EGOE: a Generating System of Multi-View Editing Environments on Eclipse Platform

Cheng-Chia Chen
Department of Computer Science, National Chengchi University
chencc@cs.nccu.edu.tw

ABSTRACT

This paper describes the design and implementation of EGOE, a generating system of multi-view editing environments on Eclipse platform. EGOE was designed to generate a language-specific editor (LSE) on Eclipse for any domain specific language. Given an editing-related specification of a target language, EGOE applies a model-driven process to generate an editing environment for the target language. The generated editor works on Eclipse and has synchronized multiple views, each possessing either structured or textual editing capability. The structured editing capability is mostly generated by EMF[4], which our system cooperates with, while the textual editing capability is generated by EGOE to work on top of the complicated editor framework of Eclipse. In addition to required basic functionalities, the editors generated by EGOE have also the features of syntax highlight, content assist and preferences setting etc. for the target language.

1: Introduction

An editor with special editing aids for a language can increase efficiency and reduce errors while editing sources of the target language. The editing aids found on a modern editor or integrated development environment (IDE) are very versatile as can be seen in Eclipse JDT[3] and Microsoft Visual Studio[17] etc. In addition to traditional basic commands like insert, delete, select, undo, copy, and paste etc., a modern high quality editor would give the user instant help in various ways. First, it can usually do instant compilation so that any syntax error while editing can be detected and informed instantly without the need of waiting until the whole text content is given. Next, it is usually language aware and hence can provide instant aids like code assistant/completion/template to remind the user and helps him to select the right token or construct skeleton to continue the editing and avoids possible errors in advance. Thirdly, a modern editor has the knowledge of the user's working codebase so that it can make use of the codebase and associated documentation to help the user to remind instantly the usage of a method or class via various views and handover aids. Moreover, it can help the user to select the right method or class to use from the bulk codebase which, like the JDK, is so tremendous that an average user can hardly be adept at all its content. Furthermore, a modern editor should also have the features of syntax highlight, preference setting, wizard, refactoring, formatter, multiple views for editing, code folding, and content outline etc.

It is therefore very desirable to have an editor with all above features designed for a target language. However, although of great use, this kind of editor with advanced editing aids is difficult to construct. Most such editors were developed only for the most popular dominant languages such as C/C++, Java and XML etc. On the other hand, for a domain specific language (DSL) which may have only a small user community and perhaps is just for temporary use and short-lived, there is usually no language-specific editor designed for it and its users can only resort to traditional general purpose text editors. The reason for the lack of a language specific editor (LSE) for a DSL is mostly economic consideration: due to the very high cost of developing a LSE, it is not worthwhile to invest a LSE on a less frequently used language.

As the release of Eclipse[2][5][15], however, the situation is changing because it give the developer a chance to construct an editor rapidly at a very low cost. Eclipse is an open source and loyalty free IDE firstly published in late 2001 but quickly emerged as a most used IDE. In fact it is also a tool integration platform. The most distinguished feature of Eclipse is its extensible plug-in architecture. Every tool can be developed as a plug-in and integrated seamlessly into Eclipse platform as if it were part of the platform. All plug-ins in the platform are interoperatable and can make use of the platform resource such as the workbench widgets and the workspace resources. As a result, every tool deployed as a plug-in in Eclipse can be operated in the same way by using the same style of user interface. Eclipse Modeling Framework (EMF)[4][6] is such kind of modeling tool developed on Eclipse. It can generate java application program interface (API) and structure editor for an arbitrary given model definition conforming to the Ecore metamodel[6]. The text editor framework[14] provided in Eclipse platform can help developers to create a text editor in Eclipse. Using the framework, developers can develop a customize text editor with additional editing aid functionalities. It is relatively easy for a developer to construct a high quality LSE in Eclipse than from scratch. Even so, however, Eclipse and, especially, its

text editor framework, are still very complex[20], and people need a long and steep learning course before they can get adept at it.

This motivates our interest in providing alternatively the EGOE generating system of editing environments on Eclipse platform, which, when given an editing-related specification of a target language, can apply the model-driven approach as suggested by OMG's MDA[10,11] to generate an editing environment for the target language.

The rest of the paper is organized as follows. In Section 2, we first introduce some backgrounds required for our work and then some related works. Then in Section 3, we describe the overall structure of EGOE and its components, and in Section 4, we introduce the details of the code generation process that EGOE adopts. Finally, in the last section, we give a simple conclusion and discuss some future directions we would continue to explore.

2: Background and related works

2.1: Eclipse Platform Text and EMF

Eclipse text editor framework[14], formally named Platform Text, is a component of the Platform subproject of Eclipse project at eclipse.org. It provides the basic blocks for text and text editors and supports a text editor framework with an abstract and a default text editor. Its text infrastructure provides facilities for text manipulation, position management, and change notification. Meanwhile, its JFace text part provides UI components for editing and presenting text. This part also offers support for rule based styling, content completion, formatting, model reconciling, hover help, and vertical rulers. Finally, it supports text file buffers which allow for shared access to the content of a text file.

The framework uses traditional MVC architecture to build its text editor: the model is a document capable of processing the inner text data in the editor; the view is a SWT StyledText showing the text in a fix manner; the control is played by an Editor class controlling how the text is to be showed. The developer can configure various editing aids by setting a configuration object accessible form the controller.

With the framework available we can develop text editors of our own for different target languages; various editing aids can be attached to them by implementing corresponding processors required by the configuration object of the editors. Because the framework and the interfaces that need to be implemented are fixed, we can form a specification by collecting all related data and use a code generator to generate the implementation code. This approach can help reduce most development time of an editor.

Eclipse Modeling Framework (EMF)[4,6] is a modeling framework and code generation facility for building tools and applications based on structured data model in Eclipse platform. It has three main components. The core EMF framework includes a metamodel (Ecore) for describing models, runtime support with default XMI serialization and reflective API for manipulating EMF objects. The EMF.Edit part provides support to help the user implement a user interface for any model based on EMF. The supported features include generation of tree viewers, property sheet views, action and model modification via command framework etc. It acts as a bridge between the Eclipse UI and the EMF core framework. Finally, the third component is the code generator part which, after importing or finishing editing an Ecore model, is able to generate a Java implementation code for it. The generated code is optionally packaged to at most four plug-ins: two of them are the API implementation of the model as well as testing code; in contrast, the other two are model editor implementation, one being Eclipse UI independent and the other Eclipse UI dependant.

2.2: Model-driven architecture

Model Driven Architecture (MDA)[10,11] is a software development architecture defined by OMG[13]. It aims to raise the level of software abstraction to model and promote the reuse of software design by model and model transformation. The main feature of MDA is to use higher layer platform independent model (PIM) instead of source code as the central of software development and use model transformation and code generation to generate lower layer platform specific model (PSM) and implementation code on different platforms from a given PIM.

Using MDA, developers can use the same PIM to generate multiple PSM's and corresponding implementations, each targeting at a different platform. MDA thus enables the reuse of system analysis result. On the other hand, a model transformation module can be designed to transform any PIM compliant with a domain specific metamodel into a PSM compliant with a platform specific metamodel. This module can be used to transforming all PIM's compliant with a domain metamodel to corresponding PSM's for a platform. It thus enables the reuse of system design for a specific domain on a target platform.

2.3: Editing environment generators

A language environment is valuable for it can give users immediate and extensive help on activities of application development like editing, testing, analysis and translation etc. Even so, however, only a small portion of existing languages have their own environments since it is usually thought hard and requires a lot of efforts to develop a functional language environment from scratch. It would thus be great if there is a system that can generate a language environment whenever we feed it with a declarative specification of a target language containing possibly also the user's requirements on the language. In fact, such kind of environment generator has appeared for decades. What

make new ones continue to appear are, on one hand, stronger UI supports provided by today's UI technologies and, on the other hand, the user's increasing demand on instant help arising from today's knowledge intensive editing and programming. The rest of this section introduces three representative generating environments found in the field. They are ASF+SDF Meta-Environment [1,12], SmartTools [7,8,9,16], and EMF, respectively.

ASF+SDF Meta-Environment is a language environment generating system working on Unix platform and was developed by CWI. It accepts as input the syntax and semantic definition of a target language. The input is expressed in ASF+SDF formalism and can be edited interactively in the Meta-Environment. With the specification of syntax, pretty print, type checking and program execution of a target language etc. given, the system can generate a C implementation of an interactive language environment containing optionally text editor, pretty printer, debugger, language analyzer and translator etc.

SmartTools from INRIA is a system written in Java. Its input includes the abstract syntax and the concrete syntax definition of a target language together with the configuration specification of the intended environment. SmartTools can then generate a structured editor for the target language according to the abstract syntax. The user can use the structured editor to edit the structured content of its input. After structured editing, SmartTools will serialize the abstract structure into textual content according to the concrete syntax and show the result in an immutable textual viewer. From the concrete syntax definition, SmartTools generates also a parser, which is used to generate the corresponding structured content while loading textual contents from a file. Besides a structured editor and a textual viewer, SmartTools also generates an XML-based tree viewer for browsing and editing input source.

EMF, as introduced in previous subsection, is a modeling framework developed at eclipse.org. It can generate a Java implementation for a model it supports. The generated implementation includes the API for model object with XML/XMI serialization ability and a structure editor on Eclipse for editing model instances. So we can get a structured editor for a language for free if we could produce an Ecore model for the abstract structure of the target language; meanwhile, the persistency implementation enable us to save and load documents in XML/XMI format very easily.

Although the environments generated by these systems perform well in various aspects, there are still defects on editing support. ASF+SDF Meta-Environment generates an environment from the grammar of a target language. The grammar is expressed in the format of ASD+SDF having no provision for specification of abstract syntax. As a result, the generated environment has no structured editor (unless the user develops his own by defining one using ASF). On the other hand, environments generated by SmartTools or EMF can edit source code only in a structured manner, since they both lack a textual editor.

3: The EGOE generating system

This section describes the structure of EGOE. As showed in Figure 1, the system can be understood from two different phases of time: the generation time and the run time. In the generation time the system is viewed from the perspective of an editor developer, who uses EGOE to generate an editing environment, while in the run time the system is viewed from that of an editor user, who uses the generated editing environment to edit his source.

There are three program modules used in the generation time. EGOE Generator is our main implementation; ANTLR[18] and part of Extended EMF are external. Extended EMF is the original EMF with an extended code generator; ANTLR is an open source LL(k) parser generator.

During the generation time EGOE takes the input specification of a target language and uses EGOE Generator, Extended EMF and ANTLR to generate an implementation of the intended editor. The implementation can then be installed and run in Eclipse as plug-ins. The result is a functional editor with both structured and textual editing capacities. It provides also various editing aids while in text editing mode.

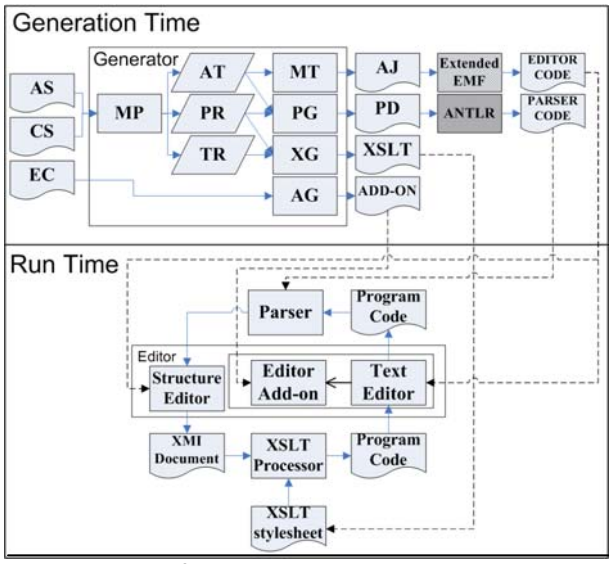

Figure 1. EGOE system

The input to EGOE at the generation time includes three parts: Abstract Syntax (AS), Concrete Syntax (CS) and Editor Configuration (EC), all expressed in XML format. AS contains the structural definition of the target language. It is basically the specification of a recursive type system, consisting of a type hierarchy, all language constructs pertaining to each type, and the signature of every construct describing how it is composed of other types of constructs. CS declares the concrete syntax as well as the lexical structure of the target language. Aside from the lexical definition, it

contains rules describing how each language construct is serialized into a concrete token sequence. CS is used to generate textual presentation from structured language constructs and is also used to generate structural representation from textual source.

The rational for decomposing language syntax into abstract and concrete syntaxes is to enable the sharing of the abstract syntax in both the textual and structural definitions of the target language. While both are needed to form a complete definition of a textual language, the concrete syntax is not used in the structural definition of a language.

The Editor Configuration (EC) contains information related to syntax highlight of the generated editor. We allow the developer to group lexical tokens into possibly overlapped categories. We also allow the developer to define preferred settings of text attributes ahead of time. A setting of text attributes contains information about the foreground color, background color and font style of textual sources it is applied to. Finally, the default text attributes of all language tokens at runtime are determined by a set of provided binding rules which associate every token category to at most one setting. Various defaults are assumed so the rules the developer need to specify are lowered to minimum. EC is consumed by the add-on generator (AG) to provide an implementation of an Eclipse preference page, which, during runtime, can be integrated into Eclipse to allow the setting of the user's preference on the text attributes of different tokens.

All parts of the input will be processed by EGOE Generator to produce four outputs: annotated Java code (AJ), parser definition (PD), XSLT stylesheet (XSLT) and editor add-on (ADD-ON). AJ is used further by Extended EMF as a model definition to generate the main editor code (EDITOR CODE in Figure 1) while PD is used by ANTLR to generate a text-to-structure parser code (PARSER CODE). We can then form a complete implementation of the intended editor by combining the four modules: EDITORE CODE, PARSER CODE, XSLT and ADD-ON. EDITOR CODE together with EMF runtime provides the implementation of an editor with both structural and textual editing capabilities. PARSER CODE and XSLT are used to translate the edited contents between structured and textual formats. Lastly, ADD-ON is the program module implementing the editing aids for syntax highlight, content assist and preference setting.

The EGOE Generator generates its outputs in a model-driven way. There are five function modules and three PIMs used in EGOE Generator to generate outputs. The five function modules are model processor (MP), model translator (MT), parser generator (PG), XSLT generator (XG) and add-on generator (AG); the three PIMs are AbstractDocumentType (AT), ParserRule (PR) and TransfoRule (TR). EGOE Generator translates these PIMs to some internal PSMs and then generates output code. We will discuss these modules in more detail in next section.

The extended EMF is the original EMF with some of its code generator part extended. We extend this part to make it possible to generate an editor with both structured and textual editing capability in place of its original capability of generating merely structured editor. We will explain how the EMF is extended in next section.

The generated code in generation time is packaged as Eclipse plug-ins. After installing and activating these plug-ins in Eclipse, we will get a working LSE for the target language. During the run time, we can choose to use the structured editor to edit the structured content and can also choose to use the text editor to edit the textual content of the source. While editing textual contents, we can use the provided editing aids such as syntax highlight, content assist and preference setting provided by ADD-ON. Although both editors each function well, their edited content still need to be synchronized. As a result, the generated Parser and XSLT stylesheet were used to perform the task of synchronization. We save the structured content edited in the structured editor to an XMI document by the XMI serialization functionality provided by EMF, and then use an XSLT processor instructed by the generated XSLT stylesheet to translate the structured content into source text (Program Code in Figure 1), which is then placed in the content of the textual editor. On the other hand, in order to get the structural contents from a textual source, we use the generated Parser to parse the textual content and at the same time reconstruct the structured content in the structured editor during the parsing process. In this way we synchronize the contents of both editors.

4: Code generation

As showed in Figure 1, EGOE Generator, Extended EMF and ANTLR generate all implementation code of the intended editor. EGOE Generator requires three input specifications AS, CS, and EC to generate AJ, PD, XSLT and ADD-ON. Then extended EMF takes AJ as input to generate EDITOR CODE and ANTLR takes PD to generate PARSER CODE. We developed the Generator and extended the EMF code generator to generate most code we need and use the ANTLR parser generator to generate parser implementation.

As stated in previous section, EGOE Generator has five function modules: MP, MT, PG, XG and AG. The MP translates the inputted AS and CS to three PIMs: AT, PR and TR. The AT model contains the abstract structure of the target language. The PR model contains rules about how to parse textual source and translate it into abstract language constructs. TR is the model containing rules about how to translate abstract language constructs to textual source. After the three PIMs are created, MT, PG and XG will use them to generate output code. MT will get AT and use the contained abstract language constructs it defines to generate annotated Java codes (AJ), which are just Java interface declarations annotated with special command

tag defined by EMF. AJ will later be used by Extended EMF to generate the main part of the implementation. The PG module will read the abstract language constructs contained in AT and the parsing rules contained in PR to create an intermediate model and then use it to generate a parser definition (PD) acceptable by the ANTLR parser generator. The XG module will first combine the tokens recorded in parsing rules of PR and the translation sequence contained in translation rules of TR to create an inner model representing all derived XSLT-style translation rules. XG then use it to produce a desired XSLT stylesheet. The AG module is responsible for generating add-on code from EC. It uses the StringTemplate [19] template engine to generate the ADD-ON code. AG first parses the definition contained in EC to get some model objects. It then triggers the template engine to generate the desired code by applying these model objects to a template group. The template group is part of EGOE and is prepared by us in advance.

In addition to contributing the EGOE Generator, we also extend the code generator of EMF to make it capable of generating our text editor code and the required plug-in manifest files. EMF puts its code generator in a specific plug-in. This plug-in uses the accompanying JET template engine to execute some code templates located in prescribed directories to generate its output. Hence it suffices to find the needed code templates which generate the editor and plug-in manifest, and then extend them by inserting additional template code which our generated editor would require.

After the generation time, four plug-ins for the target language editor would be generated. EDITOR CODE, generated by Extended EMF, is packaged as three plug-ins as it would be in original EMF. The other three generated artifacts: PARSER CODE, XSLT and ADD-ON are packaged as an additional plug-in. After installing these plug-ins, we can then see a working multi-view editor for the target language in Eclipse whenever a file with an extension name corresponding to the target language is open.

The generated editor together with associated editing aids is demonstrated in Figure 2 through Figure 5. While in the structured editor (Figure 2), the user can edit his input either from within the structured tree view or from within a property sheet in an accompanying property view (hidden in Figure 2). Besides structured editor, there is also a text editor which the user can use to edit his input in a textual from. The text editor is showed in Figure 3, where all textual contents are highlighted with various colors. As is showed in Figure 4, the textual attribute of selected kind of lexical tokens can be changed in a preference page of Eclipse's integrated preference dialog. The user can change the foreground as well as the background color of every selected kind of lexical token in a color chooser dialog and can also change the font style of the overall textual content by checking or unchecking two checked boxes. While in the text editor, the user can trigger content assist by typing the Ctrl+Space keyboard shortcut. Immediately popped up is a context menu containing a list of possible tokens, each syntactically valid to appear at the position where the caret is located. After the intended token is selected by the user and the Enter key is typed, the selected token would automatically be inserted into the desired location. The context menu for content assist is showed in Figure 5.

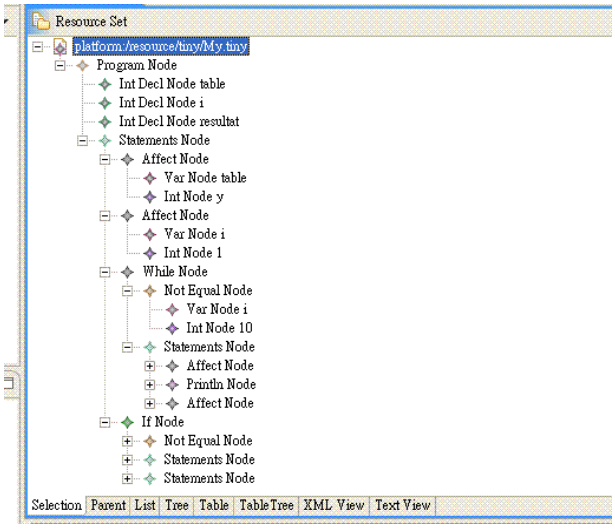

Figure 2. Structured editor

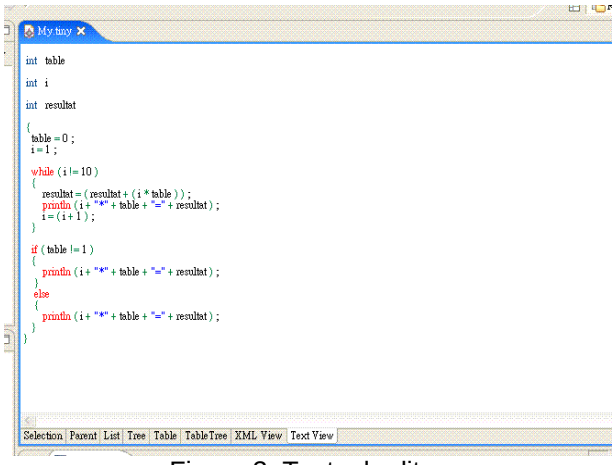

Figure 3. Textual editor

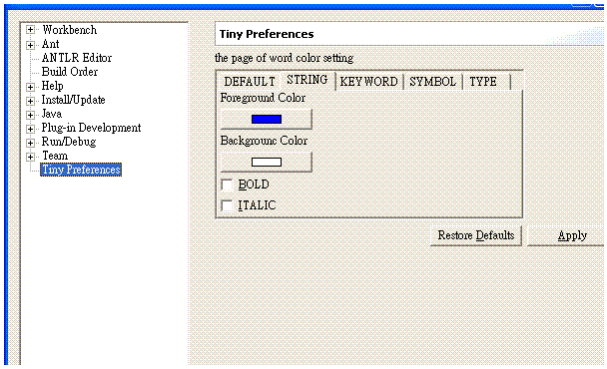

Figure 4. Preference page

5: Conclusion and future works

We have developed a system called EGOE which, when given an editing-related specification of a target language as its input, will be able generate a language-specific editing environment for the language. The input specification includes the definition of abstract syntax, concrete syntax and intended editor configuration of the target language. The generated editor is encapsulated as Eclipse plug-ins and must be installed in Eclipse to work with EMF. The main feature of the generated editor is its synchronized multi-view editing capability. Each view can provide either structured or textual editing capability. The structured editing capability is mostly generated by EMF while the textual editing capability is generated by EGOE to work on top of the editor framework of Eclipse, which, because of its complication, is usually thought to be hard to use and understand for an average developer. In addition to the required basic functionality, EGOE supports textual editing features such as syntax highlight, content assist and preferences setting for the target language.

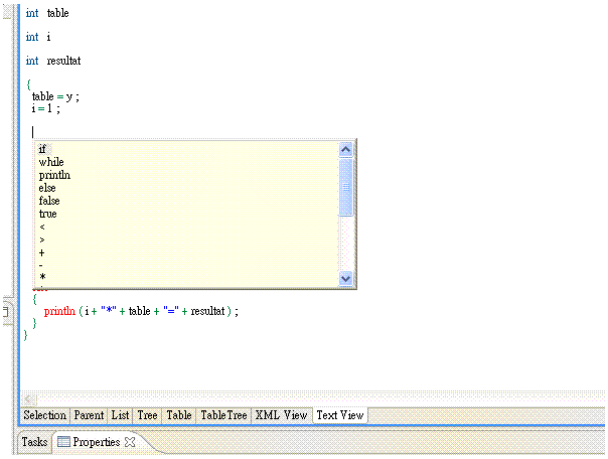

Figure 5. Content assist

While our EGOE system works well at present, the editor it generates is still very primitive lacking most features we may find in a modern high quality editor. In fact, EOGE is just in the starting stage of its evolution and our final goal is to make it capable of generating a high quality editor possessing all features of a modern editor. Therefore, we need to extend the system in many ways. First, inspired by ASF+SDF, we want EGOE to accept additional semantic description of a target language so that it can generate also interpreter, compiler, debugger and related language tools to help instant compilation, code testing, and debugging etc. Secondly, because both instant look-up and detailed explanation are very helpful while editing a language source, EOGE needs to generate code and assemble documentation about the target language for the purpose of online help. Next, the generated editor should have the capabilities of code folding and code formatter; it should also have the features of other editing aids such as code completion, code template and content wizards etc. We believe that the implementation of many features mentioned above such as code folding, code completion, code template etc. should mostly be able to be generated automatically by analyzing the lexical and syntactic structure of the target language. This is indeed the main motivation for our continuing investigation of this topic. Last, but surely not the end of the to-do list, the generated editor should be able to do refactoring and again, this should be achievable by utilizing information obtained from a language specification and the user's runtime settings.

References

[1] Brand, M.G.J. van den, A. van Deursen, et.al. The ASF+SDF Meta-Environment: a Component Based Language Development Environment. Wilhelm, R., Ed. Compiler Construction 2001 (CC'01). 365--370. Springer-Verlag., 2001.

[2] Eclipse.org. From http://www.eclipse.org/

[3] Eclipse.org,Eclipse Java Development Tools (JDT) Subproject. From http://www.eclipse.org/jdt/

[4] Eclipse.org, Eclipse Modeling Framework. From http://www.eclipse.org/emf/

[5] Eclipse Platform Technical Overview. From http://www.eclipse.org/whitepapers/eclipse-overview.pdf

[6] Frank Budinsky, David Steinberg, Ed Merks, Raymond Ellersick and Timothy J. Grose. Eclipse Modeling Framework. Addison-Wesley Pub Co., 2003.

[7] Isabelle Attali, Carine Courbis, Pascal Degenne, et.al. SmartTools: a Generator of Interactive Environments Tools. International Conference on Compiler Construction CC'01, volume 2027 of Lect. Notes in Comp. Sci., Genova, Italy, April 2001.

[8] Isabelle Attali, Carine Courbis, Pascal Degenne, et.al. SmartTools: a development environment generator based on XML technologies. In XML Technologies and Software Engineering, Toronto, Canada, 2001.

[9] Isabelle Attali, Carine Courbis, Pascal Degenne, et.al. Aspect and XML-oriented Semantic Framework Generator SmartTools. In Second Workshop on Language Descriptions, Tools and Applications, LDTA'02. 2002.

[10] OMG. OMG Model Driven Architecture. From http://www.omg.org/mda/.

[11] OMG. MDA Guide. From http://www.omg.org /docs/omg/03-06-01.pdf.

[12] MetaEnvironment. From http://www.cwi.nl/htbin/sen1/twiki/bin/view/SEN1/MetaEnvironment

[13] Object Management Group. From http://www.omg.org/

[14] Eclipse.org. Eclipse Platform Text. From http://www.eclipse.org/eclipse/platform-text/

[15] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. The Java Developer's Guide to Eclipse. Addison-Wesley Pub Co., 2003.

[16] SmartTools Software Factories. From http://www-sop.inria.fr/smartool/

[17] Microsoft Inc. Visual Studio 2005. From http://msdn.microsoft.com/vstudio/

[18] Parr, Terence. ANTLR Parser generator. From http://antlr.org/

[19] Parr, Terence. StringTemplate Template Engine. From http://stringtemplate.org/

[20] M. Antkiewicz and K. Czarnecki. Framework-specific modeling languages with round-trip engineering. Proceedings of MoDELS, Genoa, Italy, October 2006.