

EDT - Eiffel development toolkit

Master Thesis

Author(s):

Ohnsorg, Reto

Publication date:

2009

Permanent link:

<https://doi.org/10.3929/ethz-a-006007008>

Rights / license:

In Copyright - Non-Commercial Use Permitted

EDT - Eiffel Development Toolkit

Master Thesis

By: Reto Ohnsorg
Supervised by: Marco Trudel
Prof. Dr. Bertrand Meyer

Student Number: 97-253-108

Contents

1 Abstract.....	1
2 Introduction.....	1
3 Existing Eclipse IDEs.....	2
4 Evaluation.....	4
4.1 EEDT project.....	4
5 My Extensions.....	5
5.1 Project Refactoring.....	5
5.2 Compiler Integration.....	6
5.2.1 Code Base.....	8
5.3 Project File Handler.....	8
5.3.1 Code Base.....	9
5.4 Compiler Setting GUI.....	9
5.4.1 Code Base.....	10
5.5 Compiler Output Parser.....	10
5.5.1 Code Base.....	12
5.6 Run Configuration / Console View.....	13
5.6.1 Code Base.....	14
5.7 Abstract Syntax Tree (AST).....	14
5.7.1 Results.....	16
5.7.2 Testing.....	18
5.7.3 Code Base.....	18
6 Status.....	19
7 Conclusion.....	19
8 Future Work	20
9 References and Background Material:.....	22
9.1 Reading list.....	22
9.2 References:.....	22
10 Appendix.....	23
10.1 Project Schedule.....	23
10.1.1 Intended Schedule.....	23
10.1.2 Actual schedule.....	23
10.2 Installation & First Steps.....	24
10.2.1 Requirements.....	24
10.2.2 Installation Steps.....	24
10.2.3 First Steps.....	24
10.3 Feature List.....	25
10.4 EEDT CVS Repository.....	25

1 Abstract

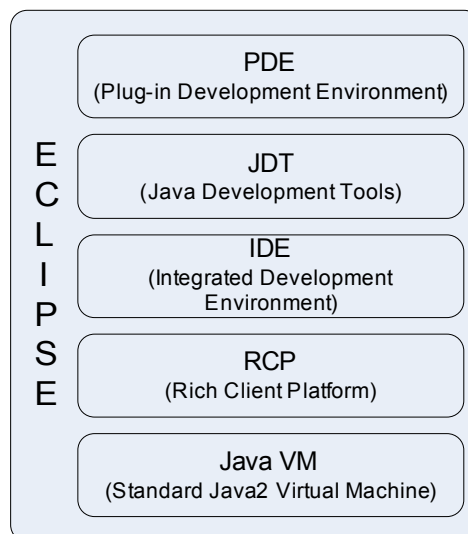
Eclipse is a multi-language software development platform comprising an IDE and a plug-in system to extend it. It is written primarily in Java and is used to develop applications in this language. Through plug-ins, Eclipse also allows to develop applications in other languages like C, C++, Cobol, Python, Perl, PHP among others.

The outcome of this thesis is an Eclipse plug-in which provides basic functionality to write Eiffel applications which is based on an inactive open source project which had been stopped in 2005.

This report covers an evaluation of existing solutions for other languages than Eiffel, an overview on what is available for Eiffel, extensions done to the existing project as well as a project status, problem section and ideas for future work on this project.

2 Introduction

Eclipse is an open platform. It is designed to be easily and infinitely extensible by third parties. The Eclipse architecture consists of five main layers as shown below:



PDE: It provides all tools necessary to develop plug-ins. This is the toolkit used to create plug-ins for Eiffel development. PDE offers a lot of wizards and templates to integrate the basic functionality needed for an IDE (Editor, console view, search functions, property pages...)

JDT: This is a complete java IDE and a platform itself. It is the standard IDE that comes with an Eclipse distribution.

IDE: IDE is a tools platform and rich client application. We can build various form of tooling by using IDE for example Database tooling.

RCP: On the bottom is RCP which provides the architecture and framework to build any rich client application.

Java VM: The standard Java virtual machine is the very bottom of the Eclipse framework.

Most implementations done during this work are extensions of the PDE framework.

3 Existing Eclipse IDEs.

To get an idea of the look & feel of an Eclipse based IDE as well as the basic functionality provided by them, several open source products were installed and run.

Plugin	Language	Version	Latest Activity	Reference
mrrnmhrm	D	0.3.0	October 2008	[1]
Emonic	C#	0.3.0	October 2007	[2]
CDT	C/C++	6.0.X	November 2009	[3]
ODT	OCaml	1.2	February 2009	[4]

Those four plug-ins were used to create a simple “Hello world” application. It has shown that all of them look & feel familiar to JDT and aim to provide the same features according to its forum entries & project road map. The most common JDT features are:

- Editor – syntax coloring, code resolving, import assistance, content assistance (auto completion, quick fix, help & tips)
- Resource explorer view – shows the resources included in a project as a tree view (projects, classes, libraries, includes...)
- Outline view – shows the structure of a Java compilation unit (class file) including methods, fields, type hierarchy
- Problem view – lists problems occurred during compilation and links them to the corresponding resource file
- Console view – displays text output of a console application and redirects text input to this application
- Refactoring support – method extraction, safe renaming and moving with reference updates etc.
- Wizards to create new elements – project, package, class, interface, etc.

- Search engine – find declarations of and/or references to packages, types, methods, and fields, scoped to the workspace, a working set or the current selection
- Update site – an easy to use Eclipse specific mechanism to install and update plug-ins online

The four plug-ins were evaluated with respect to the base features mentioned before and listed in the following table.

	CDT	Emonic	mrmnmhrm	ODT
Language specific editor	yes	yes	yes	yes
Resource explorer	yes	yes	yes	yes
Outline view	yes	yes	yes	no
Problem view	yes	yes	yes	yes
Console view	yes	yes	yes	yes
Wizard	yes	yes	yes	yes
Refactoring support	yes	yes	yes	no
Content Assistance	yes	yes	yes	no
Debugger integration	yes	no	no	no
Update site	yes	yes	yes	yes

Using these plug-ins has shown that CDT is the most elaborated plug-in and offers lots of features available in JDT. It is the only plug-in that has a fully integrated debugger. The CDT project is an official Eclipse project (supported by IBM, Nokia, Ericsson among others) and has a lot of contributors. Not only that the project is mostly in sync with the current Eclipse / JDT edition, it also comes with a lot of tutorials, help forums and also print media are available.

Mrmnmhrm and Emonic are still in beta release and have by far not as much community support as CDT. There hasn't been any project activity recently. In case of Mrmnmhrm, the fact that D is not that widely spread and hence has not a huge community might be a reason. C# is very commonly used with Microsoft Visual Studio or the MonoDevelop IDE [5] which might be a reason that the Emonic plug-in has not much support either.

ODT provides neither content assistance (auto completion) nor refactoring features and does not come with an integrated debugger. There hasn't been much activity the last few months but the project is still alive and seems to be ongoing.

4 Evaluation

The following three open source projects were found while researching on the internet.

Project Name	EFE	EDT	EEDT
Host	www.eclipse.audaly.com	https://sourceforge.net/projects/edt	https://sourceforge.net/projects/eedt
Status	obsolete	inactive	inactive
Last release	May 2004	December 2004	July 2005
Artifacts available	Source code	Discussion topics	Source code
Features implemented	<ul style="list-style-type: none"> - Source Highlighting - Compile / Build - Document generation - Supports SmartEiffel Compiler 	-	<ul style="list-style-type: none"> - Source Highlighting - Compile / Build - Cluster Browsing - Supports SmartEiffel Compiler

All listed projects are open source projects where there hasn't been any activity for at least four years. While the EDT project has remained at kind of brainstorming stage (no code, binaries available), the EFE project provided an implementation for Eclipse 2.0 but is marked as obsolete and no longer available on the Eclipse plug-in website [6], where it was originally hosted.

The EEDT project was not active either but a source code base and its binaries were available on SourceForge [2]. According to its description, some basic features needed for this project seemed to be already implemented and I considered it as worthy to test them and see whether they were working and usable for this project.

4.1 EEDT project

Download and installation had shown that the project was not running under Eclipse Galileo V3.5.1. The EEDT was originally developed using Eclipse 2.1. After some code refactoring, it was possible though to figure out what features these plug-ins basically provide although most of them were not working properly.

The following table depicts the status of each feature implemented in EEDT V.0.1.0

Feature	Code Status	Documentation
Editor (Syntax Highlighting)	Runnable code	Not available
Refactoring	erroneous	Not available
Resource browser	Runnable code	Not available
Project Creation Wizard	Not runnable, basic implementation only	Not available

Class Creation Wizard	erroneous	Not available
Feature Creation Wizard	Not implemented	Not available
Compiler Integration*	Not runnable,	Not available
Debugger Integration*	Not runnable	Not available

*Support for SmartEiffel compiler [8] & GOBO Libraries [9]

Eclipse provides extensive support to migrate to a current version (wizards, tools, FAQ, forums). A code walk through as well as reading the migration section of the eclipse documentation had led to the conclusion to reactivate the existing EEDT project and extend it with the components needed to provide a running Eiffel IDE.

Pros	Cons
Many hours (useful) software development done by several implementers	No documentation available, code walk-through takes time to get an overview of the current code base. Bug-fixing might become hard and time-consuming.
Editor plug-in works as is	Refactoring needed due to framework update (Eclipse V2.1 to V3.5.1).
Resource browser plug-in works as is	

5 My Extensions

5.1 Project Refactoring

For this study, EEDT V 0.1.0 was used as base version to do further implementation. According to Chapter 4.1, most of the basic features needed were not working properly. To get a solid code base to work on, a code refactoring had been done first which included the following tasks:

- Upgrade to JDK 1.5
- Infer generic type arguments where possible and code was reused (Generics were introduced in JDK 1.5)
- Code removed where no longer needed, mostly compiler integration code since the EEDT was intended to work with a SmartEiffel [8] compiler
- Project structure updated since the project didn't build anymore with Eclipse 3.5.1 due to binary cycles
- Plug-in definition file for each sub plug-in adapted to Eclipse 3.5.1 since some parameters were no longer supported or substituted by other ones.

The EEDT plug-in consists of nine Eclipse projects. During the refactoring work done, I had to browse through each of the projects to fix errors and figure out what is the main purpose of a project and its packages. The list below gives an overview of these projects:

Project name	Purpose	Packages	Classes	TLOC*
ch.ethz.edt.core	Contains most of the domain logic including Eiffel definition, project model, AST parser, builder, compiler driver, utilities...	45	417	42806
ch.ethz.edt.ui	Contains the user interface dialogs and views (editor, resource navigator, property pages, preferences...)	156	624	45217
ch.ethz.edt.debug.core	Contains base classes to run and debug applications. The debug support was implemented for GOBO Eiffel and is not reusable.	4	5	124
ch.ethz.debug.ui	Contains the user interface to setup and start a launch configuration.	4	7	397
ch.ethz.edt.runtime	Provides runtime dependent functionality like resource resolving, preferences, etc.	9	19	1158
ch.ethz.edt	Contains only text artifacts and help content (cheat-sheet), no source code.	-	-	-
ch.ethz.edt-feature	Defines the set of plug-ins to be built and exported to use in Eclipse. No source code, just XML definitions.	-	-	-
ch.ethz.edt-update	Update site project. Holds an update site definition and all binaries needed to install the plug-in.	-	-	-
ch.ethz.edt.doc	Contains artwork, templates, diagrams, wiki...no source code	-	-	-

* Total lines of java source code

The metrics were calculated with the 'state of flow' open source metrics plug in [10] for Eclipse.

5.2 Compiler Integration

One of the main tasks was to integrate the ISE Eiffel compiler into the existing plug-in. Integration in this context means to start a compiler and provide expected input as well as reacting on its output e.g. show status messages and errors in Eclipse views. The expected input of a compiler is usually a single class file, a few command line parameters and/or a configuration file containing the source files and libraries needed to build an

application. Such a configuration file has to be created manually according to the compiler specification and therefore is not part of the Eclipse framework. Nevertheless, Eclipse provides classes and mechanisms to keep track on what resources have changed in a project, by firing events. A potential consumer can subscribe for events like MOVED, DELETED, REPLACED, PRE_DELETE, etc. and react when they occur. Getting informed about resource changes allows to decide on what resources have to be (re)compiled. Since in many applications there is more than a a single compilation step involved, 'building' is a more general notion. A builder component knows all the details to transform a project from its source files and libraries to the output needed e.g. binaries, intermediate code, executables, etc.

The PDE framework supports basically two different build modes:

- Full build
- Incremental build

A full build compiles an entire project from scratch whereas an incremental build calculates a delta to a previous build and determines all resources to be recompiled. The calculation of this delta is a tedious task and needs profound knowledge of the compiler used, build order calculation, dependencies involved, etc. Overcoming these difficulties leads to a reduced compilation time since only parts of a project have to be rebuilt.

Unlike many compilers which just support simple batch compilation, the ISE Eiffel compiler has the ability to do incremental compilation on its own and takes care of resource changes. This feature is known as 'Melting Ice Technology' [11]. Using this ability makes it useless to implement an incremental builder in Eclipse PDE. A simple builder does the job since the compiler decides on what files have to be built. One has to provide a configuration file though to let the compiler know what files and libraries are part of the project to be built.

The three variants presented to integrate an ISE Eiffel compiler are:

- A) Implementing a simple builder in Eclipse PDE which allows to do full builds only
- B) Implementing an incremental builder in Eclipse PDE
- C) Implementing a simple builder and take advantage of the compiler's incremental build ability

The pros and cons are listed in the table below:

Variant	A	B	C
Pros	- simple implementation - easy to test	- efficient for large scale projects - automatic build possible - fine grained control over build process	- incremental compilation feature - significantly less development effort than variant B - well-tested since used in Eiffel Studio

Cons	- inefficient - unusable for large scale projects	- high development costs - extensive testing needed - no 'built-in' support for files outside Eclipse workspace.	- not much control of build process → harder to integrate user feedback like error messages, markers, etc.
------	--	--	--

Due to the fact that the Eiffel compiler already has a well-established incremental compilation mode and a new implementation of such a mechanism would have led to a more time consuming task, I decided to implement variant C.

5.2.1 Code Base

Package	Class	Description	ToDo
ch.ethz.edt.core.compiler.driver	CompilerDriverISE	Provides methods to run the compiler and set the parameters used	-
ch.ethz.edt.internal.core.build	EiffelProjectBuilder_ISE	Executes the framework build commands and sets error and warning markers	-

5.3 Project File Handler

The ISE Eiffel compiler uses a configuration file to determine what resources are part of a build process, as well as to read user settings which are not part of the input argument list. Eiffel project files [12] are formatted in XML notation and have a '.ecf' file extension. The implemented project file handler is able to create a project file and read and write its settings. Changes in the project file are done by the compiler setting user interface. The currently implemented project file handler supports the most important settings to compile an Eiffel project, but not yet all of them. Adding more settings might be done with ease analogous to the existing ones.

The following settings are already supported:

- Option warning
- Option syntax level
- Options assertion (supplier-) precondition / postcondition / check / invariant / loop
- Parameter precompile name and location
- Parameter precompile location
- Parameter system name
- Parameter library name and location
- Parameter root class

- Parameter root feature
- Parameter cluster name
- Parameter target name
- Setting multithreaded
- Setting console application

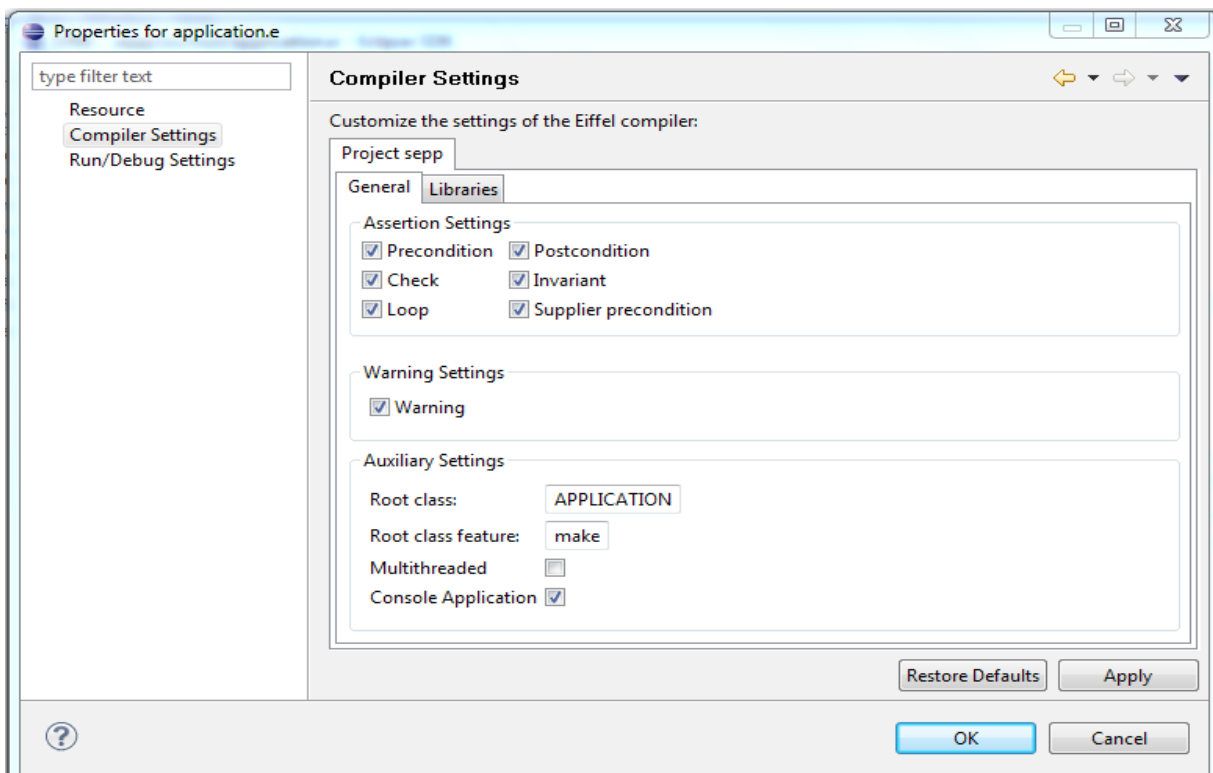
The handling of the XML structure was done by using the JDOM [13] API. JDOM is an easy to use and java based 'document object model' for XML.

5.3.1 Code Base

Package	Class	Description	ToDo
ch.ethz.edt.core.compiler.configuration	ProjectFileHandler	Creates, reads and writes the compiler configuration file (.ecf)	-

5.4 Compiler Setting GUI

To change the basic project settings like source folder, output folder, application type, libraries to link etc., a GUI is provided as depicted below



All settings are made persistent in the project file [See Chapter 5.3].

5.4.1 Code Base

Package	Class	Description	ToDo
ch.ethz.edt.ui.properties.compiler	CompilerPropertyPage	Creates the property page GUI.	-

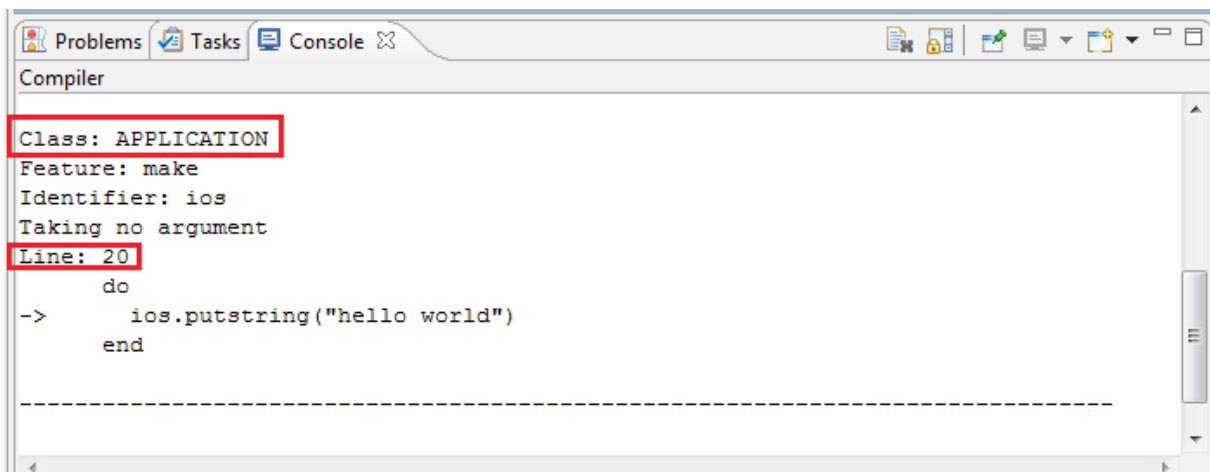
5.5 Compiler Output Parser

To react on the compiler output and mark possible errors and warnings in source (class) files, a parser is needed to process the output stream. Research on the internet has shown that there is no widely used definition or standard to define how to format errors and warnings, for example in an XML structure, to make parsing independent of a compiler. This was expected since most of the compilers provide similar information like error number, error message, file name, line number, severity and so on. For unknown reason it is not this way and parsing a compiler output becomes a very specific task.

According to Manuel Stapf from ISE Eiffel, there's no output parser for the current ISE compiler available and had to be done myself. Also, there is no specification available on how and what kind of output is generated and how this output is formatted. Many tries have shown that there are two different kind of errors possible:

- Configuration file error / warning
- Source file error / warning

The following compiler output indicates a source file error:



```

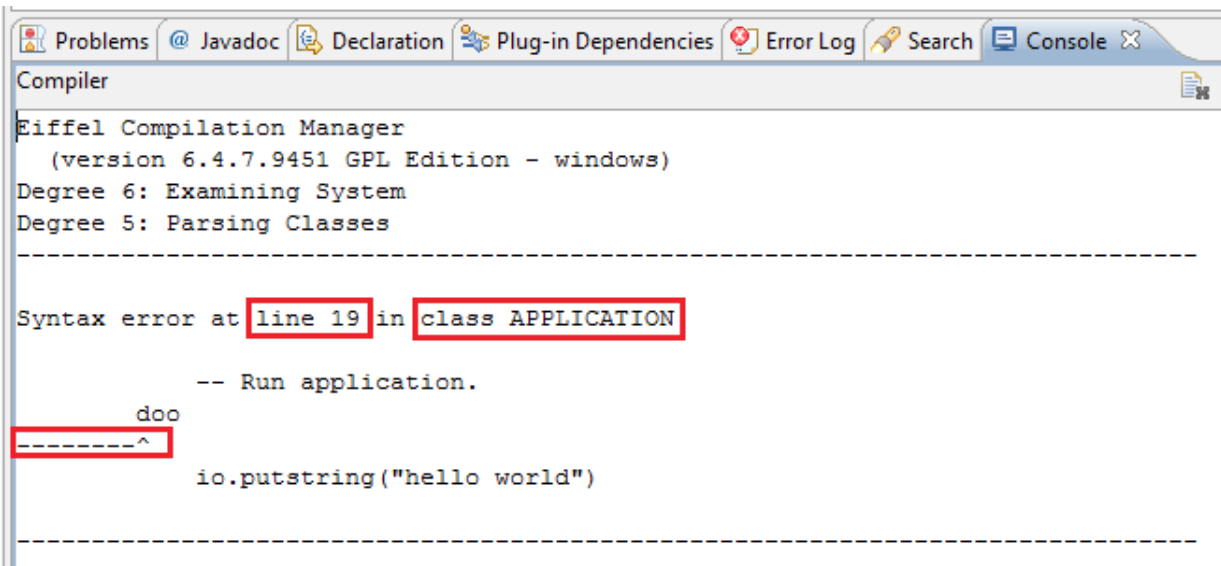
Compiler
Class: APPLICATION
Feature: make
Identifier: ios
Taking no argument
Line: 20
    do
->    ios.putstring("hello world")
    end
  
```

From this output string, we can extract class name, line number, and column number. Since it is common practice that a source file is named after the class it contains, one can easily derive the source file name.

```
6
7 class APPLICATION
8
9 inherit
10 ARGUMENTS
11
12 create
13     make
14
15 feature {NONE} -- Initialization
16
17     make
18         -- Run application.
19     do
20         io.putstring("hello world")
21     end
22
23 end -- class APPLICATION
```

Having source file name and line number, it is possible to set an error marker in the Eclipse view to help a user to find this error.

Although this looks pretty straightforward, problems occurred while using this approach. While testing the parser with a few different compiler errors, it appeared that the output format varies depending on the type of error.



The screenshot shows the Eclipse IDE's Console view. The title bar includes 'Problems', 'Javadoc', 'Declaration', 'Plug-in Dependencies', 'Error Log', 'Search', and 'Console'. The main content area displays the following text:

```
Compiler
Eiffel Compilation Manager
(version 6.4.7.9451 GPL Edition - windows)
Degree 6: Examining System
Degree 5: Parsing Classes
-----
Syntax error at line 19 in class APPLICATION
    -- Run application.
do
-----^
    io.putstring("hello world")
-----
```

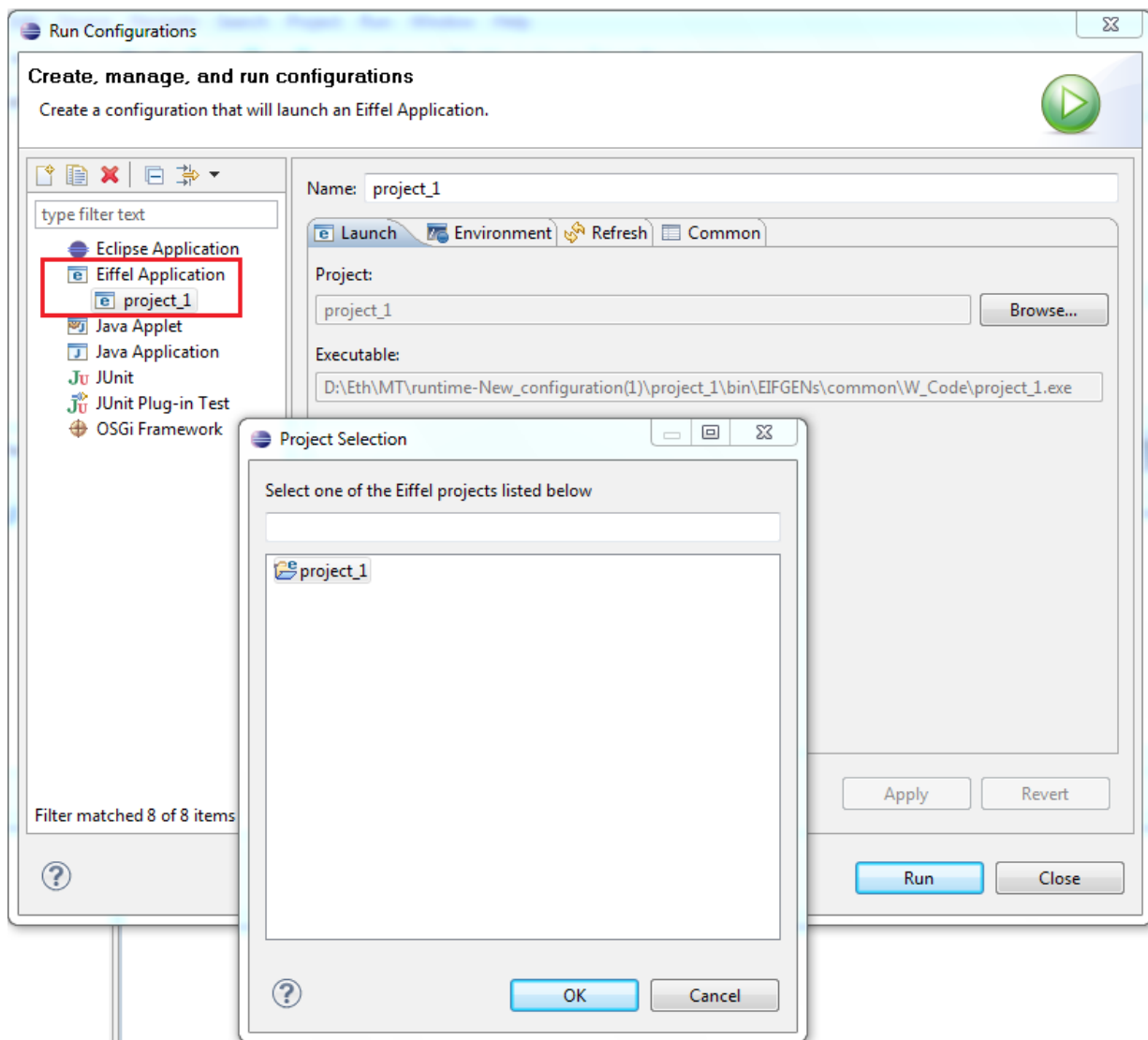
Although both error output strings contain a line number and class name, they are different to parse. The currently implemented version is able to recognize both error types. Using the parser during the last few weeks have shown, that these two types of error messages cover most of the cases, but there is no guarantee that there are no other cases possible. A specification of the error output format would have helped to save development time and make sure that the implementation works for the cases specified.

5.5.1 Code Base

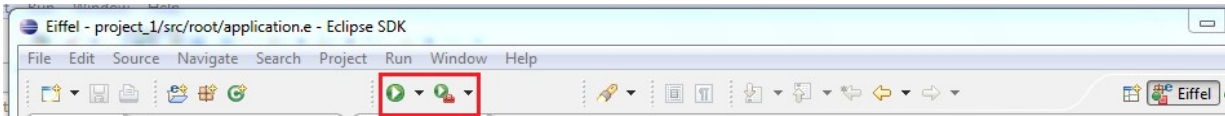
Package	Class	Description	ToDo
ch.ethz.edt.core.compiler.driver	CompilerOutput ParserISE	Parses the output stream of the ISE Eiffel compiler and creates an event for each error or warning.	-


5.6 Run Configuration / Console View

To run a program from the Eclipse workbench, a run configuration can be created manually by using the run configuration GUI:



A new run configuration can be created by double clicking on the 'Eiffel Application' icon and selecting a project on the 'Launch' tab. Once the configuration is created, it appears in the run menu:



Hint: If the 'run'  icon is being pressed and no configuration has been defined yet, a default run configuration named after the project's name will be created automatically.

5.6.1 Code Base

Package	Class	Description	ToDo
ch.ethz.edt.debug.internal.ui.launch	EiffelLaunchShortcut	Handles a framework launch event. Reads data defined in the launch configuration.	-
ch.ethz.edt.debug.internal.ui.launch	EiffelLaunchConfigurationTab	Creates the configuration tab GUI.	-
ch.ethz.edt.debug.core.launch	EiffelLaunchConfigurationDelegate	Executes the process (defined in the launch configuration).	-

5.7 Abstract Syntax Tree (AST)

To provide elaborated functionality like method renaming, dynamic auto completion for features etc., there is a need for an AST to access and change class information in a source file efficiently. For instance, to write a dynamic auto completion functionality for Eiffel, we're interested in the public features defined in a class and all base classes derived from. These features are then presented in a pop up menu from which a user can select one.

PDE comes with an AST parser for Java but not so for any other language. Research on the internet has led to the conclusion that there is no AST parser for the Eiffel language available, which is written in Java and provides an API to use within the PDE.

Various lexer/parser frameworks are available which generate a target language specific API (Java in this case) from a language specific definition (grammar). Such a grammar has to be defined mostly in BNF notation. The list below is an overview of some products available:

Tool	Target language	Eiffel Grammar available	Open Source
Yacc/Lex [14]	C/C++	yes	Yes
ANTLR [15]	C / C++ / C# / D, Java, JavaScript, Python , Ruby, ActionScript	Yes, but outdated (2003 – not working with current ANTLR since written for v2 and not compatible with v3)	Yes
JavaCC [16]	Java	No	Yes
UltraGram [17]	C++, Java, C#, VB.Net	No	No

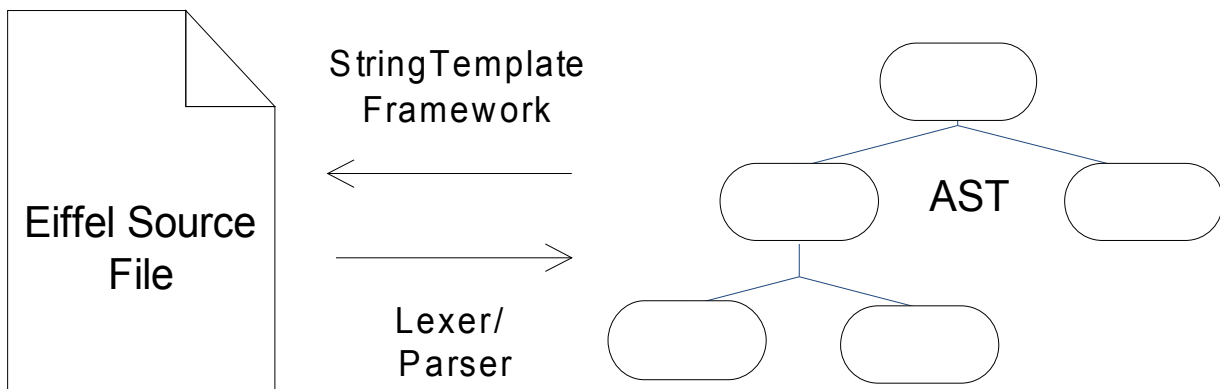
No product was found that supports Java as target language and comes with an Eiffel grammar. Several ideas came up to get an Eiffel AST for Java:

- Rewriting existing Yacc/Lex grammar for JavaCC
- Updating the existing grammar for ANTLR
- Generating C++ code from an existing Eiffel grammar and rewrite it in Java by hand

The outdated Eiffel grammar for ANTLR was written for version 2.X and is not compatible with version 3 and above. There is an experimental migration tool (v2v3.exe) in the download section of the official ANTLR website [15] available. It is supposed to migrate grammars written in v2 to v3, but all tries to run it with the existing Eiffel grammar have failed and a post in the ANTLR forum has not been answered so far.

Another aspect to keep in mind is that this AST is used to implement functions like auto completion and refactoring support, but not to implement a compiler as Yacc/Lex is often used for. The intended applications of the AST differ from building a compiler in the sense that there is a need to do changes on the source file parsed e.g. to rename variables, add a feature, definition body, etc.

ANTLR provides *StringTemplate*, an engine to change the source file (where the AST was created from) based on user defined templates. This provides a bidirectional way to parse an AST from a source file, do changes on it, and save it back as source file. The modified source file has to be compilable after any refactoring!



I decided to use ANTLR for the following reasons:

- Pure Java solution, but other popular target languages available
- Provides useful features beyond parsing and lexing – the *StringTemplate* engine.
- Open source
- Widely used, ongoing development activities for 20 years
- Lots of help material online

5.7.1 Results

The following components are implemented:

- Partial definition of an Eiffel grammar
- Code generation for Java
- Wrapper around generated code to create AST
- Experimental implementation of a feature look-up functionality
- Basic function tests with commonly used classes

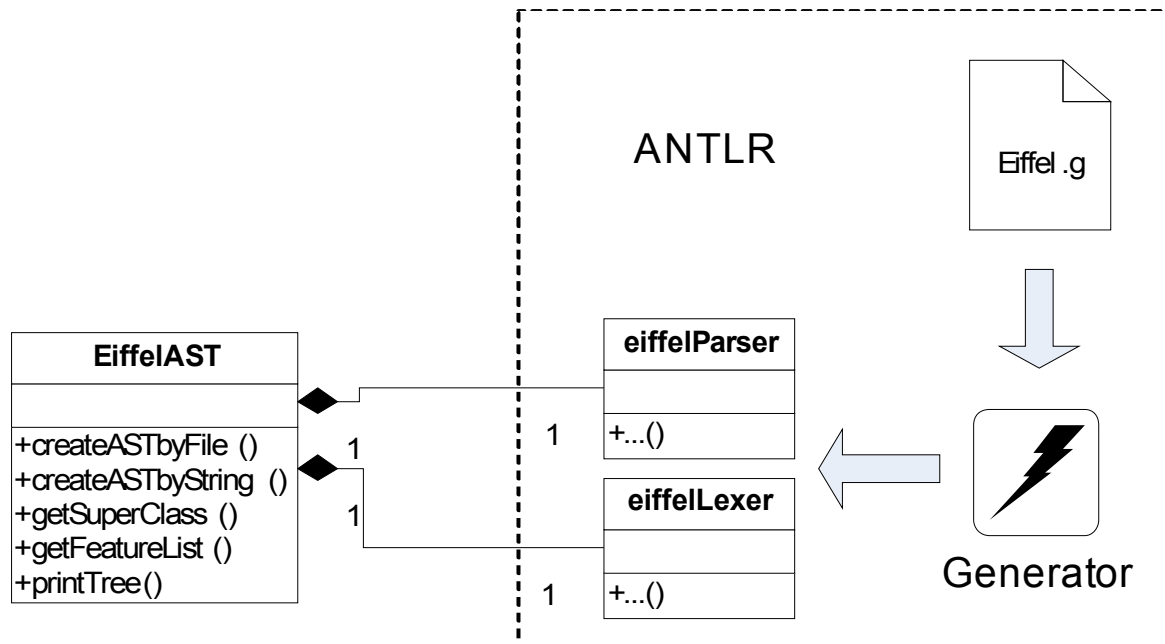
Eiffel Grammar:

The Eiffel grammar at this point is able to resolve an Eiffel source file down to feature declaration. It is not possible to get information about the implementation body within a feature though. I stopped at this level due to time constraints. The following overview shows what information can be extracted from the AST implementation:

Group	Type	Comment	Node
Index			INDEXING
Class	Name		CLASS_DEF
	Modifier	Class modifiers: Deferred, expanded, frozen	MODIFIER
	Creator	All creator names	
	Base Class	All base classes inherited from	SUPERCLASS_DEF
	Invariant		INVARIANT
Feature	Name	Feature name	FEATURE_DEF
	Modifier	Feature modifiers: Deferred, expanded, frozen	MODIFIER
	Argument Name		ARG_DEF
	Argument Type		ARG_DEF
	Return Type		RET_DEF
	Precondition clause		PRECONDITION
	Postcondition clause		POSTCONDITION

Java AST Interface

The picture below shows all parts of the implemented Eiffel AST:



Basic Auto Completion Function

A first application of the AST described above is a very basic, not to say experimental, auto completion function. It takes advantage of the facts that

- libraries included in EiffelStudio are distributed with its source files
- Eiffel class files are named after the class they implement

This allows to look-up features in libraries and display them in the Eclipse editor a similar way Eiffel Studio, Eclipse or Microsoft Visual Studio do.

So far, the AST interface provides only three methods needed to implement the basic auto completion function:

- String[] getSuperClass()
- String[] getFeatureList()
- createsASTFromFile(String aFileName)

The basic process to get all features is straightforward.

- 1) Create an AST for the source file currently edited in the editor
- 2) collect the features defined by calling `getFeatureList()`
- 3) call `getSuperClass()` to get all classes inherited from
- 4) resolve file name, find file in the library path
- 5) repeat 1-5 until `getSuperClass()` returns no more base classes

5.7.2 Testing

It has been shown that creating a new language grammar is quite a tricky and time consuming task, especially when it comes to testing (Although Eiffel is a straightforward implemented language and there are not as many legacies to take care of as in other languages).

ANTLR provides ANTLRWorks as a helpful tool to create and test grammars. It includes a debugger to lex and parse source files step by step, which I used to test my grammar with the following sources files from the Eiffel base library:

- any.e
- iterator.e
- disposable.e
- arguments.e
- container.e
- random.e
- arguments.e
- comparable.e
- primes.e
- bilinear.e
- numeric.e
- c_string.e
- linear.e
- exceptions.e
- active_list.e
- list.e
- memory.e
- interactive_list.e
- bounded.e
- pointer.e
- tree.e
- hierarchical.e
- file.e
- rt_extension.e

These classes were chosen because they are fundamental in Eiffel applications.

It is completely clear that testing a grammar with a few base classes is not enough at all to ensure proper functionality. One of the next steps would have been to set up an automated testing environment to do tests repetitively with ease. ANTLR provides *gUnit*, a testing framework analogous to widely spread JUnit.

One approach to test a grammar would include to configure gUnit the way, that any committed changes in the grammar file yield to a test run of all class files included in the base libraries that come with EiffelStudio (This includes around 370 commonly used classes).

5.7.3 Code Base

Package	Class	Description	ToDo
ch.ethz.edt.core.model.eiffel.ast	EiffelAST	Wrapper which provides access to AST	-
ch.ethz.edt.core.model.eiffel.ast	EiffelParser	Generated Parser	-
ch.ethz.edt.core.model.eiffel.ast	EiffelLexer	Generated Lexer	-

ch.ethz.edt.core.model.eiffel.ast	Eiffel.g	Grammar file	<ul style="list-style-type: none"> - extend testing - Resolve local variables, assignments, function calls and boolean expressions for refactoring support like renaming or reference finder
-----------------------------------	----------	--------------	--

6 Status

The following overview shows the intended results and its current status

Priority	Task	Development status	Description
1	Base Plugin (Editing, Compilation, Run, Browse)	Working	EDT plug-in allows to write, compile and run simple Eiffel applications. Editor, resource navigator, console window & error view are the base components
2	Installer	Working	EDT can be easily installed from an update site
3	Project Wizard	Working	Creates a new Eiffel project and a simple application class.
4	Syntax highlighting	Working	Highlights Eiffel specific syntax
5	Code completion	Concept & experimental implementation	Code completion depends strongly on the AST/DOM. A first approach is designed and implemented, but only on an experimental level and still error prone.
6	Refactoring support	Concept & experimental implementation	Renaming is possible on file/folder level. No other support is implemented yet.
7	Debugging support	Not started	-

7 Conclusion

I've collected lots of experiences with Eclipse and the PDE framework while working on this project. It turned out that using the PDE framework makes sense to implement a new toolkit in a similar way others have done before. Nevertheless, it needs time to get familiar and become productive and efficient. Due to the sheer size of the Eclipse project, it is not always obvious

what components are already implemented to what extend, and therefore I used a significant amount of time to read forums, help content, tutorials, etc.

The decision to reactivate a frozen project turned out to be successful in the sense that it was possible to update it to the latest Eclipse and Java release and add the features needed to provide a running Eiffel Development Toolkit. It posed some risks though, since it was not completely clear at the beginning how flexible the existing code base was to implement new requirements. It has shown that bug-fixing was a tedious task since it took time to get familiar with the existing code and get a feeling where problems originate from. On the other hand, time-consuming tasks like implementing an Eiffel editor from scratch became no longer necessary.

The AST implementation is not at the maturity level I expected at the beginning of the project. On one hand the initial effort to get a first prototype which shows whether it works at all took more time than planned, on the other hand some change requests for the GUI came up and were higher prioritized than the AST component.

8 Future Work

The implemented EEDT V 0.2.0 is useful to create compact Eiffel applications for people who are already familiar with Eclipse. It is not yet a fully fledged IDE to use in large scale applications, mainly because debugging support is missing. To integrate a debugger into Eclipse would be a major task which needs quite a bit of time.

Another task is to continue working on the AST implementation which will enable some more useful features discussed earlier. The list below gives an overview of open tasks and a time estimation based on the experiences made during my work on EDT.

Task	Description	Time estimation
Project Settings	The task includes to extract all possible project settings in EiffelStudio and update the project handler (Chapter 5.3) as well as the GUI (Chapter 5.4)	1 week
AST / DOM	The task includes to set up a test environment to test the Eiffel grammar (Chapter 5.7) thoroughly.	1 month
Auto completion	Extend the experimental implementation which just extracts class & base class features of the currently edited class by pressing <i>ctrl + shift</i> . The finished auto completion function should react on the function call operator '.', ask the AST for type information (if available) and get available features for this class, which is already implemented (Chapter 5.7).	1 month (depending on extensions needed on AST interface)
Reference finder	The reference finder allows to determine all references to a variable or feature in a project.	Depending on AST
Renaming (refactoring)	The renaming function allows to rename a class name, feature name or local variable and updates	Depending on AST

	all references. This function may base on the reference finder function.	
Outline view	An outline view shows the structure of a compilation unit (class) in a tree view. Precondition, features, postcondition and invariant would be meaningful nodes to show. Again, the AST implementation has to provide the content for this view. The view part is similar to the implemented resource browser which displays the project structure on resource level (files / folder).	2 weeks
Debugger integration	Eclipse PDE provides the most common debug views like variable view, breakpoint view, etc. A major task is to provide wrappers to connect the Eclipse UI with the debugger specific interface.	2 month
Bugfix: Compiler settings	The compiler properties are currently set in the cluster browser by selecting an Eiffel source file, clicking right mouse button and selecting properties → compiler settings. This is not a proper way since these settings are valid for the entire project and therefore have to be moved to the project node. For unknown reason they are not visible on this level though. The following forum entry in the Eclipse community has not lead to success yet but might be checked in the future to get a helpful hint or solution: http://www.eclipse.org/forums/index.php?t=msg&th=156074&start=0&S=d265b9e49675b2fedb1c53186c19608f	-

To close this chapter, I'd like to spend a few thoughts on the pros and cons on whether it makes sense to actively push this project forward or leave it to its fate. Since all open source projects which intended to provide an Eiffel IDE for Eclipse have been on hold for at least four years, I conclude the interest within the Eiffel community to have such a tool is moderate. To maintain this project, a lively community that keeps this IDE in sync with EiffelStudio and provides help, tutorials, FAQ would be necessary. From this point of view, there's not really a reason to continue working on this project at the moment. Nevertheless, lots of software engineering students at ETH have programming knowledge in both, Eiffel and Java, and might contribute to this project as a semester thesis or as part of the Java course at ETH (...where it was mandatory in the past to contribute to an open source project or start a new one). This might help to build a community and draw more interest on this project.

9 References and Background Material:

9.1 Reading list

Eclipse Plugin Development Environment (PDE) documentation:

http://wiki.eclipse.org/index.php/Eclipse_FAQs#Plug-In_Development_Environment

Eclipse migration documentation:

<http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/porting/3.0/faq.html>

PDE “Hello World” tutorial:

<http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html>

Java & XML using JDOM:

<http://www.ibm.com/developerworks/java/library/j-jdom/>

ANTLR tutorial:

<http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.7762&rep=rep1&type=pdf>

ANTLR AST application:

<http://www.antlr.org/wiki/display/ANTLR3/Interfacing+AST+with+Java>

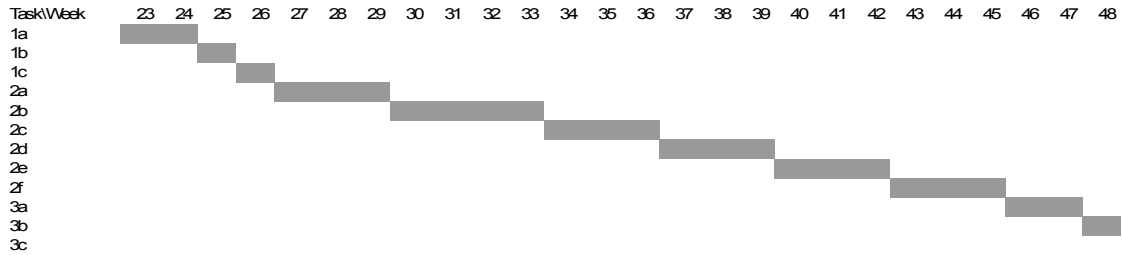
9.2 References:

- [1] <http://www.dsource.org/projects/descent/wiki/Mmrmhmr>
- [2] <http://emonic.sourceforge.net/>
- [3] <http://www.eclipse.org/cdt/>
- [4] <http://ocamltd.free.fr/>
- [5] <http://monodevelop.com/>
- [6] <http://www.eclipse-plugins.info/>
- [7] <https://sourceforge.net/>
- [9] <http://www.gobosoft.com/>
- [8] <http://smarteiffel.loria.fr/>
- [10] <http://eclipse-metrics.sourceforge.net/>
- [11] http://dev.eiffel.com/Melting_Ice_Technology
- [12] <http://dev.eiffel.com/ProposalProjectFiles>
- [13] <http://www.jdom.org/>
- [14] <http://dinosaur.compilertools.net/>
- [15] <http://www.antlr.org/>
- [16] <https://javacc.dev.java.net>
- [17] <http://www.ultragram.com>

10 Appendix

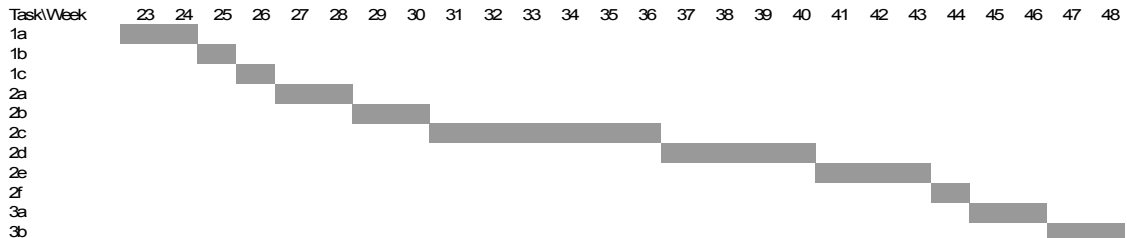
10.1 Project Schedule

10.1.1 Intended Schedule



- 1
 - a) Reading about Eclipse Frameworks, Eiffel Studio, Eiffel compiling...
 - b) Check for existing projects in this area (OpenSource)
 - c) Pro/ Con's of contributing to existing solutions vs. Starting a new project.
- 2
 - a) Implementation & Test Plugin for Project Creation Wizard
 - b) Implementation & Test Plugin (Incremental/Auto) Compilation Support
 - c) Implementation & Test Plugin (Incremental/Auto) Debugging Support
 - d) Implementation & Test Eiffel Editor (Syntax highlighting)
 - e) Implementation & Test Eiffel Editor (Syntax auto completion)
 - f) Implementation & Test Refactoring Support
- 3
 - a) Documentation Project Report
 - b) Reserve

10.1.2 Actual schedule



- 1
 - a) Reading about Eclipse Frameworks, Eiffel Studio, Eiffel compiling...
 - b) Check for existing projects in this area (OpenSource)
 - c) Pro/ Con's of contributing to existing solutions vs. Starting a new project.
- 2
 - a) ~~Implementation & Test Plugin for Project Creation Wizard~~ → Explore EDT code structure, refactor project
 - b) ~~Implementation & Test Plugin (Incremental/Auto) Compilation~~ → Project Creation Wizard (Projectfile-Handler, Registry-Handler, ...)
 - c) ~~Implementation & Test Plugin (Incremental/Auto) Debugging Support~~ → Compiler integration (Compiler driver, Eiffel builder, ou...)
 - d) ~~Implementation & Tests Eiffel Editor (Syntax highlighting)~~ → AST, Grammar, autocomplete
 - e) ~~Implementation & Tests Eiffel Editor (Syntax auto completion)~~ → Bugfixes & Changes according to MTRudel
 - f) ~~Implementation & Tests Refactoring Support~~ → Update Site / Project package structure adapted
- 3
 - a) Documentation Project Report
 - b) Reserve

10.2 Installation & First Steps

10.2.1 Requirements

The following software requirements are mandatory to install and run EEDT Version 0.2.0 :

- Eclipse 3.4 Ganymede or higher
- ISE EiffelStudio 6.4
- Microsoft Windows SDK 2008 (Optional)*

*The EiffelStudio installation is distributed with a GNU compiler to generate C code but recommends to use the Microsoft C/C++ compiler.

Hint: The plug-in may also run correctly with other ISE EiffelStudio versions, but it was developed and tested using V 6.4.

10.2.2 Installation Steps

- Install Microsoft Windows SDK first (Optional)
- Install EiffelStudio 6.4
- Install Eclipse 3.4
- Open Eclipse
- Go to Help → Install New Software...
- Press 'Add...' button, enter the EEDT installation site:

http://se.inf.ethz.ch/projects/reto_ohnsorg/edt_plugin

...and confirm with 'OK' button The EEDT plug-in appears in the list box.

- Select EEDT plug-in and press 'Finish' button. A warning appears that this plug-in contains unsigned content and the authenticity cannot be established.
- Confirm the security warning with 'OK' button
- A dialog appears and ask to restart Eclipse. Confirm by pressing 'Yes' button.
- The EEDT plug-in is now installed and ready to use.

10.2.3 First Steps

The following steps create a simple Eiffel console application with EEDT:

- Open File → New → Project. A wizard dialog opens.
- Choose 'EiffelProject' and press 'Next' button.
- Type a project name and optionally a location for this project.
- Press 'Finish' to confirm project name

The wizard creates a project, a root cluster and an Eiffel root class (application.e). The root class already contains a root class feature *make* which is the entry point for this application. Add a `io.putstring("Hello World")` in the 'TODO' section to create a runnable 'Hello World' application and save this file. The compilation process starts automatically on file changes but can be triggered manually as well.

To run the created program, a run configuration is needed and can be created manually the following way:

- Select menu 'Run → Run Configurations...'. A dialog appears.
- Double click on the 'Eiffel Application' icon. A new run configuration will be created.
- Go to the 'Launch' tab and press the 'Browse...' button. A selection dialog appears
- Select the project created before and press 'OK' button.
- Press 'Run' to save and run the created configuration or 'Close' to save the configuration without running the application.

As a shortcut, a default run configuration named after the project can be created automatically, by pressing the 'run'  icon instead of creating one manually.

Console applications write its output to a console window. If it is not already visible, one can open it by selecting Window → Show View → Console.

10.3 Feature List

- Project, Class & Cluster Wizard
- ISE Eiffel support (Full Build, Incremental Build, Finalize, Freeze, Error marking)
- Console Viewer
- Problem Viewer
- Cluster Explorer
- Editor
 - Syntax Highlighting
 - Content assist for Eiffel code constructs (branch, loop, assertions...)
 - Multi-comment/uncomment
 - "TODO" marker support
- Easy installation from update site

10.4 EEDT CVS Repository

The source code for this thesis is hosted at SourceForge. Use the following CVS data:

Host: eedt.cvs.sourceforge.net
Port: Default
Repository Path: /cvsroot/eedt
Branch: ISE_COMPILER
Tag: EDT_ISE_Eiffel_V0_2_0