T-76.115 Technical Specification
TEXlipse project
Group TeXlapse

ID: TEXLIPSE-TECH-1
Version: 1.4
Modified: February 7, 2005

Author:
Oskar Ojala (omojala@cc.hut.fi)

Table 1: Version history

Version	Date	Editor	Change
0.1	14.11.2004	Oskar	Basic structure
0.2	22.11.2004	Kimmo	File output and building
0.3	22.11.2004	Esa	Templates and preview
0.4	25.11.2004	Taavi	Viewing the outline, Basic outline navigation
0.5	25.11.2004	Oskar	Some architecture and technical descriptions added
0.6	25.11.2004	Esa	Added template syntax
0.7	26.11.2004	Esa	Modified template sections and appendix
0.8	28.11.2004	Oskar	Made corrections based on inspection, added some technical details
0.9	29.11.2004	Oskar	Added more technical detail in tasks and did some corrections
1.0	29.11.2004	Kimmo	Added some more explanations about the builder
1.1	4.1.2005	Kimmo	Updated the builder diagram and explanation of it
1.2	29.1.2005	Oskar	Added folding support and made some adjustments
1.3	1.2.2005	Kimmo	Added previewer explanation and diagram
1.4	7.2.2005	Oskar	Updated most of the document, made new architectural diagrams

Contents

1 Purpose and scope of the document

The purpose of this document is to define the technical specification and architecture of the TEXlipse system. This is intended to complement the TEXlipse requirements documentation. Thus, this document focuses primarily on specifying how features are to be implemented and why they are implemented in the specified way. Secondarily, this document focuses on defining feature behavior more specifically than done in the requirements document when that is necessary for implementing the requirement.

1.1 Prerequisites

The intended audience of this document is people interested in the architecture and implementation of TEXlipse and have some degree of programming background.

To fully comprehend the contents of this document, knowledge of the Eclipse plugin architecture, the TEX typesetting system and of compiler techniques is required. These topics are so broad that it's impossible to summarize them here, however compiler and TEX -resources are referred to when appropriate and Eclipse documentation can be found at the Eclipse www-site (`http://www.eclipse.org`.)

This document can be read with only knowledge of the requirements (see document TEXLIPSE-REQ-1) and Eclipse with the help of the domain concept descriptions, but in some places technical descriptions that require more in-depth knowledge are required and these should thus be skipped.

1.2 Document structure

The rest of this document is organized as follows; Section 2 introduces the key concept in the architecture and technical design of TEXlipse. Section 3 makes a fairly detailed architectural overview of the key concepts of TEXlipse and the software structure chosen. Section 5 expands on the architecture description and explains in more detail how the different parts are implemented and, most importantly, how they work together. Section 6 explains more detailed implementation-level issues and techniques used per implementation tasks (the tasks correspond fairly well to the functional requirements of TEXlipse.)

2 Main domain concepts

Main domain concepts:

AST Abstract Syntax Tree, a tree representation of the parsed stream. In contrast to CST, only selected tokens are represented and superfluous tokens (such as expression terminators and parentheses) are ignored in the tree.

BibTEX A bibliography citation inclusion system for LATEX, developed by Oren Patashnik. Uses a bibliography file and a style file to make a bibliography list to the LATEXdocument and to include only the cited bibliographies. See [Lam85] and [Pat03].

CST Concrete Syntax Tree, a tree representation of the parsed stream as recognized by the parser. Each token have their appropriate place in the tree dictated by the grammar.

DFA Deterministic Finite Automaton, an automaton that has deterministic state transitions, useful for representing regular expressions in computer-executable form, thus used for building lexers.

EBNF Extended Backus Naur Form, the common way of describing context-free grammars.

Eclipse IDE A free Integrated Development Environment sponsored by IBM. Intended originally for Java development, but currently emphasizes plugins for adding functionality beyond the original requirements.

Eclipse plugin A piece of Java software that integrates with the Eclipse plugin architecture and provides some additional feature for the Eclipse environment.

Eclipse plugin framework The Eclipse platform offers a rich framework for plugins, complete with interfaces and classes for implementing many common functions more easily.

Editor In Eclipse the editor view, or editor for short (as it's used throughout this document) is a view where the documents can be edited as in a normal text editor. The editor can be extended with many kinds of functionality, such as syntax highlighting.

GUI widget A component in the GUI (Graphical User Interface); can be a button, a window, a checkbox, a menu etc.

LALR Look-ahead LR, a LR parsing method that is more powerful than the SLR method, but easier than the LR-method without sacrificing too much in recognized languages. See LR.

LaTeX A popular typesetting language, based on TeX. Is written as a plain text file with a series of commands. See [Lam85].

Lexer A program for reading a stream and recognizing predefined tokens in the stream, then returning found tokens or an error if the stream doesn't correspond to the specified format.

LL Left to right, leftmost derivation parsing, an easy to understand top-down family of parsing methods. Refer to [ASU86] for details.

LR Left to right, rightmost derivation parsing, a family of bottom-up parsing methods. Refer to [ASU86] and see also [Knu65].

MVC Model-View-Controller, a design pattern where the date is held in a model, the data is presented through views and the mapping of data to views and vice versa is done by the controller.

Outline In Eclipse the outline view, or outline for short (as it's used throughout this document) is a view where the currently edited document's (the document that is currently shown in the editor) struture is shown. In the case of a Java class this would include eg. all the fields and methods, in a LaTeX-document it would eg. include the sections.

Parser A program for checking that tokens match a predefined grammar, ie. to check that the given stream is of the right form.

Parser generator A software for automatically generating a lexer and a parser from a given grammar specification.

Singleton A design pattern where the singleton class only has one existing object instance at any time, which is then shared among other runtime objects.

TeX A powerful typesetting system that permits the user to typeset documents in professional quality by using a flexible command language. See [Knu84] for a description of the language, [Knu86] for a description of how TeX works.

View In Eclipse, there are several views: the editor view, the outline view, the problems view etc. These are different views on the document or project being edited and appear visually as separate areas in the Eclipse GUI.

Visitor A design pattern where an object, which is the visitor, visits another object, thereby performing a number of operations on the visited object. The visitor implements a certain interface, so that it can be applied to the visited object. In TEXlipse visitors are used for trees, so that the visited object calls a method defined in the visitor interface when a node corresponding to the method is visited in the tree. See [Gag98] for a more thorough explanation.

3 System overview

TEXlipse is to be a plugin for the Eclipse IDE. It is to provide a LATEX-mode for editing of LATEX-documents.

Briefly, it provides code completion of references, syntax highlighting, user defined templates, automatic building, previewing, error reporting and an outline view. It does not re-implement LATEX, rather, it is intended to serve as a powerful editing tool for LATEX documents. It does not implement WYSIWYG editing of the document, as it is intended to be a poweruser-tool. Refer to the TeXlipse requirements document (document ID TEXLIPSE-REQ-1) for more information about the intended use and features of the system.

4 Architectural overview

4.1 About plugins

The Eclipse plugin architecture places many constraints on the structure of the plugin. Essentially, the Eclipse platform provides much infrastructure for building an editing environment, eg. the plugin developer does not need to program GUI widgets and basic editing functions such as copy and paste by himself. On the other hand, the Eclipse platform and the ready-made infrastructure places certain constraints on the architecture, eg. how documents are handled. In general, the wins provided by the (extensive) ready-made functionality far outweigh the disadvantages.

The plugin is not a standalone piece of software; it integrates tightly with Eclipse. Figure 1 depicts this and also shows three central components of TEXlipse: the editor, representing the editor view, the outline, representing the outline view and the builder, which handles interfacing to external programs (eg. LATEX) that are needed to build the document. The editor and outline directly represent the Eclipse views of the same names and thus

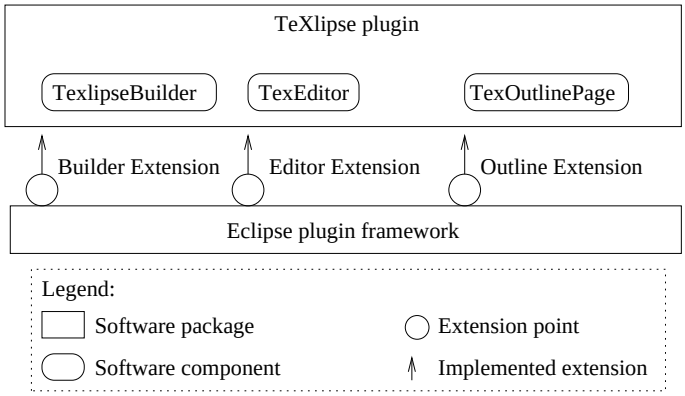

Figure 1: The plugin structure: TEXlipse extends Eclipse on certain extension points

build on the Eclipse plugin framework. The builder is the core component in a set of components handling interfacing to external programs that handle building and previewing LATEX-documents.

4.2 External interfaces

To see how the TEXlipse plugin fits in in the user's programming environment, see Figure 2, which presents the external interfaces of the plugin and the control flow. In order to work, the plugin requires (besides Eclipse) tools for actually compiling the created documents into vector representations, ie. postscript, dvi, and/or pdf. Thus, a LATEX-distribution is required to be installed separately, which TEXlipse then calls to parse the document. For implementation details, see Section 6.17.

For previewing the created document, an external previewer is called. The TEXlipse plugin permits the previewer to send messages back to the plugin, enabling bidirectional communication which makes synchronizing the Eclipse document view and the previewer view possible. For implementation details, see Section 6.20.

Due to the fact that TEXlipse is designed to run on three different operating systems, all having somewhat different facilities, preferred distributions of LATEX and different previewers, the external interfaces to programs must be able to handle all of these fairly invisibly to the user (the user is naturally required to set up the system, but setting up TEXlipse shouldn't differ much on different platforms.)

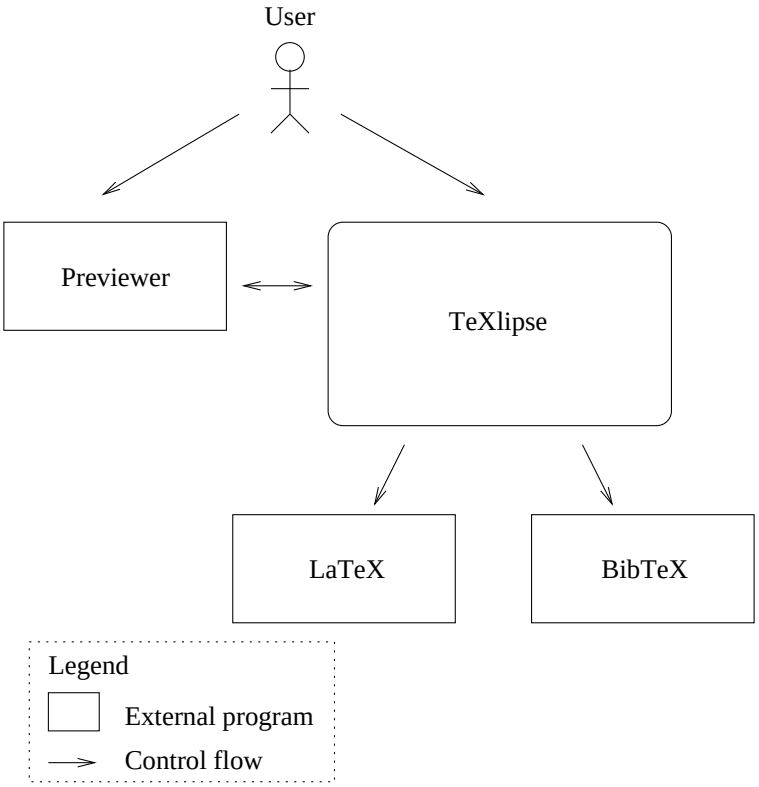

Figure 2: External interfaces with control flows depicted

Beside program interfaces such as calling LaTeX or a previewer, Figure 2 includes the user. The user mostly works with the editor, which provides the direct editing view of the document source. The user also works with the document outline, the file system browser (provided automatically by Eclipse) and the problems view in the Eclipse GUI. The user can activate the builder and the previewer. Finally, the user can specify templates and use templates, which speed up editing by inserting ready-made code to fill out.

4.3 Document model

The core concepts in TeXlipse are focused around the editor view and its functions. TeXlipse provides a LaTeX-editor and useful views on the document being edited, the central one being the document outline view (there is also the problems-view for build errors.) The outline view shows a document outline as described in requirement R2.1 (requirement document ID TEXLIPSE-REQ-1.) In order to implement some editor and outline functions, parsers for BibTeX and LaTeX are made (these are described in more detail later in this document.)

In order to facilitate the necessary communication between the outline, the editor and the document parser(s), the MVC (Model-View-Controller) pattern is applied in an adapted form. In this pattern, we have the model representing the data, the view representing a view on the data (typically a GUI) and the controller representing the logic for mapping different data to different views. This pattern is particularly useful in GUIs, since the order of user interaction cannot be known in advance, enabling the data to be edited from different views and it provides an order of abstraction between the GUI and the data model.

In an Eclipse plugin one doesn't need to implement the GUI from scratch — in fact, the GUI comes largely ready from the existing plugin infrastructure, so the "view" part is a quite thin. Also, the Eclipse plugin structure places some constraints on the document model and object hierarchy, so the MVC pattern is adapted to our needs. Figure 3 shows the coupling of the central editing views; Model keeps abstract representations of the document (autocomplete data and outline data), asking the parsers to return updated versions of the data structures when the data itself is updated. The editor essentially provides with information on editing updates as well as fetching new data structures, as does the outline.

It's worth to note that in Figure 3, `IDocument` is an Eclipse class, which contains the document being edited. The plugin architecture automatically

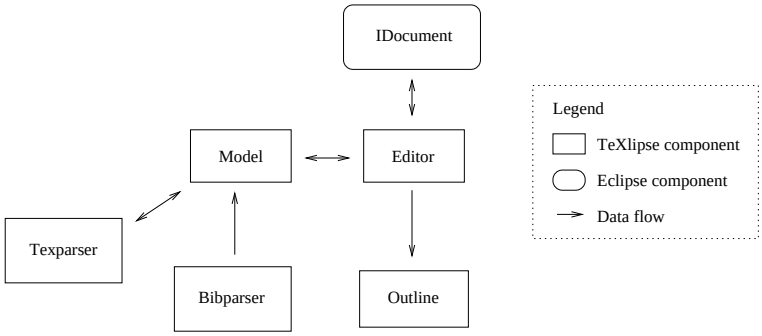

Figure 3: Editor-Model-Outline-Parser MVC-style coupling

provides for this, but `IDocument` is not alone sufficient in holding all the data required (eg. the outline structure), so we augment it with the model that contains somewhat more abstract representations of the document, in contrast to the concrete representation of `IDocument`. Thus, `IDocument` holds the model of the concrete file-based document, while our model holds the model for LaTeX-specific abstractions.

The reader might ask why use the MVC paradigm in such a way that the controller is distributed into several classes and there are essentially two models? First, the Eclipse plugin platform provides the basic way of operation for the editor and outline, as well as the `IDocument`, so the developer doesn't have too much leeway. Second, our model can be thought of as a controller, except that there are circumstances where it's more efficient and simple for the editor and the outline to go directly to `IDocument`. Third, this behavior is much better than a casual glance would suggest, since `IDocument`-class changes only when Eclipse changes and such a major change that would require a major rewrite of TeXlipse would require a major rewrite of a significant number of plugins, making the change unlikely. Fourth, the pattern described already provides a good degree of abstraction; the parsers may be changed at will, without having any effect on other components than model, since the data interfaces to it are standardized.

4.4 System architecture

Figure 4 presents the TeXlipse architecture, with external software/documents shown dashed. As can be expected, the editor is a central piece in the plugin. In Figure 4, the Eclipse plugin infrastructure is not shown for reasons of clarity. Thus, the builder appears not to be connected to anything else than the editor, even though it most certainly is — the Eclipse plugin architecture

handles calling it. This situation is depicted in Figure 1; the central parts of TEXlipse interface with the Eclipse plugin architecture, which provides the connecting framework.

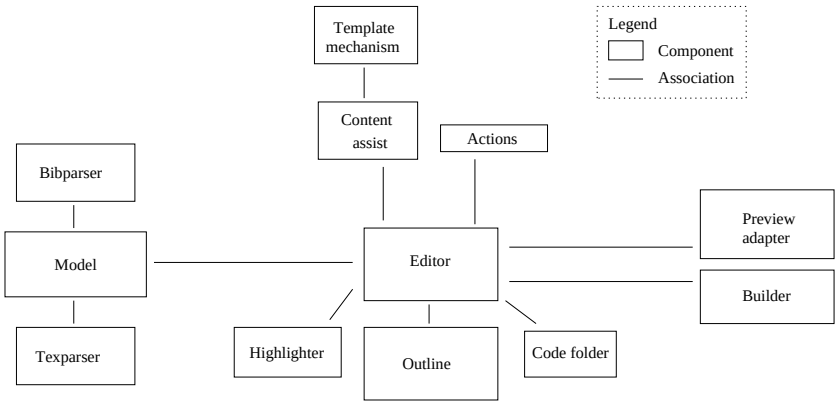

Figure 4: TEXlipse architecture shown as a component view

The architecture, as shown in Figure 4, introduces some new parts — the template mechanism, the actions, content assist, the highlighter and the code folder. The actions are the simplest — they simply contain editor actions for error messages and eventually menu options. The template mechanism is also closely associated with the editor and provides the mechanism for retrieving templates (both pre-made and user defined) as well as enabling the use of templates while editing. There are two kinds of templates: document templates and editing templates. The former can be applied to the entire document/project when starting a new project. The latter can be used via hotkeys and/or typed abbreviations during editing and insert a template into the document being edited. Due to this difference, both implement entirely separate mechanisms. The actual template completions, along with reference and command completions are handled by the content assistant –framework.

The code folder handles folding away parts of the LATEX-source from the editing view and the highlighter is a major component handling the syntax highlighting in the editor.

The external interfaces were already discussed and they consist of two major parts: the previewing facilities and the building facilities The preview adapter interfaces the document preview with the editor so that both views can be synchronized when a previewer that supports this functionality is used. The builder handles the building of the document and thus interfacing to the LATEXand BibTEX-programs installed. It calls them and they in turn produce the document in the desired format.

5 Technical overview

Based on the architecture described in Section 4 we have developed a technical design. The technical design encompasses the package and class structure of TEXlipse, as well as the interaction between the different components.

5.1 Packages

Table 2 summarizes the package structure of the plugin and briefly describes what each package does. Note that the base package is fi.hut.soberit.texlipse, which has been omitted from the table for brewity.

package	function
plugin	Plugin base functionality
actions	Editor actions (eg. code commenting)
bibeditor	BibTEX editor functionality
bibparse	BibTEX parser
builder	Builder functionality
editor	Editor and associated functionality
editor.scanner	Syntax highlighting and partitioning scanners and rules
model	Abstract document model
outline	Outline view
properties	Project property pages
templates	Template functionality
texparser	LATEX parser
viewer	Previewer functionality
wizards	Wizards (eg. project creation)

Table 2: Package structure; the base package is fi.hut.soberit.texlipse

It must be noted that Table 2 omits automatically generated parser packages (lexer, parser, node and analysis) under both parser packages — most of the automatically generated code is not meant to be human-readable and is abstracted neatly through the classes in the base parser packages.

5.2 Document model

The architecture behind the TEXlipse document model was described in Section 4.3. Here we proceed to define how we process the document and what classes are involved in some of the basic document-handling functions.

5.2.1 Parsing

(this section will be updated in the 4th iteration)

Figure 5 depicts the key classes in parsing the LaTeX-document being edited and constructing an outline from it. Many classes are omitted for clarity; the automatically generated classes alone constitute tens of classes and Figure 5 contains all the key classes anyway. The central class is `TexParser`, which contains the lexer and parser objects and provides an interface for retrieving abstract structures of the document (eg. all the labels, the outline.) Thus, `TexParser` is the class that is used by other packages in the system.

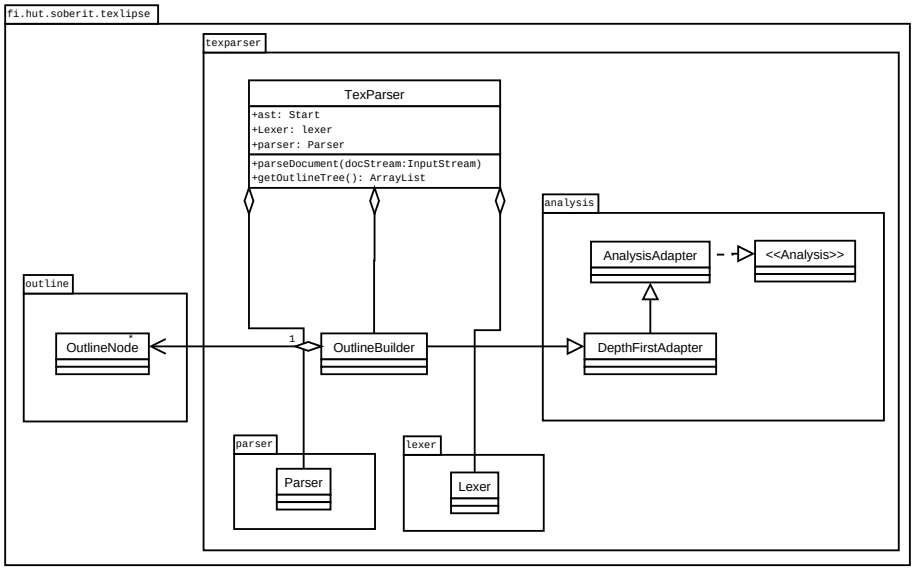

Figure 5: LaTeXparser and a depiction of the use of visitors *(To be updated)*

The inner workings of the parser-package can be explained by looking at the specific case of building an outline tree. `TexParser` in Figure 5 receives a request from the model to parse the document and receives a document stream to parse. It invokes its lexer and parser on the stream, building an AST in the process. The AST can now be transformed using the visitor pattern — applying a visitor object on the AST so that the AST calls the appropriate visitor methods of the visitor object when the nodes corresponding to the methods are visited. The visitor construction is shown in Figure 5, as is the `OutlineBuilder` -visitor and its inheritance hierarchy (the visitor methods are quite numerous and not depicted.) When the model now needs to update the outline, it asks its `TexParser` for the outline, which leads to `TexParser` invoking the `OutlineBuilder` -visitor that constructs the outline, storing the result in `OutlineNode`s forming a tree. The resulting tree is returned to

the model and can be directly used in the outline.

This visitor pattern model is employed successfully in parsing BibTeX documents, but for LaTeX documents we use a more traditional one-pass parsing approach, mainly due to the lack of benefits in the visitor approach (BibTeX has a stricter structure.) The issue is addressed more specifically in Section 6.1.

It's worth noting that the `analysis`, `lexer` and `parser` -packages are generated by SableCC and are SableCC-specific; SableCC automatically constructs a visitor interface and a visitor skeleton implementing that interfaces based on the AST structure specified in the grammar. The choice of using SableCC, its advantages and disadvantages are discussed in more detail in Section 6.1.

The BibTeX-parser is practically identical conceptually — it merely provides different data structures and methods outward and internally it implements a different parser. Hence, it forms a separate package.

The use of visitors and an AST enables easy programming and a relatively clean abstraction of functionality — our experience thus far has been that the visitors are fairly easy to program and the automatically generated grammars provide a lot of convenient abstraction, eg. changing the grammar doesn't most of the time imply refactoring everything. Abstracting the parsers serves to decrease module coupling and to easily distribute the implementation tasks. Also, it makes the system easier to understand. However, note the specific requirements of LaTeX, discussed in Section 6.1.

The Eclipse plugin framework provides for document scanners implementing a relatively easy way to do basic lexing of the document (see section 6.2 for a use of this.) However, while easy to use, these scanners are extremely tedious for more complicated grammars and they don't offer the performance and syntactical checking advantages of a dedicated parser. One problem with simpler parsing would be that the user writes a subsection without a preceding section — it might be valid, but how is the outline supposed to show it? Errors such as this are easy to catch with a dedicated parser. We can also check the validity of labels and make similar things not possible with simple lexing applications or one-pass compiling.

5.2.2 Outline

The conceptual process of parsing the LaTeX-document in order to create an outline tree was detailed in the previous section. Figure 6 now shows how the outline view is associated with the rest of the system.

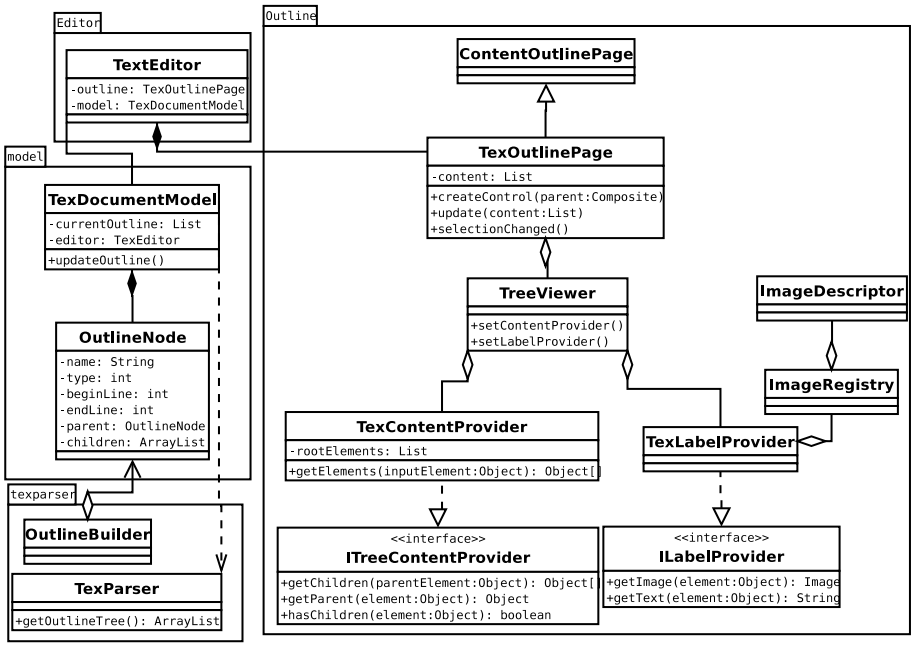

Figure 6: Outline

The way the outline works is described in more detail in Section 6.13. What is important to note here is how the `TexDocumentModel` handles calling the parser and holds the tree of `OutlineNodes` representing the outline. The task of the outline-package, in turn, is fetching the outline from the model and taking care of all tasks in displaying it (this includes displaying the actual tree as well as doing such things as choosing the correct icons for each type of node in the outline tree to display.)

5.3 External interfaces

External interfaces used by the TEXlipse plugin include builder and viewer. The builder is the module that invokes the external LaTeX program (or the likes) and creates a previewable document.

5.3.1 The Builder

Figure 7 shows the class structure of the builder package and builder's connection to the Eclipse API.

The builder starts when the user selects `Project` → `Build Project` from

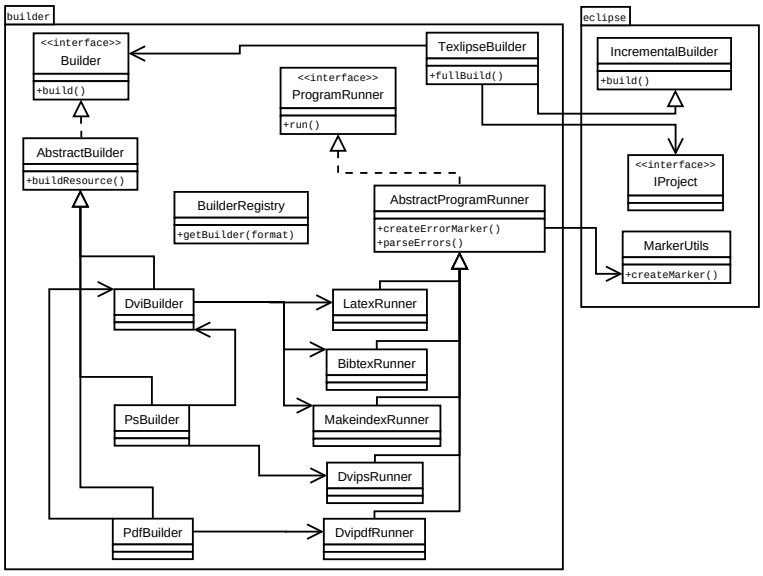

Figure 7: Builder

Eclipse's menu. Eclipse then instantiates the class `TexlipseBuilder`, because it's defined in the plugin's descriptor file. `TexlipseBuilder` does some run-time checks and then consults `BuilderRegistry` for an instantiation of the actual builder class (one of the realizations of `AbstractBuilder`). Each builder class is capable of building the input LaTeX-file to one output format. To do this, a builder uses one or more program runner classes.

A program runner is an abstract representation of an external program. These classes are implemented as realizations of the class `AbstractProgramRunner`. Program runner classes contain methods for running the program, stopping the program and parsing errors from the output of the program. To display errors, the program runners utilize `MarkerUtils` class from the Eclipse API.

The paths of the external programs are defined in the TeXlipse preferences page, as well as the default output format. The output format can be overriden per project - the same output format setting can be found on the project properties page. Not all supported external programs need to be found from the operating system. The user needs to configure only those that are required for the chosen output format.

At the center of this all is the `BuilderRegistry`, which holds all the actual instances of the builder and program runner classes. The `BuilderRegistry` class itself is implemented using the Singleton desing pattern. This

way, all the builder classes can utilize it, and it can still hold an internal global state. The `BuilderRegistry` class provides a method for looking up a builder classes for the given output format, and methods to configure program runners. The `TexlipseBuilder` class uses the registry at the start of a build process to gain a reference to the correct builder class. The builder classes, in turn, use the registry to gain a reference to the correct program runner.

5.3.2 The Previewer

Figure 8 shows the class structure of the viewer package and viewer's connection to the Eclipse API.

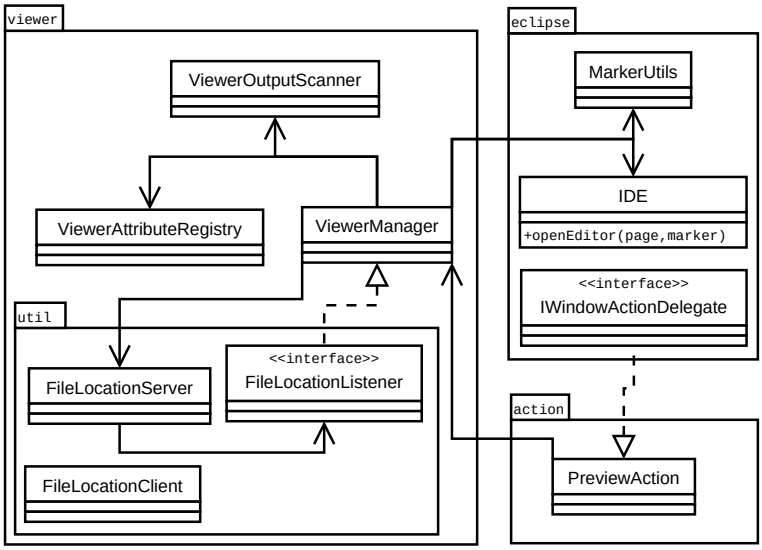

Figure 8: Viewer

The viewer can be started by choosing `Preview Document` from the Eclipse menu. This causes Eclipse to instantiate the `PreviewAction` class and call its `run` method, which calls `ViewerManager` to run the configured external viewer program. The `ViewerManager` gets the viewer program configuration from `ViewerAttributeRegistry` class, which in turn gets it from the plugin preferences. The `ViewerManager` also reads some configuration from the current project, like the file name to view. `ViewerManager` creates a running process of the external viewer program and, depending on the configuration, instantiates either a `ViewerOutputScanner` or a `FileLocationServer` or neither of them.

The `ViewerOutputScanner` runs in its own thread and reads the output of an external program as long as the program is running. The `ViewerOutputScanner` scans the output for "filename:linenumber" -strings, which tell that the user wants to navigate to the specified location in the source file. The `ViewerOutputScanner` then creates an `IMarker` object to that location, using `MarkerUtils` as helper, and then calls the Eclipse's `IDE` class to open the specified file at the given marker. This method is supported in Unix systems using the external `xdvi` program.

The `FileLocationServer` runs its own thread listening to a certain socket. The input for `FileLocationServer` is similar to that of `ViewerOutputScanner`, "filename:linenumber" -strings. This method is used in Microsoft Windows systems, where the `yap` dvi viewer is used to preview the documents. Yap can be configured to invoke an external program, when the user wants to navigate from a dvi file to its source TEX file. The TEXlipse plugin provides a client program to invoke, namely the `FileLocationClient`. The `FileLocationClient` outputs a filename and a line number, given as its command line arguments, to the socket that the `FileLocationServer` listens to. When the `FileLocationServer` receives a valid "filename:linenumber" -string, it calls the `FileLocationListener` to navigate to that location. This call propagates to the same method in the `ViewerOutputScanner` as described above.

5.4 Editor functions

The editor is a central part in TEXlipse and many of the user requirements are related to it. Many of these do not affect other packages or functions, but some use the facilities in TEXlipse already presented in this section.

Document and source code editing are key functions in Eclipse and thus the Eclipse plugin architecture offers rich functions for supporting many desirable editor functions. An example of a feature implemented within the editor framework is syntax highlighting. Syntax highlighting is achieved by using existing Eclipse document scanners by giving them rules to match and using the syntax highlighting framework. Essentially this is making a lexer which recognizes certain tokens. These document scanners can be used for other editor functions too, such as code folding. However, the expressive power of the scanners is limited, so we perform code folding using our own LATEX-parser.

Not all functions can be completely made using the classes and interfaces of the Eclipse framework. One such function is code completion. The mechanics of code completion is done using the Eclipse framework, but fetching and storing the actual completions must be done by hand — in this case using our

`TexParser` and `BibParser` -parser classes, which can parse the documents and construct the completion information.

5.5 Code reuse

Since TEXlipse is a plugin, it's already based on a large degree of reuse, as can be noted from the previous sections. Basic menus and widgets, syntax highlighting, code completion drawing etc. is eased considerably by ready-made components. However, this reuse focus on common editing tasks and it would be desirable to reuse LATEX-specific functionality, too.

The possibility of reusing large amounts (or even some amount) of code is highly desirable, since it would shorten development and testing times. Indeed, there exists and Eclipse plugin for LATEX, namely eTex. However, after studying it, we have found the documentation to be practically nil and the code to be buggy and of dubious technical quality. Thus, it was not chosen as a basis for implementation. Other LATEX-editors for Eclipse suffered from being very limited in scope - TEXlipsehas considerably more features planned for implementation, several of them being fairly complex.

There are several practical tools for solving parts of TEXlipse's problem domain, one of them being JabRef, which is a program for managing references, mainly BibTEX-databases. However, JabRef uses a hand-coded parser, which is a potential software engineering and performance problem, the internal data structures are so different than ours that refactoring would be significant and on top of it all its license (GPL) doesn't comply well with an Eclipse plugin. Due to these reasons, no code from JabRef is be used.

For aiding the construction of some LATEX-code, some good sources exist. For BibTEX, prof. Nelson Beebe's articles (see [Bee93]) are highly useful and there are many good books about TEX and LATEX, which make designing significantly easier. So while we don't have the opportunity to reuse code, we have many ideas to reuse.

6 Technical specification per implementation task

6.1 Make LATEX parser (T0.1)

`Package: texparser.*`

Define a parser (in EBNF) for a subset of LATEX. Specifically, we want to

recognize sections, references (`cite` and `ref` and \begin ... end –constructs. The preamble should be handled separately, so we can reuse the same parser for TEX–files intended only for inclusion, ie. files not containing a preamble and a \begin{document} ... end{document} –block.

An easy way to achieve this is to recognize command words and their structure (ie. we don't have a subsection without a preceding section) using a parser. For generating the lexer and parser from an EBNF description, the tool SableCC is used (see `http://www.sablecc.org`.)

SableCC was chosen over JavaCC and ANTLR primarily because it doesn't require entering action code into the grammar specification and the CST to AST transformation syntax is concise and clear. In contrast, JavaCC and ANTLR require extensive action and tree transformation code to be embedded into the grammar, resulting in messy, difficult to debug, difficult to maintain and hard to read code. SableCC solves this problem with clean grammar files and encouraging the use of a visitor pattern to transform the generated AST for different uses. In TEXlipse, one such use is to extract all the data necessary to make an outline and present it in a tree structure.

There is, however, one disadvantage with this approach: TEX and BibTEX contain constructs of type $A \rightarrow \{A\}$, which are not recognizable by regular expressions but are with context-free languages. Beebe [Bee93] solves this with action code in the Lex-definition. This would be possible in eg. ANTLR, but not directly in SableCC. The SableCC object-oriented framework does, however, offer the possibility to subclass the lexer and implement the `filter()` method, where such action code can be embedded. There are other ways to solve the problem; the constructs can be recognized in the parsing phase and then concatenated (in practice, we want to handle BibTEX-strings of the form { ... { ... } ... } as one string) by visiting the AST. In practice, subclassing the lexer is very performance efficient and makes the later stages much simpler. The only drawback is that the lexer is not fully understandable from the SableCC-definitions alone.

Other reasons for choosing SableCC was the support for unicode lexers (which can be useful in the future) and the fact that it makes an LALR-parser, not LL(k) as does JavaCC and ANTLR. The latter suffer practically no penalty in terms of expressive power by the use of predicates, but these come with significant penalties in maintainability and debuggability. Also, they don't have mechanisms to check for the validity of the formed AST, leaving this to the programmer unlike SableCC. For further comparison and details of SableCC, refer to [Gag98].

In practice, however, further study of the syntax and possibilities of TEX and LATEX and the requirements of making TEXlipse, it became clear that the

fancy AST generation with visitors is not that advantageous for LaTeX as it is for BibTeX or programming languages. We could perform all the necessary functions (outline building, label and command extraction etc.) in a single pass, making the parsing simpler and faster. In particular, LaTeX doesn't have strict semantics in the way that programming languages have, so we would simply have had a grammar defining word interspersed by commands. Also, the possibility to define own commands and the bad-but-not-forbidden –constructs make LR parsers less useful. The eonly drawback with hand-coding the parser (the lexer is naturally automatically generated) was the somewhat massive parser class. However, due to the relative simplicity of the parsing task and the fact that the visitor would be equally complex but just have more methods, this approach was pursued.

See [ASU86] and [Knu65] for basic information on parsing and particularly LR-parsing. See [Knu86] for information on how the original TeX parses its syntax.

6.2 Syntax highlighting, basic case (T1.1)

`Package: editor`

Syntax highlighting can be made easily by using a simple lexer/DFA that recognizes TeX's keywords and colorizes them. This can conveniently be done using Eclipse's built-in scanner-facilities, which can be given rules and then lex the document automatically. The highlighting itself is easy, but the expressivity of the premade rules is limited, so we need to make our own rule-classes.

6.3 Code folding (T1.2)

`Package: editor (outline handling in model, parsing in texparser)`

Eclipse provides a framework for code folding and the foldable sections can be recognized either with the document scanners (as in Section 6.2) or the outline tree made by `TexParser` can be used. For the foldable sections, their positions in the document must be stored. We do this in the LaTeX-parser by simply reusing the document outline tree that we need to create for the outline. The same positions needed in the outline are used as positions for code folds.

The actual code folding is largely done by Eclipse-classes, but we need to create the folding annotations, which means traversing the outline tree and

making suitable annotations from each node to be placed into the code folder. This is somewhat tricky, since the folder has a flat datastructure, which makes it somewhat difficult to determine which annotation in the folder corresponds to which node in the tree (eg. for maintaining folding across a save.)

6.4 Automatic indentation (T1.3)

`Package: editor`

Classes for supporting automatic indentation are provided with Eclipse. The indentation strategy can be determined by using the document scanners mentioned in Section 6.2. In addition to this, an entirely own logic of when and how much to indent is made. It bases it's decisions on the previous lines, as do practically all other Eclipse editor plugins.

6.5 Make BibTEX parser (T1.4)

`Package: bibeditor`

The BibTEX grammar is more strict than TEX and merely defines an entry format to specify bibliography entries. Due to this, it is fairly well suited to LALR-parsing.

The grammar is made using SableCC, which creates an AST automatically. Section 6.1 explains the rationale behind using SableCC. The framework for parsers in TEXlipse is explained in Section 5.2.1. It is worth noting that the framework permits elegantly adding support to TEXlipse for some other bibliography format, which might be desirable due to the problems with BibTEX(problems recognizing string literals, somewhat badly defined comment syntax among others.)

The BibTEX grammar is not very well defined (or designed), so some .bib files using uncommon syntax might not parse correctly (use prof. Nelson Beebe's tools for pretty printing them.) However, the TEXlipse bib-parser recognizes all the common BibTEX-conventions. The grammar is based pretty much on Beebe's observations in [Bee93].

It should be noted that LR-parsing is considered significantly harder to debug than LL, but having done extensive testing with SableCC for use in Eclipse we have not found this to be a problem, in part due to the excellent automation and error-detection of SableCC.

See [ASU86] and [Knu65] for basic information on parsing and particularly LR-parsing.

See [Lam85] and [Pat03] for further information about the BibTEX format.

6.6 Code completion (content assist, T1.5)

`Package: editor (completion handling in model, generation in tex-parser and bibparser)`

For code completion we need both the user's BibTEX-file's contents and the labels defined in the document. The .bib -files are parsed at startup and when saving the bib-files. What bib-files to parse are read from the document's `\bibliography` -command. The labels are retrieved whenever the project documents are parsed. They are stored into two similar datastructures (one for completing `ref` and the other for `cite` commands) in the model, from which the editor's code completion classes can fetch them. The data structure containing the completions is stored so that every model in the project can access it and it supports partial compilation so that recompiling one bib-file doesn't require recompiling all the others to enable completion. Thus, performance can be increased by splitting the bib-files.

The Eclipse plugin framework provides a number of classes and interfaces for supporting code completion in the editor view.

Storing the completions in a linear structure (array) and searching it takes $O(n \cdot m)$ time, where n is the size of the array and m is the time for partial matching a string. With a B-tree, the time is $O(\log n)$, but constructing it is more difficult and the constant terms dimnish the advantage. A third approach is to make a sorted array and use modified binary search to fetch the entries. The modified binary search (to fetch a range of values) is still $O(\log n)$ and sorting can be done in $O(n \log n)$ time, but this is only done after a modification on the reference source document. The constant terms are smaller than with a B-tree and the implementation is much simpler, in part since we can use Java's built-in mergesort.

Performance must be evaluated to make hard conclusions. In practice, the third option was implemented based on theoretical merits and seems to provide very good performance.

6.7 Template mechanism (T1.6)

`Package: templates, editor`

There are two different types of templates – project templates and LaTeX templates. The former ones are whole LaTeX documents (they can be compiled directly), which may be used when a new LaTeX project is created (i.e. selected template is copied to the main project file as it is). The latter templates are smaller pieces of LaTeX code (for example *lists* or *theorem & proof* structures), that can be inserted anywhere into the document.

The user can define her own templates, both project and LaTeX. The system has two directories, one for each template type (namely, *<TeXlipse plugin>/templates/project/* and *<TeXlipse plugin>/templates/latex/*), in which the templates reside. New templates are added simply by copying them to the corresponding template directory. In addition, the user may specify her own template directories and copy her own templates there (this is for multiuser environment, where the user may not be ably to modify the system's template directories due to insufficient rights – i.e. the user is not the *root* user).

Where as the project templates are simply copied when they are used, the LaTeX templates are a bit more flexible. The template handling is really a special case of using content assist. Thus, editor templates can be used as content assist is used and they can be edited, exported and imported in the Eclipse Preferences.

6.8 User defined templates (T1.7)

`Package: templates`

The user can freely add her own templates and add them to the system's LaTeX template directory (or define her own LaTeX template directory via preferences, if the user does not have sufficient rights). Editor templates may be added on the Templates-page on the TeXlipse –page of the the Eclipse Preferences.

6.9 Commenting blocks (T1.8)

`Package: actions`

Blocks (region in emacs-parlance) can be commented by inserting a % -sign at the start of each line in the block. They can be removed by reversing the process and ignoring leading whitespace.

Alternatively, `\begin{comment}` and `\end{comment}` -commands can be used, but noticing them is not as obvious (especially if one has to use a

non-highlighting editor due to some reason), so using the % -syntax was chosen.

6.10 Annotations for errors (T1.9)

We use the built-in annotation facility and place markers on offending lines. Offending lines can be recognized by parsing the document and examining the document references' symbol tables.

Offending lines are also recognized from the output of the build process. The builder parses the output of LaTeX , BibTeX , and such document builder programs, which report errors about the source documents.

6.11 Matching parens (T1.10)

See Section 6.2; essentially this is done with the same tools and it uses facilities provided by Eclipse.

6.12 Word counter (T1.11)

`Package: actions`

The word counter action enables counting the number of words in the selected region, taking into account the special characteristics of LaTeX-source. Due to this, this is most conveniently performed by making a simple parser that gets its input from the LaTeX-parser (see Section 6.1) and the determines how to count based on the token encountered.

6.13 View the outline (T2.1)

`Package: outline`

The outline is displayed in a tree structure similar to that of the Java editor of Eclipse. For creating the tree structure, a TreeViewer viewer will be used. The viewer allows us to avoid working directly with SWT widgets and their event handling. Instead we can concentrate on providing the model of the outline. The outline shows the outline of the document being edited. See also Section 4.3 for an overview of the document model.

The TreeViewer itself does not know much about the contents of the outline.

It retrieves the elements of the outline from `ITreeContentProvider` and uses a `LabelProvider` to get a presentation (text and icon) for each element. Thus we need to implement a `TexContentProvider` and a `TexLabelProvider`.

Parsing the document to find the elements is handled by the `TexModel` and the underlying `TexParser`. The `TexModel` then provides a tree structure for the `TexContentProvider`. The elements of this tree contain name, type, begin line number and end line number of the element.

When the user changes the document, the `TexModel` is changed too. And if needed the Model triggers the outline to be updated. Thus the outline itself does not actively monitor whether the document is changed or not.

Filtering the elements of the outline can be implemented using ViewerFilters.

6.14 Basic outline navigation (T2.2)

`Package: outline`

When the user selects an element in the outline view, the editor is focused on that element. Correspondingly the selection in the outline follows the movement of the cursor in the editor. Moving the cursor and selecting elements in the outline are functions that do not change the actual content of the document. Events for these kinds of functions are communicated directly between the `TexOutlinePage` and `IDocument`.

Catching the selection event in the TreeViewer and focusing the editor is quite straightforward. Finding out the right element after moving the cursor is a bit more complicated. It can be done either with the Partitions of the IDocument or with some kind of search structure living in the `TexModel`.

6.15 Copy/paste in outline (T2.3)

`Package: outline`

Can be done by by similar techniques such as code folding, ie. storing line numbers and moving the affected sections in the document.

6.16 Drag'n'drop in outline (T2.4)

`Package: outline`

See Section 6.15. The technique is the same, but drag and drop is enabled in the Eclipse plugin framework.

6.17 File output/building (T3.1)

`Package: builder`

Output files are produced by LaTeX. The builder is an implementation of Eclipse's `IncrementalBuilder`-interface. The builder will run the external LaTeX process when the user chooses `Build Project` from the Eclipse's Project-menu. The output files will be saved to a special output directory defined in the project properties. The temporary files may also be saved under a dedicated temporary files -directory, if the user wishes so. This may clarify the view on Eclipse's directory navigator, if the project has plenty of source files.

If necessary, the builder will also run BibTeX and LaTeX automatically so that the references are resolved in the document (this means running LaTeX, then BibTeX and then LaTeX twice in the worst case.)

Depending on the configured output format, the builder process will also run other external programs to convert the LaTeX -generated dvi file to other formats.

6.18 Displaying build errors (T3.2)

`Package: builder`

If a build fails because of invalid syntax in the LaTeX input file, the plugin will record the output of the LaTeX process and parse errors from it. Errors reported by LaTeX will be displayed in annotated form using Eclipse's `Problems`-log.

6.19 Linking errors to source (T3.3)

`Package: builder`

The builder will add `IMarker`s to the lines of source files which were reported to have errors by LaTeX. Markers are automatically linked to the error messages by Eclipse's API. User can jump directly to source by double-clicking the error message in the `Problems`-log.

6.20 Preview support (T3.4)

`Package: builder, viewer`

Previewing of the LaTeX document is done with an external viewer (dvi or pdf.) Depending on the capabilities of the viewer, different options (like line number and refresh notification) can be provided for the previewer via free form command line arguments.

The reason for not making internal (dvi or pdf) previewer is rather straightforward: firstly, the user can use the previewer she is accustomed to (instead of a predefined and, quite possibly, inferior one), and it greatly reduces the effort needed to keep the internal previewer up to date.

6.21 Linking preview to source (T3.5)

`Package: viewer`

A previewer can be linked back to the source as long as the previewer can pass the neccessary information – a filename, a line number and possibly a column number – either via printing to standard output (lines formatted as *filename:line* or *filename:line:column*) or run an external program (using arguments to pass information).

For the latter case: a small client program is provided with TeXlipse distribution, which send the information it receives via command line arguments to a port. Then the port is listened by TeXlipse plugin.

Naturally the previewer must also be able to extract the source information from the output (dvi or pdf) file. There are no restrictions about how this source information was originally included into the output file. The default way (if not configured otherwise) is to build the LaTeX source with *–src-specials* option — most previewers, like Yap (Windows, MikTeX) and Xdvi (Unix/Linux) are compatible with this source information.

6.22 Support for a LaTeX project (T4.1)

`Package: wizards, properties`

A possibility to start a LaTeX project will be provided in Eclipse's `New Project`-menu. Choosing `New Latex Project` will start the new project wizard, which is an implementation of Eclipse's wizard interface. The new project wizard will perform basic project creation tasks like creating a project

directory and the project's main file using an optionally specified template.

The LaTeX-project will also include a property page to handle such things as keeping track where the main file of the project is.

6.23 Support for partial building (T4.2)

`Package: builder`

Partial building refers to the process of creating a preview of some part of the document. If the document consists of a main file and many sub-files which are all included to the main file, the document can be built partially so that only the contents of one of the sub-files is visible in the preview. This is done by extracting the header ("premeable section") and footer from the main file and creating a temporary file by concatenating the header, the chosen sub-file and the footer. This temporary file is then built like normal LaTeX-document. Building partially is obviously much faster than building the full document, provided that the sub-files are all much smaller than the full document. Partial building can be enabled from Eclipse's menu.

6.24 BibTeX editing (T5.1)

`Package: bibeditor`

Implement an editor mode for .bib-files. Essentially, this uses some of the techniques described here for LaTeX-documents, only that editing BibTeX-files is simpler. Due to this, we try to reuse code from the LaTeX editor part as far as possible, eg. the search algorithm for code completions is shared by both editors. Generally, the bibeditor is a simpler case than the LaTeX-parser and lacks external tool interfaces (such as building), since they don't make much sense for BibTeX.

References

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, USA, 1986.

[Bee93] Nelson H. F. Beebe. Bibliography prettyprinting and syntax checking. *TUGBoat*, 14(4):395–419, 1993. December.

[Gag98] Etienne Gagnon. SableCC, an object-oriented compiler framework. Master's thesis, School of Computer Science, McGill University, Montreal, 1998.

[Knu65] Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965. This is the original paper on the theory of LR(k) parsing.

[Knu84] Donald Knuth. *The TEXbook*. Addison–Wesley, Reading, Massachusetts, 1984.

[Knu86] Donald Knuth. *TEX: The Program*. Addison–Wesley, Reading, Massachusetts, 1986.

[Lam85] Leslie Lamport. *LATEX – A Document Preparation System — User's Guide and Reference Manual*. Addison-Wesley, Reading, MA, USA, 1985.

[Pat03] Oren Patashnik. Bibtex yesterday, today and tomorrow. *TUGBoat*, 24(1):25–30, 2003.