Development of Tooling Support for the Testing of Java Programs

Rebecca Simmonds

September 2011
MSc Internet Technologies and Enterprise Computing

Industrial Supervisors: Mr Andrew Dinn and Mr Jonathan Halliday
Academic Supervisor: Professor Santosh Shrivastava

Abstract

Testing Java programs can be a problematic task. Developers commonly use additional test classes, these can produce errors or camouflage existing ones. The testing tool considered in this paper is Byteman. It addresses testing through the use of advice injection. Advice injection is used to alter code at runtime without modifying the original source code. It provides a mechanism of injecting code into Java programs. The injected code may then be used to trace code execution or alter the behaviour of a program. The tool also uses fault injection, this injects erroneous input into the program to test fault-handling capability. This paper outlines a tool, which provides the integration of the Byteman tooling with an integrated development environment. Prior to this project Byteman provided users with no IDE support. This increased the complexity of Byteman and offered users little support for development. A structured editor was developed for the manipulation of Byteman's script language. The editor recognises the Byteman rule scripts, as well as providing additional features to help simplify and quicken their development. Byteman provides command scripts which are executed in the command line. The tool provides integration of the command scripts with the IDE, removing the complexity of terminal execution. The new tooling provides support or new developers and practical tooling that current user have been without. It decreases the learning curev and provides simpler and quicker development in a popular IDE.

Acknowledgements

I would like to thank my supervisors Mr Andrew Dinn, Mr Jonathan Halliday and Professor Santosh Shrivastava for for their advice, support and assistance throughout the year. I would also like to thank the family and friends who have helped me throughout this year.

Declaration

I declare that the work within this dissertation represents my own work unless otherwise stated. The word count for this dissertation is 17,779 words.

Signed …………………………………….. Date………………………………………

Table of Figures

Table of Contents

1 Introduction

The aim of this project is to provide tooling support for the testing of Java programs. The mechanism the project will focus on is Byteman. Byteman is a tool, which makes it much easier for developers to write unit, integration and system tests during application development, to monitor trace and debug deployed applications.

The project will improve the usability of Byteman by integrating the rule language with a Java integrated development environment. A plug-in will be developed to provide the functionality in the IDE. The project will research how Byteman is used and identify usability problems. An analysis of these limitations will assist in defining how Byteman can be used within an IDE. A design will be derived from this research resulting in the development of a prototype.

1.1 Overview of Byteman

Commonly, programs are tested using custom coded unit, integration and system tests. Another frequent means of testing is through the use of a debugger, in which the user can add breakpoints to analyse certain aspects of the program. A more specialised technique of testing Java programs is JUnit testing. This is a testing framework that can be used to assert validation of methods. These can lead to tests becoming over-complicated or camouflaging erroneous results. These testing methods also provide no method of installing a test into a live program, without stopping it. Within this context a live program is a Java program already executing in the JVM. This is a disadvantage, as some live programs cannot be stopped without causing major problems and disruptions.

Byteman addresses testing through the use of advice injection. This provides the developer with a mechanism for altering code at runtime, while preserving the original source code. Injected code may simply be used to trace execution, may change the control flow or break the application in order to validate fault-handling capabilities [1].

Byteman employs a rule language to organise Java code fragments, which are to be injected into the program being tested. Advice injection provides accurate, precision testing by reusing and testing the actual code within the system. The organisation of these rules uses an event, condition and action structure [1]. This allows for the developer to control and analyse sections of a Java program and its behaviour.

Figure one displays the fundamental structure of Byteman rules, demonstrating the components that constitute a rule. The event section is the trigger point specifying the point where the code should be injected. The injection code includes a condition specified in the `IF` clause. The code in the action, following the `DO` keyword, is only executed when this condition evaluates to true.

Byteman operates as an agent program executing in the same JVM as the
user's application. The agent can be installed from the Java command line when the application is run, or after the application has started using a command script. The agent loads rules from a script file and injects them into the specified classes. Scripts can be identified on the Java command line or they may be uploaded and unloaded later so long as the application keeps running. This makes Byteman ideal for monitoring and debugging long-running applications such as internet services [2].

```
RULE <NAME OF THE RULE>
CLASS <NAME OF THE JAVA CLASS (EVENT)>
METHOD <NAME OF THE TRIGGER METHOD (EVENT)>
IF <CONDITION>
DO <ACTION>
ENDRULE
```

Figure 1: Example structure of a basic rule.

Byteman is operated either by passing arguments to the command line or
by executing shell command scripts. There is a command script to install the agent into a live program, another to upload or unload rules and one which queries the injection state of the agent, displaying which of the loaded rules have been injected and if any errors occurred during injection. A separate script can be used to parse and typecheck rules offline. All of these commands operate in a terminal and generate text output [1].

1.2 Limitations of Byteman

It has been identified that Byteman has certain limitations. These are restrictions within the Byteman development cycle. This cycle begins with the editing and generation of the script. This then leads on to the detection and correction of the rules. This will then conclude with the deployment of the rule into a Java program.

A high level concern for Byteman is the absence of IDE support. Currently Byteman provides no support for development within an IDE. The demographic for Byteman is mainly Java users. They are accustomed to exceptionally strong tool support through the use of an IDE. This results in Byteman being more complex to use.

Currently, Byteman rules must be edited in a plain text editor which does not recognise the structure or meaning of the rule text. Most Java developers expect their code editors to automatically generate content, highlight keywords, auto-complete symbol names and notify syntactic and semantic errors in their code. All of this is standard to developers who use IDEs. This provides a huge disadvantage to Byteman, and emphasises the complexity in using the tool.

A similar problem arises with the Byteman command scripts. Error messages are displayed in the command line and are not meaningful, providing little support for debugging. The error messages simply specify a line number for the error. The script also offers no method of identifying syntax errors. This can be confusing when using multiple rules, and time consuming identifying the line number for each error. The script is executed in the command line resulting in the user alternating between this and the editor. This results in the user detecting and correcting the error in the editor, then executing the script in the command line to identity if the correction was valid. This process must be completed until the error is removed. This is another time consuming and complex process when debugging. The parser is also used for runtime error checking with the same limitations apparent there.

Byteman is developed around several scripts, which necessitate the use of a command line interface for execution. This results in increased complexity when loading and unloading. This is because the developer must manually execute each script separately, remembering the different commands for each. This can be time consuming and commands easily forgotten.

Output about the agent's state is only available through the command line. This output includes all current information about injection state (all rules installed). This makes it difficult to identify relevant information, as it is unorganised and can wrap around lines in the interface.

Byteman rules reference Java classes and methods and the injected code
may even reference method parameters and local variables by name. Currently the rule and class must be viewed in separate editors. A consequence of this is more error prone code, which is because the developer must alternate between both. This can be a tasking responsibility when there are multiple scripts and classes. The developer must then validate the correct class, variables, methods and behaviour is specified in the rule script. This involves identifying the class being tested manually (from possibly thousands of classes) then recognising the specific element you are validating. There is no computerised aid for this task and the developers must rely exclusively on their own ability.

These limitations make Byteman a less appealing development tool and could therefore decrease the uptake of Byteman. The motivation for this project is to provide support for Byteman and make these limitations obsolete, generating a more attractive, quicker and simpler tool.

1.3 Problem Solution

This project's main aim is to remove the previously stated limitations. The strategy proposed to solve this problem is to integrate Byteman within an IDE. This will result in modification of a current IDE, by adding a plug-in, to include appropriate features. The features must offer support for development. The project must provide automation, simplicity and increased speed when developing Byteman rules.

The solution should offer assistance that will enable a more moderate learning curve for new users and an easier form of development for existing ones. The final project should enable Byteman with functions that currently do not exist. To provide these capabilities a design of corrective features is necessary.

The plug-in should be easily deployable and portable. This will increase its use and flexibility for users. It must also consider cross platform use, including a variety of different operating systems. As the system is kept within one IDE it will provide both languages within one application. This will remove the need to alternate between multiple applications.

The initial capability to be provided will be a structured editor. The editor will create a structured mechanism for simple and efficient development. It should recognise and customise `.btm` files. It will provide simpler error detection and correction. It will also offer a relationship between the Byteman rule and the Java class. These features remove limitations such as the absence of IDE support and the missing structured editor. It also gets rid of the disassociation of Java and Byteman, therefore making alternating between class and rule easier.

The IDE must replace the Byteman command scripts with simple dialogues
allowing the user to install the agent into a running program, upload or unload rule scripts and check the injection state of loaded rules. The command outputs can be presented in a more organised format and filtered, where necessary, to enable specific information to be located more easily. These capabilities will support the elimination of previously stated limitations, make Byteman easier to use and result in a more effective and less complex testing tool.

1.4 Aims and Objectives

The objectives have been organised taking into account that some of the features are more advanced and necessitate all prerequisite objectives to be completed beforehand. Low-level internal components will be addressed early in the project, to allow the most possible time for these technically challenging pre-requisites. Integration with the IDE's GUI will follow with features of increasing complexity added as time permits.

Aim:

The aim is to design and implement an IDE plug-in to provide tooling support for the generation and modification of Byteman rules. This will result in the automation of Byteman rule generation and provide features to aid in execution and correction of these rules.

Objectives:

1. To carry out research to identify an appropriate IDE.
    - This should include their plug-in creation capabilities, support of different languages and available software support. Advantages and disadvantages of each will be demonstrated.
2. To research Byteman, available software support and features for the plug-in.
    - This includes research into the identification of current features to provide design concepts for the system.
3. The design of the low level components.
    - These elements need to be completed to advance onto other features. This includes elements which are the foundation to the editor such as the grammar and parser.
4. To design features within the editor.
    - Design of the features to enable suitable development support in the editor.
5. To create a Grammar and Parser.
    - Development of a grammar to describe rules for the language. This will facilitate the generation of a parser.
6. To provide structured Editing of Byteman Rules.

- A back-end parse tree needs to be created to treat the code as a rule and not plain text, with a grammar generated to identify the rule parts. The grammar must include Byteman and Java rules.
- Syntax and error highlighting must be provided for rules.

7. To provide additional features of the structured editor (advanced).

   - Code completion for Byteman and Java.
   - Error markers and specific custom error messages.
   - Byteman template generation.
   - Individual GUI components presenting specific Byteman information
     displays and controls.
   - A GUI component which groups together all Byteman GUI components.

8. To create a relationship between Byteman rules and the referenced Java code (advanced).

   - This should enable the user to select a class or methods specified in the rule and then open the corresponding or containment class.

9. To implement the installation of the Byteman agent and submission of rules into a live program (advanced).

   - Facilitate the automation of installing the Byteman agent into a live Java program.
   - The previous point will allow rule loading and unloading into a live program.
   - Enable display of injection state.
   - Provide filtering of displayed state per rule or rule script.

10. To provide the implementation of a debugger (advanced).

    - Checking the code for errors by appending break points for the user to halt the code and step through it.

1.5 Dissertation Structure

The remainder of the document is structured as follows: Section two contains research into Byteman, Byteman command scripts and development options. This will analyse these areas and draw conclusions on the progress of the project. The requirements of the plug-in are included in section three. The design decisions made for the development of the plug-in are provided in section four. Section five outlines the implementation of the system, providing screen shots and a description. The implementation was then tested using an appropriate testing strategy. These tests are presented in a results section, which displays whether tests were passed and how each section of the strategy concluded. The system is then evaluated and a conclusion drawn.

1 Background Research

Information needed to be identified to provide a suitable solution. This is a new field of research so identifying the correct scope and components was essential. The sources used mainly focused on Byteman, other languages' support systems and development of the final system.

2.1 Byteman

Byteman's development cycle consists of editing, validation and deployment. This section will identify this in more detail. The initial stage in the development of a Byteman rule is writing a script. To enable this, syntax of the rule language must be understood. Rules consist of an event, condition and action. The event specifies a class or interface, a method and a location in the method where advice is to be injected. The condition and action provide the injected code. The injected rule is triggered when control reaches the trigger point, the location identified in the event. The code specified in the condition is executed. If it evaluates to true, the code in the action is executed, otherwise it is skipped [1, 2].

An example rule is demonstrated within figure two, which is taken from an application server support case. This rule is used to test if the `commit()` method in a transaction is active. The `CLASS` and `METHOD` section of the rules specify the trigger point in which the rule will be injected. The `CLASS` specifies no package name, therefore any classes with this name and method will execute the advice. The location for this rule is `AT ENTRY` which means this rule is triggered when the method is entered. This rule uses a `BIND` clause to introduce and initialise a Byteman rule variable, status, which is an integer. `$0` is a parameter variable referring to the `TransactionImple` object which is the target of the `commit()` call. The expression used as the initialiser calls the method `Transaction.getStatus()`, which returns the current status of the transaction. The variable status can now be used within the condition and action. The `IF` condition evaluates whether the status is active, if not then the condition is true. The condition ensures the rule only fires when the transaction is not active. The `traceStack` call in the action prints a message string and at most 15 stackframes, showing who committed the inactive transaction [2].

```
RULE trace inactive transaction at commit
CLASS TransactionImple
METHOD commit()
AT ENTRY
BIND status : int = $0.getStatus()
IF status != javax.transaction.Status.STATUS_ACTIVE
DO traceStack("inactive commit" + $this +
" status= " + status, 15)
ENDRULE
```

Figure 2: Sample rule taken from an application support case [2].

The Byteman agent can be installed and rules submitted as the Java program itself is started. This is achieved with an additional argument appended to the Java start up command. The argument that is used to specify this is demonstrated in figure three. This configures the JVM to load the Byteman agent and one or more rule script files. The listener option can be used to open the agent listener allowing the injected rule state to be queried and rules to be unloaded and reloaded while the program is running [1].

```
JAVA_OPTS=”javaagent:<location of byteman.jar>=script:
<script1>,...,script:<scriptN>,listener:true”
```

Figure 3: The argument to install the Byteman agent and submit scripts.

Injected code can halt an application at a specified point, monitor performance, and identify the behaviour at different sections of a program. It is presently widely utilized within the JBoss community for unit, integration and system testing [2]. Byteman provides the tracing of execution of specific code paths and displaying of the application’s state. It also provides fault injection, which allows the user to test the system when erroneous input is provided. This is essential when testing a system, to allow for user faults or malfunctions. Byteman makes it easy to monitor and measure system performance, providing the information required to tune an application.

2.2 Command Scripts

Byteman provides command scripts to aid deployment and rule validation. Inputs are supplied as command line arguments and the output is plain text.

2.2.1 bmcheck

The first of these scripts is *bmcheck*. This provides offline parsing and type-checking. The script uses the Byteman *JAR*, which contains a class that exposes a rule parser and type checker [1]. The command that is used to execute the offline parsing is demonstrated in figure four.

```
sh bmcheck.sh [-cp classpath] [-p package] * script1 […
scriptN]
```

Figure 4: The command for executing parsing.

The output from this command displays whether there is a parse or type error. It also provides the user with ambiguous error output and only the line number of the error.

Online parsing and type-checking is performed by the agent when it tries to inject rules into loaded classes. Injection is not performed if parse or type errors arise. Errors may also be viewed using the *bmsubmit* command (see Section 2.2.3).

2.2.2 bminstall

Support for the installation of the Byteman agent into live Java programs is provided by *bminstall*. The script will communicate with the JVM to provide installation of the Byteman agent into the program already executing in the JVM. The script is installed through the command line, the first step of this is to open the correct directory of the script. The developer must then enter the correct command shown in figure five. The developer must identify the process into which the agent will be installed, either by its process ID, or by the name of its main Java class. The command will only provide output if the installation is not successful displaying the error.

```
sh bminstall.sh <process id or program name>
```

Figure 5: Command for the live installation of the Byteman agent.

The script requires alteration for different operating systems, e.g. Mac OS X. This is because the engine uses VM-Attach (a tool within the JVM), this is in different locations in different operating systems due to the JDK used.

2.2.3 Bmsubmit

This is a script used to provide the ability to load and unload rules into a live Java program at any point during its execution. The agent's listener is used to listen for rules being submitted. Therefore a listener must be opened for the submit script to work. *bminstall* opens the listener automatically. If the agent is installed using the `javaagent` argument `listener:true` must be provided. The listener also provides injection state output, detailing the status of all currently submitted rules. This state can be listed by running *bmsubmit* with no arguments. To execute the submission script the user must navigate to the directory it is contained in through the command line. The user must then execute the command demonstrated in figure six. They must know the absolute path of the script containing the rule.

```
sh bmsubmit.sh <script1 , … , scriptN>
```

Figure 6: Command for the submission of Byteman rules.

2.3 Development Research

Research into development is necessary to provide a suitable solution. This will provide support and assistance for the design and implementation of the prototype.

2.3.1 Why a Plug-in?

To create support for a language there are different concerns, which must be considered when deciding on the method of implementation. There are two methods available for the creation of Byteman support.

The first method is creating a new IDE which would have the advantage of being specific to Byteman. However to edit Byteman and Java both IDEs would have to be loaded. This would not provide a solution to the limitation of no relationship between the respective tooling. The new IDE would have to provide support for Java, which is not within the scope of the project. To summarise, generating an IDE would lead to a lot of complex development and disadvantages.

The second method is a plug-in, which would be appended to a current IDE. Many users develop in multiple languages using one IDE. A plug-in would allow different language support within one application. This is advantageous for developers, enabling simpler, easier development. The only disadvantage to this method is that its availability would be limited to a singular IDE. However some plug-ins can be converted to be used within more than one [3].

The most suitable method is to create a plug-in, due to the flexibility and simplicity it will provide the user. To increase the popularity and usability a plug-in is desirable. Its structure will allow a portable system, which can be deployed easily.

2.3.2 Which IDE?

The two IDEs compared were Eclipse and IntelliJ IDEA. This choice was based on the availability and popularity of the two IDEs amongst developers, particularly JBoss developers.

2.3.2.1 Eclipse

Eclipse's architecture is composed of plug-ins. Figure seven displays this and the ease with which plug-ins merge with the IDE [4]. Eclipse is designed for the creation and addition of plug-ins providing an appropriate platform for this project [4].

As figure seven also demonstrates, there is a plug-in development environment (PDE) provided to assist plug-in developers. It provides an aid in creating plug-in projects, permitting variants on the type including standard and RCP [5].

A tooling feature for the generation of projects is the plug-in wizard. Once the wizard is completed a PDE perspective is loaded. This provides a variety of useful features. There is a modification GUI which includes the `plug-in.xml`, *extensions*, *extension points* and *dependencies*. For a plug-in to connect to all the different components correctly, a `plug-in.xml` file must be constantly updated. Two important features of this are the *dependencies* and *extensions*. *Extensions* and *extensions points* modification GUI provides simpler generation of *commands*, *menus* and other GUI elements [6]. *Commands* use a handler to execute the specified code to provide certain behaviour. Eclipse enables automation of the construction of *commands*, providing easier development. It also produces the correct code within the `plugin.xml` to reflect the newly created *command* [6]. *Dependencies* can be added and removed though the use of the modification GUI. They provide packages and classes for additional functionality n a plug-in.

Additionally PDE assists with testing of the plug-in before the deployment of the final product. Plug-ins are also available to help with the automation of creating and testing of plug-ins using PDE. “Plug-in Builder” is an example of this [7]. This provides support for manufacturing the plug-in as well as deploying the plug-in locally for testing [8, 9].

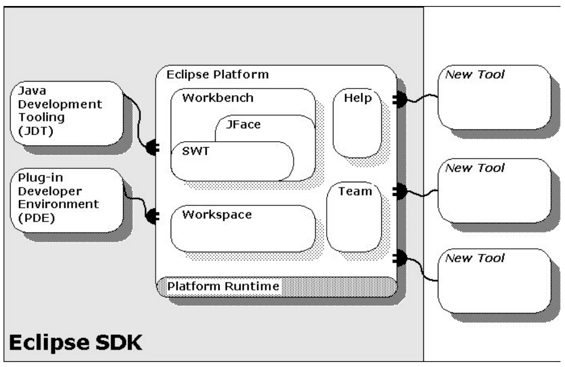


Figure 7: Eclipse SDK [4].

Eclipse also specifies explicit documentation through an API, including a variety of classes and methods. These are all specific for plug-in development [4]. The APIs are clearly defined allowing simple navigation and identification of information. These would be essential for the project providing the developer with mechanisms to implement the functionality of the plug-in. Additionally there are a variety of different tutorials available to assist developers, offering differing levels of difficulty [9]. Eclipse also has a vast amount of forums, which enables a developer to identify solutions [10]. Eclipse supplies a class with methods for specifying the content type of the new language, this is: “`org.eclipse.core.contenttype.contentTypes`” [4]. This would supply support for the Byteman type, and help with implementation of the plug-in.

Another available application type for development of professional and stylish aesthetics for the plug-in is Rich Client Platform applications (RCP). This is an adaptation, which utilizes new and improved tooling techniques. It provides simpler methods for creating and modifying menus, toolbars and other GUI elements. [5]. There are three main parts to an Eclipse RCP application:

- *Workbench adviser* is an invisible component that labours in the background and configures the appearance of the plug-in.
- *Main program* is an RCP application, which is the equivalent of a main method.
- *Perspective* is the layout of the program.

Eclipse represents each resource (files, folders and packages) as an `IResource`, then *markers* can be used to represent information about these resources. The *marker* feature of an Eclipse plug-in allows the information to be observed using different views, e.g. *Problem* view for errors [11].

Once the plug-in is developed it will require deployment. Eclipse provides a simple and quick way of deploying plug-ins. The project needs only be exported as a *JAR* file then added to the plug-ins folder in the Eclipse application folder.

2.3.2.2 IntelliJ IDEA

IntelliJ IDEA is a code-centric IDE, which encourages developer productivity and focuses on understanding and improvement of code. It allows for considerable language recognition, for example distinguishing the difference between HTML and the JavaScript within one script [12].

IDEA has an expansive range of tutorials and information available to aid with plug-in development. IDEA has a variety of libraries available to assist with the development of different language plug-ins. IDEA also supplies classes, which allow a developer to specify the type of the new language, as the language will be recognised using the extension that is attached to it. This can then be used to point to the parser. The package for extension definition is: "`com.intellij.fileTypeFactory`" [12] [13]. IDEA modifies the `plugin.xml` for the developer efficiently, allowing for the developer to concentrate on other aspects of the plug-in design.

There is however no platform for building RCP applications in IDEA, therefore limiting the aesthetics of the GUI. IDEA does however allow RCP applications built in other IDEs to be imported and modified [3]. IDEA uses *actions* instead of Eclipse's *commands and* these have a similar functionality. They are easily added to the program using a menu option. This modification is then reflected automatically within the `plugin.xml` [14].

The debugging feature allows the developer to debug the program without having to leave the IDE. This provides speed, simplicity and convenience for the developer [12].

GUI implementation is easy and simple with IDEA. The GUI development view provides the ability to drag and drop the buttons, text boxes, and other elements to the correct position on the GUI.

2.3.2.3 Analysis

Eclipse supports a variety of mechanisms to support custom language plug-in development. There is also a comprehensive selection of features for development of editors, menus, toolbar items, views and *perspectives*. There are also numerous and varying tutorials and examples. These supply the user with an extensive reference and support for developing a plug-in [15]. Debugging code is an important and practical feature within an editor and Eclipse provides support for this [16].

IntelliJ IDEA also provides numerous features to support and benefit the creation of a plug-in. It has specific assistance for development of custom language plug-ins. Features which are supported by the IDE include: syntax highlighting, quick definition look up, code completion, refactoring and other examples similar to the available features of Eclipse [12] [14]. IntelliJ also provides tutorials and videos to help the user with development, these demonstrate the features and software provided [13].

2.3.2.4 Conclusion

Both IDEs provide similar development support for implementing plug-ins, providing an extensive and varying amount of options. These features will be valuable and efficient when implementing the plug-in. The more suitable IDE available for the development of this plug-in was Eclipse. The foremost reason for using Eclipse was its popularity within the target users. Another reason was that the developer had previously used Eclipse and was familiar with its mechanics. Eclipse also supplies an extensive amount of tutorials to assist the developer with the implementation of the plug-in. This ranges from getting started to more intricate development factors. There was also a sizeable open source API available for the libraries, classes and methods, which support plug-in applications. Eclipse also provides a development environment made specifically for plug-in implementation [8].

2.3.3 Software Support

Research into plug-in development software revealed a variety of software available. This resulted in the research and analysis of three different applications: Certiv Analytics, Xtext and DLTK. DLTK is Eclipse’s plug-in for plug-in creation. It accommodates a flexible wizard utilized to generate plug-in projects with additional features for development. This software doesn’t have the simplicity of Xtext which is a major disadvantage [17] [18]. Certiv Analytics is a DSL editor, which is the intermediate of Xtext and DLTK. It uses ANTLR grammar like Xtext and develops by extending a few key classes [19]. The disadvantage of this application and the reason for its dismissal is its limited code generation capability when compared to Xtext.

Xtext was manufactured by Eclipse and provides effortless creation of new languages and their extensions. Xtext generates projects for plug-ins as well as additional features to improve development. Xtext provides ANTLR tooling to define a formal grammar and this can then be utilized to manufacture a parser [20]. It provides features to implement content assist, validation, formatting, highlighting and UI features. Xtext also allows combined use of the PDE, as well as Xtend, Check and Oaw architecture [21].

Xtext supplies ANTLR tooling which allows cross-referencing as well as terminal and enumerated rule types. It supports linking and global and local scoping. Other features available for development are syntax and error highlighting, content assist and validation. Utilization of the software is easier as it provides users with extensive documentation of features [20].

Xtext is used professionally for domain specific and general-purpose languages. It is also easily downloaded into eclipse.

2.3.4 Features for the Plug-in

Although there is no current research or technology available for Byteman IDE support, such mechanisms are accessible for other languages. Analysing the other language assistants will identify the components necessary to construct a viable solution to the problem (section 1.2). This will provide insight into possible features, display aesthetics and automation characteristics for the system.

A prominent feature implemented throughout the investigated material was an editor. This feature, often a standard text editor, maintained specialist characteristics to enhance the ability to develop the script. The editors had the capability to recognise the language and its file extension, displaying the text differently for each. Syntax and error highlighting was one of the noticeable components within these editors, demonstrating the structure of the language as well as emphasising keywords. Another obvious feature was the use of error markers, which demonstrate the location of the error. Additionally these provide error messages specifying inaccuracies in the code. These error detection methods notify the user of problems. Frequently the editor had the ability to assist the user with immediate and automated code completion. This supplemented the user with a variety of options they could choose from to complete code elements. All of these features are absent within the development of Byteman.

Perspectives were an Eclipse component used to contain all features specific to a tool. They also added views and menu options. The prominent views were an *Outline*, *Problems*, *Javadocs*, *Navigation* and a *Console*. The *Outline* demonstrates the different components of a file, emphasising the structure of the code. *Problems* is a view, which provides a view of errors in the workspace, specifying their location. *Javadocs* provides documentation for the code in a specific view. *Navigation* provides the ability to navigate through projects and files within the workspace. The *Console* view is used to display output from programs representing a collection of the output in a single location [22-24]. Again these would be completely new features to Byteman.

Debugging is an important feature for aiding developers in generating valid code. The debugger is used to identify errors with increased speed and ease, enabling the user to step through code as well as halting the program at specified points for inspection. A variety of views are available which display information such as the line number of the code being edited and the variables that have been initialized, with their values. This is described later in more detail [16].

2.3.4.1 Grammars and Parsers

A grammar is a formal definition of a language. A programming language grammar is similar to English grammar. It specifies the rules that provide the language with meaning. Just as with English grammar there are rules for defining sentences. Similarly the grammar for Byteman needs a rule to specify what defines an expression. This grammar can then be used to generate a parser. A parser is used to analyse text, therefore the grammar is used to specify the rules of this analysis. The grammar specifies how the text will be divided and analysed. For the analysis, the rule will specify patterns of the language and when the input does not match this it will produce parser errors.

The Byteman agent employs JFlex and Cup[25] to generate its tokeniser and parser from an LALR grammar. Most Java IDE development tools, XText included, only provide support for generating recursive descent parsers. They do not allow the use of left recursion and operator precedence in the grammar. So, implementation of the plug-in required construction of a new grammar for Byteman rules.

Byteman rules employ Java expressions in their `BIND`, `IF` and `DO` clauses. In theory, this meant that existing Java grammar rules could be used for the plug-in grammar. For example, ANTLR [26] provides a grammar suitable for recursive descent parsing [27]. However, Java expressions in Byteman rules comprise only a subset of the Java language. Declarations and control structures are not allowed. So, in practice the plug-in grammar needed to be written from scratch.

2.3.4.2 Debugger

To assist with error detection and correction many developers require the use of a debugger. This is a tool, which allows the user to halt code to be further analysed. The developers are usually provided with variables and their values from the point in the code at which it was halted. This can help identify errors and erroneous behaviour produced from inaccuracies. The debugger knows the location to halt through the use of breakpoints. Breakpoints are points the user can set using the editor. These usually represent a line in the code to halt on. Eclipse uses a debugger which has its own API available for users to generate a custom debugger for other languages. This could be constructive for designing a debugger for Byteman. This is however a complex and intricate process, therefore this section will investigate the Eclipse debugger and the possible aid its APIs could provide.

Available APIs

Eclipse contributes a collection of support for developing custom software to aid debugging. These include APIs, which can re-use Eclipse's own debugger and its features. These APIs are: `org.eclipse.debug.core` and `org.eclipse.debug.ui` [4]. The first of these relates to assistance with the functionality of the debugging features [25]. The aesthetics displayed when debugging are supported and implemented using the UI API. This provides an expansive spectrum of varying libraries to implement the GUI features [4].

Launcher

Through research it became apparent a launcher was necessary. This is used to launch the custom program and implemented using a `LaunchDelegate`, which is used to specify all the launch information. The launcher recognises the configuration that is required to execute the program using the extension [25]. Within this launcher the mode that the program must be launched in is specified. There are two modes available that are relevant to the specification of this project, these are *run* and *debug*. *Run* executes the program normally. *Debug* will execute the program with a debugger, which will identify breakpoints to pause the program. Within this class additional arguments are necessary to specify to the JVM to launch the program in debug mode [26].

Debug model

The debugger uses a debug model and event handling. Once the interpreter is started in debug mode then it listens for debug commands, and then sends debug events to the event socket. There are two sockets, one listening for commands, the other listening for events.

The developer must provide a debug model which implements all available debug model interfaces demonstrated in figure eight. The `IThread` is instantiated for communication and records the breakpoints. It extracts information for the breakpoint and annotates the current thread with this. The `IDebugElement` is used to implement common functions in a debugger and prevent code replication. `IVariable` is a wrapper for variable names, which delegates retrieving the value of the debug target. `IValue` is used to display the value and depending on the values used within the interpreter this can be simple strings and integers or more complex. `IStackFrame` parses and caches stack frame reply messages from the interpreter, which retrieves information about the current step and previous step of the debugger. Each `IDebugTarget` instantiates an `IProcess` (the DebugTarget will be explained in the next section).

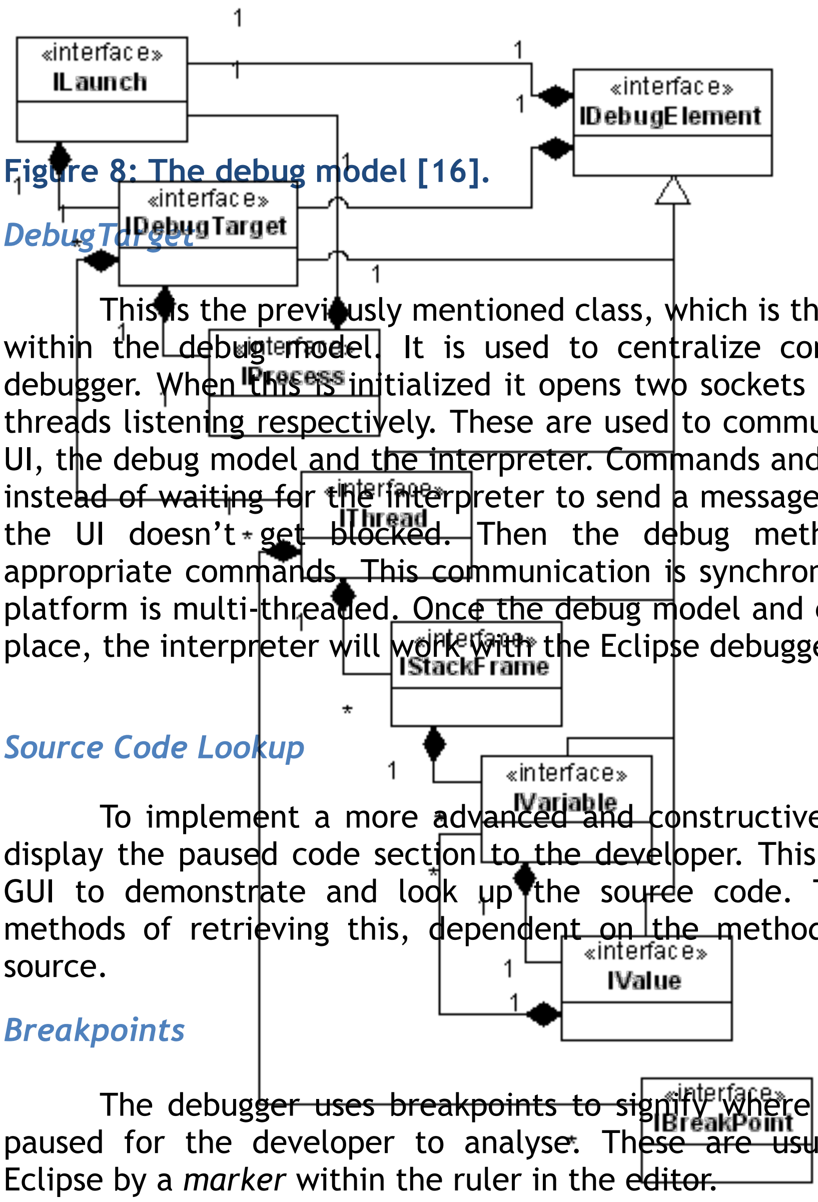


Figure 8: The debug model [16].

DebugTarget

This is the previously mentioned class, which is the prominent feature within the debug model. It is used to centralize communication in the debugger. When this is initialized it opens two sockets with read and write threads listening respectively. These are used to communicate between the UI, the debug model and the interpreter. Commands and events are used, so instead of waiting for the interpreter to send a message, events are used so the UI doesn't get blocked. Then the debug methods can send the appropriate commands. This communication is synchronised, as the Eclipse platform is multi-threaded. Once the debug model and event handler are in place, the interpreter will work with the Eclipse debugger [16] .

Source Code Lookup

To implement a more advanced and constructive debugger, this will display the paused code section to the developer. This is used to create a GUI to demonstrate and look up the source code. There are different methods of retrieving this, dependent on the method used to store the source.

Breakpoints

The debugger uses breakpoints to signify where the code should be paused for the developer to analyse. These are usually represented in Eclipse by a *marker* within the ruler in the editor.

Appending a Breakpoint:

1. Check that it is actually valid.
2. Check it is active.
3. Ensure that it is supported by the debug model and the program attempting to use it.

There is an API class available called `LineBreakpoint`, which is an abstract implementation used to help implement your own constructor and model identifier. To authenticate that breakpoints are valid the breakpoint listeners need to be filtered. Extracting the breakpoint number from the interpreter event message, and then finding the corresponding breakpoint object allows the collection of information. Then the current thread should be annotated with this information.

The breakpoints are added as *deferred breakpoints*. Breakpoints are identified as the interpreter starts and are added before the interpreter's start-up has completed. After they have been added the interpreter is resumed again. The event handler supervises the suspension of code from breakpoints and deleting them.

Defining a Breakpoint:

1. First define a breakpoint structure using the breakpoint and resource *marker*.
2. Use the toggle breakpoint menu item from Eclipse, using the appropriate debug model.
3. Toggle breakpoint action.

[16]

Byteman Alterations

To add a debugger to the plug-in is going to be a complex task and additional functionality needs to be added to the Byteman rule compiler. Research into this identified that the table of bytecode offset to line number mappings, and the name of the rule's source file needed to be injected into the bytecode. This research needs to be added to include the custom Eclipse debugger and its execution.

2.4 Research Conclusion

The research demonstrated many advantages that the project should include to provide a proficient solution. It reinforced the reasons for implementing a plug-in rather than a full IDE. The analysis identified that the most proficient IDE would be Eclipse. Therefore an Eclipse plug-in will be developed.

The software decided upon was Xtext and ANTLR, these were emphasised as the most appropriate choices for the project. They provided additional features and came as a package to support the development of the project. They will be utilized to simplify the complex process of generating the plug-ins lower level components.

The research emphasised that it would benefit users to integrate the command scripts into the IDE. It also suggested that additional features to improve the current support would be essential. This would remove the current command line interface execution improving usability of Byteman.

Research was hampered by a lack of easy access to information and support for Byteman. Similar projects were investigated and the current project has provided new insights in this area.

3 Requirements

The requirements specify the entire system and the abilities that it should provide the user with. This involves the interaction between the user and system, as well as the attributes that the system will consist of.

3.1 Requirements Specification

3.1.1 Functional Requirements

Data Entry and Outputs

R1- A text editor must be able to recognise `.btm` files.

R2- The editor must enable the user to write Byteman rules, with the editor recognising patterns and tokens of the rule.

R3- The user must be able to see highlighting of errors and syntax within the rule.

R4- The user must be able to view assistance to complete code.

R5- The plug-in should include a Byteman specific *perspective* including different views.

R6- The plug-in should contain a navigator, to enable the user to navigate through different projects.

R7- The plug-in should enable the user to navigate to the Java class defined from the rule.

R8- The plug-in should provide a *Console* for output.

R9- The plug-in will provide the user with an easy to use menu and GUI for installation of the Byteman agent.

R10- The plug-in should also provide the user with an easy to use menu and GUI for the submission and unloading of rules.

R11- The user should be able to view which rules are submitted.

R12- The user should also be able to check the state of injection, as well as filtering this by file or rule name.

R13- The plug-in should be able to generate a skeleton template of a Byteman rule for the user.

R14- The plug-in should offer the ability to utilize a debugger to validate code.

3.1.2 Non-Functional Requirements

Speed

R15- Once the plug-in has been installed it should load immediately.

R16- New `.btm` files should load straight away.

Usability

R17- The system should have a user manual to aid the user with utilization.

R18- The system should utilize the menus within eclipse so users can easily navigate and use the features.

R19- The install and submit GUIs and menus should be easy to navigate and quick to use.

R20- Any output from the plug-in should be displayed in a clear and concise way.

R21- The editor should improve the usability of Byteman.

Reliability

R22- The plug-in isn't a critical system but if properly installed should be available whenever Eclipse is loaded and working.

3.1.3 Hardware and Software requirements

Languages

R23- The plug-in will be programmed using Java.

R24- ANTLR will be used as a tool to implement the grammar to generate the parser.

R25- The plug-in should utilize Xtext to support the implementation.

Platform

R26- The plug-in will be used within Eclipse.

4 Design

4.1 Overview

The strategy for the design of the system was iterative. The system was divided into separate sections. Each section represented a feature to be developed for the plug-in. Each feature was designed, implemented then tested before progressing onto the next. This ensured that lower level components were functional before the higher level components were developed and appended.

4.2 System Design

4.2.1 Plug-in

The project will develop a plug-in, which should provide a maintainable structure. The system was split into two sections, functional and graphical. Figure nine displays `ncl.ac.uk.byteman` for the functional features and `ncl.ac.uk.byteman.ui` for the GUI features. This architecture ensures alterations can be made to either section without modification of the other. It also makes navigation through the system more manageable, and quicker. In the figure is the structure of the system, the separate projects and the features they will provide.

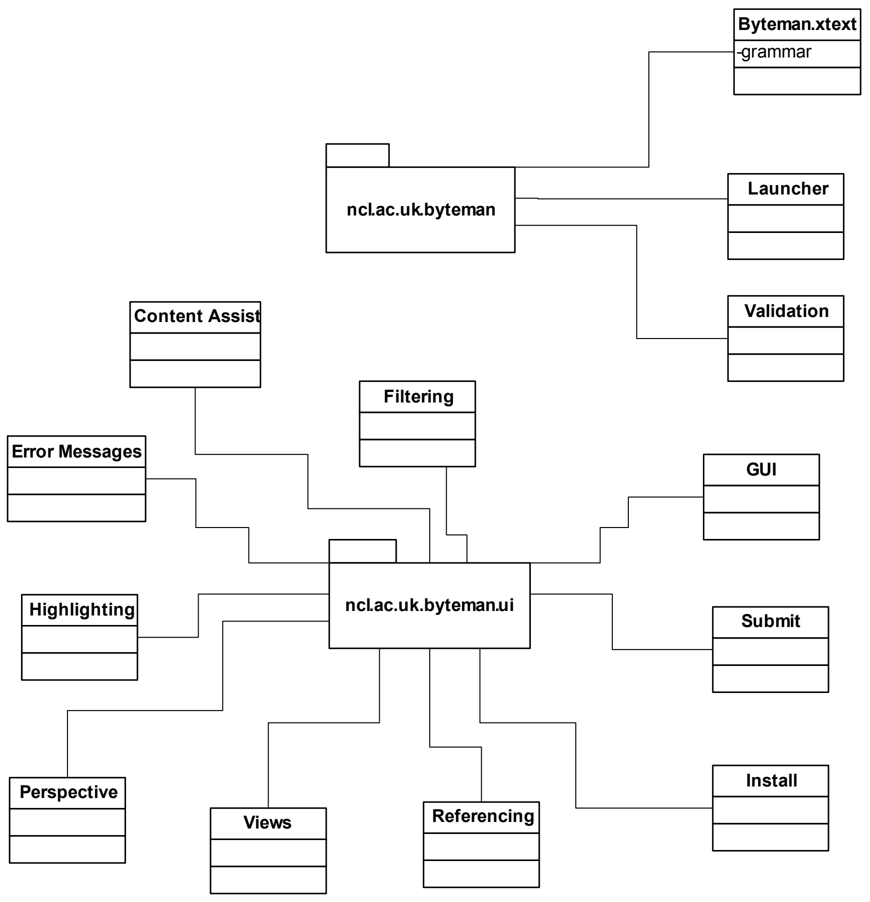


Figure 9: Diagram of plug-in structure.

4.3 Software Decisions

To produce the system structure above the correct software needs to be identified. This has to support the development and produce appropriate features. This section covers the decisions made to select the appropriate software.

4.3.1 Languages

4.3.1.1 Java and ANTLR

The principal language the system will be developed in is Java. This is due to the choice of IDE and the software used to support the development of the system. The Eclipse IDE is implemented in Java, and Xtext is a Java based DSL development software. Xtext provides use of ANTLR, a tool for generating grammar [27]. This will be utilized to develop a grammar to generate a parser for the editor to recognise the rule language. This is used to take advantage of the software capabilities.

4.3.1.2 Check

Check is an expression language provided by Xtext. This enables the definition of constraints and error messages to be used in the plug-in. These constraints use a model of the rule language produced by the parser. This will be used as the error messages will be more explanative than the current ones, and displayed to the user in the editor. The language also easily expresses the constraints to enable errors or warnings [21].

4.4 Feature Design

The features of this system aim to remove the limitations of Byteman. The design will consider available features from other IDEs, identified in the research, and design a Byteman alternative.

4.4.1 Structured Editing

Currently Byteman has no structured method for editing rule scripts. This section will provide the design of this as well as additional features to provide expansive support for the user.

4.4.1.1 Grammar

Byteman provides users with no method of recognising the rule script, it is displayed as ordinary text in editors. The first lower level component that is required to produce structured editing is the grammar. This will divide and analyse the Byteman rules. The grammar will provide a comprehensive rule set to recognise the structure of the rule language. It will allow a method to identify the patterns for the validation necessary when editing the Byteman rule scripts. The rule language allows the use of Byteman and Java syntax and semantics, these are both required in the design of the grammar.

Byteman Grammar Rules

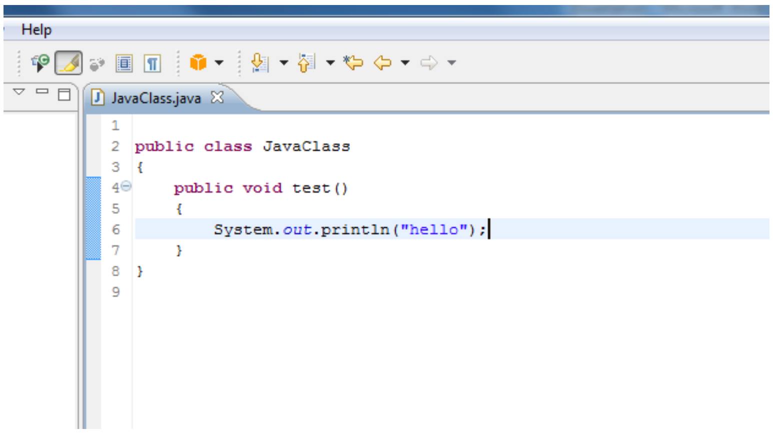
The grammar must contain grammar rules specific to the patterns that are imposed on the Byteman rule language. The domain model of the grammar must include the ability to contain multiple rules. The structure of the grammar will include patterns for individual Byteman rules in the script. Each Byteman rule will be broken into sections: event, condition and action. The event will contain the class, method and binding, if it is present. `CLASS` will contain a Java class pattern, and `METHOD` will contain a Java method pattern. The start and end keywords must be `RULE` and `ENDRULE`. The condition is a Java Boolean expression, to be evaluated. The action will also be a Java expression, which is described in the next section [1].

Java Expression Grammar Rules

Java is a general-purpose language resulting in a broader and more expansive range of patterns than Byteman. This means that the design of this section will be more complex to implement. Java contains a variety of different expressions and types, these are all given a precedence. Here is the precedence from highest to lowest: assignments, arithmetic, bitwise, logical, comparison, field access, method calls and simple expressions. This structure will be added to the grammar to allow Java expressions, for different sections of the Byteman rule. This will provide an almost complete Java grammar, as well as the Byteman grammar.

4.4.1.2 Editor

The grammar will provide the recognition of the rule language constraints and validation, the editor will use these to produce a structured editing tool. The Eclipse editor is displayed in figure ten. This offers the user multiple features. The initial feature is its ability to recognise `.java` files and display them with enforced constraints. The plug-in will implement an editor. The editor will provide the ability to recognise `.btm` files and this will provide an editor specific to Byteman. This will be achieved using the extension features provided by Eclipse PDE. The editor will offer a text editor to input the rules, and the grammar will provide the validation of the input. This will provide Byteman with the ability to develop rules quickly and easily through the use of a structured display.

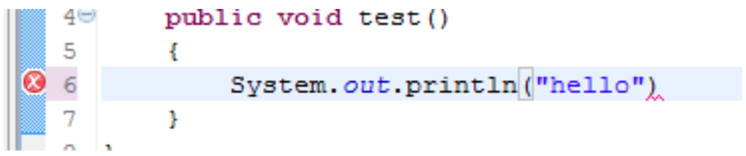


```

public class JavaClass
{
    public void test()
    {
        System.out.println("hello");
    }
}

```

Figure 10: Example of an Eclipse editor.

4.4.1.3 Error and Syntax Highlighting

The editor requires additional features for increased support. Currently Byteman's error detection technique is limited with no visual display in the code. There is also no structured method of viewing the Byteman rules. Eclipse provides this within the editor displayed in figure eleven. The editor contains highlighting of the error, and a marker notifying its location. The keywords are also highlighted, providing a more readable structure to the code. The project will provide error and syntax highlighting. This is a visual method of alerting the user to aspects of the development. Errors will be displayed in a descriptive manner demonstrating erroneous sections of the rule with highlighting and markers. Syntax highlighting accentuates keywords and areas of the code, providing a simple method of distinguishing the structure. It will provide different colours to highlight the different sections of the rules (i.e. event, condition and action). This will offer the user valuable information and the increased probability that an error will be identified and eliminated. This eradicates confusion concerning the structure of Byteman rules and displays them more coherently [20].

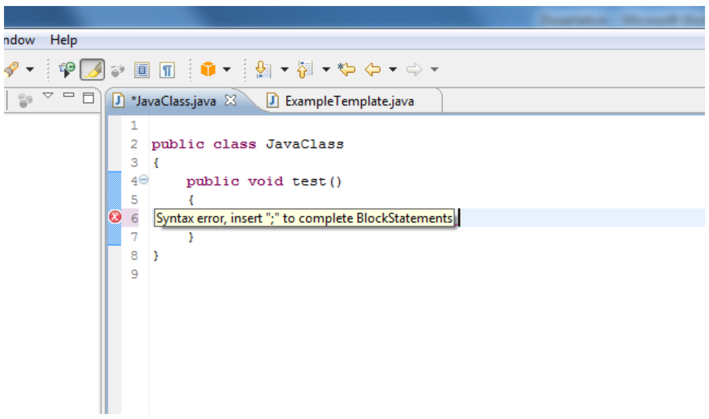


```
    public void test()
    {
        System.out.println("hello")
    }
```

Figure 11: Example error and syntax highlighting from Eclipse.

4.4.1.4 Error Messages

Byteman’s error messages are confusing and displayed in the command line. Eclipse provides comprehensive error messages in the editor shown in figure twelve, which the target users are used to. To assist with validating code, error messages should be incorporated into the plug-in. These will be for compilation errors, as they are displayed in the editor before runtime. These should clearly and concisely demonstrate to the user what is wrong, providing the developer with information to solve the problem. This will decrease the time taken to debug code, as well as assisting in producing valid code.

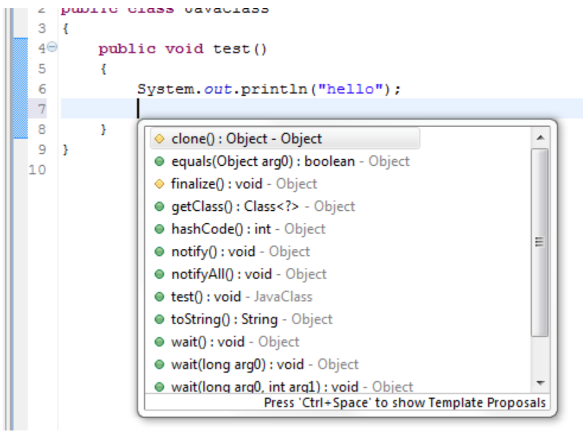


```
public class JavaClass
{
    public void test()
    {
Syntax error, insert ";" to complete BlockStatements
    }
}
```

Figure 12: Example error message within Eclipse

4.4.1.5 Code Completion

Byteman has no automation feature for completing code. Eclipse provides this through a shortcut and a list displayed in figure thirteen. The short cut decided upon for this editor is CTRL + SPACE, due to its usage within Xtext and Eclipse. It is a recognised shortcut therefore provides an easier extension for the target users. This feature will be produced using Xtext's available content assist development support. This will provide users with a list of Java and Byteman variables to choose from to complete the code element. Code completion provides a simpler and quicker method for developers to implement code, making it an attractive feature for the editor [22]. The absence of such support is a significant hindrance to user adoption of Byteman.



```
public class JavaClass
{
    public void test()
    {
        System.out.println("hello");
        
    }
}
```

```
clone() : Object - Object
equals(Object arg0) : boolean - Object
finalize() : void - Object
getClass() : Class<?> - Object
hashCode() : int - Object
notify() : void - Object
notifyAll() : void - Object
test() : void - JavaClass
toString() : String - Object
wait() : void - Object
wait(long arg0) : void - Object
wait(long arg0, int arg1) : void - Object
                    Press 'Ctrl+Space' to show Template Proposals
```

Figure 13: Eclipse example of content assist.

4.4.1.6 Referencing Java Classes

There is no method of alternating between Java classes and Byteman rules. Eclipse provides functionality which allows a user to highlight a class name and open it through the use of a menu item. This feature will be implemented using the JDT API and the search engine it provides. To use this feature the user will have to highlight the class or method that they require to be displayed. Then utilizing a provided menu feature will access the specified class, as long as it is within the workspace. This will allow users to navigate between their respective programming scripts easily, without the upheaval of having to identify the correct package and class. This will decrease development time and errors as the user can validate Java declarations more simply.

4.4.2 Perspective

This is a visual container for views in the IDE such as the editor or *Console* and is specific to a single language. Eclipse provides *perspectives* for different languages and features, as shown in figure fourteen. This provides the user with easier navigation, as all the features are contained in one location. Eclipse organises related views within the perspective using tabbed panes. Tabs within the same pane present the same type of content such as file contents or build, execution and debug output. Figure fifteen demonstrates the views specifically the *Console*, which is used for output.

These will be implemented by adding *dependencies* in the plug-in, then creating an *extension point* to specify the *perspective* and its views. The *perspective* will ensure that .btm files are opened using the Byteman editor. It will also provide a variety of views for the user, such as *Problems*, *Outline*, *Navigation* and the *Console*. This will supply the user with increased information, which will be easily accessible.

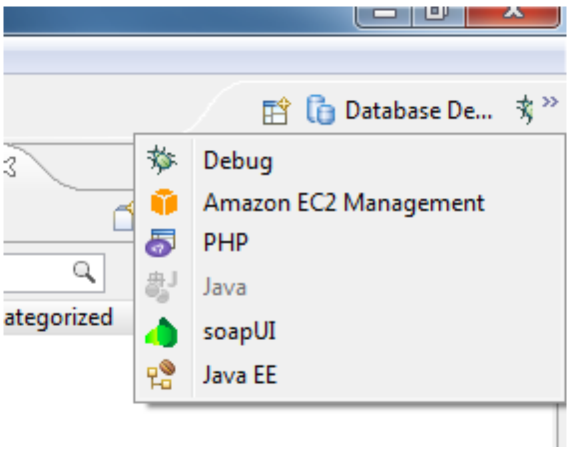
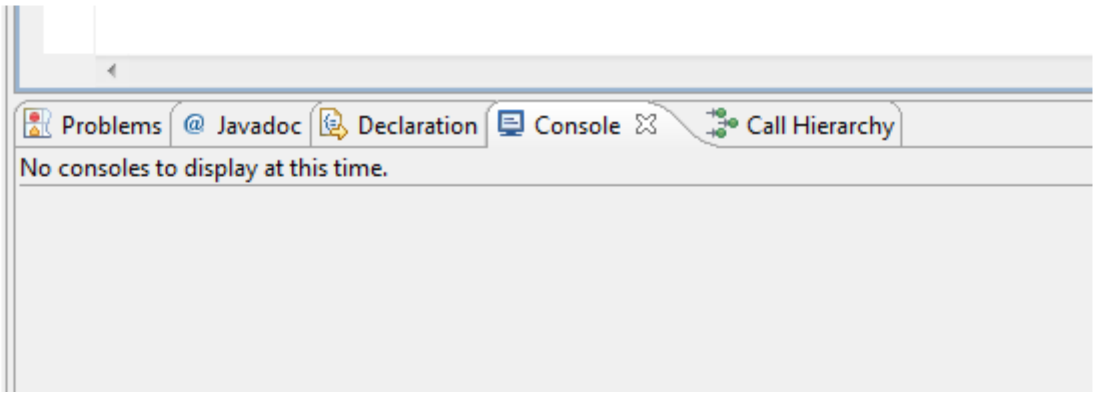


Figure 14: Example of available perspective in Eclipse.

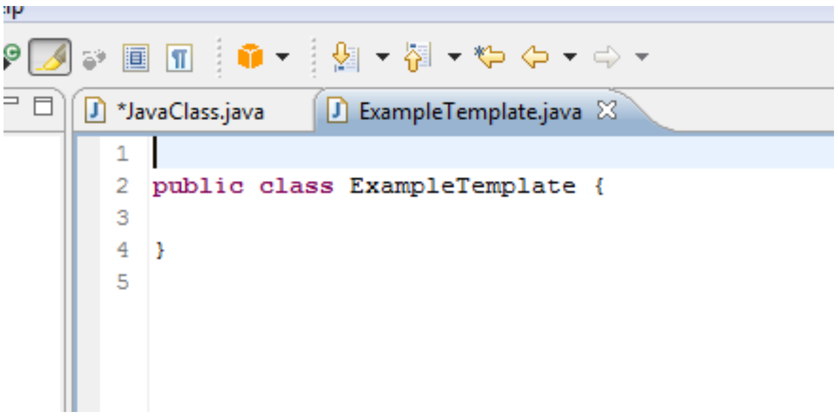


No consoles to display at this time.

Figure 15: Eclipse example of views.

4.4.3 Byteman Rule Template

Currently, each Byteman rule is developed from a blank text file, no aid is provided. Eclipse offers users a template Java class, which contains the skeleton class structure as shown in figure sixteen. Template generation is a standard feature in a multitude of IDEs. The feature will allow the user to generate a template .btm file in a project. A basic rule definition will automatically be printed in the file, decreasing the time to develop, especially for experienced users who will be accustomed to the availability of such a feature. It will also support new users by providing the principals for Byteman rules immediately. Templates will be achieved using Eclipse PDE support for wizards to generate new files. [28].



```
public class ExampleTemplate {

}
```

Figure 16: Example template from Eclipse.

4.4.4 RCP

RCP is a rich client platform, used to provide plug-ins with improved GUI features. The research into this software revealed that it wasn't suitable for this project. The plug-in that will be generated from this project will use the menus and views available within Eclipse and will only contain an inconsequential amount of GUIs. RCP is also unavailable for use with Xtext, and as Xtext provides all the features necessary for this project RCP became redundant [5].

4.4.5 Installation and Submission

Byteman already allows specialised features for the user as described in section 2.2. It was necessary to design a method to integrate and automate these within Eclipse. This would make the command line redundant, resulting in decreased complexity and no command to remember. It will provide developers with an easy and quick technique for advanced Byteman use, allowing them to install, submit and view output efficiently [1]. These features will be implemented using GUIs. These will contain a text box for the user to add the necessary information. The button on the GUI display will then allow for the completion of the necessary code to execute the script. The output produced from these actions should be displayed to the user appropriately. This should be achieved through the use of a GUI dialog box, as well as the use of a *Console* view. This will allow the user to identify and analyse output quickly and simply.

4.4.5.1 Launcher

The previous design of a submission GUI seemed too complex. The user had to provide the absolute path of the rule, even though they had developed it in Eclipse. The design was refined to provide a method of rule submission using the rule that had been developed. This produced a design for a launcher, which will submit rules in the workspace from Eclipse. This will use a specified launch configuration for Byteman and the `.btm` extension. The user right-clicking on the rule and using the Byteman launcher will submit the rule. There will be no requirement for the rule path to be entered. The launcher will allow for simpler and quicker launching of rules within the workspace.

4.4.6 Additional Byteman Support

The research identified that additional features to the current Byteman support would be beneficial to users of the plug-in. These are features to help extend the current scripts and make their methods accessible. The submission script has methods for output about all rules and deleting rules. It was decided that their availability to the user should be integrated into the IDE. This will remove their use within the command line, providing simpler execution.

4.4.6.1 Viewing Rules

The output from the injection state will be printed within the IDE. This will require the use of menus. When the menu is selected this should trigger a method to retrieve the rules that are currently submitted in the program. If there are no rules then the user should be notified of this. The output should be displayed in the *Console*. This will provide the user with a quick and simple mechanism for viewing information about which rules have been installed.

4.4.6.2 Filtering Rules

The output previously mentioned has no method to identify relevant information, therefore the design was reviewed and refinement identified. This will be a filter to specify the information required. The filter options are retrieval of information about a script or a rule. A GUI containing a text field and a button will enable this. The name of the rule or script will be entered to execute the filtering. The user should then click the button, which will execute the retrieval of information.

4.4.6.3 Uninstalling All Rules

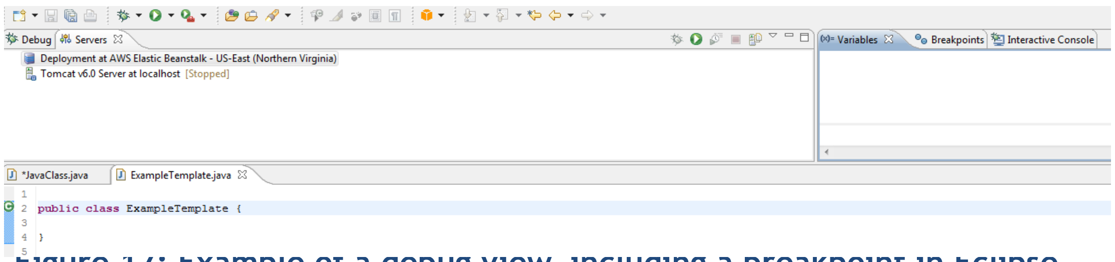
As the user has the ability to submit rule scripts, then logically they should also be able to unload them. This feature will provide the user with this capability. This feature should have a menu item in the IDE for the user to choose. When selected this should execute a code segment which uses the submission script. The method available from the script will uninstall all the rule scripts currently loaded.

4.4.6.4 Uninstalling a Selected Rule

It is sometimes necessary to unload specific rules rather than the whole of the currently loaded ruleset. The design was revised to allow the user to specify a rule script for deletion. It will use a menu item, which will open a GUI. This should allow the user to enter the name of the rule scripts they require unloaded. This will aid the user in a more specific deletion mechanism.

4.4.7 Debugger

Byteman currently has no debugger, unlike the one provided by Eclipse in figure seventeen. This is a *Perspective* on its own and provides a multitude of features. To transfer this feature to Byteman rules, the debugger would need to halt a rule when it is executing in the JVM. This would be implemented utilizing the debug APIs mentioned in section 2.3.4.2, basing the program structure on the model shown in figure eight (page twenty-five). This analysis would decrease the amount of time exhausted while debugging, and ensure simpler error identification. The debugger should give the user the ability to append breakpoints and observe halted code [16].



Deployment at AWS Elastic Beanstalk - US-East (Northern Virginia)
Tomcat v6.0 Server at localhost [Stopped]

```

public class ExampleTemplate {

}

```

Figure 17: Example of a debug view, including a breakpoint in Eclipse.

5 Implementation

This section explains the technical aspects of the project during the implementation of the system. It describes the features of the system and the mechanisms used to complete them. It displays the achievements of the features and what they provide the user with. The system's user guide can be found in appendix B.

5.1 System Overview

A plug-in was produced providing developers the integration of Byteman with Eclipse. It removed the need for command line execution. This was an advantage as the target users are IDE developers. It allowed structured editing of Byteman rules and the submission of rules from the IDE. The system adhered to the design and removed Byteman limitations.

5.2 Grammar and Parser

An editor which enforces Byteman syntax and semantics was required, so a grammar was developed. This specifies rules for the patterns of Byteman and Java expressions. The grammar provides the user with an editor to input rules, which will supports their validation.

Figures eighteen to twenty demonstrate small fragments of the implemented grammar sections. Patterns for Java expressions are displayed within figure eighteen. It contains the highest Java precedence, assignment, as described in section 4.4.1.1. The first line is the name of the rule, the right-hand side of the second line is the pattern. The parser recursively matches the input text against the patterns, to validate the inputs are correct. Java expressions in the grammar state the highest precedence expression first continuing to the lowest one. This provides the user with all possible Java expressions within the scope of the Byteman rule language. These are constrained by the grammar to the `BIND`, `IF` and `DO` clauses of the Byteman rules.

```
AssignmentExpr:
    left=VariableExpr ASSIGN right=Expression;

OperatorExpr:
    PlusMinusOperand(ops+=(PLUS | MINUS)
    operands+=PlusMinusOperand)*;
```

Figure 18: Grammar section for Java

A grammar rule for Byteman is displayed in figure nineteen. As Byteman rules have an event, condition and action, this is the architecture of the grammar. The `BytemanRule` is the representation of an entire rule within a script. Each section of the rule (e.g. event) is defined by separate rules in the grammar. The `Rule` in the second line of figure nineteen is defined at the bottom of the figure. This grammar rule represents the name of the Byteman rule and specifies that it may be a qualified name or a JVM type (a choice of Java classes). The grammar rule allows one or more `Rule` in the name this is represented by the +. This provides a valid grammar rule that allows a Byteman rule name of any length, as long as there is at least one non-white character.

```
BytemanRule :
     keyword=KEYWORD_RULE  (name+=Rule)+
     event=Event
     condition=Condition
     action=Action
     KEYWORD_ENDRULE;

Rule:
QualifiedName | JavaType;
```

Figure 19: Grammar section for Byteman rules

Figure twenty displays a terminal rule, which defines some of the tokens accepted by the grammar. It defines the tokens for the two boolean values, true and false, which can be entered in either upper or lower case. Xtext provides default terminal rules defining Java tokens like numbers, strings, comments and whitespace. However Byteman has different conventions to Java for white space and comments so the plug-in grammar had to define its own token rules.

```
terminal BOOLEAN :
     'TRUE' | 'true' | 'FALSE' | 'false' ;
```

Figure 20: Terminal rule in grammar

5.3 Editor and Features

The grammar provides validation for the input of the editor. The next feature was the development of the editor with the ability to recognise `.btm` files as Byteman rules. This was achieved through the use of an extension in the `plugin.xml`. This extension specifies the editor to be used, and within this was an attribute stipulating the extension for its files. This provides the user with structured editing and parser generated errors. These errors are generated because of the constraints specified by the grammar rule patterns. They are displayed in the editor to notify the user quickly and simply.

The plug-in offers syntax and error highlighting, to accentuate sections of the code to the developer. Syntax highlighting was implemented using a highlighting calculator and configuration. The configuration specifies colours for the calculator to use. The calculator computes the correct sections of code to highlight. This project implements keyword highlighting.

To implement this the plug-in traverses the parse tree. During this traversal the calculator identifies the keywords and highlights them. This highlighting is provided to the user throughout the development of Byteman rules. Error highlighting provides highlighted code and so developers can now identify invalid code more quickly.

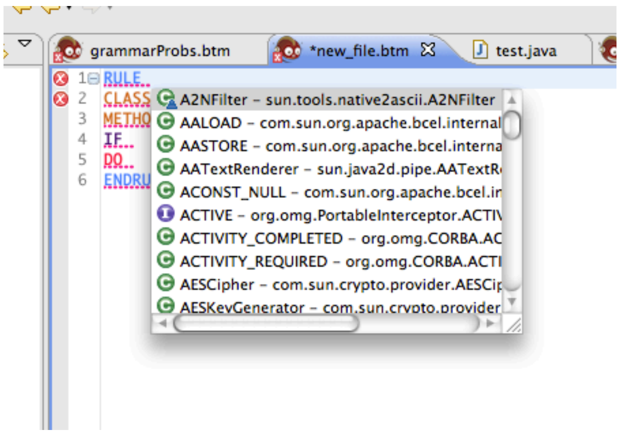
Additional error detection methods now implemented are error markers and messages. Xtext provides an expression language called Check as described in section 4.3.1.1. This language was used to identify errors and provide custom error messages. This permits for more coherent and accurate error messages. As displayed in figure twenty-one the validation specifies a type to inspect. An appropriate error message is then specified to aid the user. The example calculates if the user has neglected to state a line number or if the line number is less than zero. The check requires the condition to equate to false for the error message to be displayed. This would highlight the error and place a marker at the location, as well as specifying the error message, providing users with easier and quicker error detection.

```
context BytemanRule ERROR
"AT LINE must contain a line number above zero":
let a = "LINE":
this.event.locationSpec.atline.name.contains(a) ?
event.locationSpec.atline.line >0 : true;
```

Figure 21: Example code for errors.

Another feature provided in the editor is code completion. This supplies the user with a keyboard shortcut (CTRL + SPACE). This shortcut can be used to offer the user with a list of available options, as shown in figure twenty-two. When chosen, the code element will be completed automatically. This was implemented with the aid of two different methodologies.

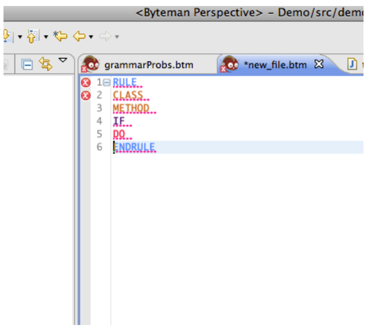
The design emphasised the necessity for content assist for Java types and common classes. This feature was completed using the JVM types package. This provides code completion of Java elements. The second method uses the “`ProposalProvider`” class. The developer implemented this class to specify the traversal of the parse tree. This will produce the content assist in the correct section of the rule, so not to violate a constraint. This class will then specify the list of available Byteman variable options for the user. The user can now quickly complete Java and Byteman elements, and the target users are provided with a feature they are accustomed to.


grammarProbs.btm
*new_file.btm
test.java
A2NFilter – sun.tools.native2ascii.A2NFilter
AALOAD – com.sun.org.apache.bcel.internal
AASTORE – com.sun.org.apache.bcel.interna
AATextRenderer – sun.java2d.pipe.AATextR
ACONST_NULL – com.sun.org.apache.bcel.in
ACTIVE – org.omg.PortableInterceptor.ACTIV
ACTIVITY_COMPLETED – org.omg.CORBA.AC
ACTIVITY_REQUIRED – org.omg.CORBA.ACTI
AESCipher – com.sun.crypto.provider.AESCip
AESKeyGenerator – com.sun.crypto.provider

```
RULE..
CLASS
METHO
IF..
DO..
ENDRU
```

Figure 22: Content assist options list

A functional feature for the plug-in was the ability to generate a template Byteman rule. When this option is executed it will open a new `.btm` file in the editor. This file displays a basic rule structure. This is displayed in figure twenty-three. The user can then fill in the missing code. As the user doesn't have to enter the keywords development time is decreased. Wizards were constructed to allow easy construction of the file for users. As figure twenty-four demonstrates the wizard provides the creation of a Byteman file. Figure twenty-five displays the subsequent wizard for locating and specifying the project the file will be placed into, as well as a text field to enter the name of the file. Once the user presses *finish* the new `.btm` file will be created and displayed in the editor. These were implemented using the PDE's wizard extension.



```
RULE..
CLASS..
METHOD..
IF..
DO..
ENDRULE
```

Figure 23: Template Byteman Rule

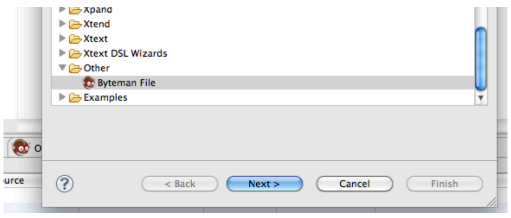
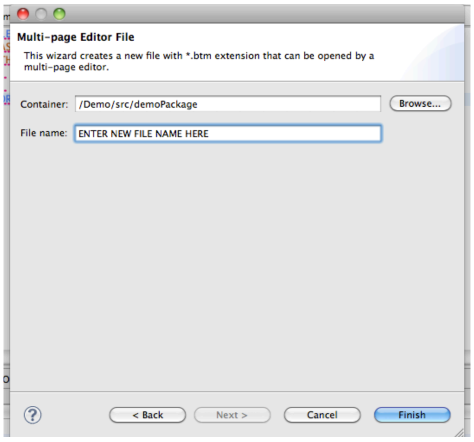


Figure 24: Wizard for creating a Byteman file


Multi-page Editor File

This wizard creates a new file with *.btm extension that can be opened by a multi-page editor.

Container: /Demo/src/demoPackage

File name: ENTER NEW FILE NAME HERE

Figure 25: Wizard to enter details of the new file

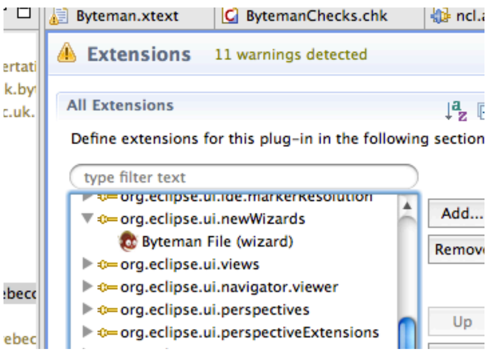

Byteman.xtext
BytemanChecks.chk
Extensions
11 warnings detected
All Extensions
Define extensions for this plug-in in the following section
type filter text
org.eclipse.ui.ide.markerResolution
org.eclipse.ui.newWizards
Byteman File (wizard)
org.eclipse.ui.views
org.eclipse.ui.navigator.viewer
org.eclipse.ui.perspectives
org.eclipse.ui.perspectiveExtensions
Add...
Up

Figure 26: Extension used for the wizard

With the plug-in the user is able to open a specified Java class from a Byteman rule. It is implemented using the Java search engine in the `org.eclipse.jdt.core.search` package. This API is used to implement a searching ability for the workspace. The user must highlight the Java element they want to search for, and select a menu option to open the reference. This executes code which uses search patterns to identify the relevant search term. If the pattern is discovered within the workspace, the class containing this is opened. This provides the user with simple and quick navigation to relevant code.

5.4 Integration of Command Scripts

The command scripts were integrated into Eclipse. Installation of the agent was provided through a GUI. Figure twenty-seven demonstrates the GUI, which can be selected through a menu item. The user must enter the Byteman home and the identified programs process ID or the main Java class name. Once the button is selected installation of the agent is executed. If the agent is successfully installed it will display a dialog box confirming this. If an error occurs this is displayed in the *Console*. The plug-in also ensures cross platform use. The system will identify the operating system being used, and execute the correct command script.

This feature provides integration of this function into the IDE and makes its execution automatic. This is easier and simpler for the user than using the command line but provides the same functionality.

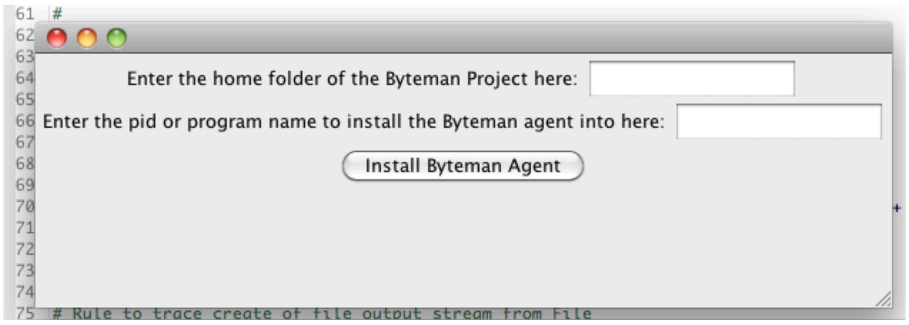

Enter the home folder of the Byteman Project here:
Enter the pid or program name to install the Byteman agent into here:
Install Byteman Agent

Figure 27: Byteman install GUI

The submission of Byteman rules is also provided through a GUI. Figure twenty-eight displays the GUI, which can be opened through a menu item. The Byteman home and rule script name are required in the text fields. The button is then selected, which initiates the submission of the rule. Successful execution outputs a listing of all installed rules, where as failed execution outputs an error listing, and these are both displayed to the *Console*. Submission is only possible if the agent has been installed prior to this, with an opened listener. If the user installs the agent through the IDE the listener will be opened automatically. This feature integrates submission of rules with the IDE, removing the need for the command line. Removal of the command line provides the user with simpler and easier development. This feature also allows automation of the submission.

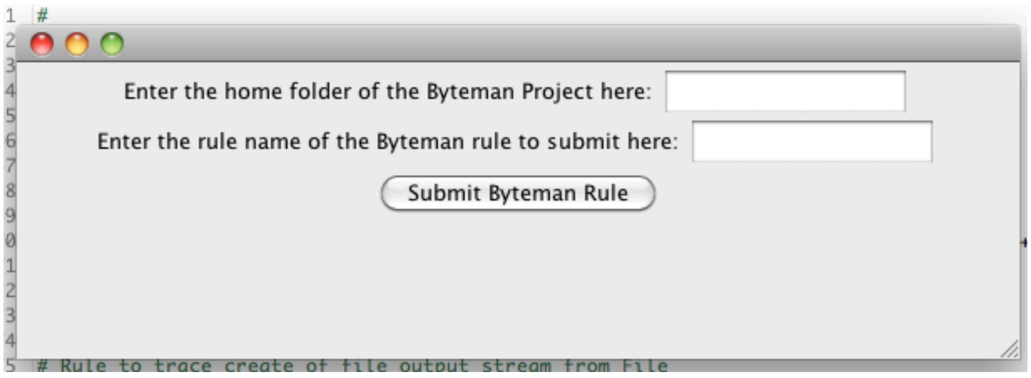
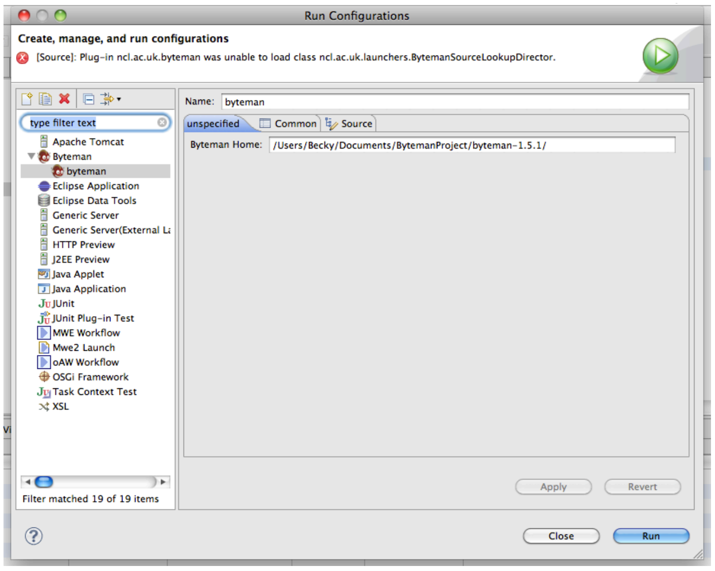

Enter the home folder of the Byteman Project here:
Enter the rule name of the Byteman rule to submit here:
Submit Byteman Rule

Figure 28: Byteman submit GUI

The design specified refinements to the submission of the rules. The launcher has implemented these refinements. Figure twenty-nine demonstrates the launch configuration tab for Byteman. This launcher provides execution of Byteman rules from within the workspace. This is implemented through the use of the `org.eclipse.core.debug`. The API allows the specification of a configuration type, resulting in the recognition of a `.btm` file. The user has to right click on the Byteman rule and click run as. The GUI will be displayed and the user then selects *run*. This will submit the rule with output for success or errors in the *Console*. The Byteman home is only required once, it is stored and displayed in the text field each time the GUI is opened. This provides the user with quicker and simpler submission of rules, as there is less information to input.


Run Configurations

Create, manage, and run configurations

[Source]: Plug-in ncl.ac.uk.byteman was unable to load class ncl.ac.uk.launchers.BytemanSourceLookupDirector.

Name: byteman

Byteman Home: /Users/Becky/Documents/BytemanProject/byteman-1.5.1/

Filter matched 19 of 19 items

Figure 29: Launch configuration for Byteman rules

5.5 Additional Byteman Support

The implementation of the additional features followed the original design specified in Section 4.4.6. These included viewing, uninstalling and filtering rules. The ability to view rules was implemented through the use of a menu item displayed in figure thirty. This then executes a method, which displays the rules in the console as demonstrated in figure thirty-one. Figure thirty displays the menu items for uninstalling rules, there being two options. The first is to uninstall all rules that are submitted. The second, a refinement of the previous, provides the user with the ability to uninstall a specified rule. When this menu item is selected it will display a GUI shown in figure thirty-two. For both of these methods the uninstalled rules will be shown in the *Console* as displayed in figure thirty-three. There are menu items to filter the rules by script or rule name. Figure thirty-four and thirty-five display the two GUIs used to filter rules. The output is then displayed in the *Console*, this is shown in figure thirty-six. These were implemented using a Java instance of the submission script. This instantiation was then used to provide methods that the script enabled. These features are integrated into the IDE. The user is provided with quicker and simpler use of them, as there are no commands to remember.

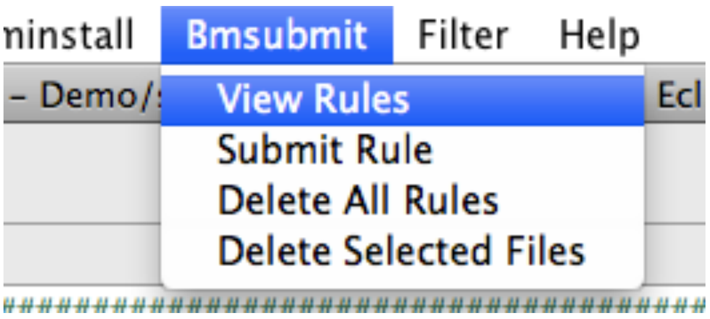
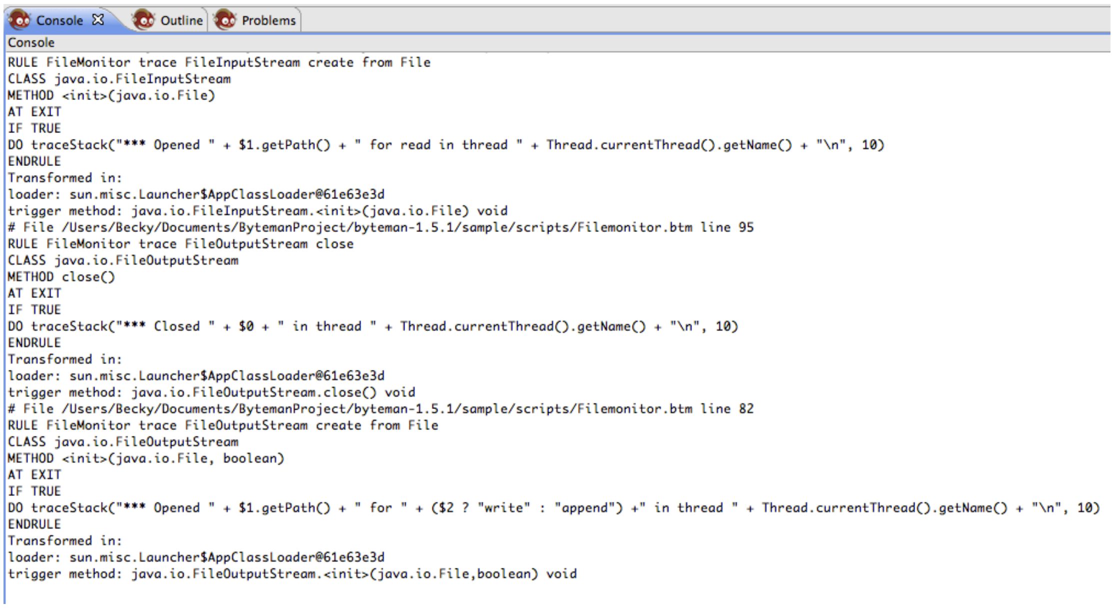
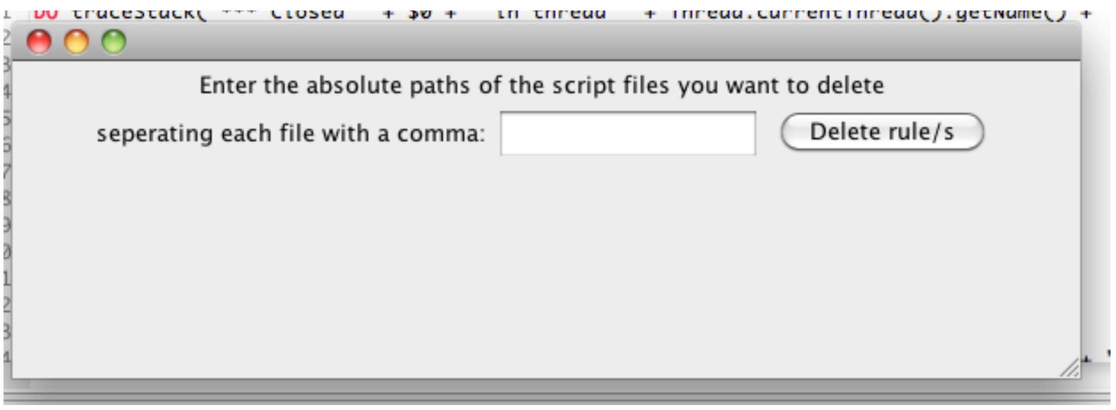


Figure 30: Menu items.



```
RULE FileMonitor trace FileInputStream create from File
CLASS java.io.FileInputStream
METHOD <init>(java.io.File)
AT EXIT
IF TRUE
DO traceStack("*** Opened " + $1.getPath() + " for read in thread " + Thread.currentThread().getName() + "\n", 10)
ENDRULE
Transformed in:
loader: sun.misc.Launcher$AppClassLoader@61e63e3d
trigger method: java.io.FileInputStream.<init>(java.io.File) void
# File /Users/Becky/Documents/BytemanProject/byteman-1.5.1/sample/scripts/Filemonitor.btm line 95
RULE FileMonitor trace FileOutputStream close
CLASS java.io.FileOutputStream
METHOD close()
AT EXIT
IF TRUE
DO traceStack("*** Closed " + $0 + " in thread " + Thread.currentThread().getName() + "\n", 10)
ENDRULE
Transformed in:
loader: sun.misc.Launcher$AppClassLoader@61e63e3d
trigger method: java.io.FileOutputStream.close() void
# File /Users/Becky/Documents/BytemanProject/byteman-1.5.1/sample/scripts/Filemonitor.btm line 82
RULE FileMonitor trace FileOutputStream create from File
CLASS java.io.FileOutputStream
METHOD <init>(java.io.File, boolean)
AT EXIT
IF TRUE
DO traceStack("*** Opened " + $1.getPath() + " for " + ($2 ? "write" : "append") +" in thread " + Thread.currentThread().getName() + "\n", 10)
ENDRULE
Transformed in:
loader: sun.misc.Launcher$AppClassLoader@61e63e3d
trigger method: java.io.FileOutputStream.<init>(java.io.File,boolean) void
```

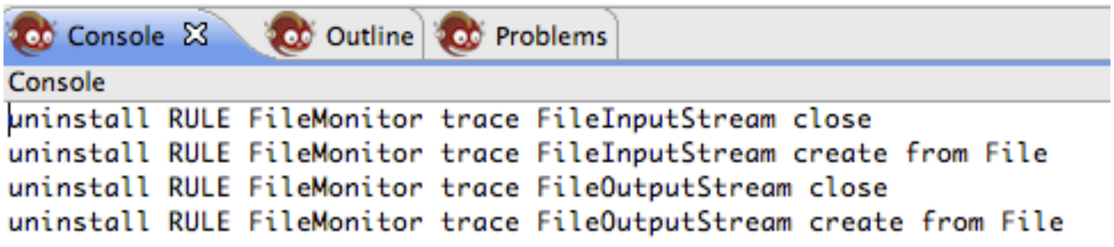
Figure 31: Rules displayed in the console.



Enter the absolute paths of the script files you want to delete

seperating each file with a comma:

Figure 32: GUI for deleting a specified rule.



```
uninstall RULE FileMonitor trace FileInputStream close
uninstall RULE FileMonitor trace FileInputStream create from File
uninstall RULE FileMonitor trace FileOutputStream close
uninstall RULE FileMonitor trace FileOutputStream create from File
```

Figure 33: Console displaying the uninstalled rules.

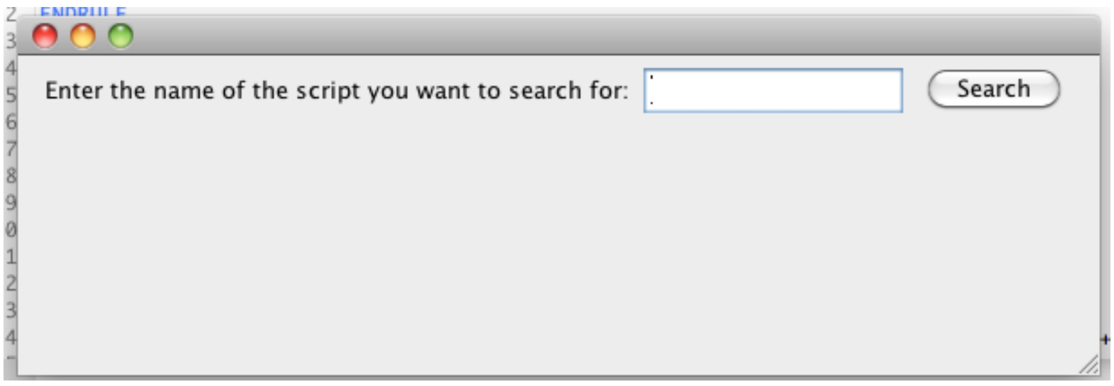


Figure 34: GUI for filtering by script name.

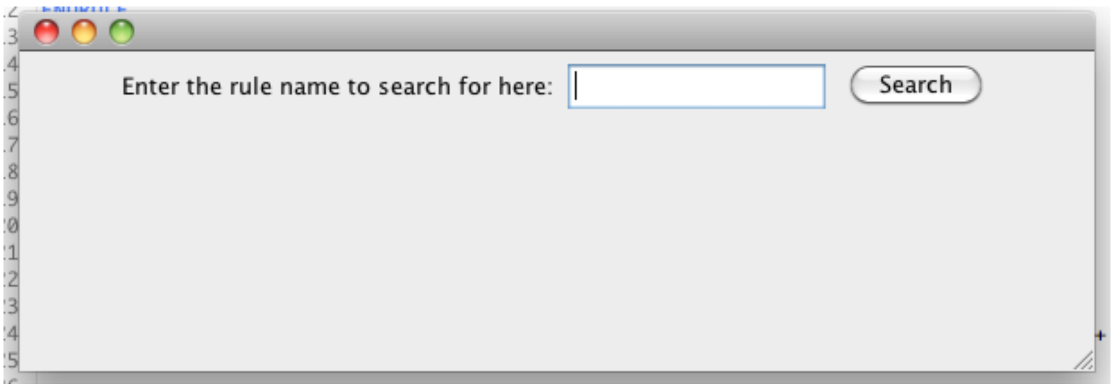
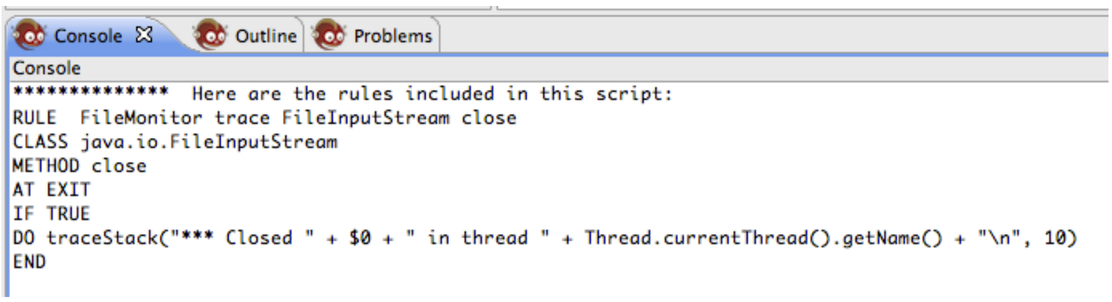


Figure 35: GUI for filtering by rule name.



```
*************  Here are the rules included in this script:
RULE  FileMonitor trace FileInputStream close
CLASS java.io.FileInputStream
METHOD close
AT EXIT
IF TRUE
DO traceStack("*** Closed " + $0 + " in thread " + Thread.currentThread().getName() + "\n", 10)
END
```

Figure 36: The console output for filtering.

5.6 Debugger

The debugger from the design section was an intricate and complex development task, which resulted in being too time consuming for completion. However the research and design is available for another developer to progress with.

6 Testing

This system was rigorously tested to validate that it is executing correctly and achieving its requirements. Testing was vital to provide evidence of a working system, and to identify bugs within it. An expansive testing range and the possible scenarios were used in the testing strategy. This section provides the choices for testing and the strategy. The full testing, screen shots and detailed testing tables, are in appendix A.

6.1 Testing Strategy

The strategy was to divide testing into unit, integration and system tests. This was chosen to provide a wide range of tests. Unit tests are the lowest level of the system and test the coding elements. This can be a class within the program or even a method. These tests identified and provided evidence of the behaviour of the components. For this system the features were divided and a program component was a unit. Integration testing combined the unit tests from the previous program element to identify and analyse whether they will work together. This provides confidence in the integration of the lowest level components, offering intermediate testing. The integration of units meant features were tested. The tests within this section were specified in the sequential order they were inputted. System testing identifies if the integration tests will work together. It combines all the components to identify if any errors were present or if the system was functional. The order of the inputs signifies the order they were executed.

These tests covered the lower level components as well as testing the combined components. This strategy was advantageous for this system as it allowed the division of the system to ensure no errors were present in even the lowest level of the code. As each of the features of the plug-in consisted of units of code, it followed that the most appropriate methodology of testing is to divide the code and test each program element.

The tests studied the behaviour of the plug-in during different circumstances specified in the *Test No.* column, which provides the number and the type of test. These tests provided evidence of the behaviour of the system under normal, erroneous and extreme test cases. Extreme test cases refer to correct input, but these will be maximal or minimal values. Normal defines a test case that is correct. Erroneous expresses a test case, which is incorrect. Some tests will have all three of the test cases, and where appropriate some will not.

Full System Test

This is the result for the system test. It demonstrated the system is functional and passed all of the tests set. This included intensive unit, integration and system testing. As previously mentioned these are available in appendix A.

Test No.	Input	Passed
1: Normal	6.4.1, 6.4.3	Yes

7 Results

The testing identified the behaviour and outputs of the system. This required analysis to determine the overall performance of the system to be evaluated. In this section the test results are discussed and evaluated to demonstrate the strengths and weaknesses of the plug-in.

7.1 Unit testing

Unit testing analysed the individual components of the plug-in. The tests presented a fully functional editor, customised for the use of `.btm` files. This is necessary as the constraints and validations are only functional for Byteman. Using this for other file extensions would cause the editor to present incorrect errors and syntax highlighting. As the system is a plug-in other languages will be supported in the IDE, therefore this would be a redundant feature.

The editor's syntax highlighting feature also demonstrated correct highlighting, only emphasising Byteman keywords with an altered colour. This was successful for normal, erroneous and extreme values. Corresponding results were found for error highlighting, only underlining erroneous code. Custom error messages have a weakness, they produce error highlighting for the whole rule. However the more explanative message will aid the user in locating the error.

Error markers were displayed in the correct location in the left hand side ruler, presenting the user with the line the error is on. These too are only displayed when there is an error in the file, this provides the user with the correct functionality.

Code completion was provided through the use of a shortcut, which displayed a list of options to the user as expected. These options included Byteman variables and formal and custom Java classes. The code completion is only available in appropriate locations. If a Java expression isn't allowed in that location then it won't be part of the code completion list. The user can automatically complete code successfully.

Templates can be generated from wizards. The template provided the user with a skeleton rule displaying the correct keywords. The wizard didn't allow the generation of a file with an incorrect extension. This was necessary as the wizard was designed as custom support specifically for Byteman.

All the GUIs were generated correctly and displayed the correct information entry points. All menus were also displayed and in the correct location.

The installation of the Byteman agent executed correctly, as well as notifying the user with success or error messages. The user was also advised about incorrect input into the text box.

Submission of valid rules executed to completion and outputted the installed rules to the user. The user was also notified of any incorrect input.

The additional features of the current Byteman support also proved correct and presented constructive output. This included viewing the submitted rules, these were correctly displayed in the *Console*. The function that enabled all rules to be uninstalled, unloaded all the rules successfully. Unloading also including the ability to specify a rule to be uninstalled, this was also successful. These both displayed the uninstalled rules in the *Console*. If incorrect information was inputted then an error was displayed in the *Console*. The plug-in also supplies the user with an option to filter rules, this feature worked for filtering by rule name and script name. It displayed nothing when an invalid name was entered, which was correct as the filter wouldn't identify the incorrect input.

The *perspective* and views all appeared and worked correctly. The available views included a *Console*, *Outline*, *Problems* and *Navigation*. Another feature of the IDE was the creation of a relationship between the declared Java class and the Byteman rule it was referenced in. This correctly opened the Java class for the highlighted Java element.

7.2 Integration Testing

Integration testing combined the unit tests to evaluate if the components can execute collectively. This will identify if the system performs properly at a higher level than unit testing, providing the user with more functionality. These tests also move towards more correct system use, providing an insight into actual user test cases.

The editor successfully provided the user with all the correct features. It provided such features as template generation and highlighting. Combined editor features executed correctly and no errors were found.

The plug-in provided the combination of installation and submission successfully. Their additional features also provided the user with extended information and executed functionally. This demonstrated evidence of the successful integration of Byteman with Eclipse.

All of the features that were combined and tested executed accurately and with no errors. This demonstrated evidence of the system working correctly for user case tests.

7.3 System Testing

The combination of all components executed correctly. The system displayed no errors or faults when fully tested with user case tests. The test cases employed rule scripts from real life scenarios. The integration tests were combined with no issues, ensuring easy and simple use for the user. This demonstrated evidence that the plug-in was functional and executed as expected.

8 Evaluation

This section evaluates the prototype against the aim and objectives of the project. This identifies if the proposed objectives of the system were met and what advantages the system provides. The depth in which the feature of the system met the objectives will be evaluated. This section will also specify the advantages the project provided and the progress within the subject field it made.

8.1 Discussion

The aim of the project was to develop a system to provide integration of Byteman with an IDE. Discussion and analysis of the objectives in Section 1.4 will identify if they have achieved the aim of the project.

The IDEs were compared showing the advantages and disadvantages of each, identifying that Eclipse would be used for the project. This decision allowed simpler and quicker implementation of the prototype. It also aided the design and the choice of technologies used. The choice of Eclipse was advantageous and improved the quality of the final plug-in. This meant that objective one was met, which was essential for the project's success.

Research was continued into current technologies, this was for objective two. The initial development stage researched and analysed Byteman. This included how it works, what it is used for, how to use it, its syntax, its limitations and practical uses. IDEs and plug-ins, which provide language support, were analysed. These provided features to be used within the system. This also involved investigating plug-in development and the possible software to be used. This meant objective two was met.

Design of the low level components such as the grammar and parser were completed. This included the Byteman and Java patterns for the grammar rules. As a result, objective three was met.

The design section uses the research in this project to include or exclude features. The design section shows a comprehensive and extensive design of each feature, therefore meeting objective four.

Using the design from objective three, objective five was met. A grammar was developed from scratch, which described patterns for both Byteman and Java. The parser used these to enforce constraints within the editor.

The plug-in provides a structured editor, which is customised for Byteman rule scripts. It uses the grammar from objective five to provide validation of input. Error highlighting is provided, which is advantageous to the user as it supports error detection in the editor. It also decreases the time taken to locate an error. These features of the editor achieve objective six. The editor additionally provides code completion, error markers and messages, template generation, *perspectives*, views and information displayed in the *Console*. This completes the seventh objective.

The eighth objective was to provide a relationship between the Byteman rule and the Java class it references. This was achieved through a menu item, which will open the referenced class. The advantage of this was that it allows the user to validate classes and methods they have referenced, quickly and simply. They can also validate that the behaviour they are trying to induce is correct and relevant. This ensured objective eight was met.

The installation and submission of the Byteman agent and rules was successfully implemented. Additionally further features were implemented to view, uninstall and filter the rules. This objective was achieved as well as including additional functionality. This meant that objective nine was met.

The final advanced objective was partially achieved in that it was researched and designed. However further work could be carried out in future (see Section 9.4).

The aim of the project was achieved, providing a functional prototype. This provides Byteman development support, integrated into Eclipse. The prototype offers a wide range of features for Byteman users, which weren't available before this project.

9 Conclusion

The project aimed to integrate development support for Byteman with an IDE, which was achieved. The project helped to create and expand research into this area as well as providing crucial designs and a resultant prototype. Initially the project was heavily research based, defining the tools to use and system features. Tool definition included research into a collection of libraries available for plug-in development. The features chosen provide a wide scope of support and met the objectives.

9.1 Achievements

The most prominent achievement of the project was that it solves the original problem by providing a wide range of support. This was provided through the multitude of features designed and implemented in the system.

All the objectives were met except the implementation of one of the advanced features, although it was researched and designed. This was an achievement, which provided an expansive solution to the problem. It has provided a solution which is simple and quick to use. Installation of Byteman is easy, the user is provided with two *JAR* files which only require copying into the plug-in folder within the Eclipse application. Then after Eclipse is restarted the plug-in will be available. This provides a system which is portable and easy to deploy. It is also simple to use as it provides users with prompts and success and error messages.

The project has provided research and analysis into a new subject area for the Byteman language. It has offered innovative designs, which have been implemented to produce a prototype, displaying the work undertaken by the project.

The testing and results have provided evidence of a functional system, which provides accurate features from the design. The results show the system's ability to deal with user errors, which demonstrates a robust and well-formed prototype that meets the aim of the project.

The project removes the limitations of Byteman, which is achieved through a number of features, one of these is a structured editor. The system also provides the integration of Byteman execution into an IDE, which makes the use of a command line redundant. It offers a relationship between Java classes and the Byteman rule, which contains them. This allows the relevant class to be opened from the rule, which is a huge advantage when you have thousands of classes.

This project covered an extensive scope of research and design work to produce the final product. It was necessary to generate analysis into a new subject area. The project is functional, robust and provides a significant amount of features to support developers with the creation of Byteman rules within the Eclipse IDE. This has never been accomplished before and will provide new and current developers with a variety of features in an IDE for Byteman.

9.2 Problems

An initial problem was the absence of previous research into IDE support for Byteman. This was counteracted by initiating research into alternative IDE language support and into Byteman itself. These two directions of research provided am insight into the capabilities and limitation of Byteman, and possible practical features for the plug-in. Combined the features were analysed to identify which would be appropriate to solve the limitations that were apparent in Byteman, thus generating original research into the problem area, and reviewing it to generate an innovative solution.

The project presented a steep learning curve for the developer. The initial programming language that had to be mastered was Byteman, as this was the prominent feature of the system. This was essential for testing, so the features of the plug-in could be analysed. Therefore all practical use Byteman was made more difficult by the limitations previously stated in section 1.2. The lower level components required implementation using a tool called ANTLR to generate a grammar for the editor. This was difficult, as the developer had never used a grammar and parser before. This necessitated knowledge of the syntax and possible options for the grammar rules. The ANTLR tooling was unlike any programming language the developer had used before. The project also used languages the user had never used before, for different features of the plug-in. The solution to this involved research and trial and error. The developer had never used plug-in development either. This is a huge task due to the different libraries and classes available. This problem was solved through time, research and trial and error.

9.3 Learning Outcomes

The project provided a broad scope of new technical knowledge for the developer. It presented new languages, programming skills, techniques, tooling and introduced new concepts. These were advantageous to the developer and aided in improving and developing their skills.

The research involved in this project extended the developers knowledge of Byteman, Eclipse, plug-in design and development and software support for custom languages. This developed their original skills, improving their analytical and reviewing techniques. Throughout the duration of the project independent working, planning and organisation skills were improved and strengthened. The developers design skills were enhanced, which was through the design of the system from lower level components to the higher-level functionality.

This project was undertaken as an industrial placement with a company called Red Hat. The project has given the developer an insight into working on a professional project and writing professional documents. The experience provided them with a chance to use new technology from industry. It also provided them with the chance to create open source software, providing a system for the JBoss and Byteman community to take advantage of.

9.4 Future Work

Due to the time constraints not all of the objectives were fully implemented in the final system. Future work would be to implement a debugger with such features as breakpoints, and code stepping. This research should be integrated with the Byteman compiler and interpreter. Additional future work would also be to refine the current features.

9.5 Concluding Remarks

Overall the project was a success providing functional features, which integrated Byteman into an IDE. This project provides new research into an innovative subject area, which successfully solves the original problem. The project provides a wide and interesting range of research, design and technical aspects, which have been beneficial to the developer. The project achieves the aim, providing a design and implementation of the objectives.

10 Glossary

D

DSL:

G

GUI:

I

IDE:

J

JVM:

P

PDE:

Perspective:

R

RCP:

S

SDK:

11 References

[1] A. Dinn. (2011, 11/03/2011). *Byteman Programmer's Guide*. Available: dpwnloas.jboss.org/byteman/1.5.1/ProgrammersGuideSinglePage.html

[2] A. Dinn. (2011). *Flexible, Dynamic Injection of Structured Advice using Byteman*. Available: http://delivery.acm.org/10.1145/1970000/1960325/p41-dinn.pdf?ip=128.240.229.65&CFID=36046115&CFTOKEN=88806712&__acm__=1311241502_9d4ce5dce044cebf00f91b990206fdde

[3] I. IDEA. (2011, 11th April 2011). *Using IntelliJ IDEA for Eclipse RCP Development*. Available: www.jetbrains.com/idea/documentation/usingIDEAforEclipse.html

[4] Eclipse. (2011, Accessed: 11th April 2011). *Eclipse Documentation*. Available: http://help.eclipse.org/helios/index.jsp

[5] L. Vogel. (12th March 2011, 13th April). *Eclipse RCP Tutroial*. Available: www.vogella.de/articles/EclipseRCP/article.html

[6] L. Vogella. (28th March 2011, Accessed: 15th April 2011). Eclipse Commands Tutorial. Available: www.vogella.de/articles/EclipseCommands/articles.html

[7] P. Builder. (2008, Accessed: 16th April 2011). *What is Plugin Builder*. Available: http://www.pluginbuilder.org/

[8] W. M. a. D. Glozic. (8th September 2003, 23rd April 2011). *PDE Does Plug-ins*. Available: www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html

[9] L. Vogel. (20th September 2010, Accessed: 14th April 2011). Eclipse Plugin Development Tutorial. *Version 1.6*. Available: www.vogella.de/articles/EclipsePlugin/articles.html

[10] DZone. (2011, 1st June 2011). *EclipseZone*. Available: www.eclipsezone.com

[11] E. P. Development. (2008, 20th April 2011). *Introduction to Builders, Natures and Markers*. Available: www.eclipsepluginsite.com/builders-natures-markers.html

[12] O. S. Maxim Shafirov, Kieth Lee, Sascha Weinreuter. (11th April 2011). *Developing Custom Language Plugins for IntelliJ IDEA*. Available: www.jetbrains.com/idea/documentation/idea_5.0.html

[13] D. Jemerov. (20th October 2010, 24th April 2011). *Getting started with plug-in development*. Available: http://confluence.jetbrains.net/display/IDEADEV/Getting+Started+with+Plugin+Development

[14] D. Jemerov. (2011, 23rd April 2011). *Developing Custom Language Plugins for IntelliJ IDEA*. Available: http://confluence.jetbrains.net/display/IDEADEV/Developing+Custom+Language+Plugins+for+IntelliJ+IDEA

[15] C. Aniszczyk. (12 Feburary 2008, Plug-in development 101, Part 1: The fundamentals. *Learn the basics of Eclipse plug-in development*. Available: www.ibm.com/developerworks/library/os-eclipse-plugindev1/index.html?ca=dgr-eclipse-1

[16] D. W. a. B. Freeman-Benson. (August 27th 2004, 14th April 2011). How to Write an Eclipse Debugger. Available: http://www.eclipse.org/articles/Article-Debugger/how-to.html

[17] A. P. Andrel Sobolev, and Jeff Norris. (2010, 23rd April 2011). *DLTK IDE Guide: Step 1 Skeleton*. Available: http://wiki.eclipse.org/DLTK_IDE_Guide:Step_1_Skeleton

[18] J. N. a. A. P. Andrel Sobolev. (2009). *DLTK IDE Guide:Step 2. Towards an Editor*. Available: http://wiki.eclipse.org/DLTK_IDE_Guide:Step2._Towards_an _Editor

[19] C. Analytics. (2011, 23rd April 2011). *DSL Editor*. Available: www.certiv.net/projects/dsl-editor.html

[20] Eclipse. *XText User Guide*. Available: www.eclipse.org/Xtext/documentation/1_0_1/xtext.html

[21] O. A. Wear. Accessed: 20th June 2011). Check / Xtend / Xpand Reference. Available: http://www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html

[22] Eclipse. (2010, 23/04/2011). *PHP Development Tools Project*. Available: www.eclipse.org/pdt/

[23] Oracle. (2011, Accessed: 14th April 2011). *NetBeans IDE 7.0 Features*. Available: http://netbeans.org/features/cpp/

[24] N. Ford. (24 Jul 2008, Accessed: 15th April 2011). Using the Ruby Development Tools plug-in for Eclipse. Available: http://www.ibm.com/developerworks/opensource/library/os-rubyeclipse/

[25] M. Bozeman, "Extending an Eclipse Embedded Debugger," *White Paper*.

[26] J. Szurszewski. (8th January 2003, Accessed: 01/08/2011). We Have Lift-off: The Launching Framework in Eclipse. Available: http://www.eclipse.org/articles/Article-Launch-Framework/launch.html

[27] T. Parr. (2010). *ANTLR: FAQ Getting Started*. Available: www.antlr.org/wiki/display/ANTLR3/FAQ+-+Getting+Started

[28] Eclipse. (2011, 11th April 2011). *Eclipse Documentation*. Available: http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.cdt.doc.isv/guide/projectTemplateEngine/Howtoregistertemplates.html

12 Appendix

Appendix A

Appendix B

Byteman Plug-in User Guide

Contents

1. Installing the plug-in

1.1 Pre-requisites

To use the plug-in the following prerequisites are necessary:

- A Java 5 or 6 SDK

- Eclipse SDK 3.3 or higher.

1.2. Installation

To install the plug-in firstly copy the two plug-in jars called: "`ncl.ac.uk.byteman`" and "`ncl.ac.uk.byteman.ui`". These two files then need added to the Eclipse plug-ins folder in the application's root directory, as shown below in figure one. Eclipse should then be restarted, the plug-in features should now be available for use.

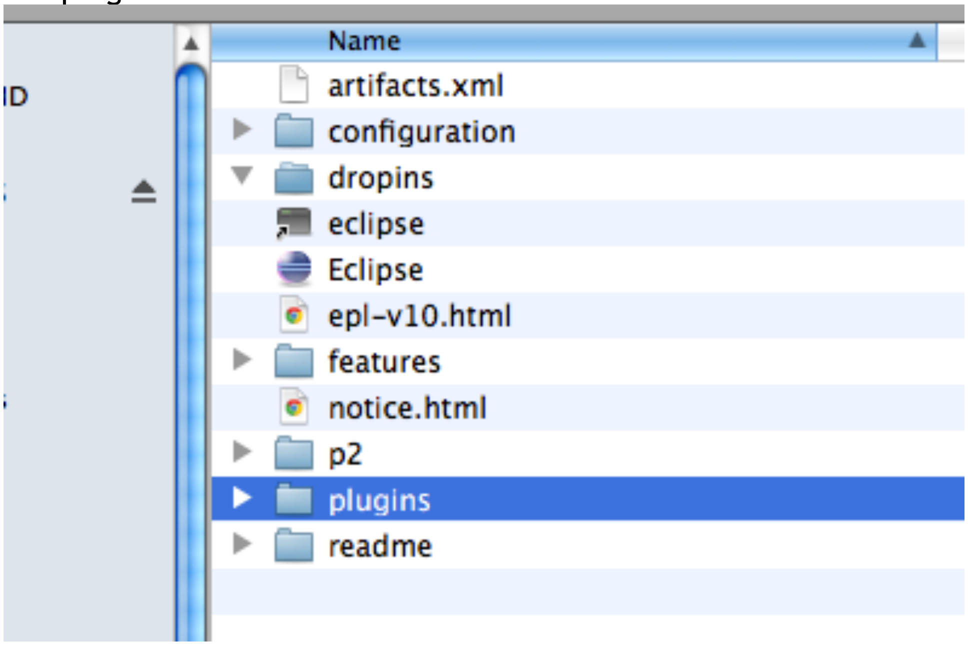
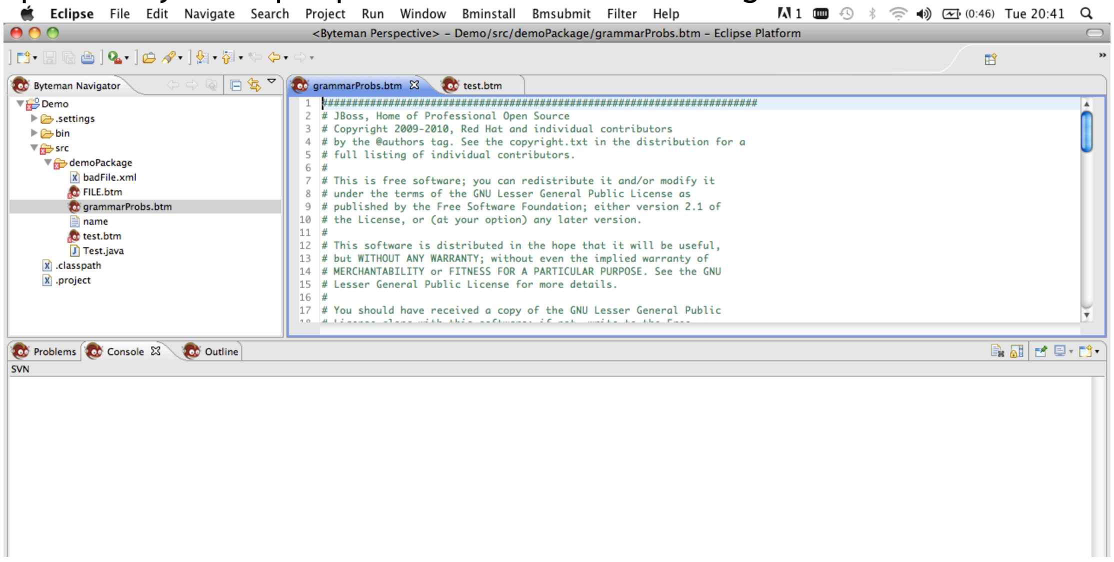


Figure 1: Plug-in folder for Eclipse.

2 Starting the plug-in

2.1 Opening the perspective

Once the plug-in has been installed the user should open the "Byteman Perspective", they can do this by going to **Window>>Open Perspective>>Other...>>Byteman Perspective**. The screen should then open the Byteman perspective as shown below in figure two.



```
######################################################################
# JBoss, Home of Professional Open Source
# Copyright 2009-2010, Red Hat and individual contributors
# by the @authors tag. See the copyright.txt in the distribution for a
# full listing of individual contributors.
#
# This is free software; you can redistribute it and/or modify it
# under the terms of the GNU Lesser General Public License as
# published by the Free Software Foundation; either version 2.1 of
# the License, or (at your option) any later version.
#
# This software is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
# Lesser General Public License for more details.
#
# You should have received a copy of the GNU Lesser General Public
```

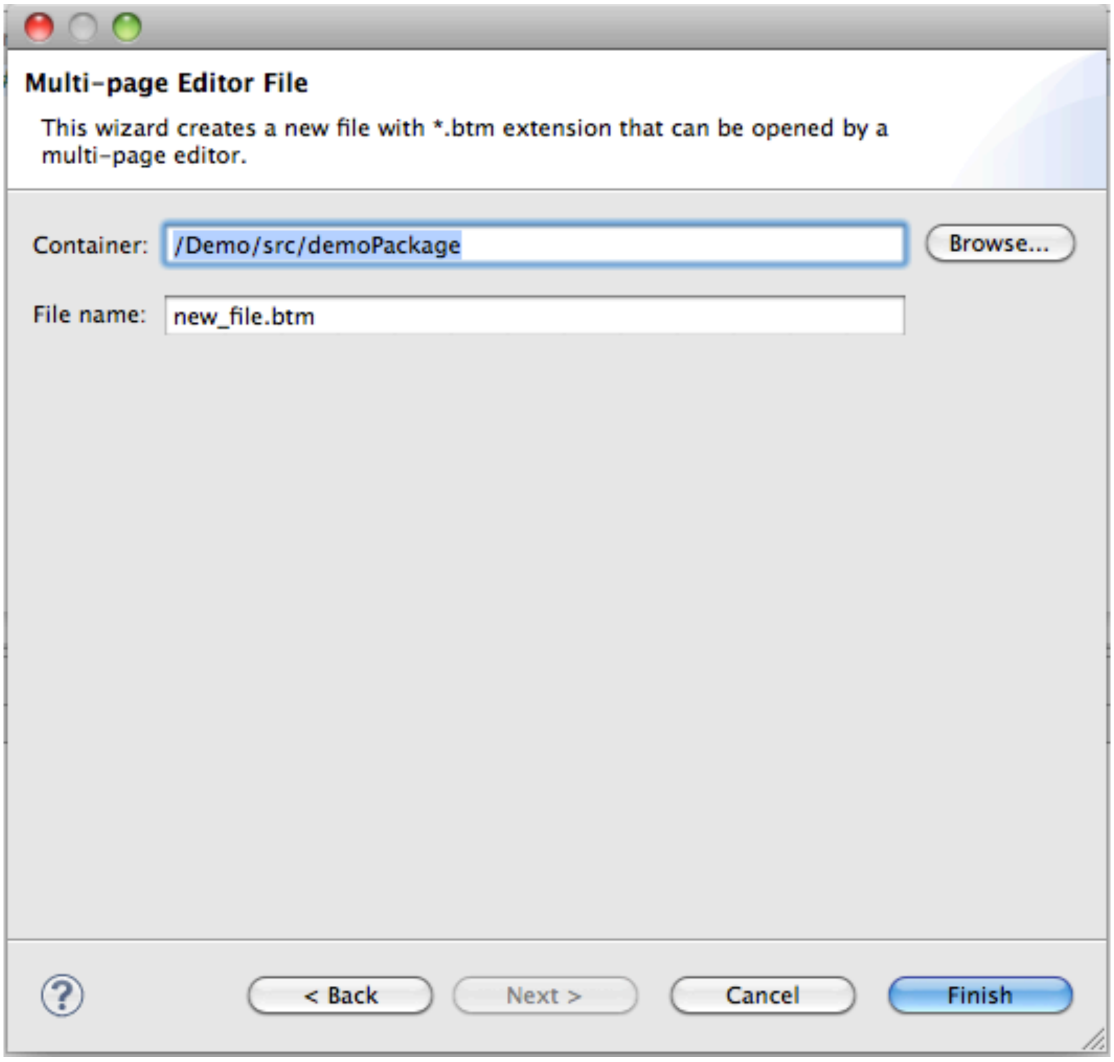
Figure 2: The Byteman Perspective.

2.2 Opening Views

The Byteman perspective provides the user with a set views for the perspective, these views include: *Outline*, *Console*, *Byteman Navigation* and *Problems*. To open one or more of these go to: **Window>>show view>>Other..>> <Name of the chosen view>**. Shown in figure two are screen shots of the different views available.

2.3 Opening a new .btm File

Finally the new `.btm` file must be opened in the editor for development to start. This is done by going to: **File>>new>>other>>Byteman Wizard>>Byteman File**. This will automatically open a wizard. The user must enter the name of the package they wish the file to go in and the name of the file, as shown below in figure three. This will then automatically generate a `.btm` file with keywords for a Byteman skeleton printed, as shown in figure four. The keywords within the rule file will be automatically highlighted to provide the user with syntax highlighting as they continue to program.
NB: enter all the Rule details before writing the comments, otherwise the comments may not be recognised and may be treated as an error.

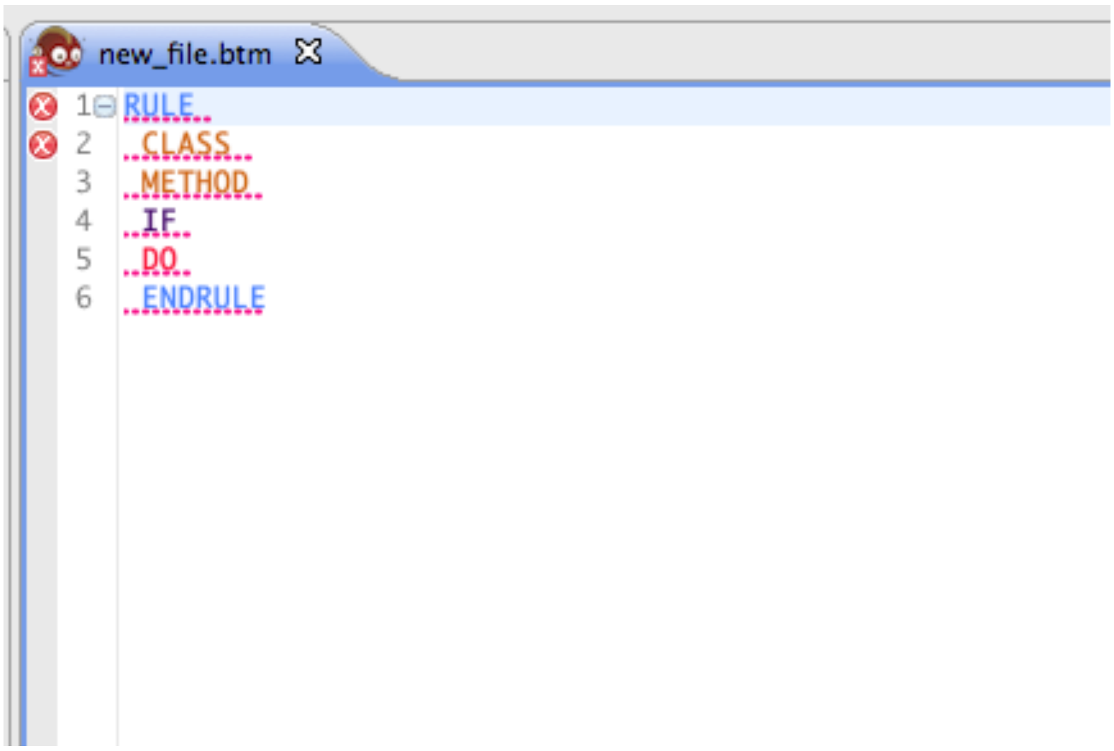

Multi-page Editor File

This wizard creates a new file with *.btm extension that can be opened by a multi-page editor.

Container: /Demo/src/demoPackage

File name: new_file.btm

Figure 3: The wizard for Byteman template generation.



```
RULE
  CLASS
  METHOD
  IF
  DO
  ENDRULE
```

Figure 4: The template Byteman rule.

3 Features in the Editor

3.1 Content Assist

To use content assist the user must be in the correct location in the code for a list to be displayed. Then the user must press CTRL + SPACE, this will then display a drop down menu for the user. The menu will have options to complete that code element, to use an option use the mouse and click on the appropriate menu item. This is shown in figure five below.

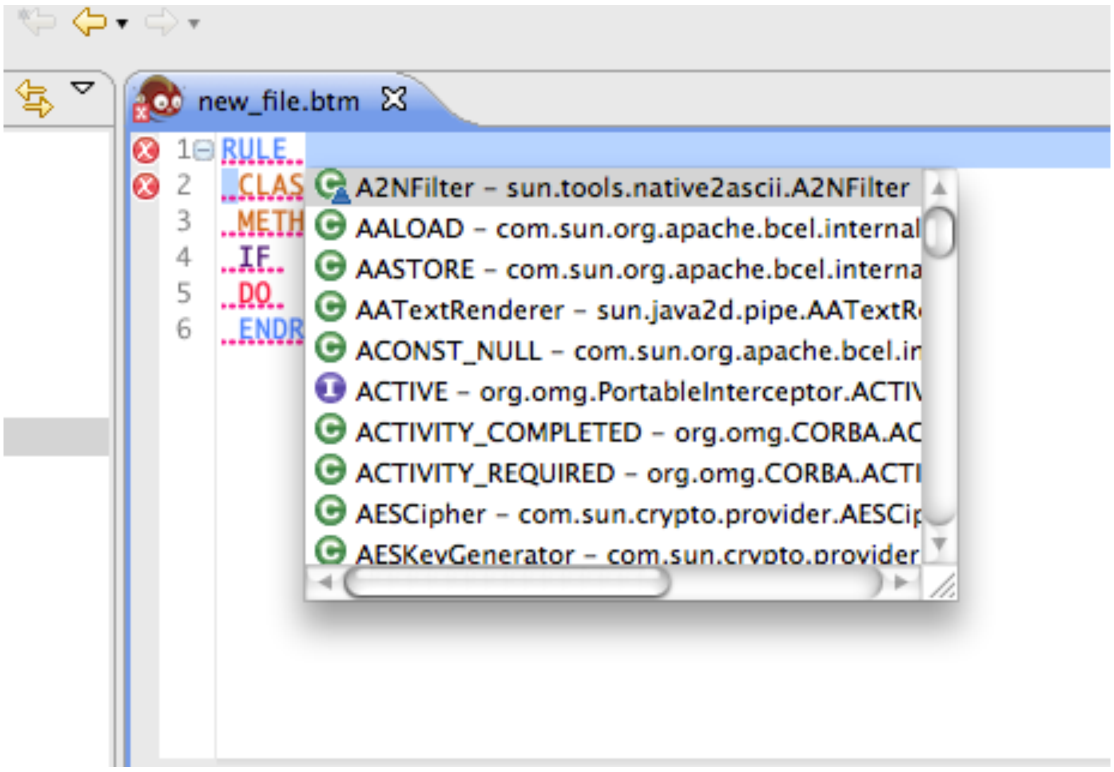

new_file.btm
RULE
CLAS
METH
IF
DO
ENDR
A2NFilter – sun.tools.native2ascii.A2NFilter
AALOAD – com.sun.org.apache.bcel.internal
AASTORE – com.sun.org.apache.bcel.interna
AATextRenderer – sun.java2d.pipe.AATextR
ACONST_NULL – com.sun.org.apache.bcel.in
ACTIVE – org.omg.PortableInterceptor.ACTIV
ACTIVITY_COMPLETED – org.omg.CORBA.AC
ACTIVITY_REQUIRED – org.omg.CORBA.ACTI
AESCipher – com.sun.crypto.provider.AESCip
AESKeyGenerator – com.sun.crypto.provider

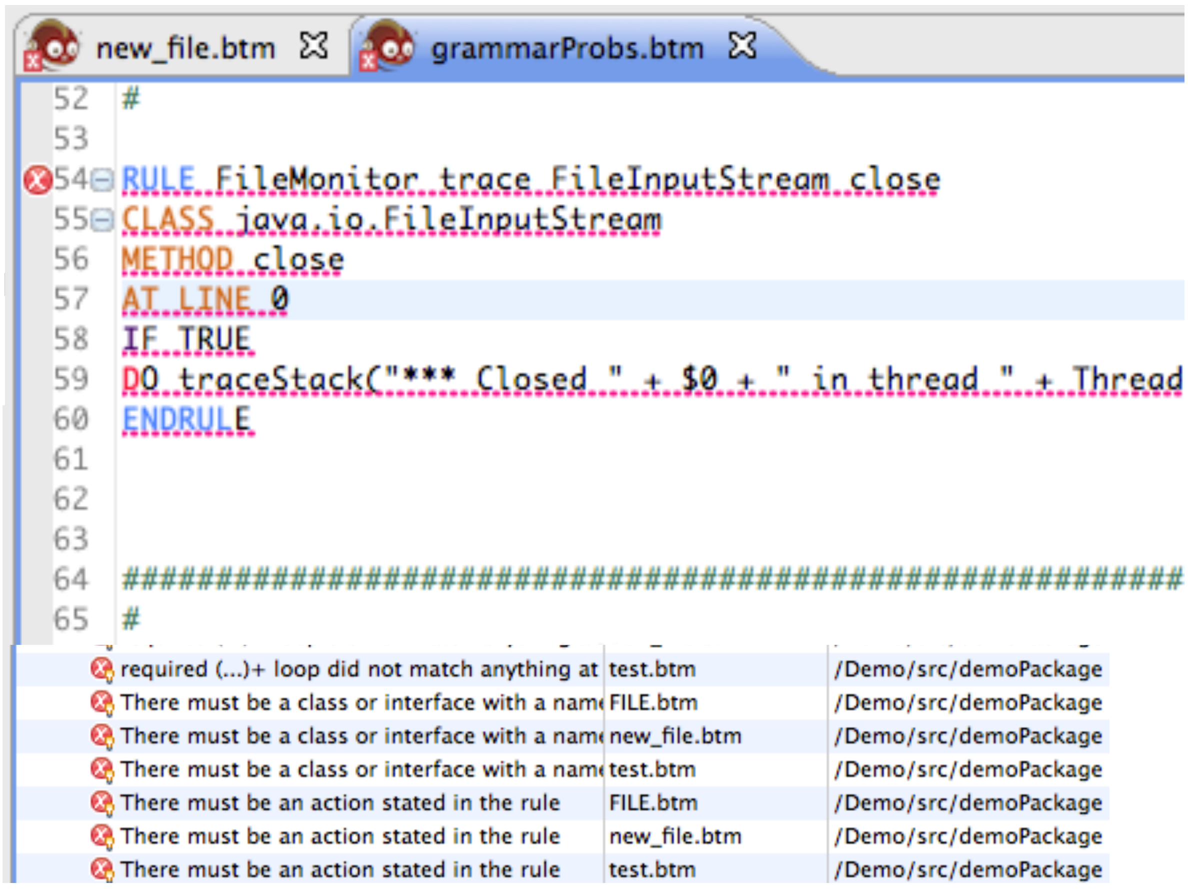
Figure 5: A list for content assist.

3.2 Error Messages

There are two methods available for error detection, they are:
1. Through the use of the editor.
2. Through the use of the *Problem* view.

To view the error within the editor the user just needs to go to editor ruler. There an error marker will be present. There will also be highlighting specifying the location of the error. If the user hovers over the error marker an error message will be displayed. This is shown in figure six.

The second method is the use of the *Problem* view, to initialize this use section 2.2. Then to bring the tab forward click the top of it. The user should then scroll down identifying the relevant errors. This is shown in figure seven.



new_file.btm grammarProbs.btm

```
#

RULE FileMonitor trace FileInputStream close
CLASS java.io.FileInputStream
METHOD close
AT LINE 0
IF TRUE
DO traceStack("*** Closed " + $0 + " in thread " + Thread
ENDRULE



##########################################################
#
```

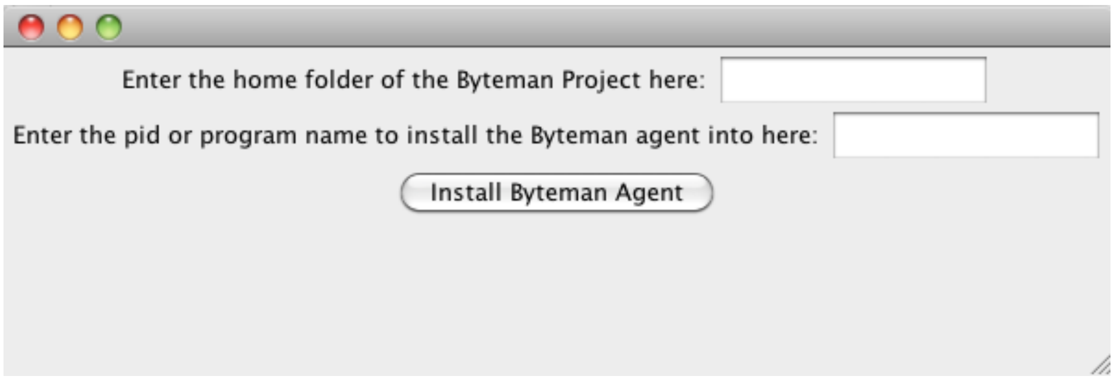
```
required (...)+ loop did not match anything at  test.btm       /Demo/src/demoPackage
There must be a class or interface with a name  FILE.btm       /Demo/src/demoPackage
There must be a class or interface with a name  new_file.btm   /Demo/src/demoPackage
There must be a class or interface with a name  test.btm       /Demo/src/demoPackage
There must be an action stated in the rule      FILE.btm       /Demo/src/demoPackage
There must be an action stated in the rule      new_file.btm   /Demo/src/demoPackage
There must be an action stated in the rule      test.btm       /Demo/src/demoPackage
```

Figure 7: The problems view.

4 Added Features to the IDE

4.1 Installing the Byteman Agent

To install the Byteman agent the user must go to **Bminstall>>Install Rule**. They will then be presented with a GUI interface allowing them to enter the Byteman home and the process id or program name that the agent will be install into. The Byteman home path is the path to the folder for the Byteman project. The process id can be found by using the activity monitor or the name of the Java program. The GUI is displayed in figure eight. The user should then click the “Install Byteman Agent” button, if it has been installed a dialog will appear to inform the user.



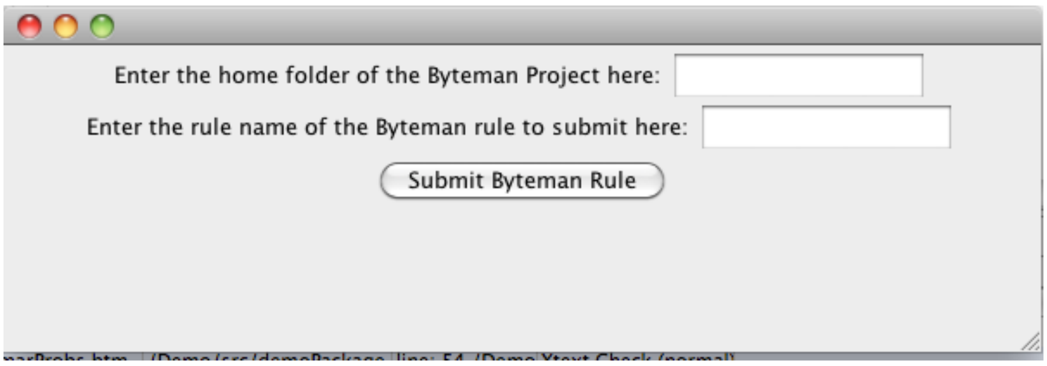
Enter the home folder of the Byteman Project here:

Enter the pid or program name to install the Byteman agent into here:

Figure 8: Install GUI.

4.2 Submitting a Byteman Rule

To submit a Byteman rule the user must first go to: **BMSubmit>>Submit Rule**. They will then be shown a GUI interface where they must enter the Byteman home path (as describe in section 4.1). They must then ether the absolute path of the `.btm` file they wish to install rules from. The output from this action will be displayed in the console. Figure nine displays the GUI.



Enter the home folder of the Byteman Project here:

Enter the rule name of the Byteman rule to submit here:

Figure 9: Submit GUI.

4.3 Viewing All Rules

The user must go to: **Bmsubmit>>View Rules**, the rules that are loaded will then be displayed in the *Console*.

4.4 Deleting All Rules

To delete all the rules the user must go to: **Bmsubmit>>Delete All Rules**, the deleted rules will then be displayed in the *Console*.

4.5 Deleting Selected Rules

To delete a set of rules of the users choice the user must go to: **Bmsubmit>>Delete Selected Files**. The user will then be presented with a GUI, as shown in figure ten. The user must enter the absolute path of the rule they wish to delete into the presented text field. Then they need to click the "Delete rule/s" button. This should then delete the specified rules, the output being displayed in the *Console*.
NB: The file paths must be entered with a comma "," in between the names for the system to find the correct file path e.g. "filepath,longfilepath" .

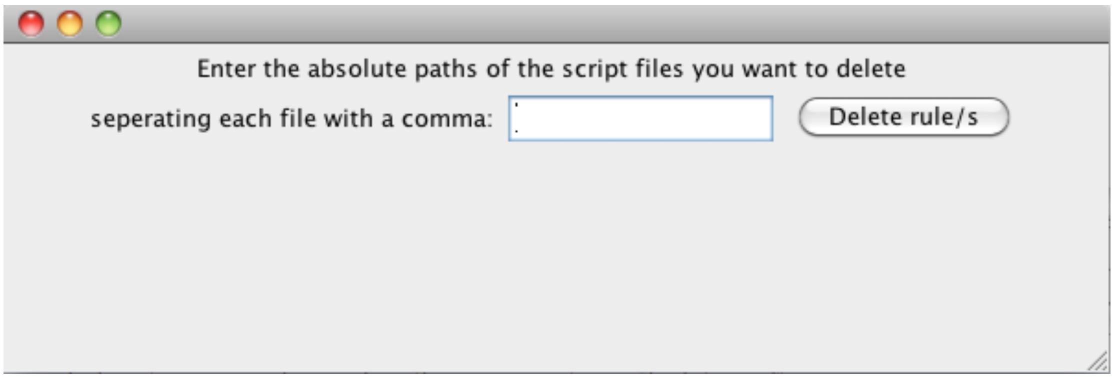

Enter the absolute paths of the script files you want to delete
seperating each file with a comma:
Delete rule/s

Figure 10: Delete GUI.

4.6 Filtering Injection State by Script Name

To filter all the rules that have been loaded the user must go to: **Filter>>by script/file name**. This will then present a GUI, displayed in figure eleven. The user must enter the name of the file or rule they wish to display information about. Then click on the "search" button. The filtered output will be displayed within the *Console*.
NB: The name of the file is just the relative path e.g. `name.btm`.

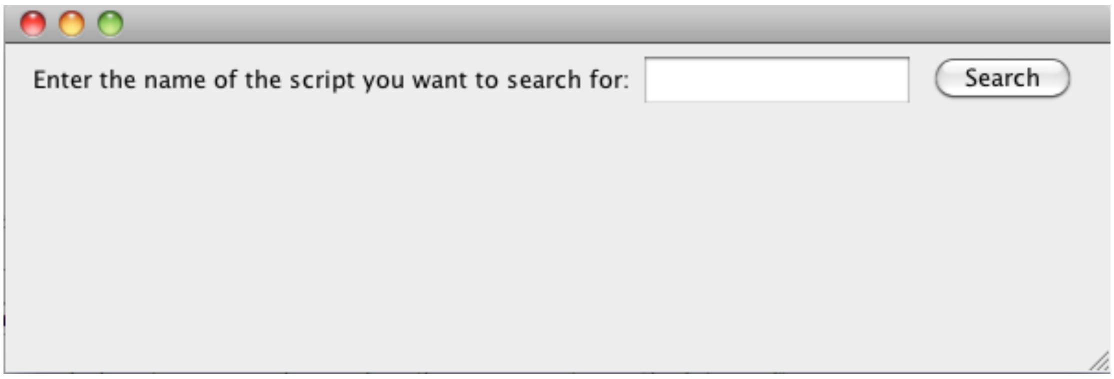


Figure 11: The filter GUI.

4.7 Filtering Injection State by Rule Name

To filter the Rules by name then the user must go to: **Filter>>by rule name**. This will then display a GUI, as shown in figure twelve. The user must then enter the name of the rule into the text field and press the “Search” button. The rule will then be displayed in the *Console*.
NB: The name of the rule must be the one specified within the Byteman rule e.g. inside the `.btm` file is: RULE rule name, then the user would enter: “rule name”.

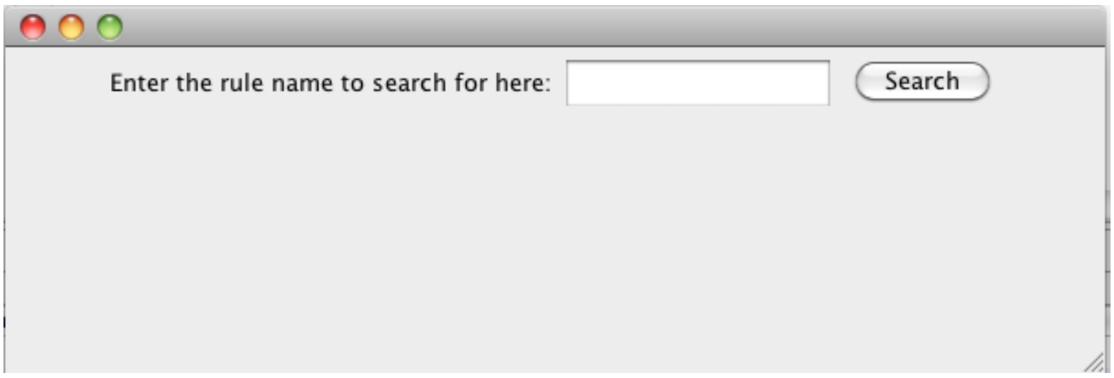


Figure 12: The filter GUI.