td>
</tr>
<tr>
<td></td>
<td>Deleting members from an interface.</td>
<td>Any implementing class.</td>
<td>NoSuchMethodError thrown at runtime.</td>
<td></td>
</tr>
<tr>
<td>Field declaration</td>
<td>Adding a field to an interface.</td>
<td>Pre-existing binaries that are not recompiled.</td>
<td>Linkage errors.</td>
<td>See related material in 13.4.7 “Field Declarations”.</td>
</tr>
<tr>
<td>Rule</td>
<td>Change</td>
<td>Affected</td>
<td>Error/Notice</td>
<td>Other</td>
</tr>
<tr>
<td>------------------</td>
<td>-------------------------------</td>
<td>----------------------------------------</td>
<td>-------------------------------</td>
<td>-------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Non-final fields changed to final.</td>
<td>Pre-existing binaries that attempt to assign values to that field.</td>
<td>IllegalAccessError thrown at runtime.</td>
<td>Changing final to non-final or changing the initial value does not break BC. However, pre-existing binaries will have to be recompiled to see the new value. See related material in 13.4.8 “final Fields and Constants”.</td>
</tr>
<tr>
<td>abstract Method Declarations</td>
<td>Non-final method changed to final.</td>
<td>Existing binaries that depend on the ability to override this method.</td>
<td>VerifyError at runtime.</td>
<td>Changing a static method from non-final to final does not break BC. Removing a final modifier does not break BC. See related material in 13.4.15 “final Methods”.</td>
</tr>
<tr>
<td></td>
<td>Adding or deleting a native modifier.</td>
<td>None.</td>
<td>None.</td>
<td>See related material in 13.4.16 “native Methods”.</td>
</tr>
<tr>
<td>Rule</td>
<td>Change</td>
<td>Affected</td>
<td>Error/ Notice</td>
<td>Other</td>
</tr>
<tr>
<td>------</td>
<td>------------------------------------------------------------------------</td>
<td>--------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>Class method added to a subclass, which overrides an existing method in a superclass.</td>
<td>Pre-existing binaries that call the method.</td>
<td>The pre-existing binary will not find the new method.</td>
<td>The binary will find the method if the qualifying type is of the subclass type. Also, a static method that overrides a method will be found, and the binary will be unaffected. See related material in 13.4.22 “Method Overriding”.</td>
</tr>
</tbody>
</table>

**Evolution of Packages**

A new top level class or interface may be added, provided it does not share an unrelated name, and not break BC. Changes in non-public classes and interfaces affect only types within the package in which they are declared. These can be deleted or changed, provided that affected binaries are also updated.
MAIN LITERATURE REVIEWS

CLIENT SPECIFICATION: THE DOOM SCENARIO

S. Eisenbach, C. Sadler, Y. Lam.

This document recognises the need for a tool to maintain software being developed in a distributed environment. First it introduces object orientated programming (OOP), and the concepts of classes, interfacing and inheritance. It tells us how problems arise regarding clients of altered objects that will cease to function due to these alterations. It explains how sources are currently compiled to avoid these problems. It then adds the problem of a distributed environment, and how current solutions cannot work there.

In detail, the paper describes “the fragile base class problem”, “dynamic loading and linking” mechanisms, and their formal underpinnings. On the way, it introduces an existing project called “The Java Version Control System”. It finishes with a review of related work currently going on.

THE JAVA LANGUAGE SPECIFICATION, CHAPTER 13

Sun Microsystems.

This chapter of the official Java specification explains what binary compatibility is. As well as this, it presents the main changes (evolutions) to packages, classes, and interfaces that would break BC. For the more complex rules, it gives some clear examples.
Although it is quite detailed, a summary of its rules can be found in Appendix A.

**JAVA PRODUCT VERSIONING SPECIFICATION**

Sun Microsystems.

This paper specifies how classes and resources are versioned. It talks about packages, and how they represent a consistent unit that can be developed, updated and distributed. It talks about why versioning is needed and whom it would affect.

**JAVA VERSION CONTROL SYSTEM**

Y. Lam et al.

This is a third year group project that implemented a version control system. This system detected a few (ie not all) rules that enforced BC. The main difference, however, between this and the DJVCS is that all source files belonging to a project have to be in one directory.

It constructs an algorithm to check for anything that could break BC, after a source file had been edited. It discusses the functionality and UI of the final product. It then ends with an evaluation and ideas for future work.

**DEVELOPING DISTRIBUTED SYSTEMS**

B. Cornelius.

This is lecture material from the University of Durham. It introduces the basics of client-server computing, including sockets. It follows on with more technical topics, including RMI, CORBA, JDBC and signed applets.

The whole document is very Java orientated, and all of its examples are written in this language. It explains the technology in a clear manner,
very good diagrams, tutorials, and steps. It even tells you how to get a specific example running. However it seems to only go so far as to introduce the various technologies, so it does not go on to the more advanced aspects of these technologies.

**Binary Component Adaptation**

R. Keller, U. Hölzle

This is quite a technical paper explaining what BCA is and how it can be used. BCA is a way of adapting Java binaries (in the class file format) after they have been compiled, without necessarily having access to its source. This enables users to adapt third party objects to fix bugs, conform to interfaces or anything else in order to evolve or integrate them.

We see how their BCA system intercepts a class file after a class loader has loaded it. It then uses an adaptation file (called a “delta file”) to make changes to this loaded class, before passing it to the default verifier to check that the class is still ok to use. Note that these changes could be made at load time of each class (sacrificing loading times as a result) or beforehand, by storing the adapted class file on disk (sacrificing disk space).

It describes two situations (with examples) that show where BCA could be used as a solution. The first of these is in the integration of two components with conflicting symbols. These symbols could be changed by BCA to comply with each other. The second scenario is when evolution of software is required. This problem occurs when it is required to change an interface, without knowledge of the objects implementing that interface. BCA is used to adapt the interface when extensions are required, to be added to any objects implementing them.
We are given a way of implementing a BCA system, by modifying the class file itself. The class file is dissected and then altered using instructions contained in a delta file.

The paper then ends with an evaluation of the work, a review of related work, and a conclusion.

**Automatic Program Transformation with JOIE**


Another load time transformation “engine”, the Java Object Instrumentation Environment is a toolkit for constructing transformations of Java classes. Its implementation differs, however, in that the transformations are applied at runtime, via a modified class loader, rather than a separate processing block as in BCA.

This paper raises the same issues as the previous, however it goes on to consider more wider concerns, such as the possible weaker security in modified class files, as well as any legal problems caused by modifying third party code.

**Adaptable Binary Programs**

R. Wahbe, S. Luccio, S. L. Graham.

This is a third paper covering binary adaptation. “Binary Adaptation” is the term that this paper uses to describe their method of altering post-compiled code. However, it seems like the emphasis is to create analysis code to discover the effects about the client code running. Also, it is not java specific, rather it tends to deal with native binary code. Rather than using intermediary “delta files”, the system provides operations to alter the code. The operations available fall into three categories: control, edit and register operations. These
deal with providing control flow information, editing machine instructions and using registers respectively.

**Distributed Events in Jini Technology**

S. Roberts, J. Byous.

This is a short paper introducing the Java Jini technology, what it does, how it works and where it can be used. It tells us how Jini is based on RMI, to provide secure discovery and join protocols, as well as a lookup service. As well as these, there are some leasing and transaction facilities to provide for a changing distributed environment.

A bit of detail is used to describe current event models, and how the Jini RemoteEventListener interface is used instead. It specifies this interface and what fields it has, its size, etc.

It ends quite abruptly, without really explaining how Jini is a suitable replacement for existing technologies. This means that further research may be needed on this area.

**The Jini Architecture Specification**

Sun Microsystems.

This paper expands on the above. It uses a good level of detail, suited to someone who wishes to implement or use a Jini system. It sticks to the high-level architecture of the system and defines the different components that make it up.

Before continuing to an overview of the system, it tells us the goals of Jini, and what environment it will work in. It then goes in to further detail about the components of the system, and the architecture of the system.
It finishes with an example of a Jini printing service that a device such as a digital camera may use. It steps through all the phases that the printer has to do to offer its services on the network, and what the camera must do to find and use this service.

**Why Jini Technology Now?**

*Sun Microsystems.*

This is a non-technical paper underlining the problems with today’s common computing model, and how wonderful life would be in a Jini enabled world. It is very readable – rather like a piece of marketing than a white paper.

It tells us what a typical Jini system requires, what the designers assumed about the future of Jini (most interestingly that services may not be Java objects) and what it is capable of.

It then presents us with a relatively comprehensive account of the history of computing, in terms of networks, processors, and the use of them. It does this as a lead in to what it think holds for the future – Jini.
The following flowcharts represent the rules implemented in this project. Rules A, B, E, F and G are subclasses of RuleClass, rules C, D, H, I and J are subclasses of RuleMember. This relationship is reflected in the structure of the charts.

The functionality delegated to RuleClass and RuleMember's applyRule() method has been enclosed in the dotted boxes. The rest of the algorithm's work is done in RuleClass and RuleMember, as they are shared by all of the rules.
Get a class from the old library

Is the class in the new library?

Yes  

Checked all classes?

Yes  Return true  Return false

No
Rule B

Get a class from the old library

Is the class a subclass?

Yes
  
  Is the new class a subclass?

  Yes
    
    Are the superclasses the same?

    Yes
      
      Checked all classes?

      Yes
        
        Return true
      
      No
        
        Return false
  
  No
    
    Return true

No

Checked all classes?

Yes
  
  Return true

No
Rule C

Get a class from the old library

Get a field from this class

Is this field in the new class?

Checked all fields?

Checked all classes?

Return false

No

Yes

No

Yes

Yes

Return true

No

Figure C.3 - Rule C.
Figure C 4 - Rule D
Get a class from the old library

Has the new class become abstract?

No

No

Checked all classes?

Yes

Return true

Yes

No

Return false

Figure C.5 - Rule E.
Get a class from the old library

Has the new class become final?

No

Checked all classes?

Yes

Return true

No

Return false

Figure C 6 - Rule F
Get a class from the old library

Has the new class become non-public?

No

No

Checked all classes?

Yes

Return true

Return false

No

Yes

Figure C.7 - Rule G
Get a class from the old library

Get a constructors from this class

Is this constructor in the new class?

No

Checked all constructors?

No

Checked all classes?

Yes

Return true

Yes

Return false

Yes

No
Rule I

Get a class from the old library

Get a fields from this class

Has this field got less access in the new class?

Yes

Return false

No

No

Yes

Yes

Checked all classes?

Yes

Return true

No

Checked all fields?

Yes

No

Figure C.9 - Rule I
Rule J

Get a class from the old library

Get a method from this class

Has this method got less access in the new class?

Yes

No

No

Checked all methods?

Yes

No

Checked all classes?

Yes

Return true

No

Return false

Figure C.10 - Rule J
The following OMT diagrams reflect the final design of the tool. They are grouped by package, and only important classes have been shown.
Figure D 1 - Package bcrules

D2
Figure D 2 - Package ruleserver
Figure D 4 - Package classsync

- <interface>
  RuleRemoteCLIF

- <interface>
  RuleRemoteObjIF

- <constructor> +RuleSubmit()
  topic(String), void (static)
  -ruleRef(String), String, var, float, ro: RuleRemoteObjIF, void (static)
  -main(arg, String[]), void (static)

- <constructor> +ClassSync(cid: String, cd: String, s: String, d: boolean)
  -out(s: String), void
  -syncClasses(): void
  -transferFile(name: String, var, float, ro: RuleRemoteObjIF, void)

- buffer: int = 524288 (final)
- DVSName: /string = "DvCS" (final)
- server: String
- classid: String
- cacheOf: String
- debug: boolean

- <constructor> +LocalVersionSettings()
  -currentVer: float
  -classid: String
  -descIcsip: String
  -setVersion(): String
  +getVersion(): String
  +setDescIcsip(desc): String
  +setCurrentVer(Ver): float
  +getCurrentVer(): float
  +save(): boolean
  +reset(cid: String, cid: String): boolean

D5
Appendix E

JAVADOCS

To keep the size of this report down, the JavaDocs generated from the tool source will not be reproduced here.

They are available in the project directory in the subfolder /html online at http://www.doc.ic.ac.uk/~sas97/project.

A copy is also contained on the project CD-ROM.
Appendix F

USER GUIDE

FOR TOOL ADMINISTRATORS

Make sure the network is available and there is an RMI Registry running somewhere on the network.

Run the Rule Server:

\texttt{java \ ruleserver.RuleServer \ repository \ server}

Where \texttt{repository} is the path to the library repository and \texttt{server} is the network name of the RMI registry.

Make sure the following properties are set, either by the \texttt{–D} switch when running the java program, or some other means:

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>classpath</td>
<td>Path(s) to all required packages, \texttt{bcrules}, \texttt{ruleserver} and \texttt{rmifiletransfer}.</td>
</tr>
<tr>
<td>\texttt{java.rmi.server.codebase}</td>
<td>Path to package ruleserver</td>
</tr>
<tr>
<td>\texttt{java.rmi.server.hostname}</td>
<td>Network name of RMI registry.</td>
</tr>
<tr>
<td>\texttt{java.security.policy}</td>
<td>Java security policy file. Must allow file and network access.</td>
</tr>
</tbody>
</table>

Alternatively, the runserver.bat file can be edited for any particular circumstances.
FOR LIBRARY DEVELOPERS

Make sure the network is available and there is an RMI Registry running somewhere on the network, with a registered Rule Server.

Compile your libraries as normal. Move them to a known empty directory.

Run the library submitter:

```java
java rulesubmit.RuleSubmit class_ID transferdir server
```

Where `class_ID` is the ID you wish to register the classes under, `transferdir` is the directory containing the library classes and `server` is the network name of the RMI registry. Make sure the classpath and policy file properties are set as outlined above. Alternatively, the runclient.bat file can be edited.

The following is output:

<table>
<thead>
<tr>
<th>Last published version: v1.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latest version on server: v1.8</td>
</tr>
<tr>
<td>Enter a float of the form x.x, or return to accept 1.9:</td>
</tr>
</tbody>
</table>

Figure F 1 - Submitter Output (1)

Enter a float greater or equal to that suggested, or hit return to accept the automatically generated version number. After the classes are uploaded, the following will be asked:

<table>
<thead>
<tr>
<th>Do you want to check for BC breakage?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you want to publish v2.0 as the latest class set?</td>
</tr>
</tbody>
</table>

Figure F 2 - Submitter Output (2)

Answer ‘y’ to each. Any other reply will be interpreted as no.
FOR APPLICATION DEVELOPERS AND USERS

Application Developers must instantiate a ClassSyncr object, and invoke the syncClasses() method to update the classcache:

```java
ClassSyncr cs = new ClassSyncr(cid, cd, s, d); cs.syncClasses();
```

Where `cid` is the Library ID being updated, `cd` is the path to the local classcache, `s` is the network name of the RMI Registry, and if `d` is true debug mode is activated.

Run the client application as normal. Make sure the Java policy file is set as described above.
Appendix G

DEVELOPMENT AND TESTING ENVIRONMENT

Only relevant software and hardware has been presented.

HARDWARE

Desktop PC: Intel Pentium III 450, 256Mb RAM, 20Gb IDE HD, 10/100 Ethernet.

Laptop PC: Intel Pentium III 600, 128Mb RAM, 10Gb IDE HD, 10/100 Ethernet.

Network: Single 10Mb 8-port hub, two 1m cat-5 UTP cables.

SOFTWARE

Microsoft Windows 98 SE.

JDK 1.3/JRE 1.3.

Forte For Java, Community Edition Version 1.0, Update Release 2.

Kawa Enterprise 5.0.

ServU FTP Server v3.0.

Leech FTP Client v1.3.

Microsoft Office 2000.
SmartDraw 5

Metamill v1.0
## PROJECT LOG

Reading time not included. SE = Susan Eisenbach.

<table>
<thead>
<tr>
<th>Date</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>10/11/00</td>
<td>First meeting with SE. Discussed outline of project, split it over two areas - versioning and distribution. Received last years group project JVCS. Also obtained chapter 13 of Java spec - binary compatibility.</td>
</tr>
<tr>
<td>14/11/00</td>
<td>Researched into versioning, object storage, class loading. Second meeting with SE. Continuing with research, going to concentrate on Sun’s Jini for the distributed part.</td>
</tr>
<tr>
<td>19/11/00</td>
<td>Wrote up notes on binary compatibility rules in the form of a table that lays out what changes affect what and how</td>
</tr>
<tr>
<td>24/11/00</td>
<td>Presented binary compatibility notes to SE. Also expressed my concerns over using jini technology in my project. She advised me to see James Bradley, someone who is already using it in his.</td>
</tr>
<tr>
<td>1/12/00</td>
<td>Brief meeting. Had not done much since coursework is due. SE gave me three names that may bring up more on the topic. I told her that I had spoken briefly to James.</td>
</tr>
<tr>
<td>5/12/00</td>
<td>Reported back on the two names that I had found. I could not find the third. We decided that I should carry on researching.</td>
</tr>
<tr>
<td>12/12/00</td>
<td>Just a quick meeting on how the research was going.</td>
</tr>
<tr>
<td>Date</td>
<td>Task</td>
</tr>
<tr>
<td>------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>24/12/00</td>
<td>Further research on the third person. Also deeper research into Jini and related technologies.</td>
</tr>
<tr>
<td>29,30,31/12/00</td>
<td>Began Project Outsourcing exercise.</td>
</tr>
<tr>
<td>1,2/1/01</td>
<td>Completed outsourcing report, sent to SE for verification.</td>
</tr>
<tr>
<td>4/1/01</td>
<td>Began corrections to outsourcing report.</td>
</tr>
<tr>
<td>8/1/01</td>
<td>Met with SE to discuss further improvements to my OR, implemented some of these.</td>
</tr>
<tr>
<td>9/1/01</td>
<td>Another meeting with SE on how to polish up the report. Put the finishing touches to it.</td>
</tr>
<tr>
<td>12/1/01</td>
<td>Handed in outsourcing report.</td>
</tr>
<tr>
<td>22/1/01</td>
<td>Met with SE to discuss direction of project. Decided to concurrently look at the two aspects – communications and BC breakage detection – separately.</td>
</tr>
<tr>
<td>28/1/01</td>
<td>Wrote document “Choosing a Communication Protocol” to decide which communication technology to use.</td>
</tr>
<tr>
<td>29/1/01</td>
<td>Meeting with SE. Told her my decision to use RMI, she seemed ok with that. I was advised to look into custom classloaders as well.</td>
</tr>
<tr>
<td>1/2/01</td>
<td>Created a little toy RMI program. At the moment, it outputs something written at the client’s side onto the server output. I am confident that it works ok, so I will use it for the project.</td>
</tr>
<tr>
<td>2/2/01</td>
<td>Created a custom classloader. At the moment, does not do anything special, apart from looking in a specific directory for unknown classes. Adapted it to write a message to the server via RMI, when loading specific classes. The whole thing seems to be quite stable.</td>
</tr>
<tr>
<td>5/2/01</td>
<td>Meeting with SE. Discussed current progress, and decided on certain design issues.</td>
</tr>
<tr>
<td>7/2/01</td>
<td>Started experimenting with BC rules. Decided to uses a rule interface to define all of the rules that I will have. It should make the addition of rules easier. Created test classes and a harness to check if rules will work ok. The custom class loader came in handy as that is used to load the different versions of the test classes.</td>
</tr>
<tr>
<td>8/2/01</td>
<td>Finished rule C as defined in the JVCS group project.</td>
</tr>
<tr>
<td>9/2/01</td>
<td>Replaced “rule” interface with abstract class, moved code into the abstract class, rules now act on groups of classes. See design decisions for more info.</td>
</tr>
<tr>
<td>10/2/01</td>
<td>Started implementing rule B. having trouble due to the previous projects interpretation of the rule.</td>
</tr>
<tr>
<td>Date</td>
<td>Task</td>
</tr>
<tr>
<td>----------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>11/2/01</td>
<td>Completed rule B, although not too happy with it. Completed rule D, as a combination of rule D and E.</td>
</tr>
<tr>
<td>12/2/01</td>
<td>Meeting with SE. Decided to carry on with the rule implementations.</td>
</tr>
<tr>
<td>14/2/01</td>
<td>Implemented rules: A, E, F, G, H, I.</td>
</tr>
<tr>
<td>15/2/01</td>
<td>Implemented rule J. Also changed rule class hierarchy. Added diagnose function. to-do: push rule methods to its subclasses. Create “rule engine”.</td>
</tr>
<tr>
<td>19/2/01</td>
<td>Meeting with SE. Decided who my second marker was, SCD.</td>
</tr>
<tr>
<td>15/2/01</td>
<td>Implemented rule J. Also changed rule class hierarchy. Added diagnose function. to-do: push rule methods to its subclasses. Create “rule engine”.</td>
</tr>
<tr>
<td>25/2/01</td>
<td>Meeting with SE. Decided who my second marker was, SCD.</td>
</tr>
<tr>
<td>26/2/01</td>
<td>Implemented rules: A, E, F, G, H, I.</td>
</tr>
<tr>
<td>27/2/01</td>
<td>Meeting with SE. Demoed rule engine, and decided to progress into the other aspects of the project, although I still have to fix rule B.</td>
</tr>
<tr>
<td>8/3/01</td>
<td>Finished off rule engine. Started on a server-client pair for developers to submit their classes, managed to get file transfers working OK.</td>
</tr>
<tr>
<td>21/5/01</td>
<td>Meeting with SE. Demoed rule engine, and decided to progress into the other aspects of the project, although I still have to fix rule B.</td>
</tr>
<tr>
<td>22/5/01</td>
<td>Project review with second marker, SCD. She raised a lot of issues, some were trivial to handle, whilst others may require further thought.</td>
</tr>
<tr>
<td>24/5/01</td>
<td>Finished rule B. Found that it had already been covered by the Java compiler and rule D. Changed rule B to detect a change in superclass. Also elaborated on the file transfer mechanism – the submitter now submits all class files in the current directory to a location on the server, a directory defined by the project’s name and version.</td>
</tr>
<tr>
<td>21/5/01</td>
<td>Spent most of today getting back into project. Fixed file transfer – now actually transfers files! Also developed server file organisation.</td>
</tr>
<tr>
<td>22/5/01</td>
<td>Fixed the binary trouble that I was having with file transfer. Binaries are now maintained after transfer. Also added code to the server to load classes to be compared. Need to test.</td>
</tr>
<tr>
<td>23/5/01</td>
<td>Server is now ready to archive class files. It can also be instructed to check a specific version set with the latest classfiles. Need to provide a method to set one version of classfiles as the “latest” version.</td>
</tr>
<tr>
<td>24/5/01</td>
<td>Created “rulesettings” class. Started work on remoteclassloader.</td>
</tr>
<tr>
<td>25/5/01</td>
<td>Adapted server and libclient to solve problems outlined in design decisions. Started work on appclient.</td>
</tr>
<tr>
<td>26/5/01</td>
<td>Further work on appclient. No milestones broken.</td>
</tr>
<tr>
<td>Date</td>
<td>Task</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>27/5/01</td>
<td>Started new “classcache” implementation of appclient. Completed remoteinputstream.</td>
</tr>
<tr>
<td>30/5/01</td>
<td>Started on version control logic. Spent most of the day doing “theoretical” stuff. Decided and started to implement a final design.</td>
</tr>
<tr>
<td>31/5/01</td>
<td>Completed application client using adjusted logic. Started testing the application.</td>
</tr>
<tr>
<td>1/6/01</td>
<td>Demoed current functionality to SE. Seems to be ok.</td>
</tr>
<tr>
<td>2/6/01</td>
<td>Began “the great code tidy”. Consolidated redundant code, removed all hard coded variables. Added new functionality for appdevs – they can now see what changes they need to make to use the latest version of a library.</td>
</tr>
<tr>
<td>3/6/01</td>
<td>Continued from above. Specifically, tidied up output to screen, extended use of exceptions for error handling. Also “packaged” relevant classes together for preparation of Javadocs.</td>
</tr>
<tr>
<td>4/6/01</td>
<td>Completed packaging structure. Began actual Javadoccing.</td>
</tr>
<tr>
<td>5/6/01</td>
<td>Completed Javadoc comments.</td>
</tr>
<tr>
<td>6/6/01</td>
<td>Contrasted my solution with Javabeans and Jini. Completed proof of BC algorithm.</td>
</tr>
<tr>
<td>7/6/01</td>
<td>Started bulk of report. Half way through background.</td>
</tr>
<tr>
<td>8/6/01</td>
<td>Almost finished background.</td>
</tr>
<tr>
<td>9/6/01</td>
<td>Finished background. Completed requirements.</td>
</tr>
<tr>
<td>10/6/01</td>
<td>Started design. Notes made in design decisions came in handy.</td>
</tr>
<tr>
<td>11/6/01</td>
<td>Completed design, started implementation.</td>
</tr>
<tr>
<td>12/6/01</td>
<td>Completed implementation, started evaluation.</td>
</tr>
<tr>
<td>13/6/01</td>
<td>Completed evaluation, started conclusion.</td>
</tr>
<tr>
<td>14/6/01</td>
<td>Completed conclusion, started appendices.</td>
</tr>
<tr>
<td>15/6/01</td>
<td>Polishing up – diagrams etc.</td>
</tr>
<tr>
<td>16/6/01</td>
<td>Completed appendices, touched up, created some diagrams.</td>
</tr>
<tr>
<td>17/6/01</td>
<td>Completed diagrams.</td>
</tr>
<tr>
<td>18/6/01</td>
<td>Finalising report – proof reading, formatting, etc. Last log entry before report hand-in on 20/6/01.</td>
</tr>
</tbody>
</table>

Table H 1 - Project Log