FMSL Plug-in for the Eclipse Platform

Jesse Englert

Computer Science Department

California Polytechnic State University, San Luis Obispo

June 13, 2006

Abstract

The Formal Modeling Specification Language (FMSL) is a programming language used by students in software engineering courses. The FMSL development process is arduous due to the lack of an Integrated Development Environment (IDE). The problem this senior project solves is how to design an IDE for FMSL. This senior project describes a design that integrates FMSL into the Eclipse Platform.

FMSL is integrated into the Eclipse Platform in the form of a plug-in. The plug-in offers support for an editor with syntax coloring, a content outline view, syntax error markers, and compilation. A proof-of-concept implementation is presented using the Eclipse 3.1 Platform. Since this project is a proof-of-concept, there are many possible improvements to be made.

Contents

1 Introduction **7**

1.1 Problem . . . 7

1.2 Solution . . . 8

1.3 Limitations . . . 8

1.4 Outline . . . 9

2 Scenario **10**

3 Background Information **18**

3.1 Eclipse Architecture Overview . . . 18

3.1.1 Platform runtime and Plug-in Architecture . . . 18

3.1.2 Workspaces . . . 18

3.1.3 Workbench and UI Toolkits . . . 19

3.2 Eclipse Language Integration Issues . . . 20

4 Architecture and Design **23**

4.1 Plug-in Architecture . . . 23

4.2 Parsing . . . 25

4.2.1 SableCC Parser Generator . . . 25

4.2.2 Parser Manager . . . 29

4.3 Building . . . 31

5 Usage Guide **34**

5.1 End Users . . . 34

5.2 Developers . . . 34

5.2.1 Eclipse Requirements . . . 34

5.2.2 CVS Checkout . . . 34

5.2.3 Debugging . . . 35

5.2.4 Deployment . . . 36

5.3 Testing . . . 36

5.3.1 JUnit Unit Testing . . . 36
5.3.2 Manual Testing . . . 37

6 Related Work **39**
6.1 Texlipse Plug-in . . . 39
6.2 Java Development Tool . . . 39

7 Conclusions and Future Work **43**

A Acronyms **44**

B Java Development Tool (JDT) Features **45**

C Requirements **48**
C.1 Overall Description . . . 48
C.1.1 Product Perspective . . . 48
C.1.2 Product Features . . . 48
C.1.3 User Classes . . . 48
C.1.4 Operating Environment . . . 48
C.1.5 Constraints . . . 49
C.1.6 User Documentation . . . 49
C.1.7 Assumptions and Dependencies . . . 49
C.2 UI Overview . . . 50
C.2.1 Workbench . . . 50
C.3 Text Editor . . . 51
C.3.1 Syntax Highlighting . . . 51
C.3.2 Automatic Indentation . . . 52
C.3.3 Problem Markers . . . 52
C.3.4 Content Assist . . . 53
C.3.5 Code Folding . . . 53
C.3.6 Line Level . . . 54
C.3.7 Quick Fix . . . 54

C.3.8 Text Hover . . . 54
C.4 Outline View . . . 55
C.4.1 Structure . . . 55
C.4.2 Linking . . . 56
C.5 Project View . . . 56
C.5.1 File Hierarchy . . . 56
C.6 Preferences Dialog . . . 57
C.6.1 Editor Preferences . . . 57
C.6.2 Builder Preferences . . . 57
C.7 Wizards . . . 58
C.7.1 New Project Wizard . . . 58
C.8 Non Functional Requirements . . . 58
C.8.1 User Interface . . . 58
C.8.2 Software Interface . . . 59
C.8.3 Delivery . . . 59
C.8.4 Installation . . . 59

List of Figures

1 New Project Wizard . . . 10
2 New File Wizard . . . 11
3 Properties . . . 12
4 Builder Preferences . . . 13
5 Editor Preferences . . . 14
6 Syntax Preferences . . . 15
7 Editor . . . 16
8 Content Outline View . . . 16
9 Editor and Problem View After Parsing . . . 17
10 Editor and Problem View After Build . . . 17
11 Eclipse Platform Architecture [2] . . . 19

12 Eclipse Platform Workbench [2] 21
13 Context Diagram 23
14 Plug-in Architecture 24
15 How the fmsleclipse plug-in extends the Eclipse Platform 24
16 Builder Data Flow Diagram 32
17 Builder UML Diagram 33
18 Java Perspective [2] 41
19 JDT Architecture [2] 42
20 Workbench 50
21 Detached View 51
22 Problem Marker 53
23 Content Assist 53
24 Quick Fix 54
25 Text Hover 55
26 Outline View 55
27 Navigator View 57
28 File Structure 58
29 Syntax Highlighting In Editor Preferences 59
30 Builder Preferences 60
31 New Project Wizard 61

1 Introduction

This senior project report documents the research and work performed on the FMSLEclipse project. FMSLEclipse is an Eclipse tool that adds support for the FMSL to Eclipse. Eclipse is a general purpose development platform and application framework for building software.

The intended audience of this report includes Dr. Gene Fisher, students performing further work on this project, and Eclipse plug-in developers. Dr. Gene Fisher plans to continue development of this project for use in his undergraduate software engineering courses at Cal Poly State University, San Luis Obispo. Eclipse plug-in developers may find this report, along with the FMSLEclipse source code, useful as a starting point or reference for integrating a programming language into Eclipse.

1.1 Problem

At Cal Poly State University, San Luis Obispo, most students taking an undergraduate software engineering course are required to learn and use FMSL. FMSL is a programming language used to specify the external structure of a mechanized system and to define the system's requirements. A system is a mixture of hardware, software, and human processing elements.[3]

The empirical way to develop FMSL code on a standard campus supported UNIX system is to write it in VI or EMACS and then compile it at the command prompt. The resulting data dictionary is in HTML and may be viewed by a web browser. This is a tedious and archaic way to write code.

Almost all modern programming languages[1] are supported by at least one IDE. An IDE greatly eases the development process by providing numerous tools that work together. Standard tools included with an IDE are a text editor, compiler, and debugger.

An IDE for FMSL is needed to simplify the development process, lower the learning curve, and raise productivity. The IDE should have functionality found in existing IDEs such as Eclipse and Visual Studio.

Creating an IDE for FMSL is an interesting project for two reasons. First, it has not been done before. Second, such a tool has the possibility of being used by many real students in the

[1] C++, Java, Ruby, and PHP are examples of modern programming languages

class room rather than sitting in the library unused and collecting dust.

1.2 Solution

Several options exist to create an IDE for FMSL. The purist option is to build it from scratch. This is no good because it is outside the scope of a senior project. The simple option is to extend an existing text editor such as VI or EMACS to recognize FMSL. For example, a configuration file exists that extends EMACS to do syntax coloring on FMSL code. The robust option is to integrate FMSL into an extensible development platform. This is the option chosen for the FMSLEclipse project.

We chose to use Eclipse because it is an extensible development platform that is open-source and widely used. It is the most popular Java IDE in the world, which means students are likely to be familiar with it. It is also actively updated and maintained by a large community, so we do not expect it to follow the path of www.pets.com.

The solution takes the form of several plug-ins that extend Eclipse to support FMSL (see section 3). The main features of the plug-ins include syntax coloring, content outline view, error reporting, and compiling. These features are explained in further detail later on.

1.3 Limitations

This project is intended to be a proof of concept and therefore contains several limitations. At the beginning of the project a requirements document was created (see appendix C). Many of the non high priority requirements were not met.

The main limitation is an incomplete parser (see section 4.2 for more information on the parsing architecture). The current parser was created from a reduced grammar and therefore does not support the entire FMSL syntax. This issue will be addressed in a later release.

Several features outlined in the requirements document were not implemented. These include automatic indentation (see section C.3.2), content assist (see section C.3.4)), code folding (see section C.3.5), quick fixes (see section C.3.7), text hover (see section C.3.8), outline view linking (see section C.4.2), and project view (see section C.5).

A following known bugs exist and should be addressed in the future:

- Source files must reside in the source directory specified by the project properties when building a project.
- The output directory specified in the project properties does not contain the output files after a build.
- The FMSL -> View Data Dictionary menu option does nothing.
- Changing the syntax coloring in the preferences dialog does not effect editors that are currently open.

1.4 Outline

The remainder of this report will describe the FMSLEclipse project in detail. Section 2 contains screen shots of the project in a walk-through fashion. Section 3 provides background information on the Eclipse platform. Section 4 details the architecture and design of the software. Section 5 contains information for the different users of this project including end users, developers, and testers. Section 6 summarizes related work done in the area of integrating a language into Eclipse. Lastly, section 7 discusses conclusions and future work possibilities.

2 Scenario

This section contains screen shots of the FMSLEclipse project in action. The screen shots show the life cycle of a simple FMSL project. Figures 1 and 2 show the wizards used to create a new FMSL project and file.

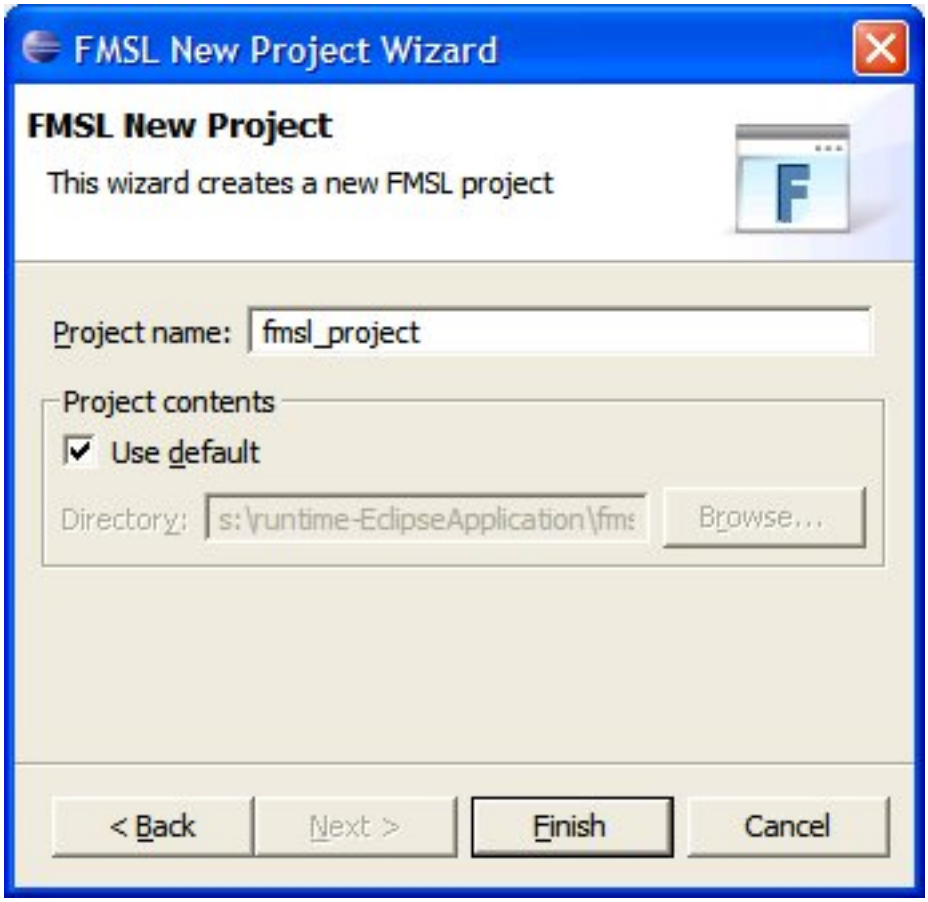

Figure 1: New Project Wizard

The new project wizard requires the user to select a name for the new project. The new project will be placed in the current workspace unless the *Use default* checkbox is deselected and a new project locaton is specified in the *Directory* text field.

The new file wizard requires the user to select a parent folder and a name for the new file. The wizard ensures the file name ends with the .fmsl extension by preventing the user from selecting the finish button until the proper extension is used. The *Advanced* button revealse a checkbox and a text field that allows the user to create a link to a file outside of the current workspace.

Figure 2: New File Wizard

After creating a new project it is important to set the project properties in figure 3. The *Source dir* text field allows the user to enter the name of the directory that holds all of the source *.fmsl* files. The *Output dir* text field allows the user to enter the name of the directory that holds output files generated from building the project. The *Build data dictionary* checkbox determines whether the data dictionary is generated or not during a build.

The plug-in preferences are displayed in figures 4, 5, and 6. The builder page is where the user must specify a path to the FMSL compiler and FMSL doc compiler. Paths to these executables are necessary in order to build an FMSL project. The *Builder Consol Output* preference will display output from the FMSL compilers to the consol if selected.

The editor preference page is where the user can control the parsing behavior. Selecting

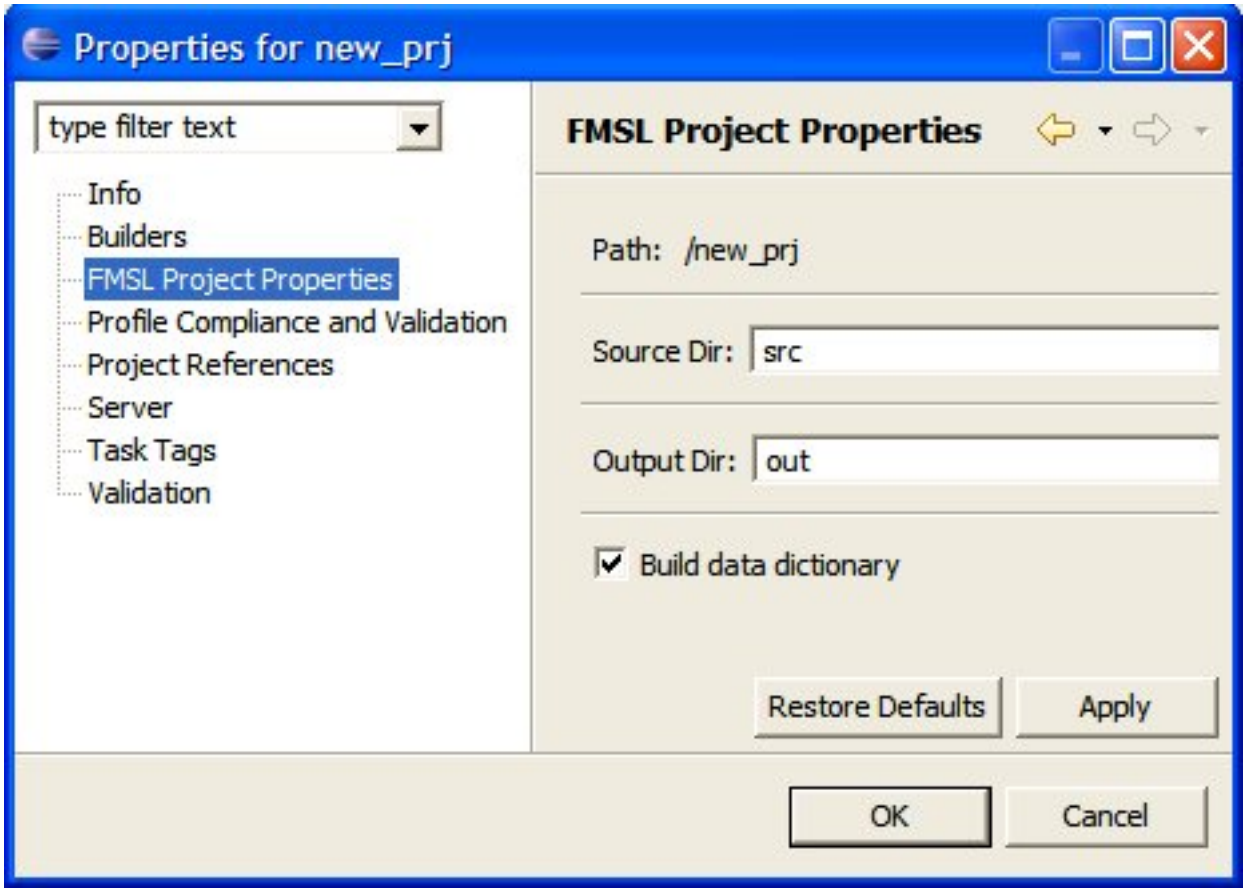

Figure 3: Properties

the *Enable Automatic Parsing* checkbox will ensure the current document is parsed everytime it is changed. The *Automatic Parsing Delay* text field will delay the actual parsing the specified amount of time. It is important to delay parsing so that the document is not parsed immediately after every keystroke.

The content outline view in figure 8 shows an outline of the FMSL document in figure 7. The content outline view is populated only if automatic parsing is enabled.

Lastly, figures 9 and 10 show what happens when there is a syntax error during parsing and building.

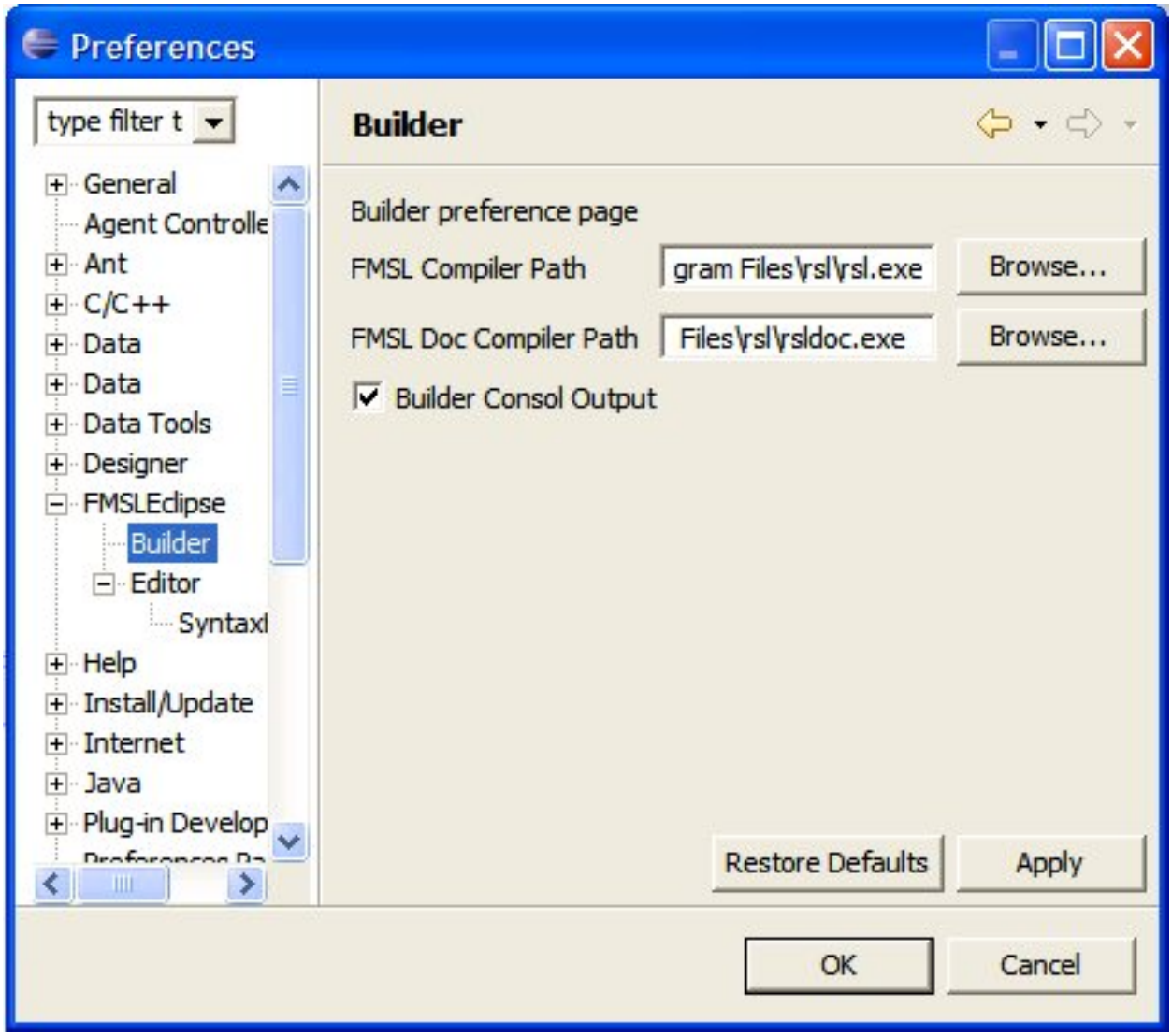

Figure 4: Builder Preferences

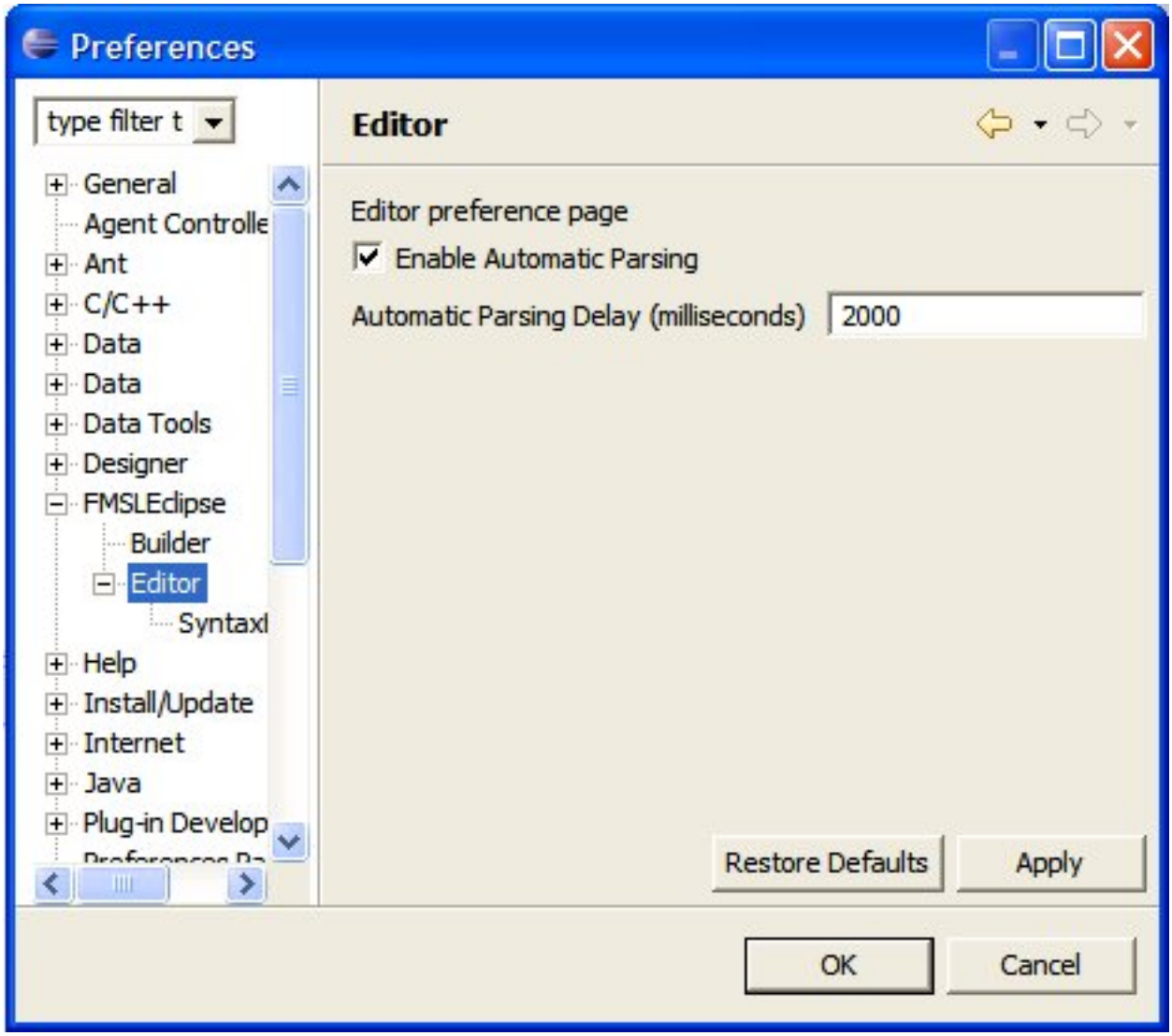

Figure 5: Editor Preferences

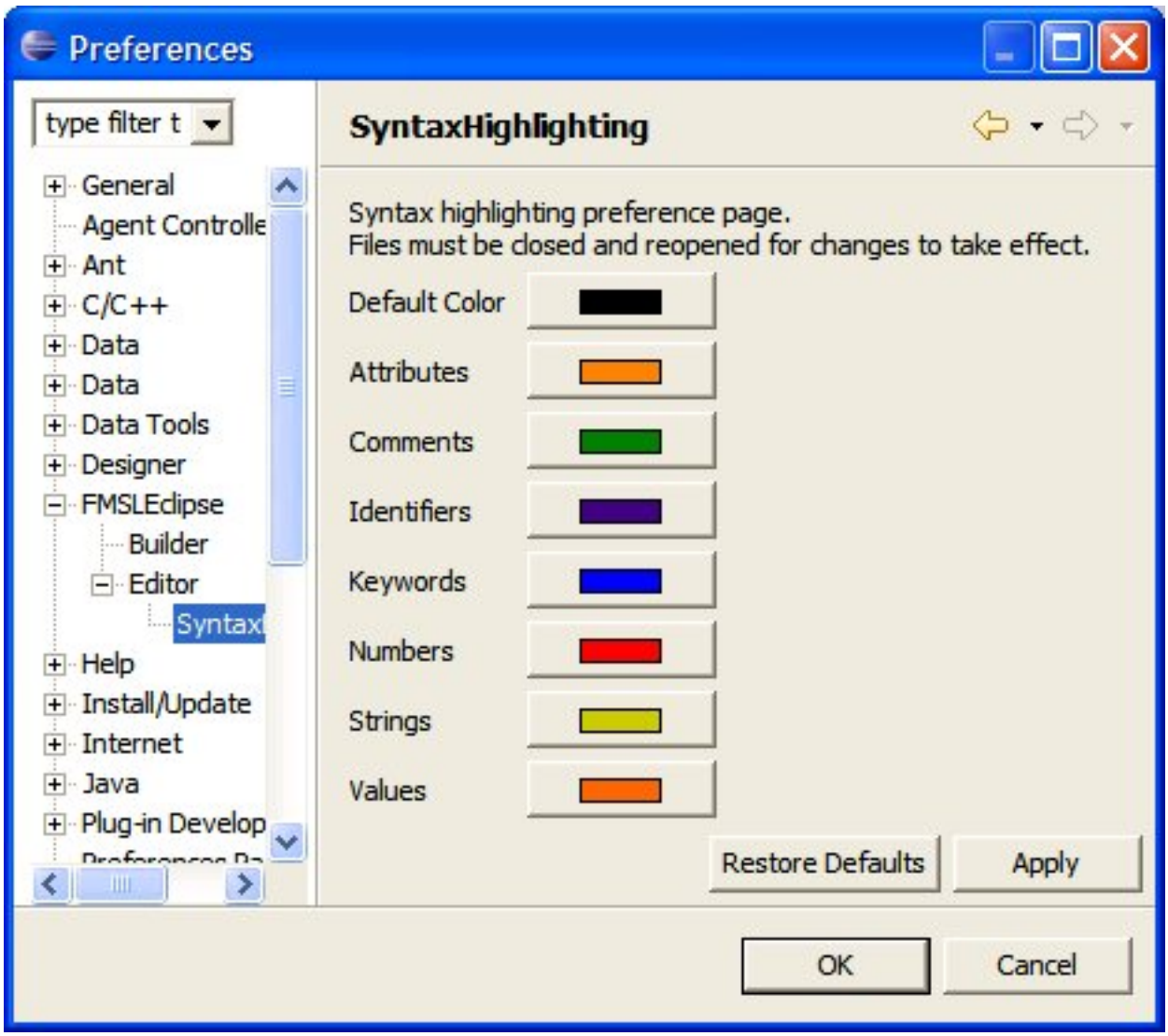

Figure 6: Syntax Preferences

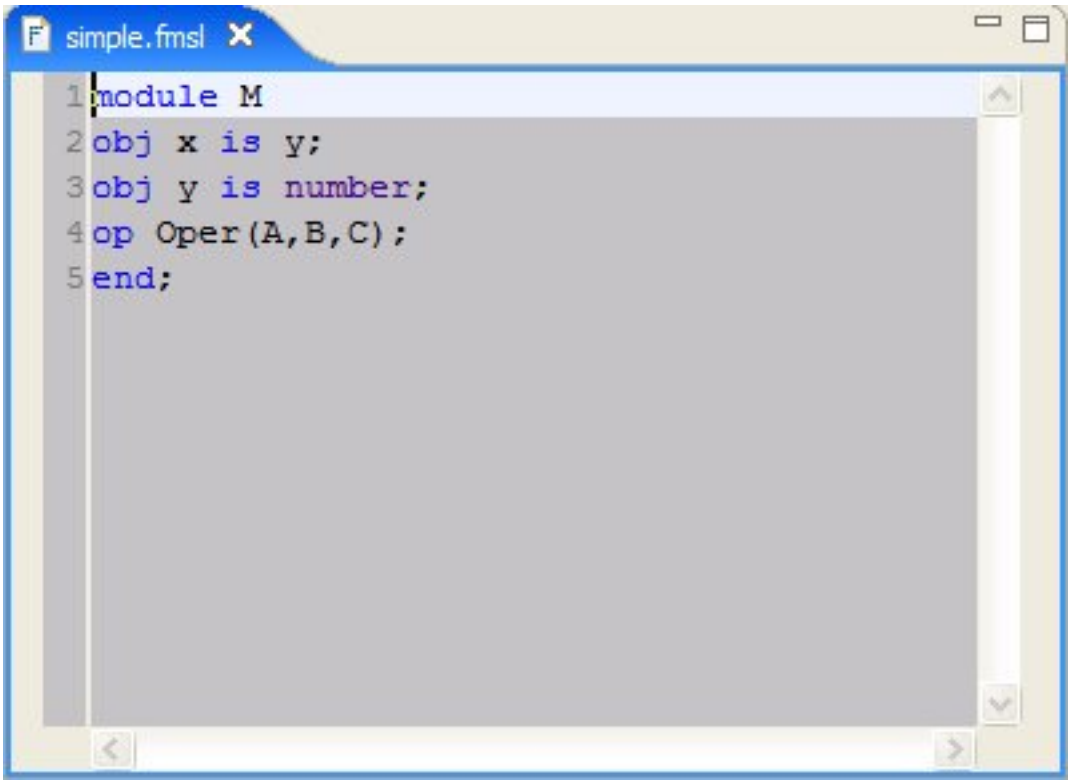

Figure 7: Editor

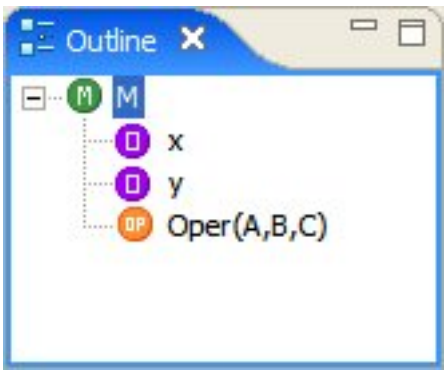

Figure 8: Content Outline View

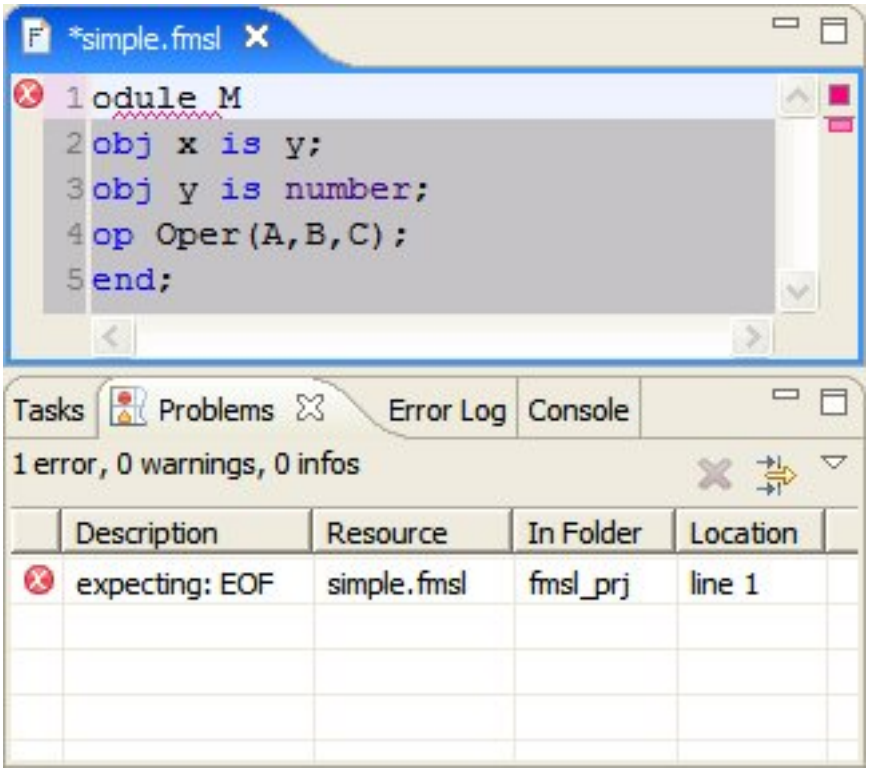

Figure 9: Editor and Problem View After Parsing

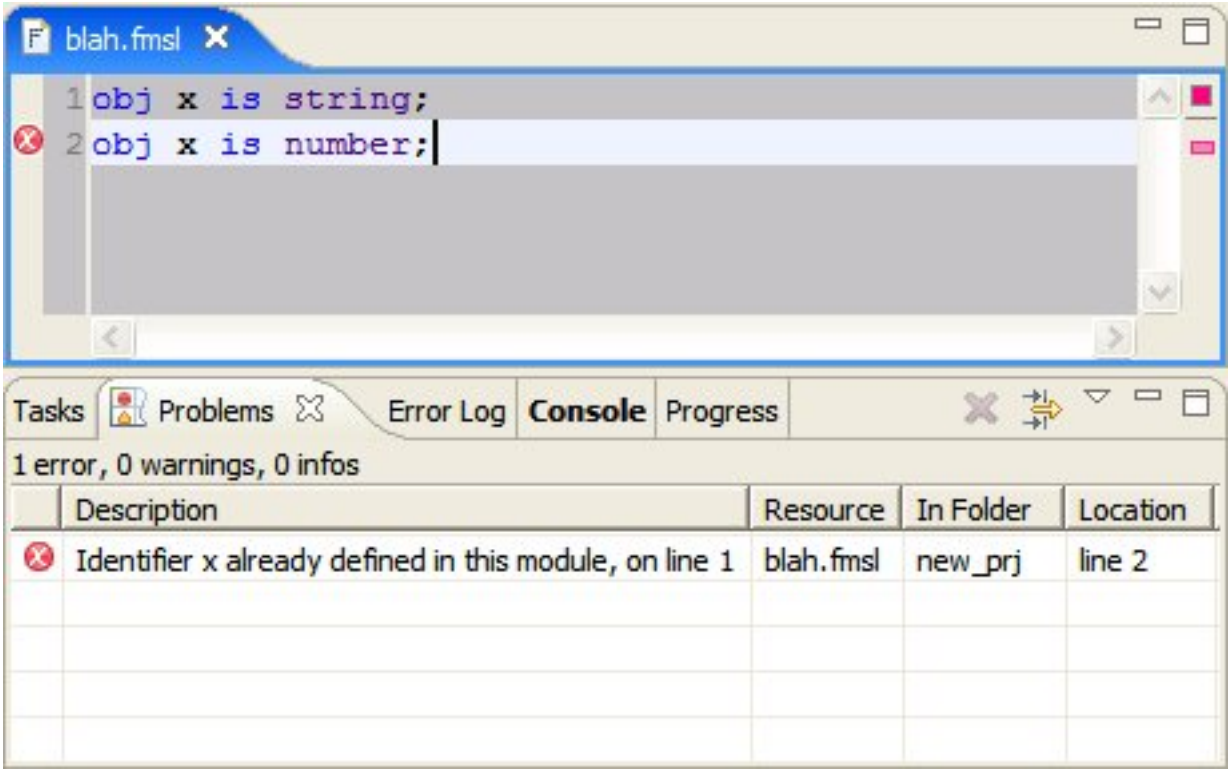

Figure 10: Editor and Problem View After Build

3 Background Information

3.1 Eclipse Architecture Overview

3.1.1 Platform runtime and Plug-in Architecture

The fundamental unit of the Eclipse architecture is the plug-in. Eclipse is extensible through the ability to add new functionality in the form of a plug-in. New functionality may be contained within a single plug-in or split across multiple plug-ins. With the exception of the Platform Runtime kernel, all functionality within the Eclipse Platform is contained in plug-ins. [2]

Plug-ins themselves are extensible by providing extension points. An extension point allows another plug-in to contribute some piece of functionality. For example, the workbench plug-in declares an extension point for the menu bar. A plug-in my extend the menu bar by contributing a menu to that extension point. [2]

Each plug-in has a manifest that declares how it is associated with other plug-in. The manifest consists of two files: the manifest.mf and the plugin.xml. The manifest.mf file is an OSGi bundle that contains the plug-ins runtime dependencies. The plugin.xml file contains the plug-ins extension points and extensions. At start-up, the Platform Runtime reads in all the plug-in manifests to build a plug-in registry. The plug-in code is not actually loaded until it is required. This allows the Eclipse Platform to support a large number of plug-ins. [2]

Figure 11 shows the major plug-ins of the Eclipse Platform. Each rectangle with the exception of the Platform Runtime is a plug-in.

3.1.2 Workspaces

Put simply, the workspace is a container for its resources. A workspace resource is a file, folder, or project. A workspace has one or more projects, where each project maps to a directory under the workspace directory by default but may map to any directory on the file system. [2]

A project contains files and folders that may be used or modified by the user. The Eclipse Platform provides an API that plug-ins may use to access workspace resources. A project may

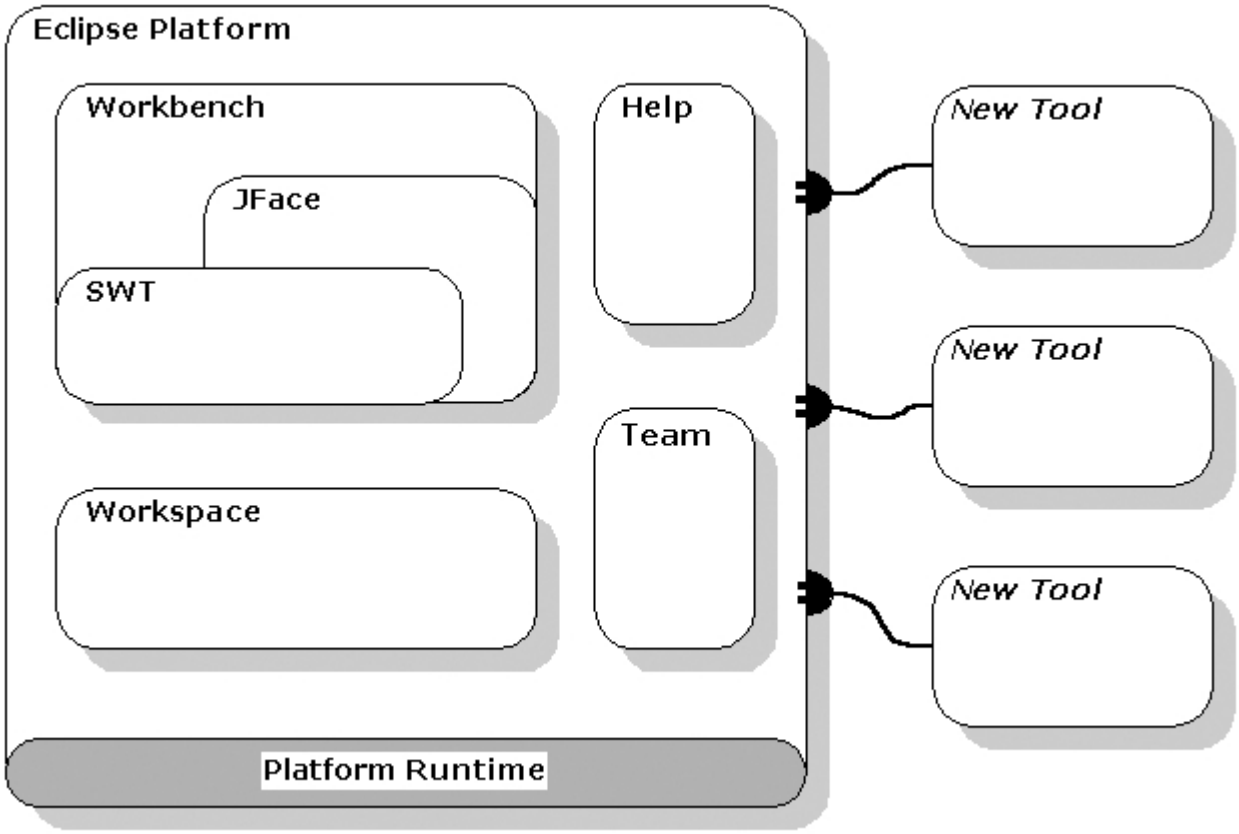

Figure 11: Eclipse Platform Architecture [2]

be tagged with one or more natures by plug-ins. A nature gives the project a personality. For example, a project with Java source files has a Java nature that attaches the Java builder to the project so that the Java files are compiled. [2]

The workspace provides several other conveniences to the user. A local history mechanism tracks the revisions of a file. A marker mechanism allows resources to be annotated with descriptive elements such as compiler errors, to-do list items, bookmarks, search hits, and debugger breakpoints. An event notification mechanism allows plug-ins to listen for specific resource events such as creations, deletions, and changes to individual resources. [2]

3.1.3 Workbench and UI Toolkits

The workbench provides the main UI to the user which is extensible through an API. It is built using the SWT and JFace UI toolkits rather than AWT and Swing. Both toolkits will be

discussed briefly in order to understand the foundations of the workbench.

The SWT stands for Standard Widget Toolkit. It is a heavyweight toolkit that provides a common OS-independent API for widgets and graphics. Although it provides an OS-independent API, it still relies on an OS-dependent implementation that uses a mixture of the native window system and emulation. The Eclipse Platform and all plug-ins use the SWT rather than AWT or Swing for presenting information to the user. [2]

JFace is a completely OS-independent UI toolkit that works with the SWT but does not hide it in the same way Swing hides AWT. It provides many frameworks and APIs for common UI programming tasks. For example, it has a dialog, preference, and wizard framework. [2]

The workbench is what the user sees when they start the Eclipse Platform as in figure 12. It provides the structure that plug-ins use to interact with the user. This structure is based around the concept of editors, views, and perspectives.

Editors are used to view and edit the contents of workspace resources by allowing the user to open, edit, and save them. An active editor may contribute actions to the workbench menus and tool bar. Multiple editors may be opened and are always located in the center of the workbench as in figure 12. Editors may be assigned to a specific content type such as .java or .html and new editors may be supplied by plug-ins. [2]

Views are used to provide information about some object in the workbench that the user is working on. The outline view, for example, displays a structured outline of the currently edited resource. Views may interact with other views and editors. For example, double clicking a file in the navigator view will open that file inside an editor. [2]

A perspective is used to control the layout of views and editors inside the workbench. It decides what views to initially open and where to place with respect to the editor area. For example, the Java perspective opens the package explorer view while the resource perspective opens the navigator view.

3.2 Eclipse Language Integration Issues

According the the Official Eclipse 3.0 FAQ book there are several phases involved when integrating a programming language into Eclipse:

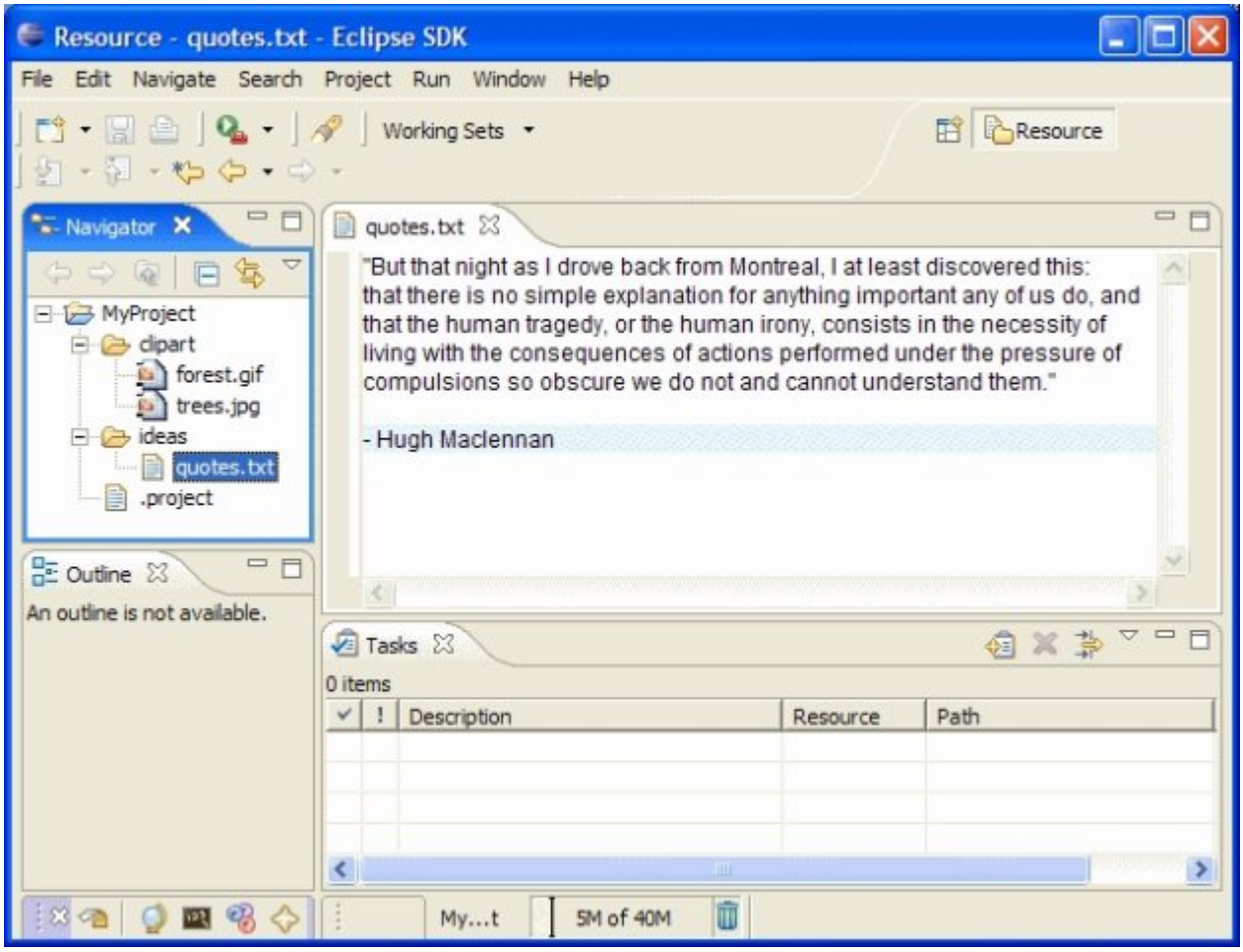

Figure 12: Eclipse Platform Workbench [2]

1. Compiling Code and Building projects
2. Implementing a Document Object Model (DOM)
3. Editing
4. Finishing touches

The first phase requires integrating a compiler and builder with Eclipse. The compiler may be an external program or an internal implementation. The internal implementation is the ideal solution but requires the compiler to be written in Java. The builder is in charge of coordinating the compilation of the project at the necessary time.

The second phase is implementing a DOM for the programming language. A DOM represents the structure of you programming language. It is used to generate information for the content outline view, content assist, text hovers, and refactoring. It is essentially the Abstract

Syntax Tree (AST) that is obtained from either the compiler or a lightweight parser. The lightweight parser route is more efficient and is how the JDT implements its Java model.[1]

The third phase is to implement a language-specific editor and all of its amenities. The editor is built on top of Eclipse's extensive editor framework. The amenities include content assist, quick fixes, refactoring, a content outline view, text hover, and problem markers.[1]

The fourth phase involves implementing the finishing touches. The finishing touches include wizards, project nature, perspective, documentation, and a debugger.[1]

4 Architecture and Design

4.1 Plug-in Architecture

Let us begin the discussion of FMSLEclipse architecture at a high level starting with the context diagram. The context diagram (see figure 13) shows the interaction of FMSLEclipse with external entities. The system represents an instance of Eclipse with the FMSLEclipse feature installed. A user sends commands to the system in the form of typing and mouse clicks. The system is dependent on an external FMSL compiler for the following reasons:

1. Problems are reported to the system during compilation (see figure 10).
2. The FMSL is translated into another form (an HTML data dictionary for instance).

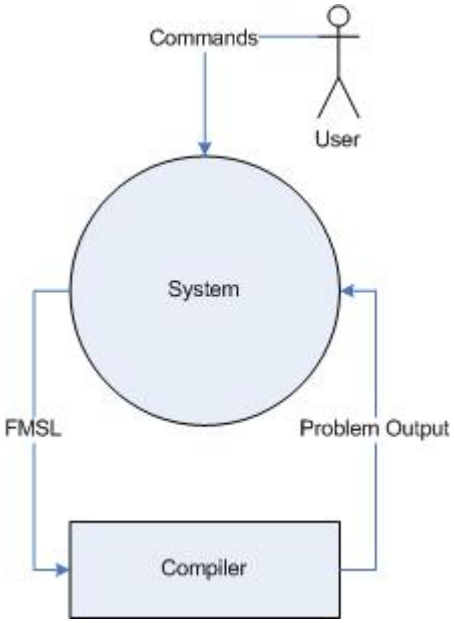

Figure 13: Context Diagram

FMSLEclipse is divided into three modules where each module is contained in a separate plug-in. Only one plug-in extends the Eclipse platform as seen in figure 14.

The fmsleclipse plug-in is the only plug-in that extends the Eclipse platform. All of the UI code is contained inside this plug-in. It contributes the FMSL editor (see figure 7), content outline view (see figure 8), properties (see figure 3), preferences (see figures 4, 5, and 6), new file wizard (see figure 2), and new project wizard (see figure 31). It uses and extends classes from the Language Integration Toolkit (LIT) and parser plug-ins.

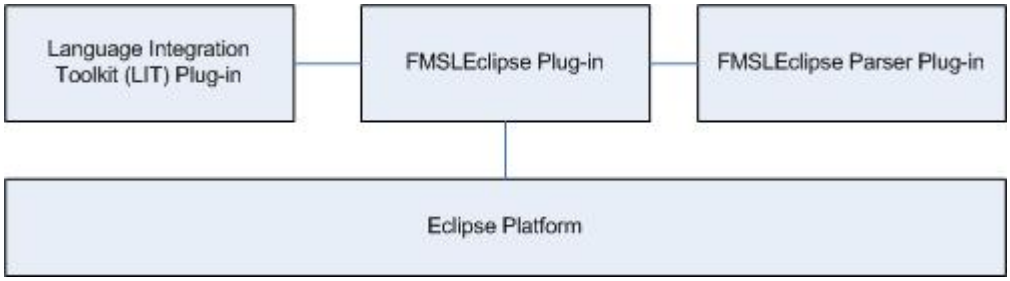

Figure 14: Plug-in Architecture

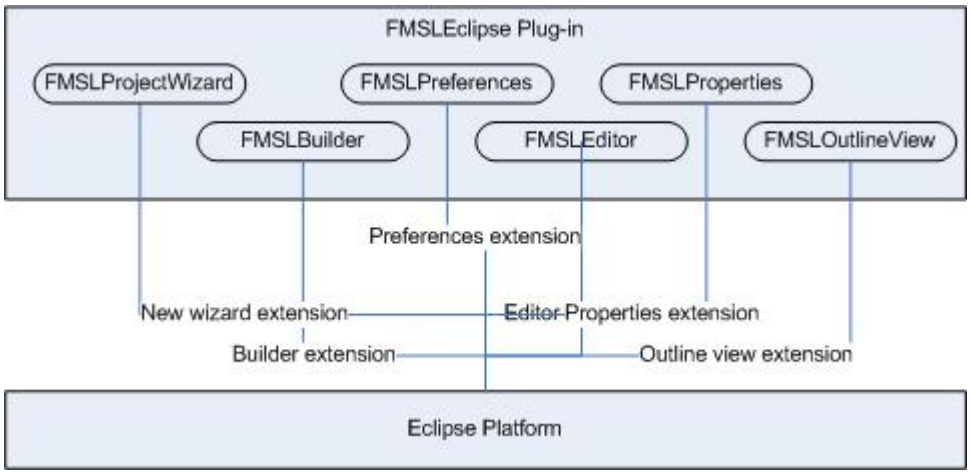

Figure 15: How the fmsleclipse plug-in extends the Eclipse Platform

The LIT plug-in was created during the development stages of the project, when it was recognized that during development certain patterns could be applicable to integrating any language in Eclipse. These patterns are abstracted into the LIT plug-in for other developers to use when integrating a language into Eclipse. The main pattern abstracted is the parser manager which is discussed in detail in section 4.2.2. The LIT plug-in also includes various utility classes that are not specific to FMSLEclipse .

The parser plug-in was similarly created during the development stages of the project. It contains the lightweight parser code created from a parser generator. Parser generators are discussed in further detail in section 4.2.1. The reason this code is in its own plug-in is because a different parser generator may be used in the future. Decoupling the parser allows for greater flexibility.

4.2 Parsing

The main functions of the lightweight parser in FMSLEclipse are to provide syntax errors and information for the content outline view. A lightweight parser allows for many future possibilities such as content assist, code folding, refactoring, and quick fixes. It is the backbone of any robust Eclipse based programming language tool because it provides a link between the document structure and the tool itself. The rest of this section will discuss the parser generator used to create the lightweight parser for FMSL and the language independent parsing manager.

4.2.1 SableCC Parser Generator

A parser generator or compiler-compiler is a program that generates the source code for a parser that is capable of parsing a specific programming language. The parser generator usually takes a grammar describing the programming language as input.[7]

Several parser generators were considered for creating the FMSL lexer and parser. Among the ones not chosen were ANother Tool for Language Recognition (ANTLR) and Java Constructor of Useful Parsers (JCUP) which are based on PCCTS and Yet Another Compiler Compiler (YACC) respectively. ANTLR has a built in lexer while JCUP uses Java Lexer (JLex) to generate a lexer. ANTLR and JCUP both generate Java code.

Below are the advantages and disadvantages of ANTLR and JCUP taken from Etienne Gagnon's masters thesis on SableCC.[6]

ANTLR Advantages:

- Tight integration of the lexer and the parser.
- Lexer is LL(k)-based (with predicates) and share the same syntax as parsers.
- Action code is much easier to debug with ANTLR than with LALR(1) table-based parsers, due to the natural behavior of recursive-descent parsers.
- Supports EBNF.
- Option to generate AST automatically.
- Support recursive-descent tree-parsers.
- Support for AST is very convenient for multiple-pass compilers.

- The range of languages that can be parsed is much bigger than LL(1), and is relatively comparable to LALR(1). The use of semantic predicates enables the parsing of context-sensitive grammars.
- Free and source code is in the public domain.
- Well supported, with dedicated newsgroups on the Internet.

ANTLR Drawbacks:

- Lexer does not support 16 bits Unicode character input.
- While semantic predicates allow the parsing context-sensitive grammars, they also are a software engineering problem. Semantic verifications must happen along with parsing in order to enable semantic predicates. Furthermore, the predicates are somehow an integral part of the resulting grammar. This obscures the grammar.
- Syntactic predicates are very expensive in computation time.
- A majority of predicate uses would not be needed by an LALR(1) parser to recognize the same grammar.
- LL(k) grammars cannot be left recursive. This handicap can be fixed by widely know grammar transformations and the use of EBNF syntax. A = Aa—a =¿ A = (a)+.
- ANTLR specifications suffer from the same important software engineering problems as JLex/CUP. They tend to become huge, and debugging action code involves the same tedious cycle.
- As with JLex/CUP, the responsibility of keeping the specification and the generated code synchronized is left to the programmer. So taking a shortcut in the debugging cycle by fixing errors in the program directly can result in unsynchronized specification and working code. This is most likely to happen when debugging semantic predicates in an Integrated Development Environment.
- The integrity and the correctness of the AST is left in the hands of the programmer. There will be no warning if a transformation on the AST results in a degenerate tree. Such bugs are extremely difficult to track, because they may result in a null pointer exception, or

some other error condition in unrelated code thousands of instructions after the transformation occurred. This is comparable to C and C++ array out of bound problems.

JCUP JLex Advantages:

- JLex DFA based lexers are usually faster than hand written lexers.
- JLex supports macros to simplify the specification of complex regular expressions.
- JLex supports lexer states, a popular feature found in GNU FLEX.
- CUP generates LALR(1) parsers and can deal with some ambiguous grammars using options to resolve LALR conflicts.
- The set of languages that can be recognized by an LALR(1) parser is a superset of LL(k) languages. In addition, LALR(1) grammars can be left recursive whereas LL(k) grammars can't.
- LALR(1) parsers are usually faster than equivalent PCCTS LL(k) parsers for the same language, because PCCTS uses costly syntactic predicates to resolve parsing conflicts.
- Both JLex and CUP are available in source code form.

JCUP JLex Drawbacks:

- JLex supports only 8 bits characters. But, Java has adopted 16 bits Unicode characters as its native character set. JLex has a directive, but it is not yet implemented.
- JLex still has known bugs in presence of complex macros.
- JLex macros are treated much like C macros. This means that they are textually replaced in regular expressions. This can lead to very hard to find bugs similar to those found in C in presence of un-parenthesized macros.
- JLex and CUP have not been specially designed to work together. So, it is the programmer's job to build the links between the code generated by both tools. JLex has
- CUP options for handling ambiguous grammars can be quite dangerous in the hand of a novice, because it is hard to clearly determine the recognized grammar.

- Action code embedded in CUP parsers can be quite difficult to debug. The abstraction required to clearly understand the operation of a table-based LALR parser is the source of this difficulty for casual users of CUP.
- With today's low memory prices and faster processors, many programmers prefer to work on an Abstract Syntax Tree (AST) representation of parsed programs. CUP offers no support for building ASTs. So, the programmer has to write the appropriate action code to build nodes for every production alternative of the grammar.
- The lack of support for ASTs renders CUP ill suited for multiple pass compilers.
- The fact that actions are embedded in the specification is a big software engineering problem. It means that resulting specifications are often enormous. Since CUP does not allow the specification to be broken into multiple files, the specification may result in one huge file. Furthermore, the duplication of action code in the specification and in the resulting program is quite bad. It leaves to the programmer the responsibility of keeping the specification and resulting program consistent. So safely debugging JLex/CUP action code involves the following tedious cycle:

  Repeat

  1. Writing or modifying action code in the specification file.
  2. Compiling the specification.
  3. Compiling the resulting code.
  4. Executing the resulting program to find errors.
  5. Locating the errors in the program.
  6. Looking back in the specification for the related erroneous action code.

  Until success
- Taking a shortcut in the previous cycle by fixing errors directly in the generated program can result in unsynchronized specification and working code, if the programmer does not take the time to update the specification accordingly. This is most likely to happen when a programmer debugs actions in an Integrated Development Environment.

SableCC is the parser generator we chose to generate a lexer and parser for FMSL. Both

the FMSL lexer and parser are generated by SableCC from a grammar created by Dr. Gene Fisher. FMSLEclipse is currently using a reduced grammar as a proof of concept. A full grammar to support the FMSL language will be implemented in the future.

SableCC was chosen because it generates Java code, builds a lexer and parser, provides extensible AST tree walkers, and is maintainable. The design of SableCC favors good software engineering practices over speed. The framework uses object-oriented design patterns to provide modularity that makes the resulting lexer and parser maintainable.[6]

Given a grammar, SableCC generates Java code grouped into the following four packages:

- lexer
- parser
- node
- analysis

The lexer package contains the Lexer and LexerException classes. The Lexer class is the lexer generated from the grammar and the LexerException is thrown when there is an error while lexing. The parser package contains the Parser and the ParserException classes. The Parser class is the parser generated from the grammar and the ParserException is thrown when there is a parsing error. The node package contains a class for each node in the typed AST generated by the Parser class. The analysis package contains several classes that may extended to walk the AST tree.[6]

SableCC uses the visitor design pattern to walk the AST. Two classes are provided in the analysis package that walk the tree in normal and reverse depth-first traversal. To implement actions it is necessary to create a new class that extends one of the tree-walker classes and to override methods for the required nodes in the AST that should be visited.

4.2.2 Parser Manager

A pattern encountered while developing FMSLEclipse is that parsing a document as the user types code can be abstracted for any programming language. This became the main motivation for creating a separate LIT plug-in. It is important to parse a document while the

user types to provide instant feedback. Feedback such as syntax errors and an updated content outline view improve the user experience and productivity.

On a high level the parsing process involves the following steps:

- Parse the document
- Gather information
- Update UI

Parsing the document is abstracted into the IParser interface in the LIT plug-in. This interface is intended to be implemented by a class that knows about the specific lexer and parser to use. This abstraction decouples the lexer and parser implementation so that any parser generator may be used to create a lexer and parser. It also provides a method to parse the document.

Gathering information about the document is also abstracted into the IParser interface. The interface provides methods for getting errors, tasks, content outline view information, and a symbol table. Errors are used to annotate a document with markers that provide a description of the specific error. Tasks are also used to annotate a document with markers but provide information about a particular task that needs to be completed at that spot. Tasks are usually in the form: comment TODO task. In Java an example syntax of a task is: //TODO my task goes here. The grammar must be programmed to look for tasks. Information for the content outline view is obtained by walking the AST generated by the parser. The symbol table is generated in the same manner as the content outline view.

After the information has been gathered the UI must be updated. Error and task markers are added to the marker bar and the problem and task views are updated respectively, The content outline view is passed new information and updated.

All of this is orchestrated by the ParserManager class in the LIT plug-in. The ParserManager is notified each time the user types and begins the process. First, a thread is spawned to parse the document and gather the information. Before that thread dies it spawns another thread that makes the necessary UI updates. These threads are non-blocking and set to a low priority so the user will not notice any change in UI responsiveness.

4.3 Building

Eclipse provides a framework for building a project. In most cases building a project involves passing the project's source files through a compiler. Syntax errors may be parsed from the compiler's output and output files may be generated. The Java builder, for instance, annotates Java source files with errors and warnings, and generates .class files.

The FMSL builder may build a project in two ways. It will either run the source files through a compiler that generates syntax errors only or through a compiler that also generates the output data dictionary files. The builder is run automatically during each save operation if the build automatically preference is checked or manually. The user must specify where the two FMSL compiler executables reside in the file system in the preferences (see figure 4).

Each FMSL project has an FMSL project nature assigned to it that is responsible for attaching an FMSL builder to the project. Any project that contains an FMSL file will be assigned an FMSL project nature automatically when the file is opened in an FMSL editor. See section 3 for more background information of project natures and builders.

The build process in figure 16 begins when the user selects Project -> Build Project or the file is saved when the build automatically option is set. The FMSL source files are then sent to the FMSL compiler which is launched as an external program from within Eclipse. Any output from the FMSL compiler is parsed for errors which are used to annotate the FMSL code and update the problems view.

The builder class diagram in figure 17 shows the class interactions of the plug-in with itself and Eclipse. The main class in charge of building a project is the FmsleclipseBuilder class. It extends the Eclipse framework and is attached to a project as the FMSL builder by the FMSL project nature. The IBuilder interface defines the classes that are capable of performing the actual build. The IProgramRunner interface defines classes that run a specific external compiler and parse the errors from the compiler's output. Each IBuilder has knowledge of the IProgramRunner it should use. The BuilderRegistry class knows about each IBuilder and IProgramRunner and is used to obtain the correct one.

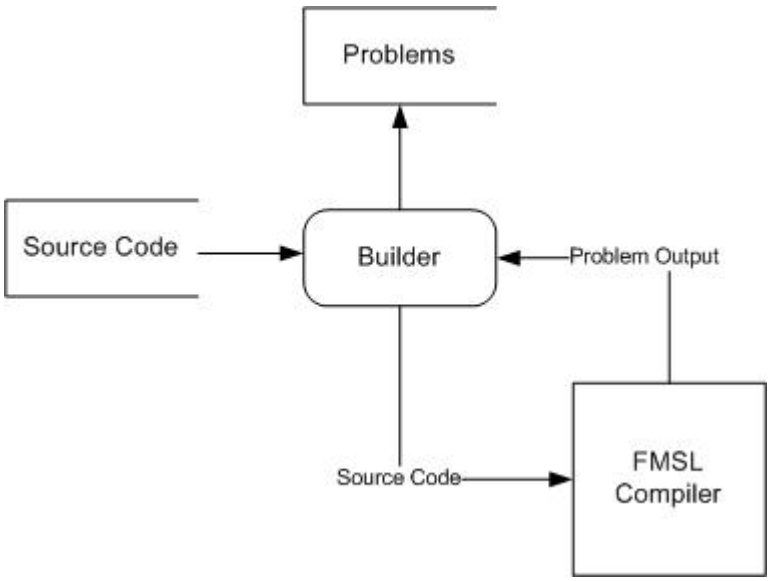

Figure 16: Builder Data Flow Diagram

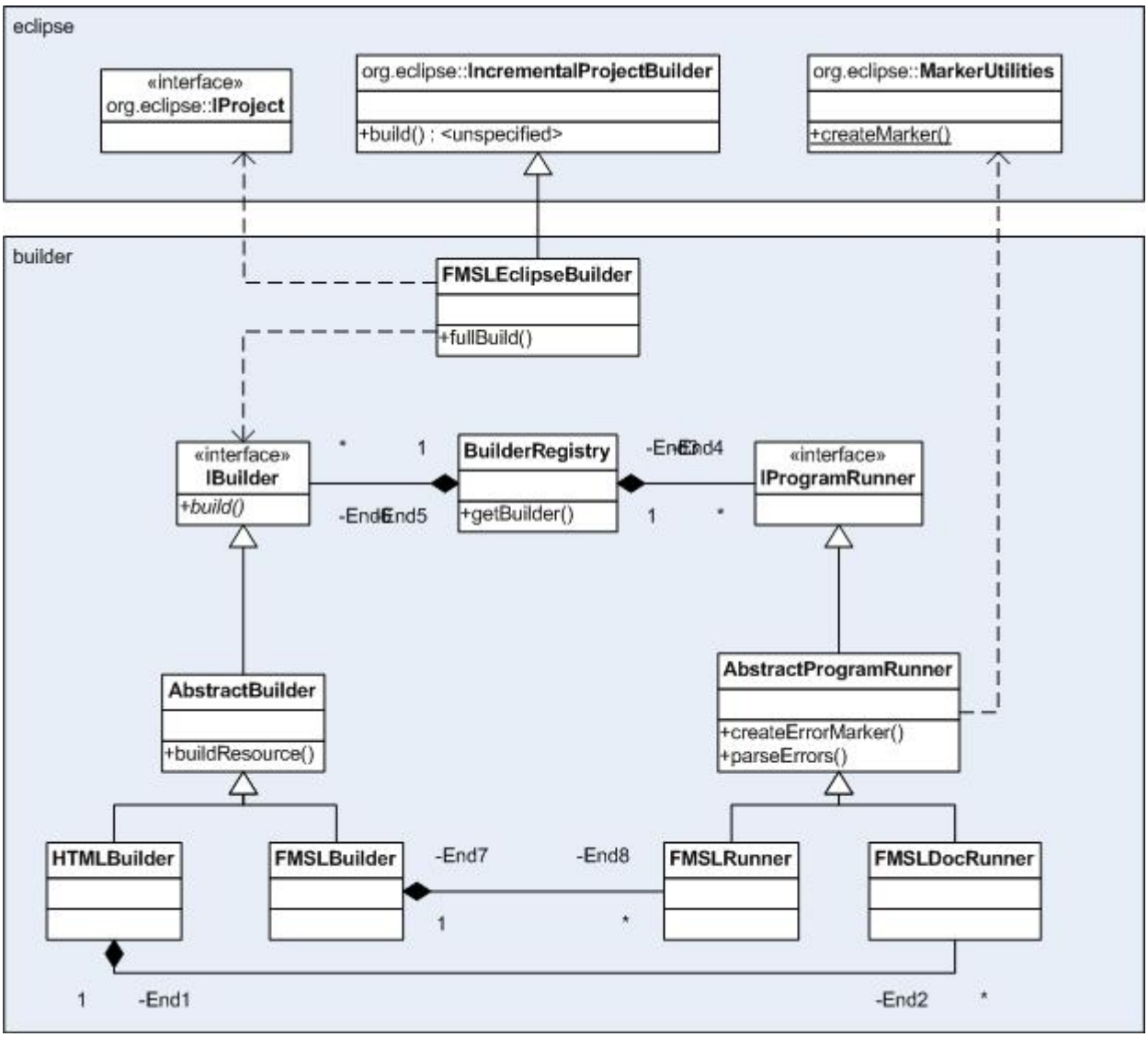

Figure 17: Builder UML Diagram

5 Usage Guide

This section contains information for end users, developers, and testers of FMSLEclipse . The end users section explains how to install the FMSLEclipse tool while the developers section describes how to setup an Eclipse environment to work on FMSLEclipse . The testing section outlines the framework for testing the plug-in.

5.1 End Users

FMSLEclipse is deployed as an Eclipse feature (see section 5.2 for more information on features). The following procedure is used to install the FMSLEclipse tool:

1. Obtain the FMSLEclipse feature (i.e. fmsleclipse.zip)
2. Unzip to your Eclipse installation (i.e. C:\Program Files\eclipse)
3. Start or restart Eclipse
4. Verify the installation by checking Help -> Software Updates -> Manage Configuration

5.2 Developers

Setting up FMSLEclipse within Eclipse for development is a simple procedure if these steps are followed.

5.2.1 Eclipse Requirements

Verify your Eclipse installation meets the following requirements:

- Version: Eclipse 3.1 . FMSLEclipse may or may not work with newer versions of Eclipse.
- Plug-ins: JDT and Plug-in Development Environment (PDE). These come with the standard Eclipse Software Development Kit (SDK) distribution.

5.2.2 CVS Checkout

Checkout the feature and each plug-in from Concurrent Versioning System (CVS):

1. Switch to the CVS perspective.
2. Add a new repository location:
   (a) File -> New -> Other -> CVS -> Repository location
   (b) Use the following information in the wizard:
      - Connection type: extssh
      - Host: hornet.csc.calpoly.edu
      - Repository path: /home/gfisher/projects/CVS
3. Use the CVS Repositories view to browse the newly created repository to:
   - HEAD\fmsl\subprojects\eclipse\implementation\source\java.

   Checkout the following modules as projects by right clicking on their respective folders in the CVS Repositories view:
   - features\fmsleclipse
   - plugins\fmsleclipse
   - plugins\lit
   - plugins\parser

5.2.3 Debugging

It is possible to debug the FMSLEclipse tool by using two instances of Eclipse: one that uses the FMSLEclipse plug-ins and the other to debug it. Debugging an instance of Eclipse is similar to debugging a normal Java app. Use the following steps to create a profile and start debugging:

1. Run -> Debug
2. Select ”Eclipse Application” in the configurations tree and click the ”New” button at the bottom of the dialog.
3. The default values should be fine so click the ”Debug” button to open a new instance of Eclipse with the FMSLEclipse tool installed.

5.2.4 Deployment

It is necessary to deploy FMSLEclipse as an Eclipse feature because it consists of several plug-ins. An Eclipse feature is used to manage a set of plug-ins that usually relate to a common feature. They are managed by the Update Manager, which is part of the Workbench. They may be branded to include information such as: provider, supported environment, license, and update sites.[4]

A feature consists of a single zip file or a directory. The zip file or directory contains two directories: features and plug-ins. The features directory contains meta data about the feature. The plug-ins directory contains the feature's plug-ins. Use the following steps to create the FMSLEclipse feature:

1. File -> Export
2. Select "Deployable features"
3. Ensure the dialog settings are correct and then click "Finish"

5.3 Testing

This section describes the testing framework for FMSLEclipse . The framework follows the same methodology used by the JDT tool, which is considered to be a best practice for testing plug-ins. The framework consists of automated regression testing using JUnit and manual regression testing using written procedures. All the tests are contained in a separate "test" plug-in and may be found with the rest of the FMSLEclipse plug-ins in CVS (see section 5.2).

5.3.1 JUnit Unit Testing

JUnit is an open source unit testing framework used by developers to create automated regression tests. It was written by Erich Gamma and Kent Beck.[5] Refer to www.junit.org for more information on how to use the JUnit framework and how to write JUnit tests.

Fortunately, JUnit is integrated well with Eclipse and should be included with the standard SDK distribution. JUnit tests are usually written per class so the package structure of the test

plug-in mirrors the FMSLEclipse plug-ins package structure. At the top level package there is an ”AllTests.java” class that is used to run all of the individual JUnit tests. New unit tests should be added to this class.

```
public class OutlineNodeTest extends TestCase {
protected void setUp() throws Exception {
super.setUp();
mParent = new OutlineNode(”parent”, FmslNodeType.MODULE, 0, null);
mChild = new OutlineNode(”child”, FmslNodeType.OBJECT, 1, mParent);
mChildren = new ArrayList<OutlineNode>();
mChildren.add(mChild); }
/*
* Test method for ’net.sourceforge.lit.model.OutlineNode.addChild(OutlineNode)’
*/ public void testAddChild() {
Assert.assertTrue(mParent.getChildren() == null);
mParent.addChild(mChild);
Assert.assertTrue(mChildren.equals(mParent.getChildren()));
}
private OutlineNode mParent, mChild;
private ArrayList¡OutlineNode¿ mChildren; }
```

5.3.2 Manual Testing

Manual testing is more free formed than JUnit testing and requires a real person to perform the test. Each manual test is contained in an HTML file under the ”Manual Scenario Tests” directory of the test plug-in. The format of a manual test was taken from the Eclipse Platform UI test plug-in (org.eclipse.ui.tests) and is considered to be a best practice. An example of a manual test for the new file wizard:

FMSL New File Wizard

Purpose: To test the FMSL New File Wizard in Eclipse. We will complete the FMSL New File Wizard, verify its appearance, and verify its behaviour.

Setup:

1. Install Eclipse Platform.

2. Install FMSLEclipse plugin.

3. Create a project.

Method:

1. Start the workbench.

2. Invoke File $>$ New $>$ Other. In the ”Select a Wizard” dialog, select FMSL Wizards $>$ FMSL File. Verify that the icon next to the wizard depicts a file with an ”F” on it. 3. Select Next button. Verify the image banner at the top of the wizard is correct.

4. Enter ”newfile” into the file name text box and verify the finish button is still ghosted.

5. Enter ”newfile.fm into the file name text box and verify the error ”File extension must be ”fmsl” ” is displayed at the top of the wizard.

6. Enter ”newfile.fmsl into the file name text box and verify the finish button is available.

7. Select Finish button. Verify the file has been created in the resource view and that it has been opened in an fmsl text editor.

8. Repete the above steps and verify the wizard displays the error ”The same name already exists.”

6 Related Work

6.1 Texlipse Plug-in

The texlipse plug-in adds support for the Latex typesetting language to Eclipse. It was created by the Texlapse-team at Helsinki University of Technology and is currently maintained as an opensource project on sourceforge.net. It is a good example of how to integrate a language into Eclipse.

Texlipse was used as a best practise example while developing FMSLEclipse and much of the design is reused. Because it is an open source project some of the code is used in FMSLEclipse . All of the code reused has bee updated to take advantage of Java 5.0 and abstracted when possible.

The two major designs taken from Texlipse were parsing (see section 4.2) and building (see section 4.3). The parsing design was greatly improved upon by genericizing it to be used by any programming language as described in section 4.2.2.

6.2 Java Development Tool

The JDT refers to an Eclipse feature that provides a full Java IDE. It is the ultimate example of a plug-in that integrates a language into the Eclipse Platform. This section describes the best practices of the the JDT.

A high level architecture of the JDT reveals that it is made up of several plug-ins. The main plug-ins are:

1. Core
2. UI

The JDT core plug-in contains all the non-GUI elements. This includes the Java project nature, Java builder, Java problem marker, and the Java model. Refer to figure 19 to see how the JDT core plug-in extends the workspace.

The Java project nature labels a project in the workspace as a Java project. It attaches a Java builder to the project that handles calling the Java compiler to build the project. The Java builder is capable of performing a full or incremental build based on the changes since the

last build. During compilation the JDT annotates Java resources with Java markers if errors or warnings are found.

The Java model plays an important role because it provides the Application Programmable Interface (API) for navigating the Java AST. It is easier to use the Java model API rather than accessing the underlying resources directly. [2]

The JDT UI plug-in contains all the GUI related elements. This includes the Java perspective, Java editor, actions for creating Java elements, and several views. Refer to figure 19 to see how the JDT UI plug-in extends the workspace.

The Java editor is the default editor for *.java files. It supports too many features to list but some important ones are syntax coloring, content assist, and automatic formating. Changes made in the editor are reflected in the content outline view and vica-versa.

Several actions for creating Java elements are contributed to the workbench. These actions include wizards for creating Java projects, packages, classes, and interfaces. These actions operate both on the file system and Java model. [2]

The contributed views include the package explorer, content outline, and type hierarchy. The package explorer is similar to the resource view but focuses on showing the Java structure of the project. The content outline shows a detailed outline of the current java file being edited. The type hierarchy view shows the inheritance tree of a selected class or interface.

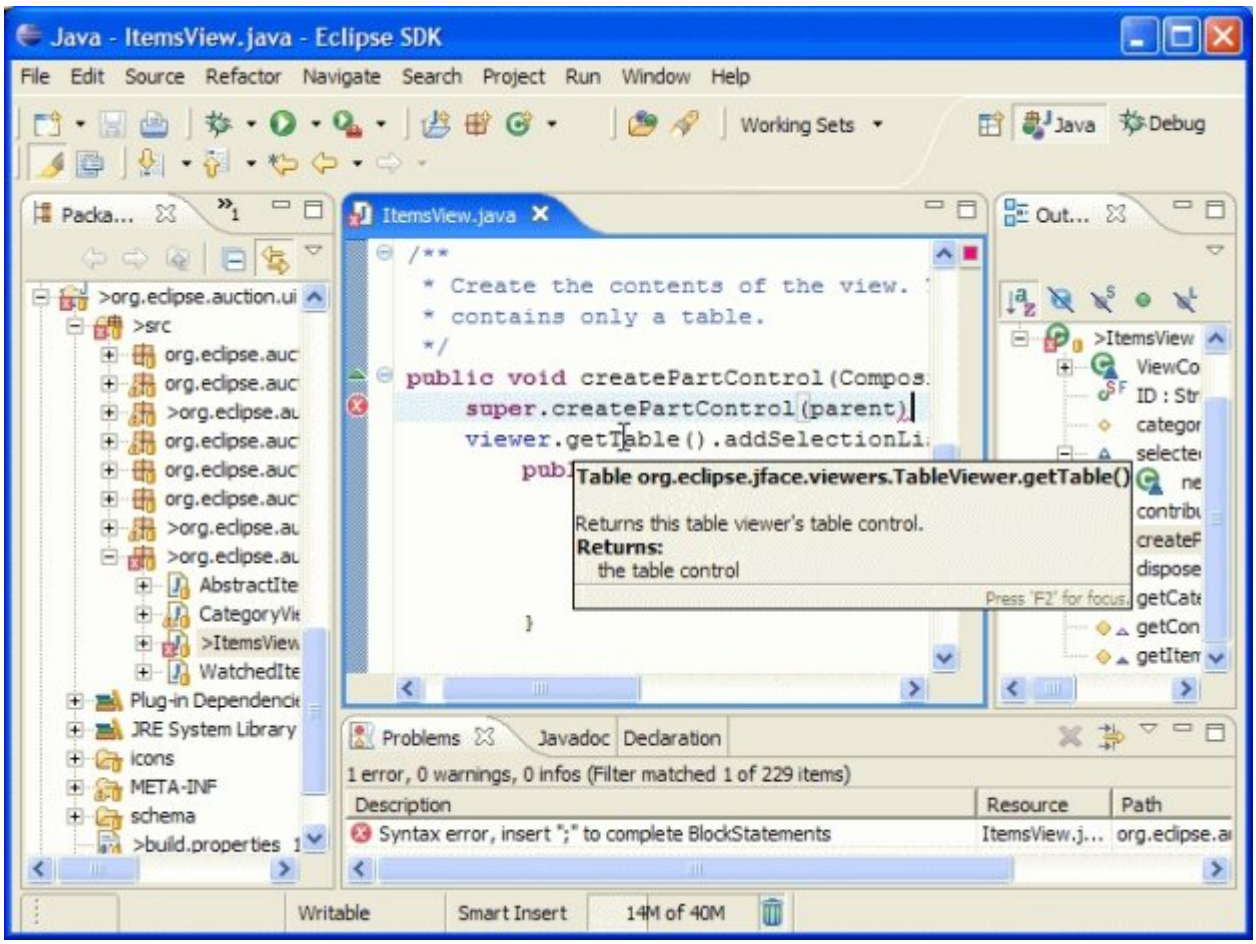

Figure 18: Java Perspective [2]

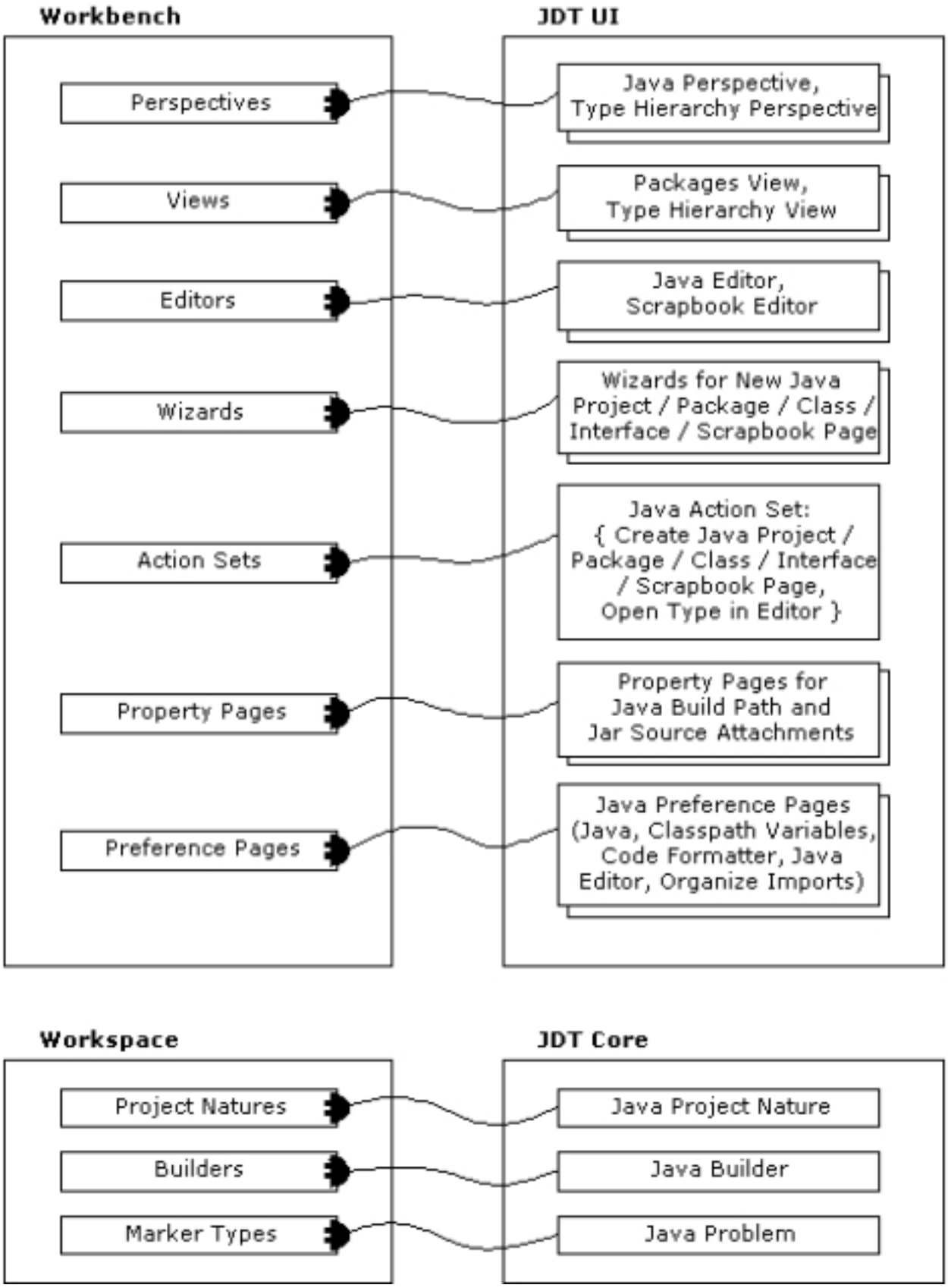

Figure 19: JDT Architecture [2]

7 Conclusions and Future Work

The main goal of this senior project is adding support for FMSL to Eclipse. The process taken to do so is detailed in this report. Issues on how to integrate a programming language into Eclipse are also covered.

We found that Eclipse is an excellent choice to use as an IDE for a programming language. It allows tools to be added via the flexible plug-in model. It provides a robust and extensible framework for programming language support. Being open source and the most popular Java IDE, Eclipse will be around for a long time.

We found a pattern while implementing the parsing process. The whole process may be abstracted into a framework for use by other programming languages. See section 4.2.2 for details.

This project is a proof of concept that the FMSL language can be supported by Eclipse. As such, there is future work to be completed. The lightweight parser is currently based on a reduced grammar offering an incomplete implementation of the FMSL language. A full implementation of the FMSL grammar is required for a usable tool. The current lightweight parser also quits after encountering a single error. A robust solution would report multiple errors. Lastly, the remaining features in the requirements document (see appendix C) should be implemented. These features are enumerated in section 1.3.

A Acronyms

ANTLR ANother Tool for Language Recognition

API Application Programmable Interface

AST Abstract Syntax Tree

CVS Concurrent Versioning System

DOM Document Object Model

FMSL Formal Modeling Specification Language[2]

IDE Integrated Development Environment

JAR Java ARchive

JDT Java Development Tool[3]

JCUP Java Constructor of Useful Parsers

JLex Java Lexer

LIT Language Integration Toolkit

PDE Plug-in Development Environment

RSL Requirements Specification Language

SDK Software Development Kit

UML Unified Modeling Language

YACC Yet Another Compiler Compiler

[2] FMSL used to be called Requirements Specification Language (RSL)
[3] JDT refers to the eclipse project that contains the Java IDE

B JDT Features

The following list is a summary of the features offered by the JDT project. It is provided here as an example of the different features to include when integrating a programming language into Eclipse. An overview discussion of the JDT may be found in section 6.2. The following list was taken directly from the ”Eclipse Platform Technical Overview” article. [2]

- Java projects
  - Java source (*java) files arranged in traditional Java package directories below one or more source folders.
  - Java ARchive (JAR) libraries in the same project, another project, or external to the workspace.
  - Generated binary class (*.class) files arranged in package directories in a separate output folder.
  - Unrestricted other files, such as program resources and design documentation.
- Browsing Java projects
  - In terms of Java-specific elements: packages, types, methods, and fields.
  - Arranged by package, or by supertype or subtype hierarchy.
- Editing
  - Java source code editor.
  - Keyword and syntax coloring (including inside Javadoc comments).
  - Separate outline shows declaration structure (automatic live updates
  - Compiler problems shown as annotations in the margin.
  - Declaration line ranges shown as annotations in the margin.
- Code formatter
  - Code resolve opens selected Java element in an editor.
  - Code completion proposes legal completions of method, etc. names.
  - API help shows Javadoc specification for selected Java element in
  - Import assistance automatically creates and organizes import

- Refactoring
  - For improving code structure without changing behavior.
  - Method extraction.
  - Safe rename for methods, etc. also updates references.
  - Preview (and veto) individual changes stemming from a refactoring operation.
- Search
  - Find declarations of and/or references to packages, types, methods,
  - Search results presented in search results view.
  - Search results reported against Java elements.
  - Matches are highlighted as annotations in the editor.
- Compare
  - Structured compare of Java compilation units showing the changes to
  - Replace individual Java elements with version of element in the local history.
- Compile
  - JCK-compliant Java compiler.
  - Compiler generates standard binary *.class files.
  - Incremental compilation.
  - Compiles triggered manually upon demand or automatically after each
  - Compiler problems presented in standard tasks view.
- Run
  - Run Java program in separate target Java virtual machine.
  - Supports multiple types of Java virtual machine (user selectable).
  - Console provides stdout, stdin, stderr.
  - Scrapbook pages for interactive Java code snippet evaluation.
- Debug
  - Debug Java program with JPDA-compliant Java virtual machine.

- View threads and stack frames.
- Set breakpoints and step through method source code.
- Inspect and modify fields and local variables.
- Expression evaluation in the context of a stack frame.
- Dynamic class reloading where supported by Java virtual machine.

C Requirements

C.1 Overall Description

C.1.1 Product Perspective

FMSLEclipse is an entirely new product. It will be written as a plugin to extend Eclipse. Eclipse is a development platform that provides a framework for different development tools. All development tools for Eclipse are written as plugins. The plugin will interface with Eclipse by using the framework API provided by Eclipse.

C.1.2 Product Features

FMSLEclipse will provide a full featured IDE for FMSL. This includes the ability to compile, a text editor, a project based structure with file navigation, and a hierarchical view of each FMSL file.

C.1.3 User Classes

The primary users of the system are software engineers who want to use FMSL to specify their software applications. The immediate users will be engineering students attending Cal Poly.

Undergraduate students under the 2005-07 catalog will take the required software engineering courses during their junior or senior year. It is assumed they know how to program at this juncture and are familiar with at least one professional IDE.

C.1.4 Operating Environment

This section describes the environment that FMSLEclipse will operate in. Each requirement is designated by [OE-#] and is referred to in the rest of this document by that designation.

OE-1: FMSLEclipse shall run within Eclipse 3.1 .

Priority: High

OE-2: FMSLEclipse shall operate on any computer and operating system that Eclipse 3.1 does. For the scope of this project, FMSLEclipse will only be tested on: an x86 based machine running Windows XP and a machine running OSX.

Priority: Low

OE-3: FMSLEclipse shall be Java 1.5 compatible. Any known portability issues between platforms will be avoided. Focusing on x86 Windows, Mac OSX, and SPARC Solaris.

C.1.5 Constraints

This section describes the design and implementation constraints of FMSLEclipse . Each requirement is designated by [CO-#] and is referred to in the rest of this document by that designation.

CO-1: FMSLEclipse shall use the existing FMSL compiler written in C and C++. Porting the FMSL compiler to Java could be a senior project in itself and is outside the scope of this project.

Priority: High

CO-2: FMSLEclipse shall be written in Java. Eclipse is written in Java and expects its plugins to be so also.

Priority: High

C.1.6 User Documentation

Any user documentation for FMSLEclipse will be found online in a section titled ”FMSLEclipse ” under the Eclipse help system.

C.1.7 Assumptions and Dependencies

This section describes the assumptions and dependencies that affect the **monitor program**. Each requirement is designated by [AS-#] or [DE-#] and is referred to in the rest of this document by that designation.

DE-1: Dr. Fisher will provide the grammar for FMSL in a format that can be read by the parser generator we decide to use.

Priority: High

C.2 UI Overview

C.2.1 Workbench

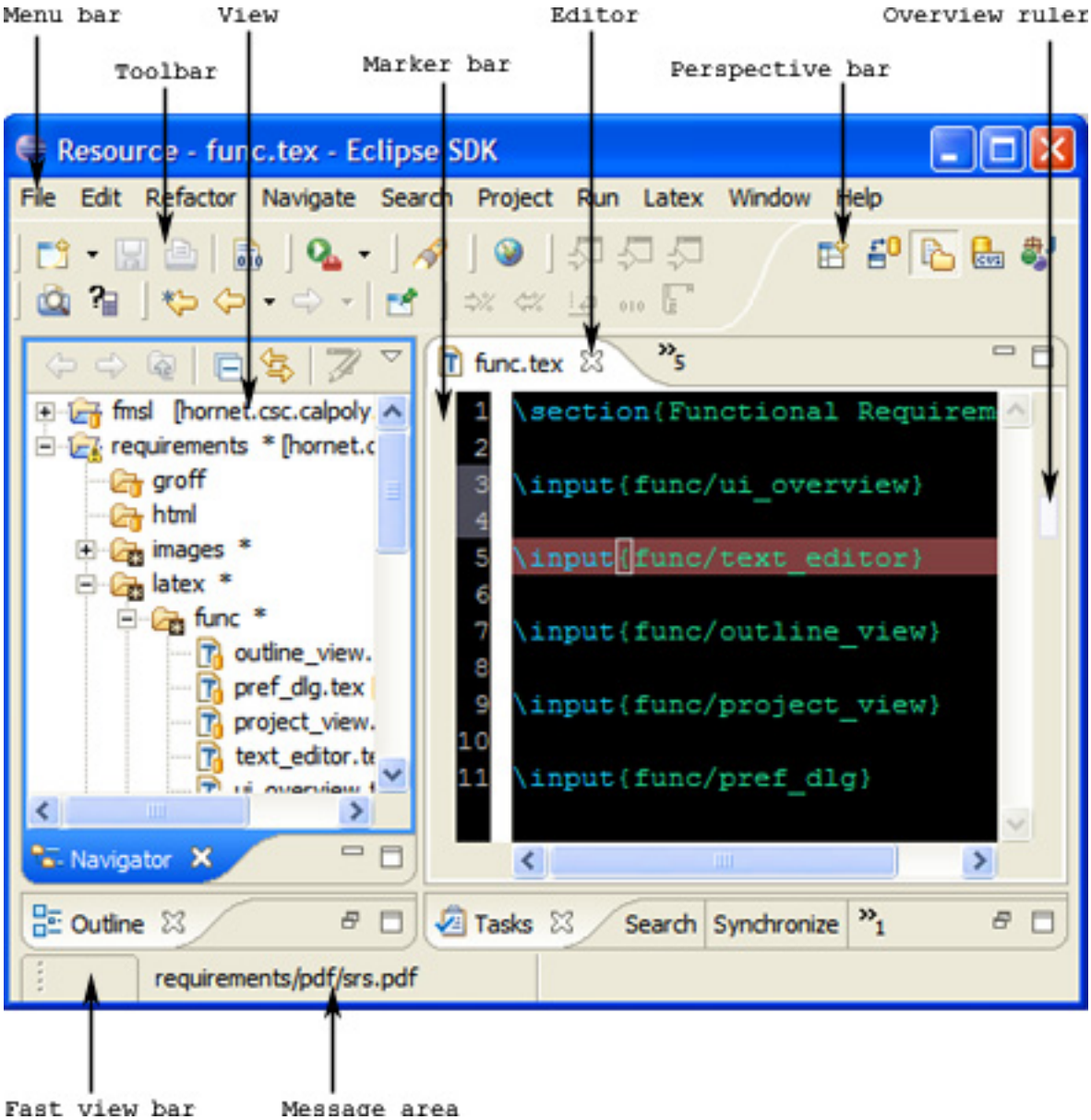

Figure 20: Workbench

The workbench (see figure 20 is the term used to refer to the main window presented to the user. It consists of a menu bar, toolbar, perspective bar, and a perspective. A perspective contains and controls the layout of multiple views and editors. An editor is used to edit a resource. A view is used to display information about the project or the resource being edited. Views may be detached from the workbench as seen in figure 21. The marker bar and overview ruler display annotations in the resource being edited. The fast view bar allows the user to place a view as a shortcut in that bar and when selected it will display temporarily. The message area displays relevant messages.

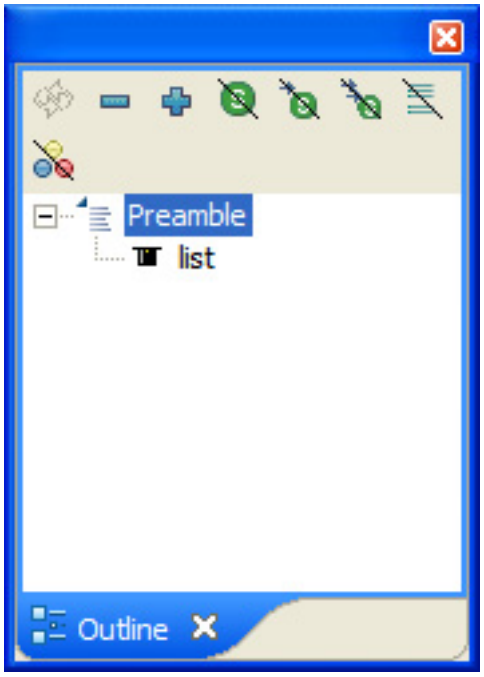

Figure 21: Detached View

C.3 Text Editor

The text editor in Eclipse allows the user to edit a source file. A specialized text editor for FMSL will provide features that help the user edit, navagate, and diagnose problems of a source file.

C.3.1 Syntax Highlighting

Priority: High

FMSLEclipse shall highlight the FMSL syntax in a source file. The colors used shall be selectable by the user (see C.6. The following syntax shall be highlighted:

- comments
- import/export declarations
- expression keywords
- objects, operations, modules
- values
- attribute names

C.3.2 Automatic Indentation

Priority: High

FMSLEclipse shall automatically indent FMSL code.

comments: indent to same level as following token.

import/export declarations: level 0

modules: level 0

objects, operations, values: level 1

attributes: level 2

multi line attribute: level 3

C.3.3 Problem Markers

Priority: High

FMSLEclipse shall indicate that a line of code is causing a problem by marking it with a problem marker (figure 22). A problem marker is an annotation to the text editor that is displayed as an icon next to the line of code causing the problem. FMSLEclipse shall support the following markers:

- Warning
- Error

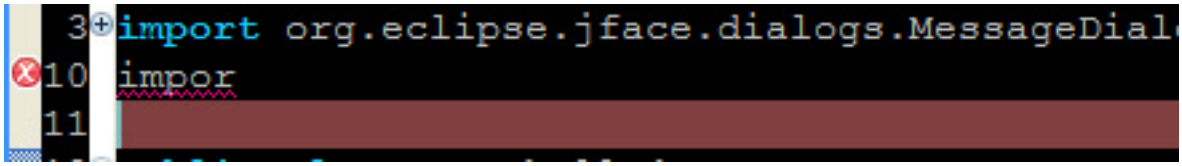

Figure 22: Problem Marker

C.3.4 Content Assist

Priority: Medium

FMSLEclipse shall provide content assist (figure 23) while editing a FMSL source file. Content assist will auto complete code as it is typed by providing a list of possibilities.

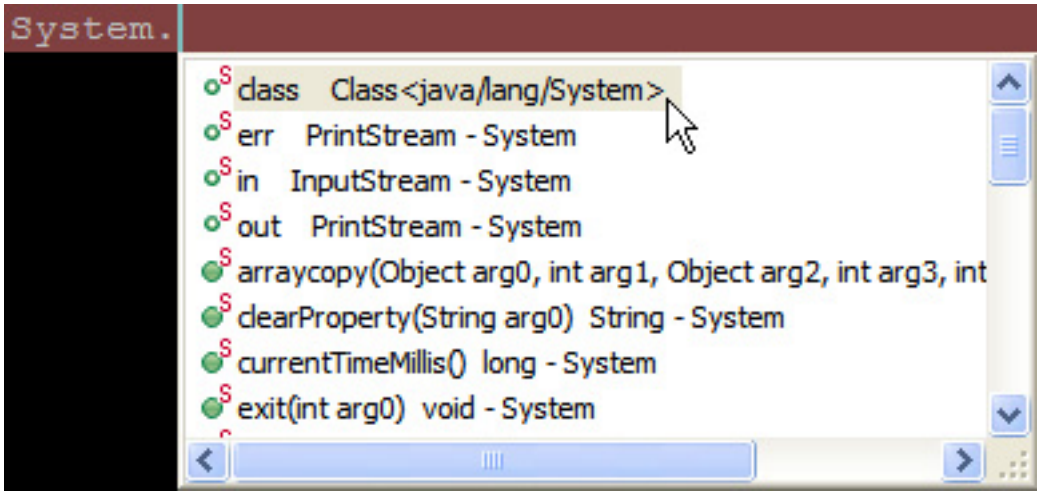

Figure 23: Content Assist

C.3.5 Code Folding

Priority: Low

FMSLEclipse shall allow the user to fold sections of code in order to improve readability. Folding code reduces a section of code so that only one line is viewable. The following FMSL sections are foldable:

- Module
- Object
- Operation

C.3.6 Line Level

Priority: Low

FMSLEclipse shall indicate that a section of code is causing a problem by underlining it with a colored wave like line. FMSLEclipse shall underline warnings in yellow and errors in red.

C.3.7 Quick Fix

Priority: Low

FMSLEclipse shall provide quick fixes for each problem makker generated (see C.3.3). A quick fix (figure 24) is a set of resolutions for the problem that the user may select from to automatically solve the problem.

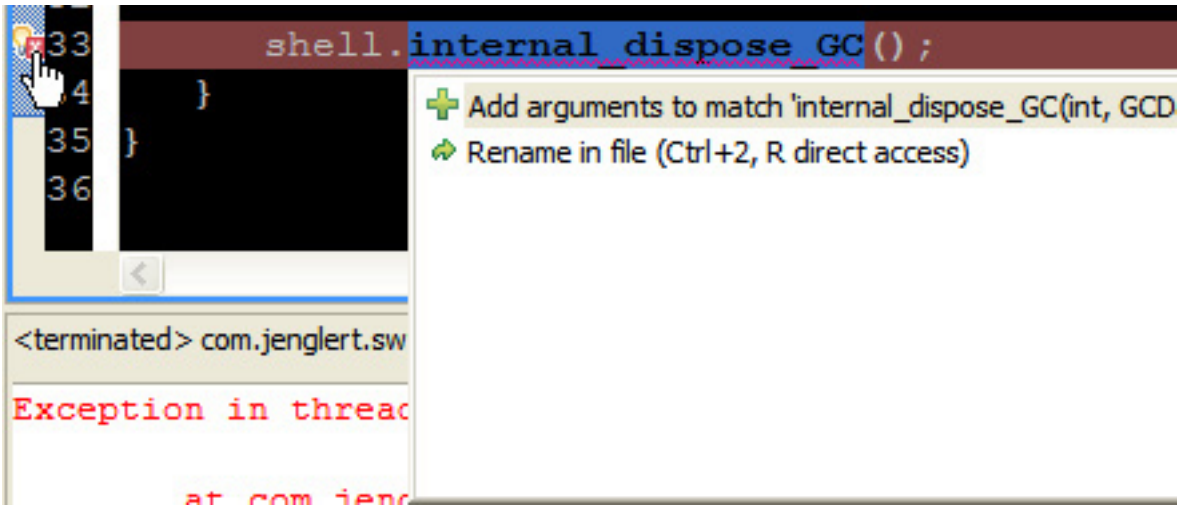

Figure 24: Quick Fix

C.3.8 Text Hover

Priority: Low

FMSLEclipse shall display text hovers (figure 25) for the FMSL syntax. A tool tip like text box is displayed when the user hovers the mouse over a FMSL keyword in the editor. There should be a user preference to turn this feature on and off.

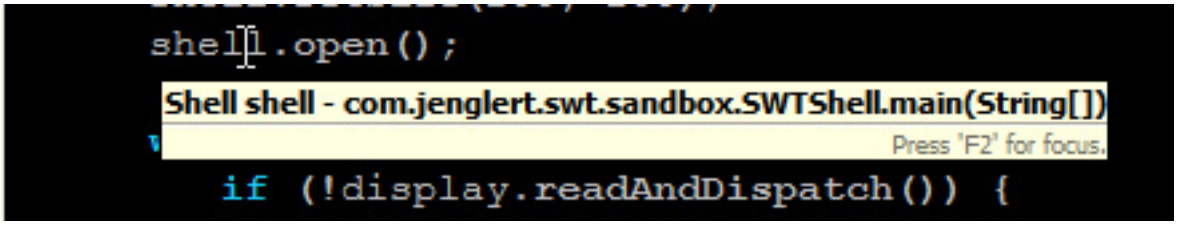

Figure 25: Text Hover

C.4 Outline View

The outline view (figure 26) in Eclipse displays a hierarchical structure of the current source file being edited. A specialized outline view for FMSL will display the structure of the current FMSL source file being edited.

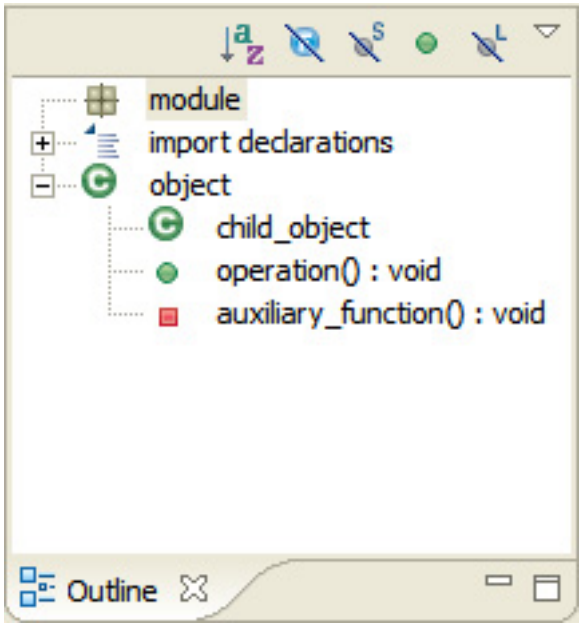

Figure 26: Outline View

C.4.1 Structure

Priority: High

FMSLEclipse shall display the structure of the FMSL source file being edited in a hierarchical manner. The following hierarchy will be used:

- Module

- Imports
- Exports
- Object
  - Child Object
- Operation
- Value
- Auxiliary Function

C.4.2 Linking

Priority: Medium

FMSLEclipse shall link elements in the outline view to the source file. When the user selects an element in the outline view, the corresponding code in the source file will be highlighted.

C.5 Project View

The navigator view (figure 27) in Eclipse displays a hierarchical structure of the workspace file system. A specialized project view for FMSL will be built on top of the navigator view to provide more information about FMSL projects and files.

C.5.1 File Hierarchy

Priority: Medium

FMSLEclipse shall display the structure of a FMSL source file when the file is expanded (figure 28). It will display the same information as the outline view (see section C.4) and use the hierarchy specified in section C.4.1.

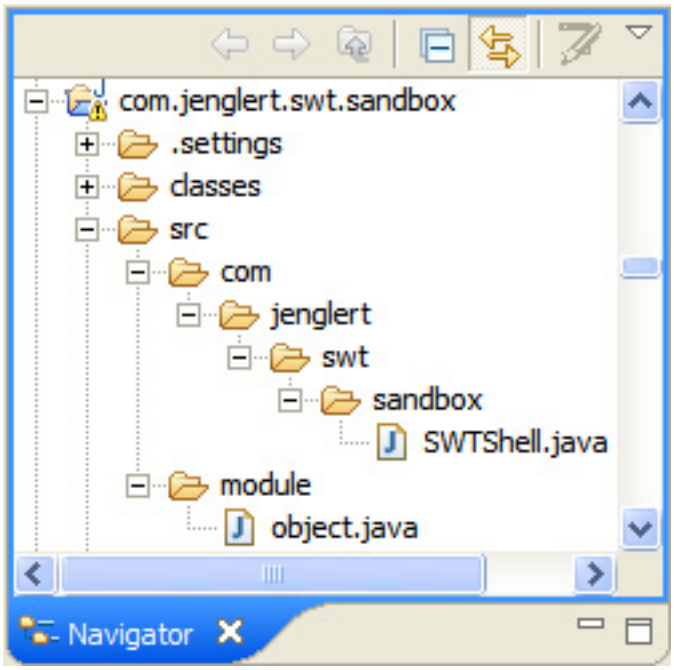

Figure 27: Navigator View

C.6 Preferences Dialog

The preferences dialog in Eclipse contains the preferences for each plugin. Plugins are responsible for contributing preferences to this dialog. FMSLEclipse will contribute preferences to the dialog that will be contained under the FMSLEclipse category.

C.6.1 Editor Preferences

Priority: High

The editor preference page is located under the FMSLEclipse section in the preferences dialog. The syntax highlighting preference page (figure 29) is located under the editor preferences. This page will allow the user to specify the color of different FMSL syntax.

C.6.2 Builder Preferences

Priority: High

The builder preference page is located under the FMSLEclipse section in the preferences dialog. This page will allow the user the specify a path to the external FMSL compiler. The

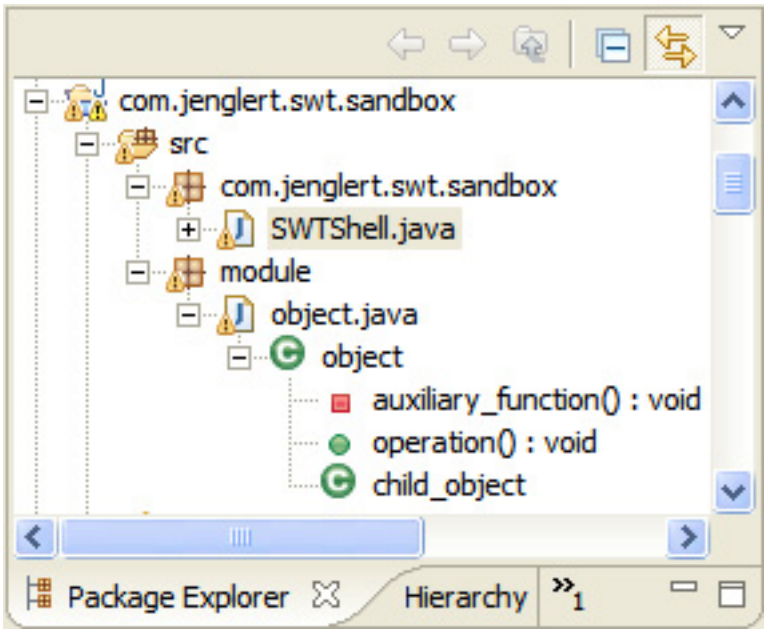

Figure 28: File Structure

page will look similar to figure 30.

C.7 Wizards

C.7.1 New Project Wizard

Priority: High

FMSLEclipse shall use a wizard (see figure 31 to guide the user in creating a new FMSL project. The wizard with prompt the user for the following information:

- Project Name
- Project Location
- Output Directory
- Source Directory

C.8 Non Functional Requirements

C.8.1 User Interface

FMSLEclipse shall adhear to the Eclipse User Interface Guidelines Version 2.1 found at:

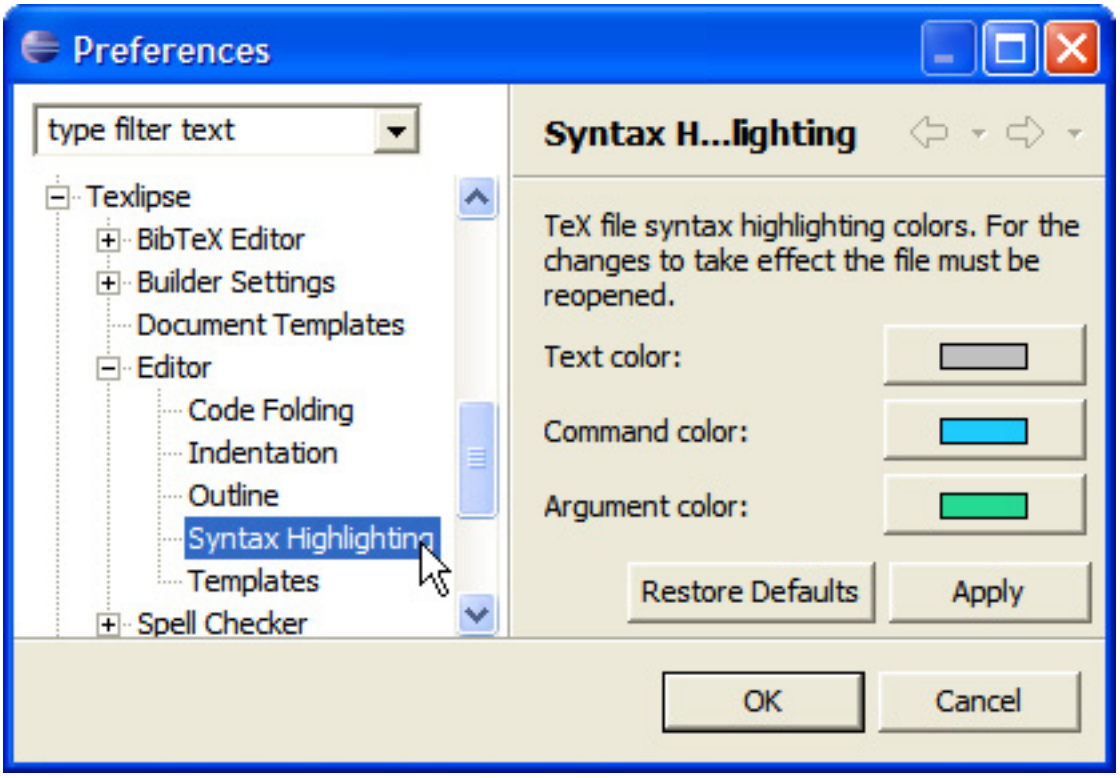

Figure 29: Syntax Highlighting In Editor Preferences

http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html.

C.8.2 Software Interface

FMSLEclipse shall interface to the existing FMSL compiler by using a compiler executable.

C.8.3 Delivery

FMSLEclipse shall be delivered as one or more plugins managed by a feature. A feature specifies the prerequisites and organization of plugins. Prerequisites are feature and plugin dependencies. Each plugin may be packaged as a jar or zip file.

C.8.4 Installation

FMSLEclipse shall be installed through the Eclipse update manager. The update manager allows an Eclipse user to specify a URL to download features from.

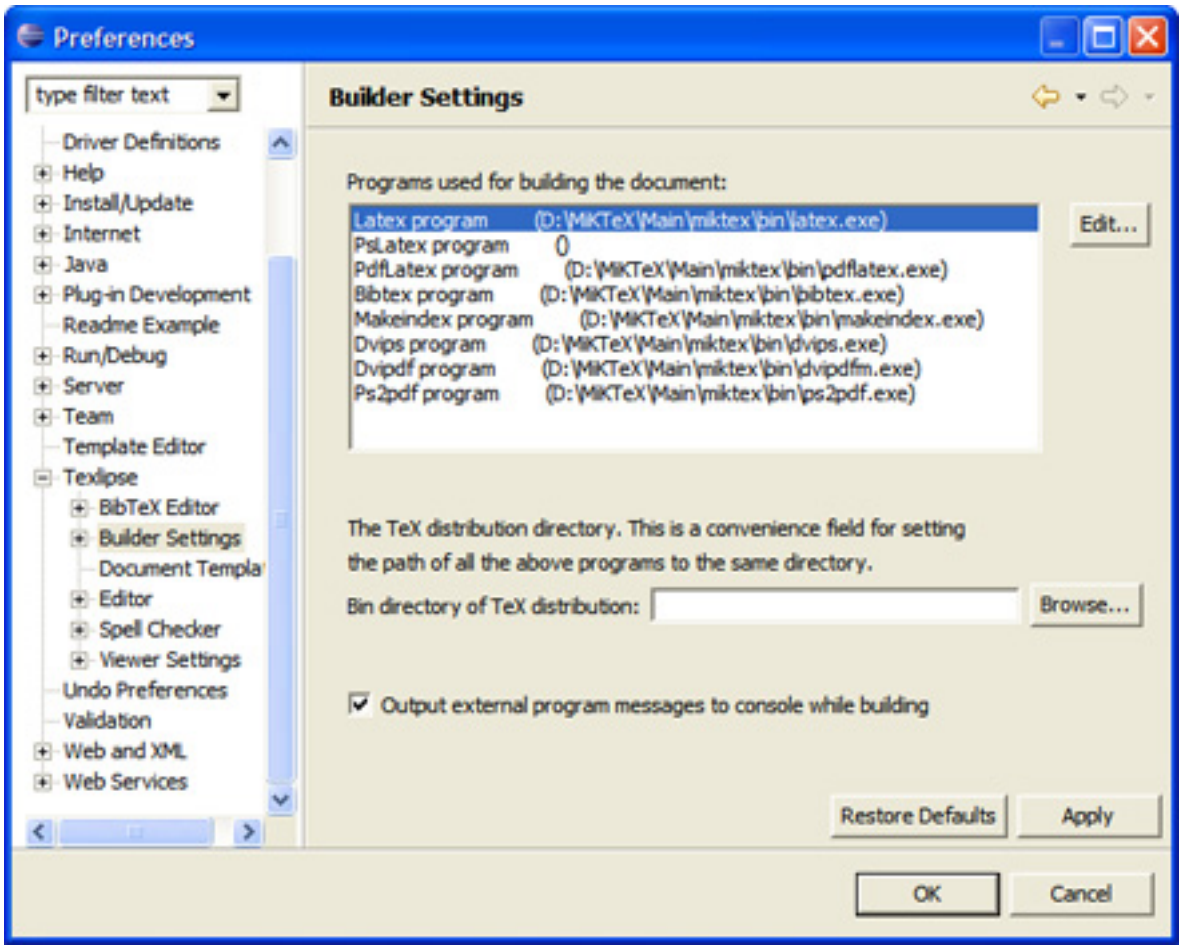

Figure 30: Builder Preferences

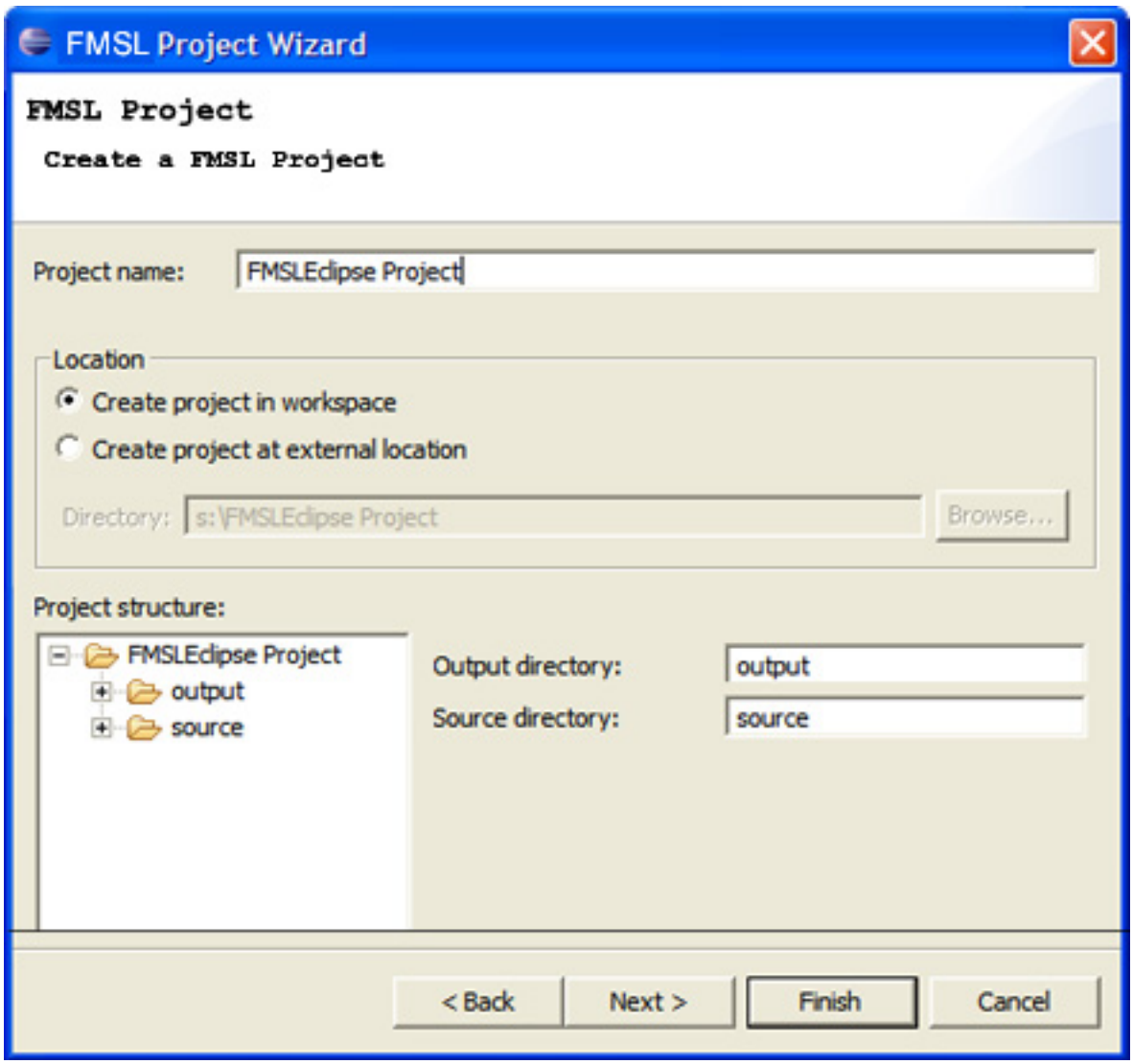

Figure 31: New Project Wizard

References

[1] John Arthorne Chris Laffra. *Official Eclipse 3.0 FAQs*. Addison-Wesley, 2004. 3.2

[2] Jim des Rivieres and Wayne Beaton. Eclipse Platform Technical Overview, Apr 2006. www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html. (document), 3.1.1, 3.1.2, 11, 3.1.3, 12, 6.2, 18, 19, B

[3] Dr. Gene Fisher. Rsl: A requirements specification language, 2000. www.csc.calpoly.edu/ gfisher/classes/308/doc/ref-man/. 1.1

[4] Dan Kehn John Kellerman Pat McCarthy Jim D'Anjou, Scott Fairbrother. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2nd edition, 2003. 5.2.4

[5] junit@objectmentor.com, 2006. www.junit.org. 5.3.1

[6] tienne Gagnon. Sablecc, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, Mar 1998. 4.2.1, 4.2.1

[7] wikipedia community, 2006. www.wikipedia.org. 4.2.1