THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

An Eclipse Environment for Z

Chengdong Xu

This thesis is submitted in partial fulfillment of the requirements for the Degree of Master of Science at the University of Waikato.

December 2006

Abstract

This thesis reports on the design, implementation and evaluation of a new interactive Z environment that is integrated into the Eclipse environment.
The Z language is a formal specification notation that is used to describe and model computer-based systems. For the widespread use of Z, it is desirable to integrate Z support with a mature and popular editing environment, such as Eclipse. Eclipse was chosen as the basis for the new Z environment because it is widely used, it provides rich functionality and it is designed to be extensible.
The Z environment described in this thesis extends Eclipse to provide a large set of useful features for editing and analyzing Z specifications, such as a table of Z symbols, constant detection of syntax and type errors, outlining facilities, cross-referencing of Z names and conversion between different Z markups. User feedback shows that the resulting Z environment is helpful for editing and correcting Z specifications.
The development of the Z support for Eclipse is a part of the CZT (Community Z Tools) project, which is a Java framework for building Z tools and provides parsers, typecheckers and other Z tools.

Contents

List of Figures

List of Tables

Abbreviations

CZT – Community Z Tools

IDE – Integrated Development Environment

GUI – Graphic User Interface

JDT – Java Development Tool

PDE – Plug-in Development Environment

SDK – Software Development Kit

WYSIWYG – What You See Is What You Get

Chapter 1 Introduction

1.1 Overview of Z Notation

"The formal specification notation Z (pronounced "zed"), useful for describing computer-based systems, is based on Zermelo-Fränkel set theory and first order predicate logic. It has been developed by the Programming Research Group (PRG) at the Oxford University Computing Laboratory (OUCL) and elsewhere since the late 1970s, inspired by Jean-Raymond Abrial's seminal work. Z is now defined by an ISO standard and is public domain" (Bowen, 2005).

"*The Z specification language was adopted as an ISO standard in 2002*" (Malik & Utting, 2005).

The Z notation is a mature, model-based notation. In Z a computer-based system is modeled by representing its state and some operations that can change its state. Z is not another programming language. Z specifications are not intended to be interpreted or compiled into a running program for execution. The Z notation is used to describe and model computer-based systems and is formal enough to be understood by computers. However, it is not intended to be executed by computers, because it is designed for human readers only. The Z notation is extremely powerful for its use in two rather different ways: a descriptive style and an analytic style. Z can be used to model some particular system such as a text editor or a radiation therapy machine, but it can also be used for the definition and extension of itself (Jacky, 1997, p.32).

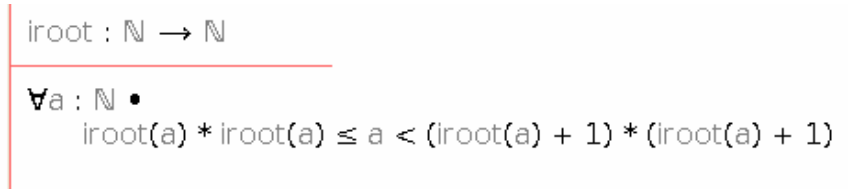
To see how a Z specification differs from the code written in a programming language and be clear why a Z specification can not be expected to be executed by computers, here is a very simple example written in the C language.

```
/* Compute the integer square root */
int iroot (int a)
{
  int i, term, sum;
  term = 1; sum = 1;
  for (i = 0; i <= a; i++)
    term = term + 2;
    sum = sum + term;
  }
  return i;
}
```

(Jacky, 1997, p.34)

As the comment in the first line shows, this tiny C function is supposed to compute the integer square root of an integer number. By calling `iroot` with a set to an integer number, for example, `4`, it should return `2` as its result. However, as you know, some numbers do not have integer square roots at all. But the function's name and comment do not explain the function's behavior for every input. They turn out to be not complete, and thus not be as helpful as they are supposed to. It is clear that a code written in a programming language can only describe its computation, but not the actual result of the computation (Jacky, 1997, p.34).

Here is the Z specification describing the behavior of `iroot`.



(Jacky, 1997, p.35)

As seen in the above snippet of Z specification, the `iroot` function is expressed in a single Z paragraph called an axiomatic definition. The text is delimited by a sort of box as shown above. The big horizontal line divides the paragraph into two parts. The text above the line is the declaration part of the paragraph, which corresponds to the declaration of the C function:

```
int iroot (int a)
```

The ℕ is the symbol for natural numbers. The declaration part in the Z paragraph tells that the input (the ℕ to the left of the arrow) to the function must be a natural number and the result (the ℕ to the right of the arrow) is also a natural number (Jacky, 1997, p.35).

The text below the big horizontal line is the predicate part of the paragraph. It explains that `iroot` returns the largest natural number that is at most the square root of the argument, `a`.

The above example clearly shows the key difference between Z and an executable programming language. "The predicate part in a Z definition describes what the function does without explaining how to do it; the body of a C function describes how to compute the function without explaining what the result will be" (Jacky, 1997, p.37). The example also shows that Z is not a programming language.

A Z notation is typically represented in the form of a mixture of texts, Greek letters, boxes, and some invented Z-specific pictorial symbols, which can help users quickly grasp the structure of the model of a computer-based system. However, the input of those special symbols is also a big pain for Z notation

writers, so suggestions for rendering the Z notation symbols in ASCII and in LaTeX are also included in the **ISO Z Standard** (ISO/IEC 13568, 2002).

One of the main benefits of formal specification languages like Z is that the specifications can be analyzed by tools to find errors and explore consequences. There are many Z tools available, but the most common tools used by Z users are the Emacs editor and command line typecheckers. The Community Z Tools (CZT) project was started in 2002 to build sophisticated tools support for the new ISO Z Standard. "The CZT project is building a set of tools for editing, typechecking and animating formal specifications written in the Z specification language, with some support for Z extensions such as Object-Z, Circus, and TCOZ. These tools are all built using the CZT Java framework for Z tools" ("Welcome to CZT", 2006)

1.2 Overview of Eclipse

Eclipse as an open source project was launched by a group set of companies – IBM and seven other companies in the fall of 2001. Since then, "*Eclipse has taken the computing industry by storm. The download data for the Eclipse SDK (Software Development Kit) is astounding and a true ecosystem is forming around this new phenomenon*" (Arthorne & Laffra, 2004, p.3). In the last few years, the Eclipse organization has grown surprisingly fast. It has been organized as a non-profit corporation known as the Eclipse Foundation from its original eight companies. Due to its increasing growth, it is hard to define what Eclipse is. It can be thought as a platform for different IDEs or a general-purpose application for some users without programming with specific languages. Of course, it can be a perfect Java development environment for Java programmers. It is also an open source community. To quote the Eclipse FAQ: "*The official party line is, Eclipse is an open (IDE) platform for anything, and for nothing in particular*" (Arthorne & Laffra, 2004, p.3). That is, the Eclipse platform is a universal

IDE for all platforms without thinking of any particular tool support initially.

The Eclipse SDK usually serves as the representation of Eclipse when being spoken of by its users. The SDK is comprised of several contributing projects, which include the platform subproject, the Java Development Tools (JDT) subproject and the Plug-in Development Environment (PDE) subproject. The minimal units that make up Eclipse are the components known as the Eclipse plug-ins. All components in the Eclipse platform are plug-ins including the famous JDT, which is also an independent project and no "backdoor" is open to its development, even though it comes with Eclipse.

Unlike other tools, Eclipse provides tools builder with a new way of thinking. Tools builders should always think of Eclipse as their student and they "*teach*" it what to do for their problems by writing the so-called plug-ins. From the users' point of view, they would prefer a universal tool with support for as many capabilities as possible so that they do not have to switch to different tools for different behaviors and programming languages. That is exactly what Eclipse does. After teaching Eclipse what it should do and re-starting Eclipse, users see new functionalities in Eclipse, rather than a new tool and they can not tell where one of the tools assembled into Eclipse starts or where another tool ends. The only limitation of Eclipse is obviously the plug-in developers' imagination. It is the developers who create the value of Eclipse as they can decide how to extend Eclipse and how Eclipse performs.

However, to get new functionalities that integrate well with Eclipse and plug-ins that others contribute, the users should know how to "plug in" their specific functionalities into Eclipse based on the integration points, known as the *Eclipse extension points*.

This section is going to give a brief overview of the Eclipse platform. As talked above, the Eclipse platform is obviously a foundation platform on

which you can build different Integrated Development Environments (IDEs) for different purposes. However, it is also a foundation platform for building arbitrary applications and purpose-centric tools. By building tools or applications based on the Eclipse platform, you can get benefits from the Eclipse platform by integrating your tools or applications with other Eclipse-based tools or applications contributed by other developers or teams. Thus, Eclipse users can have a rich and consistent experience in using Eclipse for different purposes. Different people may have significantly different reasons for using Eclipse, but most users, particularly developers, require good user interfaces which can facilitate the management of their digital resources, such as browsing or editing their projects. Thus the Eclipse platform provides a managed window system in which there are a lot of useful user interface components.

Figure 1.1 shows a screenshot of the main workbench window that Eclipse provides.

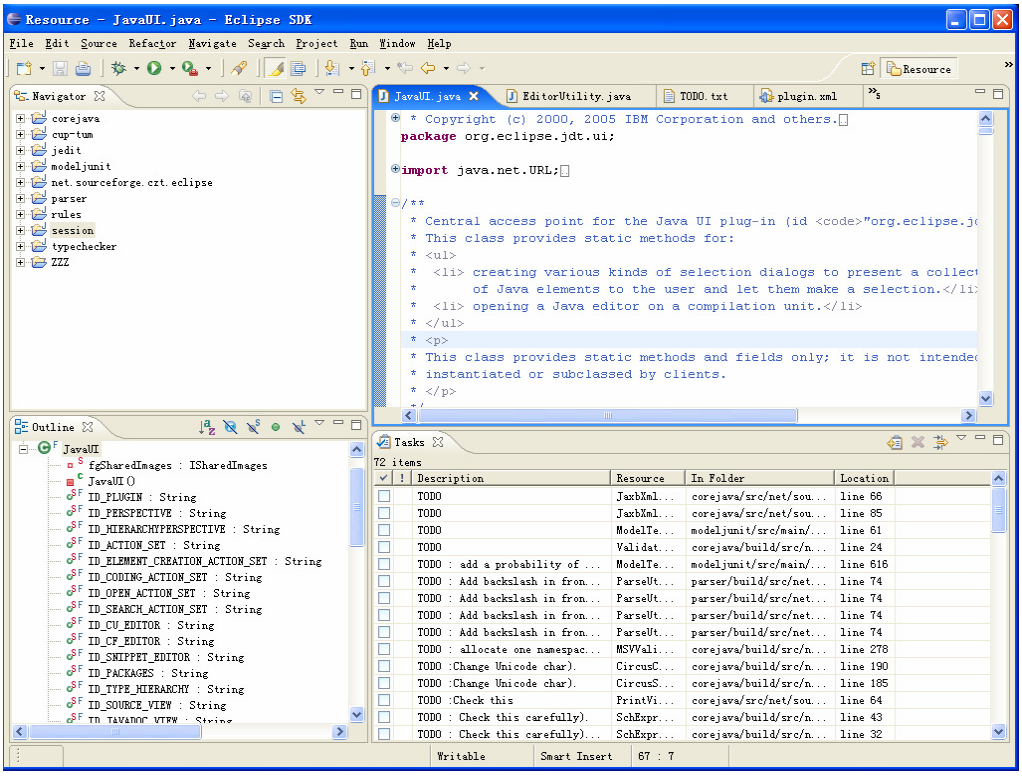


Figure 1.1 – The Eclipse Workbench Window

As seen in Figure 1.1, the Eclipse platform provides its users with a powerful window containing a comprehensive collection of standard generic user interface components such as editors and a set of views.

The text editor in the top-right corner provides the ability to browse and/or edit the contents of text-based resources in projects. Projects are typically collections of file-based resources. The resources navigator view positioned on the top-left corner enables users to browse the resources in the system. The outline view on the bottom-left corner typically outlines the contents of the resource that the user selects and often reflects the user's selection. The Eclipse platform also provides a rich set of other useful views/panels which can facilitate the use of Eclipse and other contributed plug-ins. Some of the most useful views are:

The Problems View: Plug-ins can inform users of the problems (errors, warnings, useful information, etc.) related to a specific resource.

The Tasks View: This is shown at the bottom of the above picture. It allows users to create their future tasks so that later they can see what they intended to do. They can easily jump to the correct position of the related resource by clicking on a particular entry in the view.

The Error Log View: This usually informs the Eclipse users of the errors Eclipse produced during its runs. The plug-in developers can also contribute to the Eclipse platform by adding error entries to inform their plug-in users of the problems that occur when they are using the plug-ins.

The Console View: This allows plug-ins to output some useful information that is not suitable to put into other views or output some particular information like compilation progress messages.

As well as the editors and views, the Eclipse platform also provides menu bars, tool bars, a status bar and pop-up context menus that provide commands that act on specific resources.

The above functionalities are just some of the commonly used generic functionalities which are built into the Eclipse platform, although there are lots of other functionalities that the Eclipse platform provides. The Eclipse platform provides plug-in developers the ability of extending the platform by adding new content types, adding new functionality or modifying existing functionality on new or existing content types.

As the editor framework is the most important part provided for most Eclipse users, especially the application developers, the following gives an overview of some important functionality that the Eclipse editor framework offers to its users.

Figure 1.2 shows that a .TXT file “TODO.txt” is opened with a generic text editor.

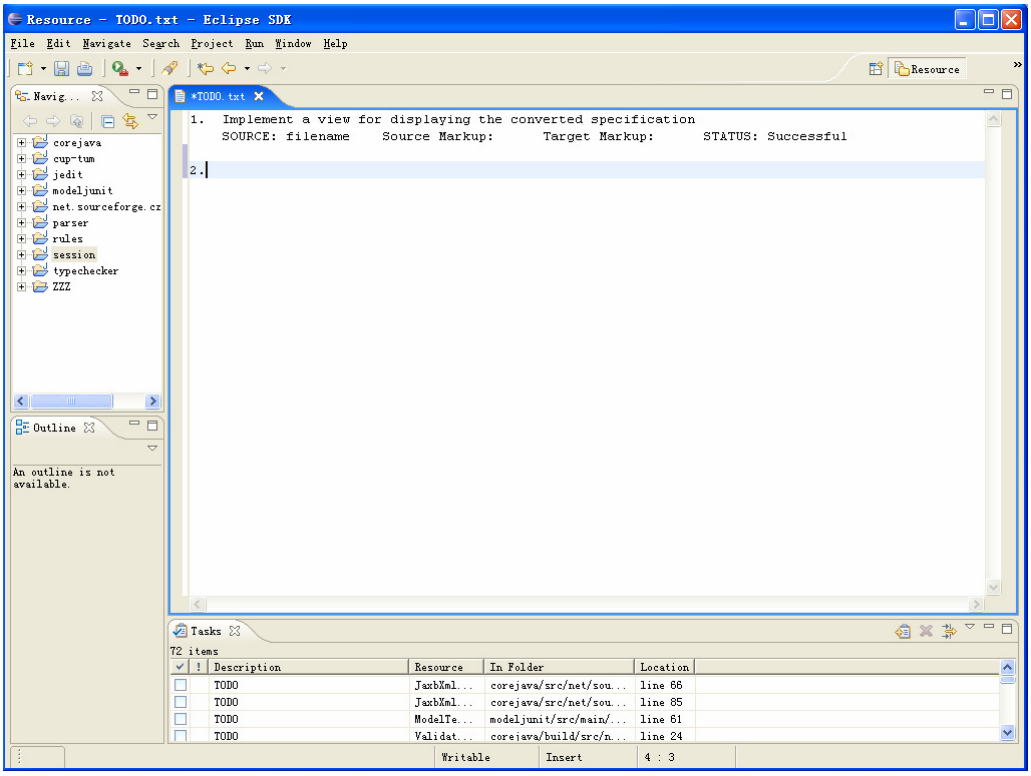


Figure 1.2 – Eclipse Generic Text Editor

When a file is opened for editing, the editor that is opened depends on the type of the file. In this case, the text file is opened with the generic built-in text editor. The name of the edited file is displayed on the tab of the generic text editor with an asterisk (*) shown before the name. The asterisk (*) indicates that there are unsaved changes associated with the opened text editor since it was saved last time. Each time the user attempts to close an editor with unsaved changes, a dialog will be presented to the user to allow the user to choose to save or discard the changes. The display of the unsaved state of an editor is a strong user requirement.

The Eclipse platform editor framework also provides the support for visual presentation of the contents of a resource (see Figure 1.3).

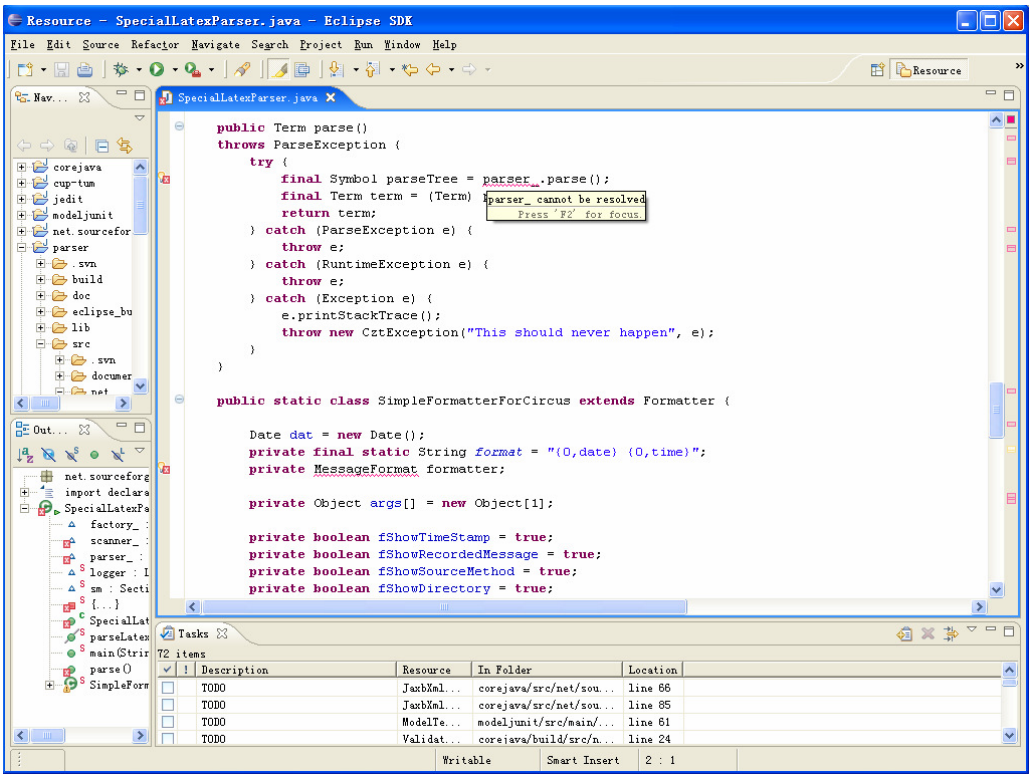


Figure 1.3 – Visual Presentation of Contents

The visual presentation support is desired by programmers during development. The Eclipse editor framework provides the support by allowing contributed plug-ins to configure the behaviors depending on their requirements. The picture above shows some visual presentation behaviors that the Eclipse editor framework provides.

The contents of a text-based resource can be colored according to some specified requirements. A particular paragraph of text can be folded up (see the signs (+/-) on the left side of the opened editor), which allows users to hide paragraphs they are not concerned with at that moment.

The editor framework also supports the visual annotation of resource-related information on vertical bars along both sides of the editor. These are normally used to indicate errors, warnings or other useful information associated with the contents of the editor on the related lines.

As seen in the middle of the editor, there is also a box of text floating in the editor. This is the Text Hover support, which is used to provide some help information regarding the text in the editor when the mouse is hovered above that text. The help information will remain until the mouse leaves the area. The hover support is also provided for the vertical bars on both side of the editor. On the left vertical bar it normally provides some information regarding the line of the text where the mouse is hovering. On the right vertical bar it provides the same information as it provides on the left vertical bar. One difference is that the information provided on the right does not correspond to the absolute position of the mouse, but to the position relative to the whole text. Another difference is that when the mouse hovers above the top end of the right vertical bar, the information provided is the count of some significant information about the text in the editor, such as the count of errors.

1.3 Why develop a CZT plug-in for Eclipse?

When we are making the decision to do something, we usually ask ourselves why we want to do it. The reasons for developing the CZT plug-in for Eclipse can be outlined as follows:

- **Specification Maintenance**: Just like most programmers, Z specification writers would like not to write Z notations manually on pieces of paper, but to use editors on computers so that they can easily maintain the specifications. When a Z specification is created in an editor program, it should be saved in a physical file on disks for future reviews and maintenance.
- **Support for Existing Eclipse Users**: The Eclipse platform is becoming more and more popular among programmers and software developers. As Dave Thomson, Eclipse Project Program Director, IBM, says, "*Eclipse is a game-changing industry phenomenon. It's*

great technology, it's open source, it has phenomenal support from a worldwide community of developers, and a whole lot of important industry players are using it to build great products" (D'Anjou & Fairbrother & Kehn & Kellerman & McCarthy, 2005, xxix). Since Eclipse has such a significant impact on the tools industry, there is no reason to force Eclipse users to edit Z specifications with other tools that they may be unfamiliar with or even dislike.

- **Modern WYSIWYG Editor for New Z Users**: "One of the biggest barriers to the widespread use of the Z standard is the issue of tool support" (Malik & Utting, 2005). For the widespread use of Z, a set of tools should be supported to facilitate the editing of Z notation so that new Z users can enjoy the Z notation and thus keep using it. To attract new users and keep existing users, it is better to have a Z editor that is based on a popular and mature editor framework. The Eclipse editor framework is mature and increasingly popular. It has formed its own community with user groups in diverse areas. It also has sufficient existing users and it is attracting more and more potential users.
- **Multiple Markups**: A Z specification can be created in one of the following types of markups: LaTeX markup, Unicode markup, ZML markup. If it is created in Unicode markup, it will be stored and reviewed in either UTF-8 encoding or UTF-16 encoding. Most text editors can not browse or edit all of these markups and encodings properly, so users have to use different editors to edit Z specifications for different markups. To solve the problem, it is better to have an editor to support all of these markups and encodings. Ideally, the editor can automatically use proper markup or encoding to browse or edit a Z specification without users' concerns.
- **Insertion of Special Z Characters**: When editing Z specifications using Unicode markup, users are concerned about how to insert into

the editor some special Unicode characters, especially the Z-specific characters, because existing editors do not provide convenient ways of finding the appropriate characters and inserting them.

- **Z-Specific Syntax Coloring**: Z users also like to browse or edit a Z specification with colored visual presentation. As the above sections discussed, the Eclipse editor framework provides a comprehensive collection of the functionalities that a CZT Editor would need.
- **Constant Background Error Detection**: It is a good practice for an editor to support constant error detection while a Z specification is being edited. The Eclipse editor framework supports constantly running a parser in the background during the editing. The background parsing work has no side-effect on the editing and thus users can keep editing without interruptions. Furthermore, the Eclipse editor framework also supports visual presentation of detected errors, such as parse errors and type checking errors, in different ways.
- **Outline Views**: When a Z specification is being browsed, Eclipse can provide an outline view of the specification. This feature is particularly useful when the specification is too long to fit into a page. The view can give the user the structure of the specification, then the user can choose an entry in the outline view and thus the corresponding part will be selected in the editor. This makes it easier to browse or edit a Z specification.
- **Support for Hiding/Showing Parts of A Specification**: The Eclipse editor framework also provides the support for hiding some parts of a Z specification that users are not temporarily concerned about and showing them later. Since Z specifications usually contain alternating paragraphs of English and formal Z, this is a useful way of being able to hide the English paragraphs while working on the Z

paragraphs, or vice versa.

- **Declaration – Use Cross-Reference**: Another good practice for a Z editor is to allow users to easily see all references of a variable they are interested in and jump to the section at which it is declared or defined. Like some other programming editors, the Eclipse editor framework supports these behaviors too.

1.4 Structure of the Rest of the Thesis

Chapter 2 introduces four editing environments that have similar goals to the CZT Eclipse editor plug-in.

Chapter 3 gives an overview of the CZT Eclipse editor plug-in.

Chapter 4 describes the **Z Character Map** view.

Chapter 5 introduces the CZT Eclipse Editor. It describes the features that the editor provides, as well as their design issues, alternatives and corresponding implementation techniques.

Chapter 6 introduces the wizards that the editor provides for creating new CZT projects or Z specifications.

Chapter 7, 8 and 9 introduces three useful views for Z development. They are the **Outline** view, the **Problems** view and the **Z Conversion** view respectively.

Chapter 10 introduces the CZT Eclipse perspective that facilitates using the plug-in.

Chapter 11 describes the first evaluation (regarding the design and features) conducted during the development of the plug-in.

Chapter 12 introduces the box rendering feature in the editor, as well as its

corresponding implementation techniques.

Chapter 13 describes the second evaluation (regarding the usability) conducted during the development of the plug-in.

Chapter 14 describes the preference settings that the plug-in offers to users.

Appendix A, B and C are the materials used for the usability evaluation (see Chapter 13).

Appendix D is the printout of the user documentation provided by the CZT Eclipse documentation plug-in.

Chapter 2 Related Work

This chapter describes four editing environments that have similar goals to the CZT plug-in for Eclipse. Two of them are for editing Z and the others are for editing other languages. The goal is to, by comparing them, knowledge what functionalities that other programming editing environments provide and then implement some of them including the functionalities desired by Z users.

2.1 CZT jEdit Plug-ins

The CZT jEdit plug-ins offer Z users a WYSIWYG (***What You See Is What You Get***) environment for editing Z specifications in both LaTeX and Unicode markups using jEdit. "jEdit is a mature programmer's text editor with hundreds of person-years of development behind it" ("jEdit – Programmer's Text Editor", 2006). "jEdit uses the Swing toolkit for the GUI and can be configured as a rather powerful IDE through the use of its plugin architecture" ("jEdit", 2006). jEdit is widely used for editing Java and is a competitor to Eclipse, so the way that it has been extended to support Z is especially relevant to and helpful for the implementation of the CZT Eclipse plug-in.

Some of its useful features are briefly described below.

- **Syntax Coloring**: The text in a specification can be displayed in different colors based on the syntax of the specification. The narrative section and narrative paragraph are particularly colored differently so that Z users can easily figure out the useful parts from the text colors.
- **Editing Mode Indicator**: The editing mode of the current buffer in the editor is shown on the status bar at the bottom of the editor.

Figure 2.1 shows how the editor displays the mode when editing a Z specification.

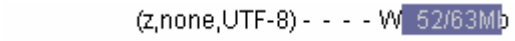


Figure 2.1 – Editing Mode Indicator

The figure can tell users that the current editing mode is Z Unicode and the encoding used is UTF-8.

- **Z Character Map**: Figure 2.2 below is the Z character map provided by the plug-in.

Paragraphs	S...	Op	[G]	Ax	Ax[]	::=	==	Sch	S...	⊢?			
Predicates	∀	∃	∧	∨	¬	⇒	⇔	≠	∈	∉	⊆	⊂	
Sets	ℙ	𝔽	{	|	•	}	∅	∪	⋃	∩	⋂	∖	⊖
Relations	↔	↦	×	◁	⩤	▷	⩥	⊕	⨾	⁻¹	⦇ ⦈	↗+↙	
Functions	→	⇸	⤖	↣	⤔	⤜	⇻	⤞	∘	λ	μ		
Sequences	⟨ ⟩	⌢	⌢/	1	↾	#							
Arithmetic	ℤ	ℕ	𝔸	+	−	-	*	div	m...	≤	≥		
Schemas	Δ	Ξ	θ	⦇..⦈	∖	↾	⨾	≫	′	↘1↖			

Figure 2.2 – Z Character Map

As described in Chapter 1, a major issue of editing a Z specification is to insert Z-specific Unicode characters. The character map solves the problem by displaying the characters so that users can insert the characters by mouse clicks. Additionally, the panel also allows users to insert templates of Z paragraphs by mouse clicks. If users are not familiar with the characters, they can also see the information about a particular character by making the mouse hovering on it.

- **Error List Panel**: as shown in Figure 2.3. The plug-ins provide this panel to indicate the parsing and type-checking errors to users.

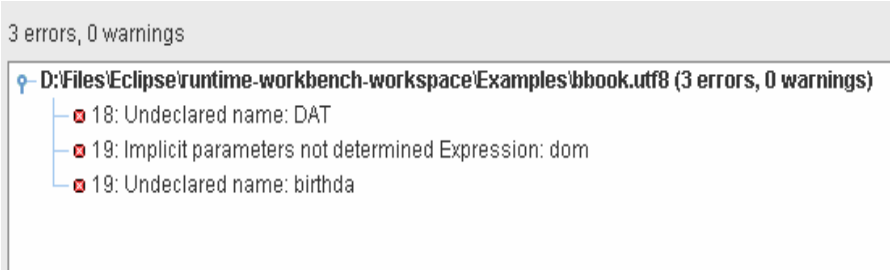


Figure 2.3 – Error List Panel

From this panel, users can easily see what errors the current specification has. Clicking on an entry in the panel will lead users to where the error occurs.

- **Z Sidekick Panel**: Figure 2.4 shows a part of the panel after parsing and type-checking a Z specification. The panel provides users an outline of the specification in the current buffer in the editor.

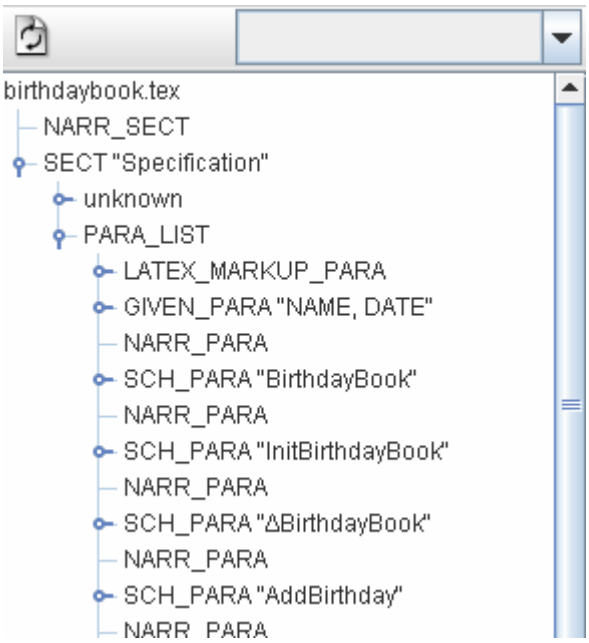


Figure 2.4 – Z Sidekick Panel

The panel is particularly useful when a specification is very long. Users can easily see the whole structure of the specification. Users can also jump to a part of the specification by clicking on the

corresponding entry of it in the panel. In addition, the panel provides users opportunities to choose different parsers for parsing Z specifications.

The plug-ins also offer users a set of commands for browsing or editing Z specifications. The major commands include:

- **Go to Previous/Next Asset**: These commands lead users to the start of the paragraph before/after the one in which the cursor is.
- **Convert to Different Markup**: A set of commands are provided for converting the current specification to different markups including LaTeX, old LaTeX, Unicode and XML. The output of the conversion is put into the main editor window as an unsaved buffer and then it can be saved for later use.
- **Highlight Term**: This command highlights the smallest term at the current cursor position by drawing a box around it. If a term is already highlighted, then the enclosing term of that term is highlighted if it has one. This allows the structure of the Z terms to be explored and terms to be selected for rewriting etc.
- **Go to Definition**: This command leads users to the definition part of the currently selected Z name.
- **Rewrite Selected Formula**: The command rewrites the selected formula for optimization of the current specification.
- **Prove Selected Predicate**: The command proves whether the currently selected predicate is true or not. It can help users to write Z specifications correctly.
- **Show Whitespaces**: This feature is particularly useful when some Z-specific whitespace characters are invisible. It can help users to see those characters and then to prevent users from deleting them accidentally.

2.2 CADiZ

"*CADiZ is a set of free software tools that aims to give direct support to use of ISO Standard Z notation. Compared to the notation of the Z Reference Manual of 1992, ISO Standard Z provides some useful generalizations. CADiZ has only minor deviations from the ISO Z standard, and resolves the looseness in that standard*" (Toyn, 2005).

Due to its comprehensive and powerful support for Z, CADiZ (Toyn, 2006) has been widely used by Z users. For this reason, the features that it provides to support Z are extremely relevant to the development of the CZT Eclipse editor plug-in. Some of its useful features have been adopted during the implementation of the CZT Eclipse editor plug-in.

Figure 2.5 shows a screenshot of its "**TypeSet**" panel for browsing Z specification.

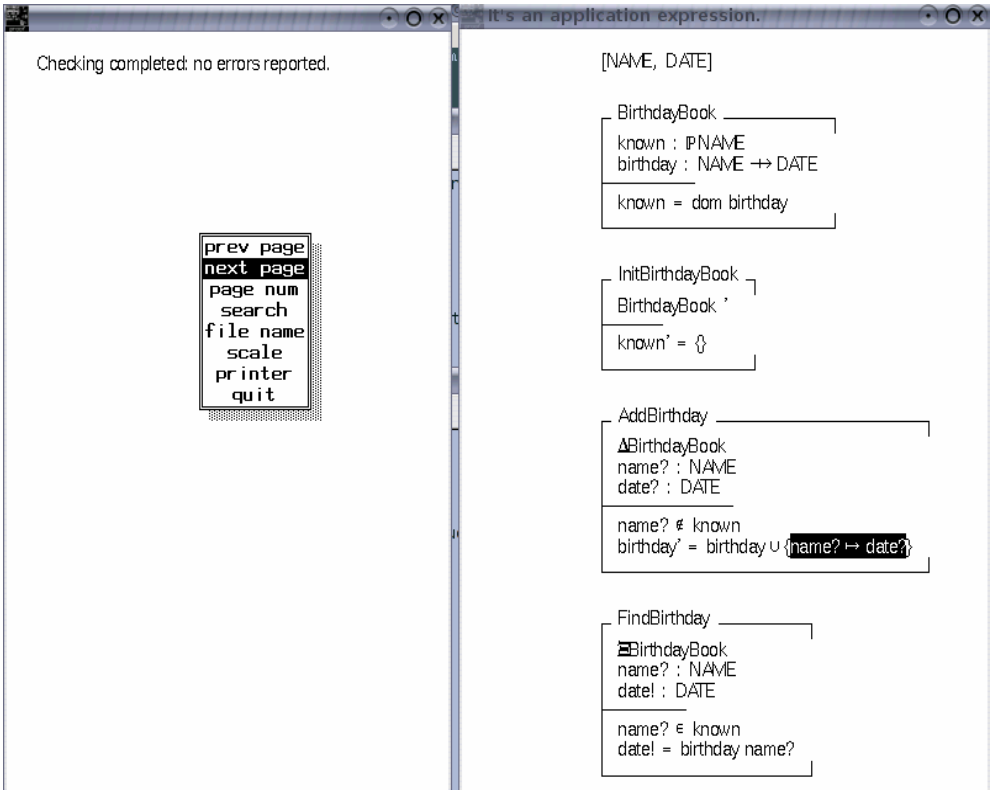


Figure 2.5 – Typeset Panel

As seen in Figure 2.5, there are two panels used for browsing Z. The left panel is used for displaying the errors from typesetting the specification on the right panel, while the right panel is the main panel used for browsing purposes. Some features are listed below.

- **Selecting a formula**: The user can use mouse-clicks to select a formula inside the specification. The first click selects the smallest formula around the cursor position. Each further click within the existing selection selects the next larger formula. If a paragraph is selected, the further click does not select a larger formula, as it is already the largest formula for a selection. Instead, it considers as no selection exists at the moment and the click selects the smallest formula containing the current cursor position. Each time a formula is selected, its syntactic category (the class of the underlying term) is displayed in the window's title bar.
- **Showing the Type of a Z Name**: When a Z name is selected by a mouse-click, the user can use the context menu entry "type" to see its type in the window's title bar.
- **Jumping to the Declaration Part**: When a Z name is selected, the user can use the context menu entry "declaration" to make the declaration part of the name being selected in the panel.
- **Showing Next Use of a Z Name**: When a Z name is selected, the user can use the context menu entry "next use" to select the next occurrence of the name in the panel.

However, CADiZ can not be used for editing a specification so that Z users have to switch to another tool, for example *Emacs*, for editing. It is not a standard interface so as to make Z users hard to use it due to its unfamiliarity.

2.3 Java Development Tools (JDT) Plug-in

When talking about Eclipse, people are often talking about the Java

Development Tools (JDT) (http://www.eclipse.org/jdt/) included with Eclipse as it provides users with a rich experience of the functionalities that Eclipse provides.

"*The JDT project provides the tool plug-ins that implement a Java IDE supporting the development of any Java application, including Eclipse plug-ins. It adds a Java project nature and Java perspective to the Eclipse Workbench as well as a number of views, editors, wizards, builders, and code merging and refactoring tools. The JDT project allows Eclipse to be a development environment for itself*" ("Eclipse Java Development (JDT) Subproject", 2006).

Here is a brief overview of some features that the JDT plug-in provides.

The JDT plug-in provides users with a set of Eclipse perspectives including the Java perspective. An Eclipse perspective defines the set of actions and parts, including editors and views, which will appear in a window when it is opened. The initial size and position of views within the workbench window can also be specified in the perspective.

The primary view within the Java perspective is the **Package Explorer** showing the hierarchy of Java files and resources within the Java projects. The **Package Explorer** offers a Java-centric view of resources, showing each package as a separate element in a flattened hierarchy, rather than a file-centric view showing a package as nested folders (Clayberg & Rubel, 2004, p.8).

The JDT plug-in provides users with a set of wizards to help users easily create different types of new Java elements. The set of wizards includes:

- **New Java Project Wizard**: This helps create a new Java project in the workbench. Figure 2.6 shows a screenshot of the new Java

project wizard page.

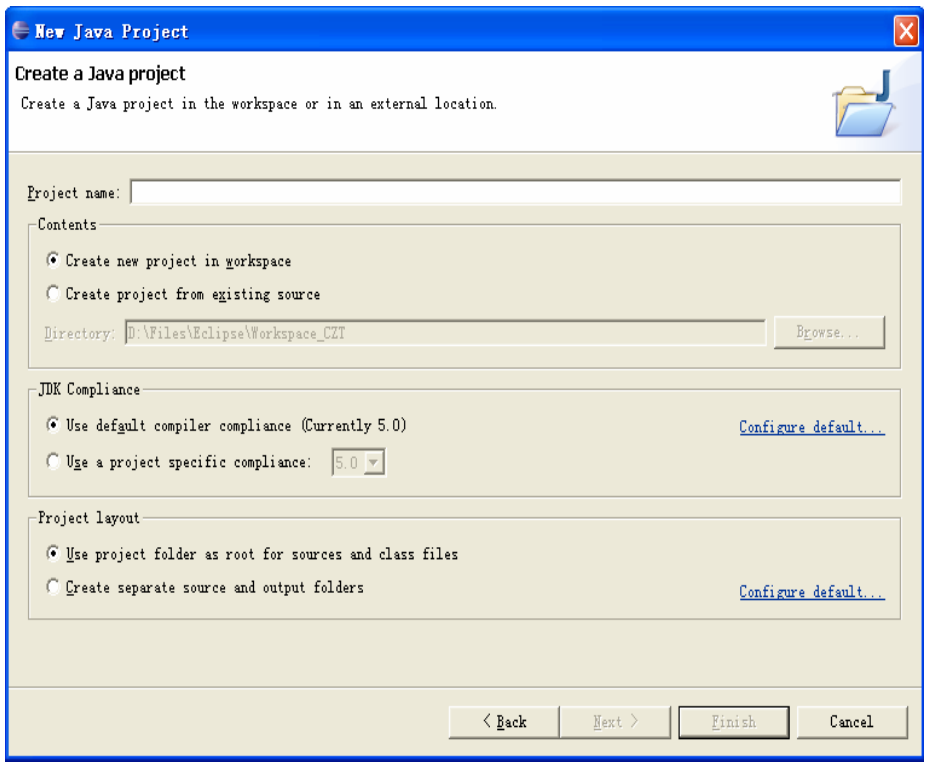


Figure 2.6 – New Java Project Wizard

- **New Java Package Wizard**: This helps create a new folder corresponding to a new Java package in a Java project.
- **New Java Class Wizard**: This helps create a new Java class file in a Java project.
- **New Java Interface Wizard**: This helps create a new Java interface file in a Java project.
- **New Source Folder Wizard**: This helps create a new Java source folder in a Java project.

The JDT plug-in provides a powerful Java Class File Editor (Java Editor in short) with a set of views for viewing and editing Java elements. Some of the features they provide are outlined below.

- **Syntax Coloring**: As seen in Figure 2.7, the text is displayed in

different colors in the Java Editor by following some coloring rules specified by the plug-in. This feature helps the user to more easily understand the contents of the class file.

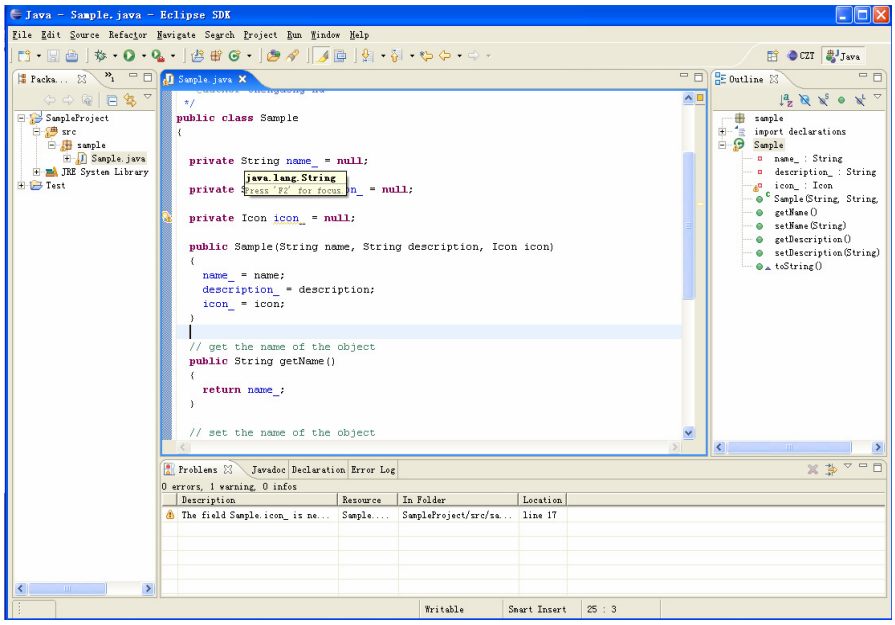


Figure 2.7 – Syntax Coloring

- **Content Assistant**: This helps the user to complete a phrase of text by providing the user with some proposed choices in a popup window upon user request depending on the context information. Figure 2.8 shows that the Java Editor displays some choices to complete the method call "`this.`".

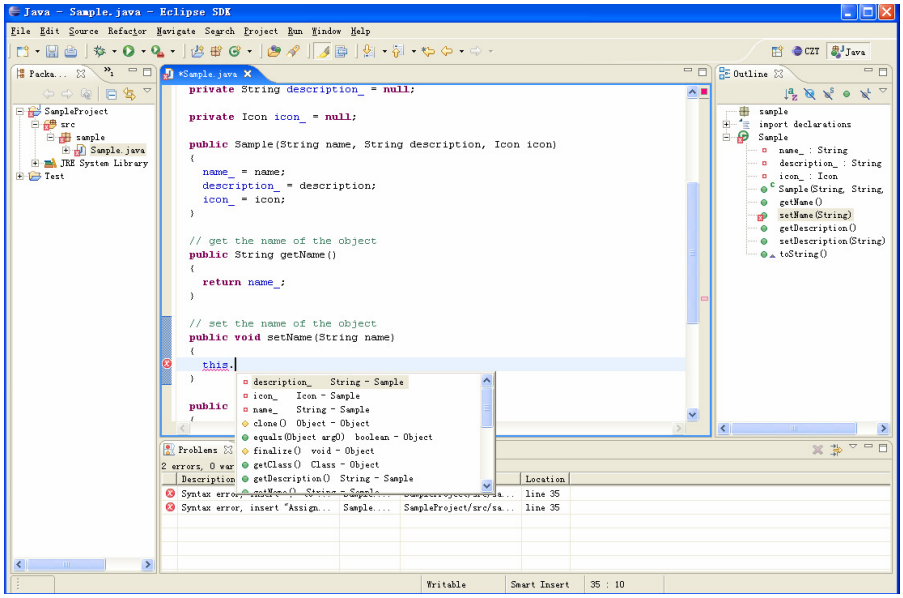


Figure 2.8 – Content Assistant

- **Visual Presentation of Text Annotations**: The Java Editor also provides the support of presenting text annotations in rulers and as squiggles in the text. Figure 2.8 shows that a shaded (blue) vertical bar on the left-hand side of the editor represents the current partition being edited. Also there are a red cross in the left-hand side, a red rectangle in the right-hand side of the editor and squiggles under the text “this”, each of which is showing the syntax errors in the text. This is one of the mechanisms for error reporting in the Java Editor.
- **Folding support**: As described in the last chapter, the Eclipse editor framework supports the folding of some texts in an editor. The Java Editor implements this feature by folding up the declarations of imports, constructors, method, classes and etc. separately. And the editor also shows a shaded (blue) vertical bar on the left-hand side to indicate where the user is editing. These presentations are automatic updated as the user edit text.
- **Hover Support Over Rulers and Text**: Each time the user positions the cursor over the text in the editor, the editor will display some help information to the user about the underlying text, which could

be error annotations, or the type, or something else of the underlying element.

- **Problem Reporting**: In addition to the reporting mechanisms described above, the Java Editor also reports errors to the user in the following ways: displaying a red cross before the filename of the underlying resource, adding entries into some views such as the Problems view, and decorating the element in the Outline view (described later) according to the problem severity.
- **Text Outline View**: As shown on the right-hand side of the editor in Figure above. This view outlines the contents of the resource that the user is editing and thus provides the users with a quick way of grasping the hierarchical structure of the text and navigating around it.
- **Preference handling**: The Java Editor adds a lot of entries to the workbench preferences dialog, through which the user can change the way in which the Java Editor behaves, such as the colors and fonts of each kind of text and whether the contents of the Outline view are automatically updated, etc.

The editor and views also provide a rich set of actions on Java elements. The following are some of the interesting actions.

- **Comment**: These commands help the user to add comments to or to remove comments from a line or block of text.
- **Format:** This set of commands help the user to format some or all of the text in the editor according to the profile specified by the user in the preferences dialog.
- **Organize Imports**: This command helps the user to add/remove the Java import declarations.
- **Open Declaration**: This command leads the user to the declaration

part of the name (variable, type, method, etc.) at the current cursor position.

- **Show References**: This set of commands help the user to find all references of a name (variable, type, method, etc) at the current cursor position. Based on the command the user selects, the user can see all references of the name in the current project, or even the whole Eclipse workspace that the user currently uses.

2.4 TeXlipse Plug-in for Eclipse

"*TeXlipse is a plugin that adds LaTeX support to the Eclipse IDE*" ("Texlipse", 2006).

The TeXlipse plug-in for Eclipse (http://texlipse.sourceforge.net) offers useful support for LaTeX projects. It is a fully featured LaTeX development environment. It is particularly useful for users who already have basic knowledge about LaTeX. It offers users an extremely comprehensive set of features including ("Introduction", 2006):

- **Syntax Highlighting**: Similar to the editing tools described above, the TeXlipse plug-in provides its users with highlighted syntax, so that users can clearly see the syntax of the article being edited.
- **Document Outline**: As the JDT plug-in does, the TeXlipse plug-in provides users a useful outline panel (as shown in Figure 2.9). When the user clicks on an entry in the panel, it will not select the corresponding contents in the editor (as JDT behaves), but will show a shaded (blue) vertical bar on the left-hand side of the editor to indicate the lines that the outline entry corresponds to.

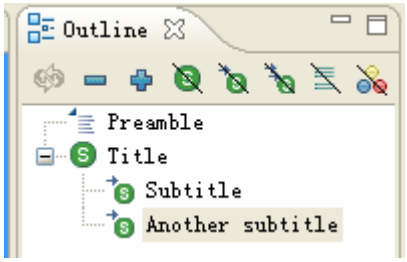


Figure 2.9 – Document Outline

- **Code Commenting**: The editor provides the commands for commenting or un-commenting multiple lines of LaTeX code in the editor.
- **Code Folding Support**: The editor provides this feature as JDT does. The code blocks that can be folded include sections, the preamble and some environments such as figures and tables.
- **Error Reporting**: As JDT does, the editor also reports parsing or building errors to users via in-text annotations, console output and entries in the Problems view.
- **Content Assist**: The editor also provides code completion support when editing. The editor does this by providing a set of templates. The available templates for the current cursor position are determined by the text before the cursor.
- **LaTeX Table View**: The simple table view can make table editing easier. Users have a set of table-specific controls, including inserting and deleting selected rows, moving selected rows up or down, duplicating a row, etc. Figure 2.10 shows a screenshot of a part of the LaTeX Table view.

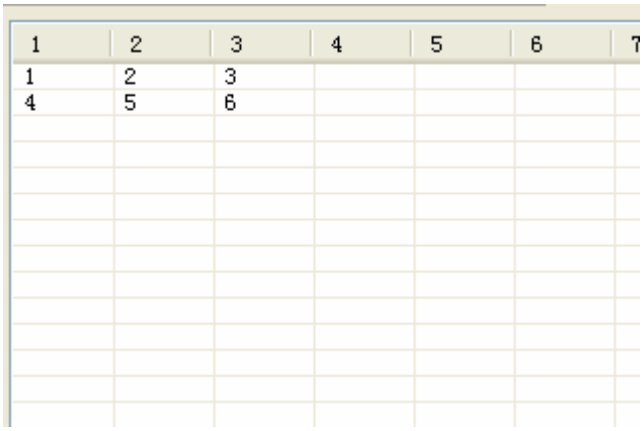


Figure 2.10 – LaTeX Table View

- **New Project Creation Wizard**: Figure 2.11 shows a screenshot of the first page of the wizard.

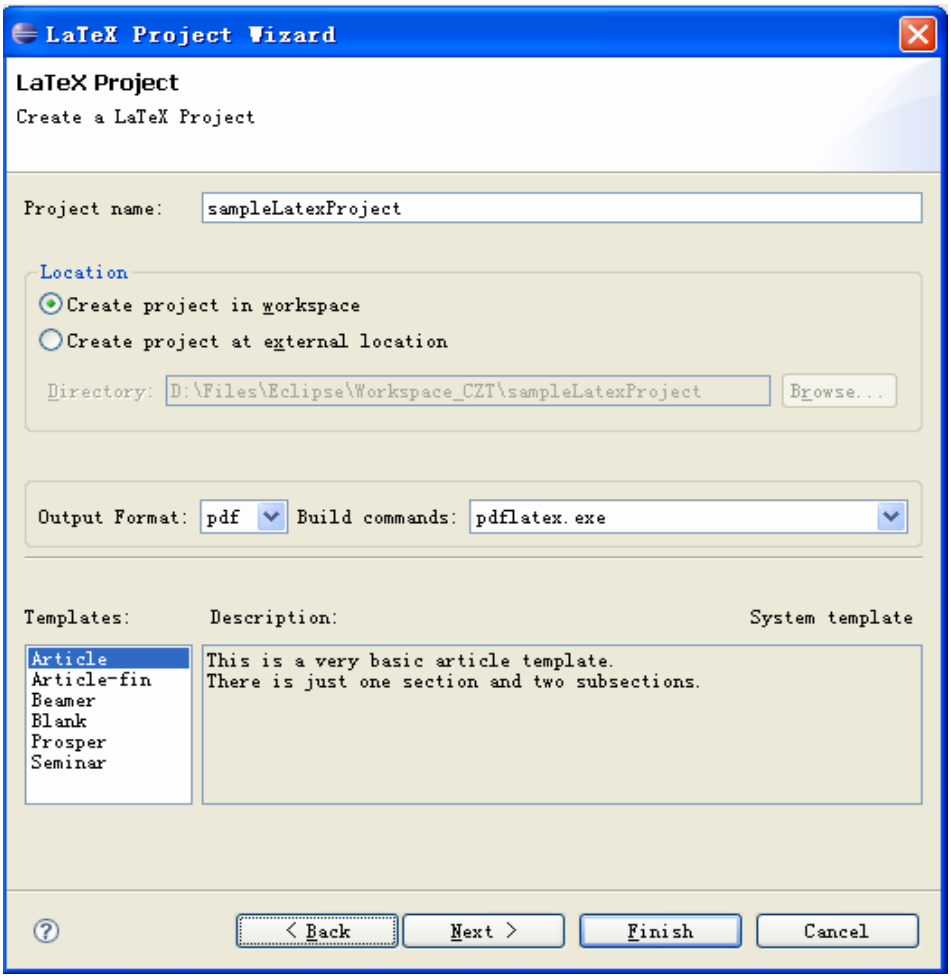


Figure 2.11 – New LaTeX Project Creation Wizard

As Figure 2.11 shows, in addition to setting the name and location, it allows users to set the output format and/or build command. In addition, it also offers users a set of templates for starting up the new project. Users can also add their own templates by setting documents they edit as templates. The second page of the wizard gives users further control over the structure of the project, including setting up the source/output directory, project output file, main .tex file and temporary files directory.

- **Preference handling**: The editor provides users a lot of control over its behaviors in its preference pages. For example, it allows users to specify the colors and fonts for each kind of text, to add new document templates, etc.

Chapter 3 Overview of the CZT Eclipse Editor Plug-in

The CZT Eclipse editor plug-in provides users with a visual environment for browsing and editing Z specifications. It makes use of the Eclipse editor framework to provide users with a visual editor. The main strategy for designing it is to follow the *Eclipse User Interface Guidelines* (Edgar & Haaland & Li & Peter, 2004) for plug-in development and the same conventions that other programming editors use, such as the Java Editor in the JDT plug-in. The CZT Eclipse editor plug-in provides a set of useful mechanisms to help Z users to edit Z specifications, including the following features:

3.1 Features

- **Syntax coloring of Z specifications** (see Section 5.3.1):

  The plug-in provides users a colored view of Z specifications. The Z keywords and comment parts are colored in the editor.

- **Partitioning of Z specifications and highlighting of currently browsed partition** (see Section 5.3.2):

  For ease of browsing, a Z specification is partitioned into a set of partitions or Z paragraphs according to its semantic meaning. In addition, the CZT Editor highlights the partition that the cursor is currently positioned in, so that the user can easily see which partition is being browsed at that moment.

- **Folding of partitions** (see Section 5.3.3)

  Each separate partition can be folded (hidden) for ease of browsing

when the user is not concerned with it.

- **Bracket matching** (see Section 5.3.4)

  When the cursor is positioned after a bracket, the semantically matching opening or closing bracket is annotated in the editor.

- **Selection by double-clicking** (see Section 5.3.5)

  By double-clicking in the editor a Z syntactical word containing the position of the cursor will be selected in the editor. This allows users to easily select a word and do further operations such as cutting and pasting it somewhere else.

- **Highlighting of all references of a variable** (see Section 5.3.6)

  When the cursor is positioned on a Z name or its reference, all references of the variable, including itself, will be highlighted in the editor.

- **Jumping to the declaration of variables**(see Section 5.3.7)

  When the cursor is positioned on a variable or its reference, users can execute a command to quickly browse to the declaration part of the variable.

- **Highlighting the enclosing element and restoring last highlight**(see Section 5.3.8)

  Users can execute a command to highlight the text associated with the smallest term at the current cursor position by drawing a box around it or changing its background depending on the user's setting in the preference page. Repeatedly executing the command will make the highlighted area larger. This allows the structure of the Z

terms to be explored and thus helps the user to understand the structure of the Z specification. Another command does the inverse of this, so that the user can change to highlight the previous smaller term.

- **Problem reporting** (see Section 5.3.9)

  During editing of a Z specification, it is constantly parsed in the background without interrupting the editing. When problems, such as errors or warnings, are found, they are immediately reported to the user in a variety of ways.

- **Hover support** (see Section 5.3.10)

  When the mouse is hovered above a Z construct, the editor will display some information about that construct in a floating box next to the mouse position.

- **Viewing current specification in different formats** (see Section 5.3.11)

  A specification can be browsed using another markup or encoding in a view, so that users can compare different markups and/or encodings of the specification.

It also provides some Z-specific views to help browse and/or edit Z specifications. Here is a brief introduction of two of those useful views.

- **Z Character Map**: This provides users with a panel that displays the special characters used in Z specifications. Some characters are Z-specific. The view also provides users a convenient way of inserting Z characters into the active editor (see Chapter 4).
- **Z Conversion View**: This provides users with a separate panel for

viewing the current specification in another markup and/or encoding. For example, if a specification is edited in LaTeX markup in an editor, it can be viewed in Unicode markup in this panel. (see Chapter 9)

Figure 3.1 illustrates some features of the CZT plug-in when a Z specification is browsed or edited. The details of the CZT plug-in will be described in later chapters.

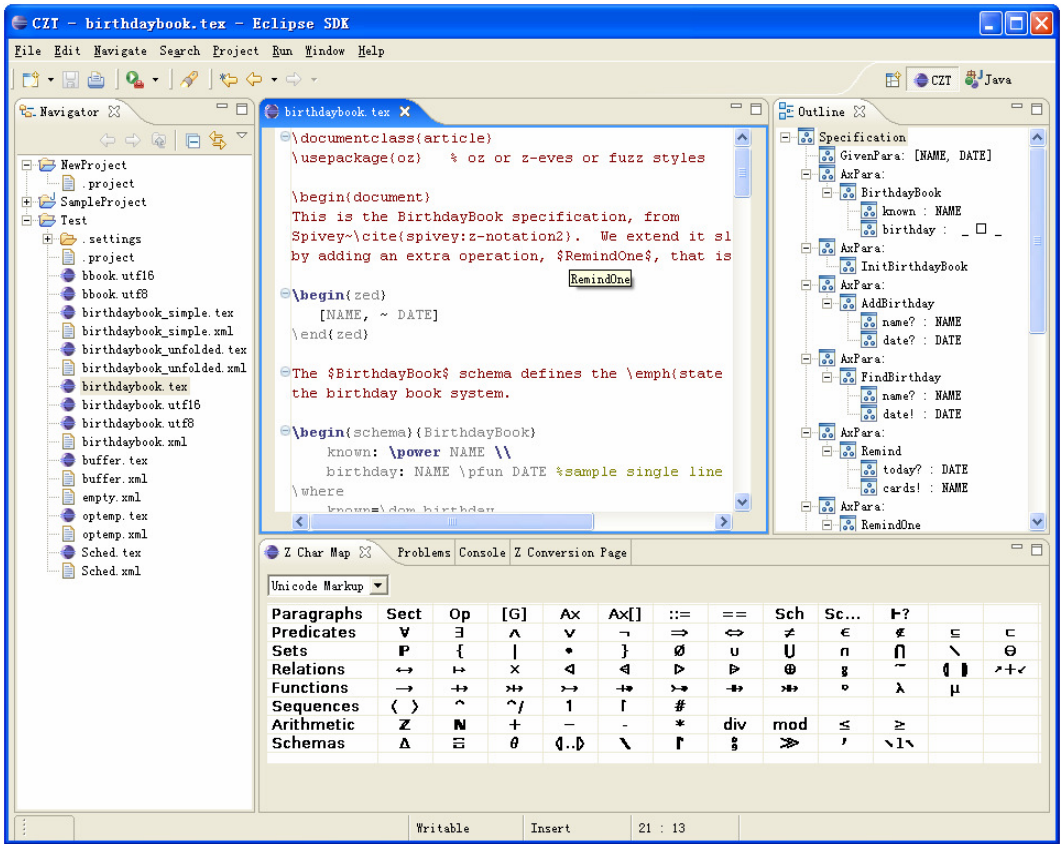


Figure 3.1 – Some features of the CZT plug-in

The plug-in also provides users with wizards to facilitate the creation of new CZT projects and Z specifications.

As the JDT plug-in does, the CZT plug-in also provides a convenient Eclipse perspective.

3.2 Implementation Techniques

The Eclipse Plug-in development environment (**PDE**) provides a special wizard for facilitating the creation of a plug-in development project with several mouse clicks. This section briefly discusses some of the settings that were chosen when the wizard was used and how these affect the users of the plug-in. The goal of discussing these settings is firstly to give enough information so that the CZT Eclipse plug-in can be recreated in future versions of Eclipse, if necessary, and secondly to explain how some of the design choices affect users of the plug-in.

The wizard can be accessed by selecting the menu "***File/New/Project…***" and then selecting "***Plug-in Project***" from the list showing, followed by the "***Next***" button. Then the first page of the wizard is shown as Figure 3.2.

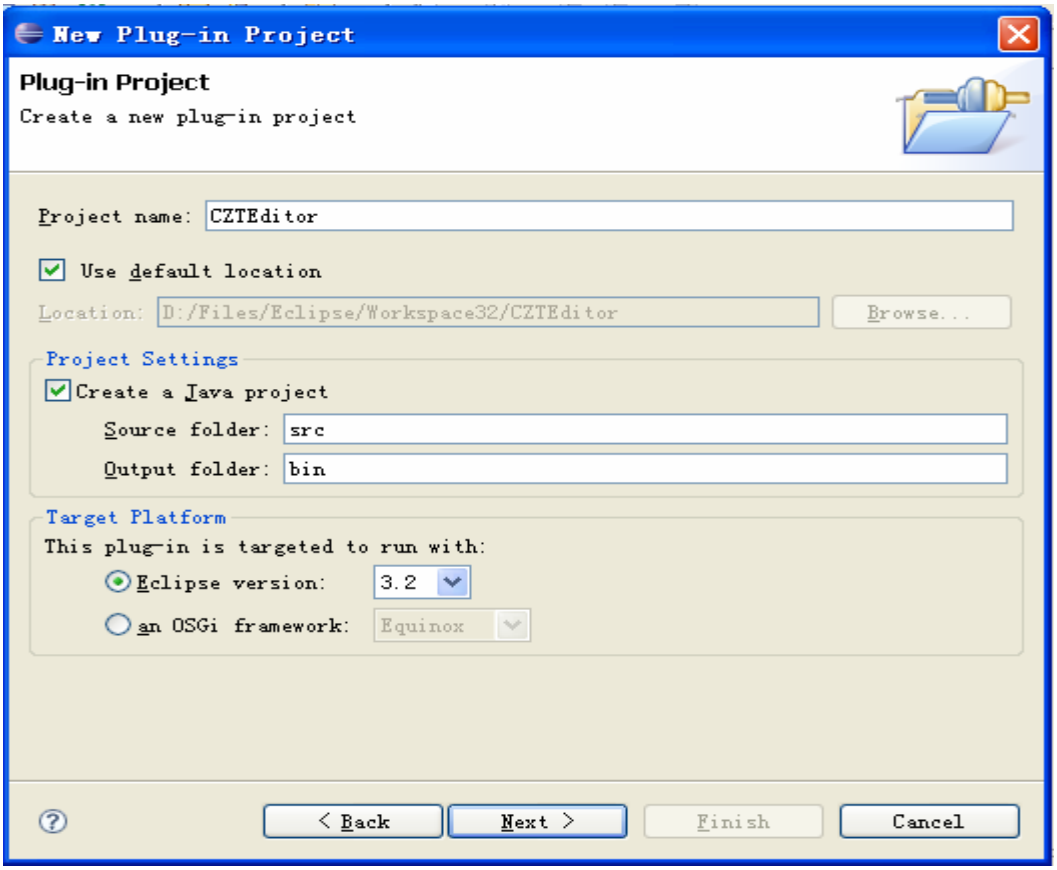


Figure 3.2 – The First Page of New Plug-in Project Wizard

In this page, the name of the new project, `CZTEditor`, was entered into the "Project name" field. This name is not visible to users, but is the name of the project used during the development. Here the default location was used. As the `CZTEditor` project definitely needed to create a Java project, the "Create a Java project" option was ticked and the default settings of source and output folders were used. As new tools are usually developed using the latest techniques so that they will not be left behind, the targeted Eclipse platform version to run the plug-in was set to the latest version, `3.2` in this case. This ensures that users of the plug-in are using Eclipse version 3.2 or higher, which avoids bugs due to incompatibility with older versions of Eclipse.

The next page of the wizard (see Figure 3.3) provides fields for generating the plug-in manifest.

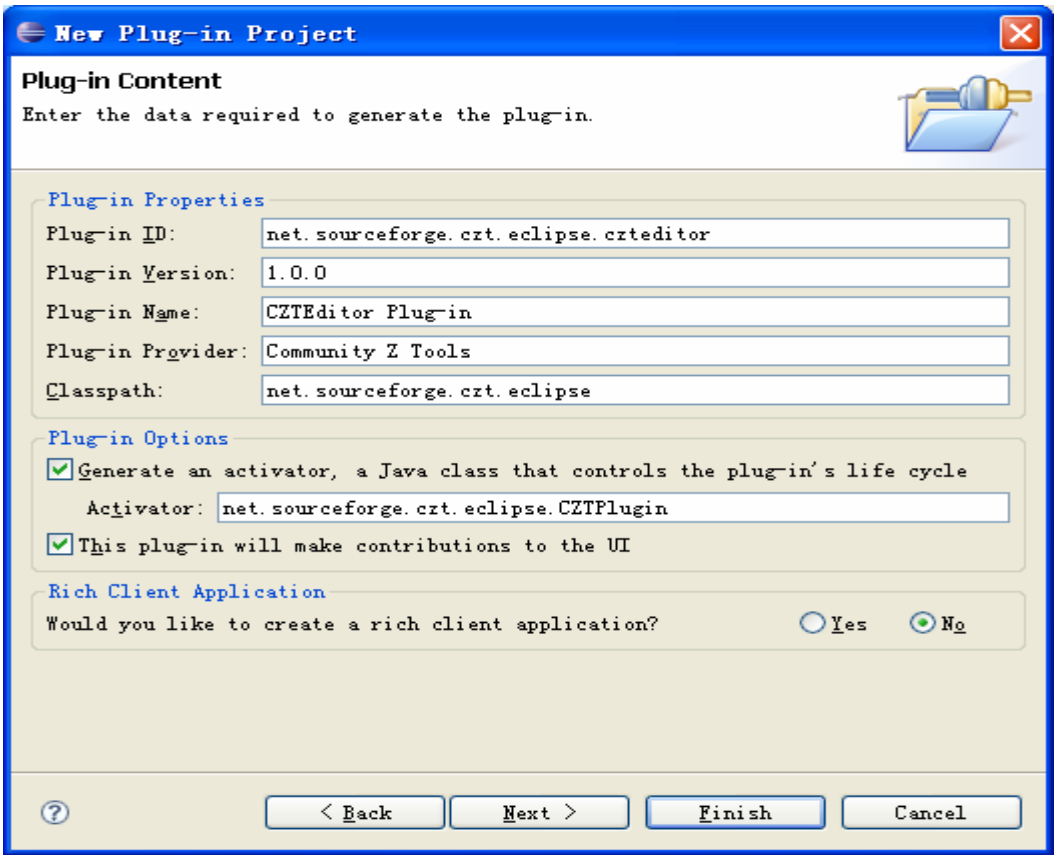


Figure 3.3 – The Second Page of new Plug-in Project Wizard

The description of some important fields in this dialog is as follows:

- Plug-in ID: the ID of this new plug-in. It was set to `net.sourceforge.czt.eclipse.czteditor` in this case. The ID is used by Eclipse to identify the plug-in and thus must be unique. The ID will also be shown to users in the plug-in list in the **Plug-ins** view.
- Plug-in Version: this is the version number of the plug-in as seen by the user. For the first version, the default number "`1.0.0`" was used. This should be incremented for each release of the plug-in.
- Plug-in Name: the name of the plug-in. The CZT Editor plug-in's name was set to `CZTEditor`.
- Plug-in Provider: (Optional) the provider of the plug-in
- Classpath: (Optional) the path to which where Java classes can be found at compile-time. As the CZT plug-in uses Java classes, so the classpath must be specified. The run-time classpath was specified later in the plug-in manifest file.
- Activator: as described in the dialog, this is the Java class that controls the plug-in's life cycle. Typically, this class describes how the plug-in behaves during its life cycle. Since the CZT plug-in uses Java classes, the activator must be specified (as shown in Figure 3.3).
- As the plug-in will create an Eclipse UI, the corresponding option "This plug-in will make contribution to the UI" was checked.

The next wizard page shows some available templates. It allows the plug-in developer to create the plug-in project with a simple template. If a template is selected to be used by the new project, the next wizard page displays the setting information for the template if applicable. As the CZT Editor project creates an editor, the editor template was chosen here. The templates give

plug-in developers a good start to create a new project, particularly when they are not familiar with it. Once the setting in this page is done, the new plug-in project is created and a set of files and folders are created at the same time. If a template is used, a plug-in manifest file named "`plugin.xml`" is created and it contains the information about how the plug-in contributes to Eclipse. Then the plug-in manifest editor is automatically opened and comes to the front (see Figure 3.4).

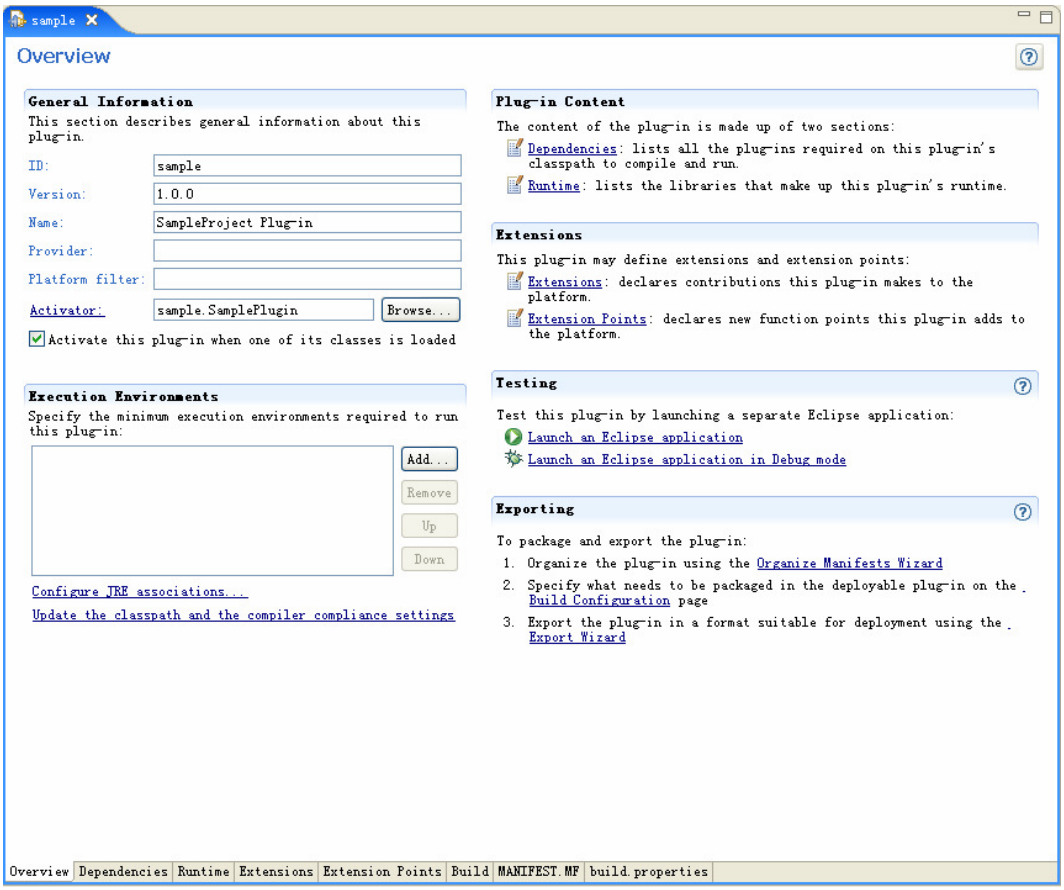


Figure 3.4 – The Plug-in Manifest Editor

The plug-in settings manifest is stored in several files, including the main XML file named "`plugin.xml`", and their names can not be changed by users. The manifest editor offers users an easy way of changing the settings for the plug-in. Some setting pages will be discussed in later chapters if necessary.

Chapter 4 The Z Character Map View

The Z Character Map view (**ZCharMap** in short) is a utility view. It provides a palette of commonly used Z characters and constructs. Its most important function is to provide users with a convenient way for inserting Z characters or constructs in the currently active text editor in the current workbench window. As talked in Chapter 1 regarding the reasons for developing the CZT plug-in, it is hard to insert the Unicode characters (particularly the Z-specific characters) in most editors, so the CZT plug-in solves the issue by providing a simple click-to-insert feature in the Z Character Map view. This view also provides useful documentation about each Z character. Figure 4.1 shows a screenshot of the Z Character Map view.

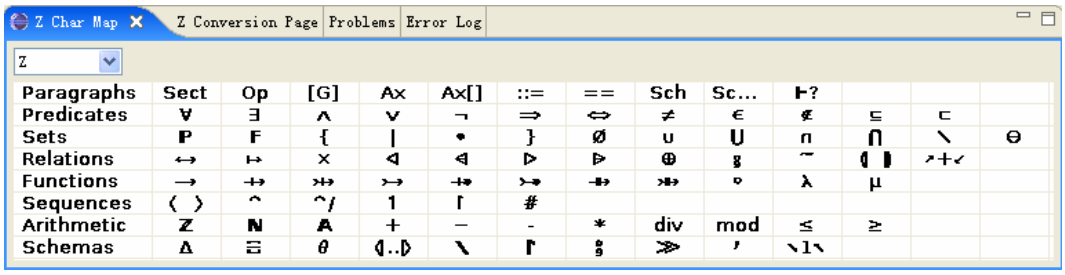


Figure 4.1 – Z Character Map View

4.1 Categorized Characters and Hover Support

As shown in the above figure, the Z characters and constructs are categorized into a list of categories such as paragraphs. The layout of Z characters helps users to easily find the characters they are interested in. For each row in the character table, the first column describes the category of the Z characters in the remaining columns of the row. All Z characters are displayed in Unicode format in the table, but a short description of a particular Z character is displayed when the mouse is hovered above the character for more than

half second. Figure 4.2 shows the sample behavior when the mouse is hovered above the Z character in the third column of the second row.

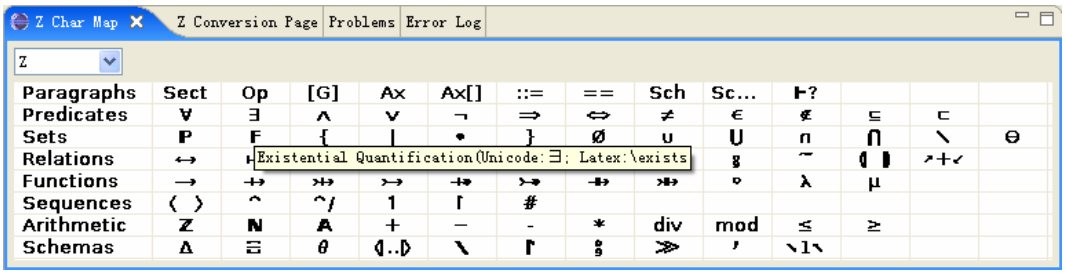


Figure 4.2 – Hover in Z Character Map View

It shows the description of the Z character (an existential quantifier in this case) and also its representation in Unicode and LaTeX format.

4.2 Actions

The Z Character Map view provides a smart solution for inserting Z characters at some point in a buffer in an active text editor by clicking on a particular Z character in the character table. The reason for saying it is smart is that the actual text inserted depends on the active editor, by following the logic below:

a) If the active editor is not a CZT Editor, the description of the selected Z character is inserted in the editor.

b) If the specification is in LaTeX markup (the name of the specification file has the extension "*.tex*"), then the actual text inserted in the active editor is the LaTeX representation of the selected Z character.

c) If the specification is in Unicode markup (the name of the specification file has the extension of either "*.utf8*" or "*utf16*"), then the Unicode representation of the selected Z character is inserted into the active editor.

With this context-sensitive insertion, users can easily insert a Z character in a

suitable format without needing to set a markup mode, look up the name of a LaTeX macro in a book or look up a Unicode character in the Z standard. The behavior is particularly useful for inserting a Z character in Unicode format, because this is very difficult without the Z Character Map.

For expert use, the Z Character Map also provides another way of inserting Z characters. The user can use the arrow keys on a computer's keyboard to select a character and then press the "***Enter***" key to insert it in a proper format into the currently active editor.

4.3 Implementation Techniques

To contribute a view to Eclipse, a simple way is to create it by selecting the view template when the plug-in is being created. In the CZT case, the plug-in was already created, so four steps were involved.

1. An Eclipse view extension point was added to the extension list in the "**Extensions**" page.. This was done in the "Extensions" page of the manifest editor as shown in Figure 3.4. This editor is opened by opening the file "`plugin.xml`" or "`META-INF/MANIFEST.MF`". Clicking on the tab "Extensions" at the bottom of the editor opens the "Extensions" page. A view extension point was added by clicking on the "**Add…**" button, selecting the extension point `org.eclipse.ui.views` from the extension point list and then clicking on the "**Finish**" button.
2. A new category for CZT views was declared by right-clicking on the new entry and selecting the menu "**New/category**". The id, `net.sourceforge.czt.eclipse`, and the name, `CZT`, of the category were supplied to their corresponding fields in the property panel on the right. The name is displayed in the list of contributed views, which is displayed by using the menu entry "Window/Show

View/Other…".

3. Then, a view was added in the similar way as adding a category. If the new view entry is selected in the extension list, the right-hand side of the page shows the initial settings for the new view. In the CZT case, the id of the view was set to `net.sourceforge.czt.eclipse.views.ZCharMapView`, the name was set to `ZCharMap` and the category id was set to `net.sourceforge.czt.eclipse`. All the CZT views are given this same category id, so that they are grouped together in Eclipse menus and lists (for example, the "Window/Show View/Others" list), which makes it easier for users to find the CZT views. Another compulsory field is the ***class*** field. The name of the class needs to be prefixed by its package name. An easy way to specify the class for the view is to click on the link "class*" beside the ***class*** field, which pops out a new Java class wizard directly. In the wizard, the name of the class was provided and its super class was also provided as all views must implement the `org.eclipse.ui.IViewPart` interface.
4. Finally, for simplifying the view creation, the code defining the view was created by subclassing an abstract class named `org.eclipse.ui.part.ViewPart`, which implements the required interface.

As the ZCharMap view uses a table to display the characters, it uses a Table Viewer (`org.eclipse.jface.viewers.TableViewer`) as its base control. As users can only select a whole row in the Table Viewer, users will not see the exact cell they select. A Table Cursor (`org.eclipse.swt.custom.TableCursor`) played this role in a `TableViewer` by creating it with the table used in `TableViewer`.

Another issue for the ZCharMap view is to determine what to insert into the editor. When users select a character in the table to insert, it checks the type of the editor and then inserts the proper format of text associated with the character based on the strategy described earlier in this chapter.

4.4 Design Issues and Alternatives

With the Z Character Map view, the user can insert a Z character by clicking on it. An alternative style would be to make the view insert a character by double-clicking on it. As the map is only used to insert a character, there is no need to make single-click to do something else. To save the user's time when editing a Z specification, it is better to make single-click to play the inserting role so that it can save the user's time for inserting characters.

Another issue involved was how to display the information about a character. Two other ways have been considered. One is to display the information on a label in the view. The disadvantage is that it will consume some space of the view. Another way is to display it in the status bar at the bottom of the window. However, the status bar is shared by the editor, so it is a good practice to leave it to the editor as most tools do. A major disadvantage of these two alternatives is that they require more screen space. It is desirable to minimize the space required to display all the Z characters, so that the user has as much space as possible for the Z specification being edited. To display it in a floating panel as the Z Character Map uses can not only save space, but also allow multiple lines of information to be displayed.

4.5 Future Implementation

In the future, it would be better to find a more suitable way of displaying the information about a character. The information normally contains the following information: a label, a description, its LaTeX format and its Unicode format. It would better to display it in a format like the popular

HTML list, so that users can easily find the useful information they want. Particularly, if the information panel could be scrollable, the view could provide users with more useful information about a character.

Chapter 5 The CZT Editor

The CZT Editor is the most important part of the CZT plug-in. Like other Eclipse editor plug-ins, the CZT Editor provides users with a visual representation of Z specifications. Additionally, it provides utility commands for some common actions needed by Z users.

5.1 Design Issues and Alternatives for the Editor

In Eclipse, an editor is associated with particular filename extensions. A Z specification file may have one of the following extensions:

- .tex: for specifications using LaTeX markup
- .utf8: for specifications using Unicode markup and UTF-8 encoding
- .utf16: for specifications using Unicode markup and UTF-16 encoding

A simple way of supporting multiple extensions would be to develop a separate editor for each extension. In that case, the editors could be developed easily. However, many similar things will be done repeatedly and thus it is not a good design and it would make future maintenance more difficult.

Therefore, a single Z editor is developed and is associated with all of the above extensions. The editor can automatically use the proper markup or encoding to browse or edit a Z specification without concerning the user.

5.2 Implementation Techniques for the Editor Creation

One simple way to create an editor is by selecting an editor template when the

plug-in is being created. Then a sample editor will be created with some features that were specified at that moment. After that, the editor can be changed to suit specific needs. Since the CZT editor plug-in (see Chapter 3) was already created, the following three steps were followed to create the Z editor.

1. Define the editor extension in the plug-in manifest file.

   On the "Extensions" page of the manifest editor, the "**Add…**" button opens a window containing a list of extension points. To define the CZT Editor, the `org.eclipse.ui.editors` extension point was selected (see Figure 5.1).

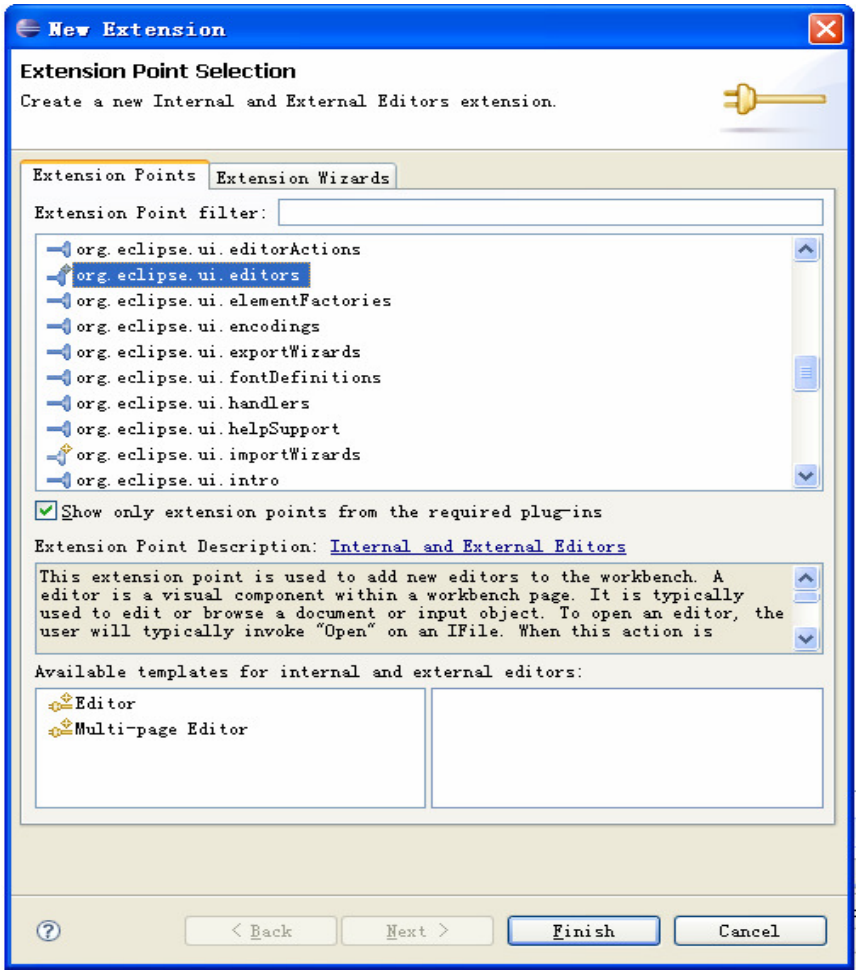


Figure 5.1 – New Extension Page

Available templates for the new editor are also listed here to be selected. For the CZT Editor, no editor template was chosen to get a clean start of creating the editor. After the "**Finish**" button was clicked, the new extension point was added to the plug-in's extension list. Then a new editor extension was added by right-clicking on the added extension point and selecting "**New/editor**". A new panel being displayed on the right shows the details about the new editor extension. The fields are described below:

id – the unique identifier of the new editor.

name – the name of the new editor. This name is displayed in the editor list, which is opened by right-clicking on a file, whose name extension is supplied in the "extensions" field below, and selecting "Open With" menu entry. This name can also be seen in the "**General/Editors/File Associations**" preference page.

icon – a path (relative to the plug-in's directory) for the image displayed at the top-left corner of the editor and the window. It is also used for decorating the entry of the file in the **Navigator** view by displaying the icon before the name of the file.

extensions – this field is for specifying the list of comma-separated file extensions to be understood by the editor.

class – this is the name of the Java class that defines the editor. This class must implement the `org.eclipse.ui.IEditorPart` Java interface.

contributorClass – this is the name of the Java class that adds new actions to the workbench menu and toolbar. This class must implement the Java interface

`org.eclipse.ui.IEditorActionBarContributor`.

2. Develop the Java code that implements the editor

   The code can be created by implementing the Java interface *`org.eclipse.ui.IEditorPart`*. However, rather than implementing this interface directly, the CZT Editor subclassed the *`org.eclipse.ui.editors.text.TextEditor`* class, which some basic features which are needed by most text-based editors and also implements the *`IEditorPart`* interface.

3. If the "contributorClass" field is supplied, the corresponding Java code must be created too. The CZT Editor implemented this class so that some actions were added into the editor's menu.

5.3 Features

Currently, the CZT Editor implements the following features. The rest of this chapter describes these features in more detail.

- Syntax coloring (see Section 5.3.1)
- Highlighting of current partition (see Section 5.3.2)
- Folding of partitions (see Section 5.3.3)
- Bracket matching (see Section 5.3.4)
- Selection by double-clicking (see Section 5.3.5)
- Marking references of a Z name(see Section 5.3.6)
- Jumping to the declaration of variables(see Section 5.3.7)
- Highlighting the enclosing element(see Section 5.3.8)
- Problem markers (see Section 5.3.9)
- Hover support (see Section 5.3.10)
- Viewing specifications in alternative formats (see Section 5.3.11)

5.3.1 Syntax Coloring

Like most visual text editors for programming languages, the CZT Editor provides syntax coloring support. The content of a whole specification is partitioned into paragraphs each of which is syntactically a Z paragraph. For a Z narrative paragraph, its whole content is colored with a specific color. For other types of paragraphs, the Z-specific keywords, operators, narrative texts or comments can be colored differently to meet the users' preferences so that the users can easily view the contents of a Z specification and thus understand the structure of it. Figure 5.2 shows a screenshot of the visual representation of a sample Z specification.

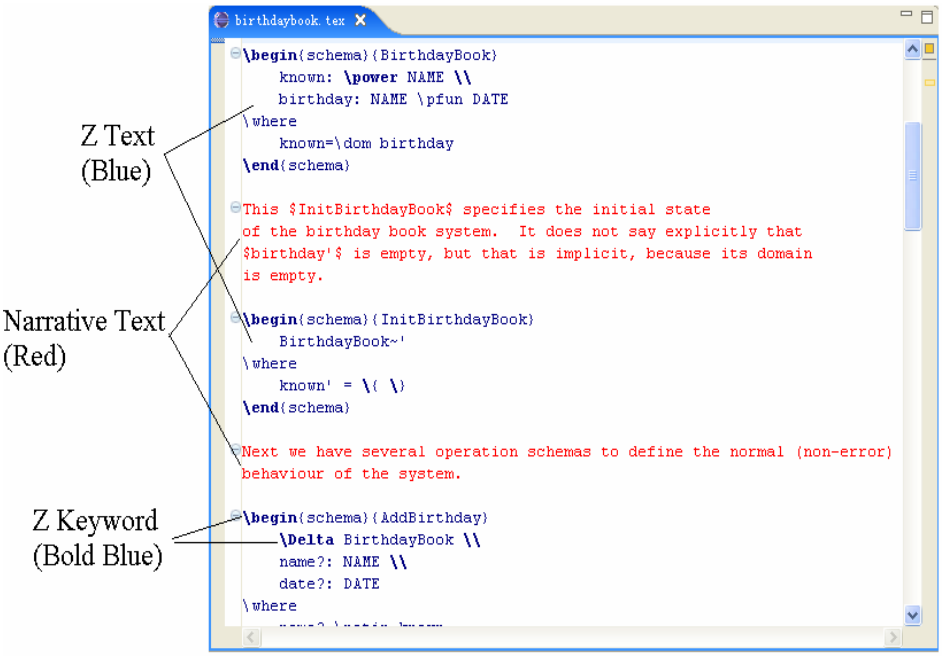


Figure 5.2 – Syntax Coloring Support

5.3.1.1 Design Issues and Alternatives

As other programming editors do, the CZT Editor can simply specify each specific color to each set of Z-specific keywords, operators, narrative paragraphs or comments. However, the **ISO Z standard** (ISO/IEC 13568, 2002) specifies different sets of Z-specific rules to each markup of specifications. Due to this issue, no general set of

rules can be applied to the CZT Editor properly.

One simple way to solve this would be to develop a set of editors and then apply a set of rules to each of the editors respectively. However, as discussed earlier, this is not a good design.

Another way to solve this is to develop a single editor that applies different sets of rules to different markups of specifications. A good reason for doing this is that different markups of specifications use different file extensions so it would be easy to determine which set of rules is to be applied to the specification when the specification file is opened in the editor.

5.3.1.2 Implementation Techniques

Eclipse provides the API for adding syntax coloring support to the editor. It uses the "Damage – Repair – Reconciling" model. Each time the user changes the document in the editor, the presentation reconciler will calculate the region of text to be displayed (known as "*Damage*") and how to redisplay it (known as "*Repair*"). Damage means the text to be redisplayed. This process is called "*Reconciling*" (Ho, 2003).

To add this support to the editor, the editor needs to be configured to use a customized `SourceViewerConfiguration` class. This class is responsible for a lot of behaviors, such as syntax highlighting, hover support, etc. More configurations inside the class will be introduced in later sections or chapters.

The method in the *SourceViewerConfiguration* class for syntax coloring support is called `getPresentationReconciler()`. This must return a presentation reconciler that implements the

`org.eclipse.jface.text.presentation.IPresentationReconciler` Java interface. The CZT Editor could provide a subclass that implements this interface. However, it was found to be simpler to use an instance of the `org.eclipse.jface.text.presentation.PresentationReconciler` class instead, with some configuration to give it the desired behavior.

Each type of content in the editor document must be configured with a pair of presentation reconcilers. One of them is responsible for providing the *damage* rules for detecting the text to be redisplayed, while the other is responsible for providing the repair rules for how to redisplay the area detected by the former reconciler. Both of them need to be configured with a `RuleBasedScanner` object. These scanners are responsible for defining the set of syntax coloring rules for a type of content.

5.3.1.3 Future Implementation

In the future, it would be better to make the editor support the syntax coloring for the elements dynamically generated by the parser, such as the variables and user-defined operators.

5.3.2 Highlighting of Current Partition

When browsing or editing a Z specification, the user can see which paragraph is being browsed or edited according to the current cursor position. Figure 5.3 shows a screenshot of the behavior.

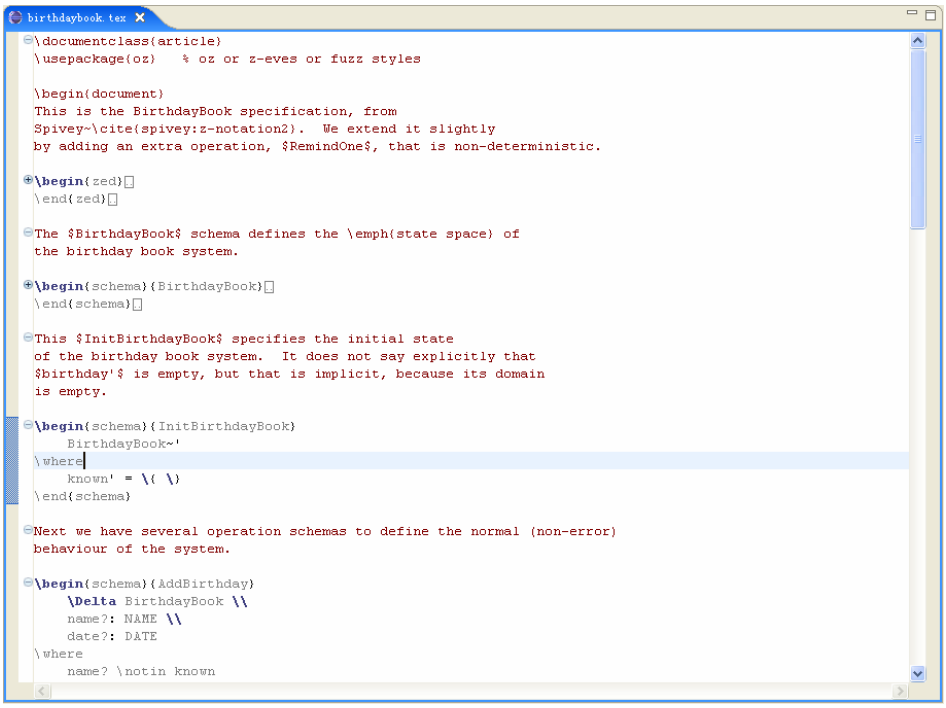


Figure 5.3 – Highlighting of Partitions

As seen in Figure 5.3, the line the cursor is positioned in is highlighted (it is colored in light blue). The shaded (dark blue) vertical bar on the left-hand side of the editor indicates the extent of the current paragraph being edited or browsed. In this way, the user can easily see where the paragraph starts or ends and whether it starts or ends as expected.

5.3.2.1 Design Issues and Alternatives

Although the syntax coloring support displays the narrative paragraphs and the Z paragraphs differently, two adjacent Z paragraphs will have the same color, so may appear to be a single paragraph. This could lead to confusion when performing operations on the current paragraph. Highlighting the current partition with the vertical bar solves this problem by directly indicating to the user which Z paragraph is being edited. An alternative approach was also considered during the design of the editor, which was to set a different color for the background of the

currently editing paragraph. This approach was adopted by the CZT jEdit plug-in. In theory, the user can figure out the editing paragraph through its background. However, doing so will give the editor too many colors and thus make the syntax coloring less clear to the user. Another disadvantage is that the background color may make it difficult for the user to see the cursor.

5.3.2.2 Implementation Techniques

Each time the cursor moves around the editor, the editor sets the range of the paragraph that the cursor is in by using the method `org.eclipse.ui.texteditor.AbstractTextEditor.setHighlightRange(int offset, int length, boolean movecursor)`. The first parameter is the start position of the editing paragraph and the second is the length of the paragraph. The third one is to decide whether to move the cursor to the start of the paragraph. In the CZT case, the cursor does not need to jump to the start, so this parameter was set to `false`. Each time the method is called by the editor, the lines that the paragraph covers are indicated by the vertical bar on the left-hand side of the editor.

5.3.3 Folding of Partitions

Figure 5.3 also shows a "-" sign or a "+" sign on the left-hand side of the starting point of a paragraph. The "+" sign indicates that the paragraph starting there is currently collapsed and can be expanded by clicking on the sign. Reversely, the "-" sign indicates that the paragraph starting there is currently expanded and can be collapsed by clicking on the sign. Since Z specifications usually contain alternating paragraphs of English and formal Z, this is a useful way of being able to hide the English while working on Z, or vice versa. This function is also very useful when the

user is editing a big specification containing a lot of paragraphs. The user can focus on the few paragraphs that relate to the paragraph being edited and can collapse all other paragraphs.

5.3.3.1 Design Issues

When the specification gets larger, it becomes more difficult to see the overall structure of the specification. To get a clear view or context information about a particular point, the user needs to know the structure of other parts. The folding support enables the user to collapse the paragraphs the user is not concerned about and thus give the user a brief overview of the collapsed paragraphs and details about the expanded paragraphs. This also enables the user to focus on an interesting subset of the paragraphs in the specification, by hiding the others.

5.3.3.2 Implementation Techniques

The folding behavior is referred to as a special kind of annotation in the editor. It is called a "*Projection Annotation*". The implementation of supporting folding involved the following three steps (Deva, 2005):

The CZT Editor simply creates an instance of this class in the `createControl` method so that the painter is created when the editor is created

1. In the editor class, the `createPartControl` method was overridden so as to configure the editor to install an instance of `org.eclipse.jface.text.source.projection.ProjectionSupport`.
2. In the editor class, the createSourceViewer method was

overridden so as to return a return an instance of `org.eclipse.jface.text.source.projection.ProjectionViewer`.

3. Each time the Z specification is parsed, the editor calculates the positions of all Z paragraphs and creates a `org.eclipse.jface.text.source.projection.ProjectionAnnotation` for each of them. After that, the editor removes all old projection annotations from its `ProjectionViewer` and puts the new ones in via the `ProjectionViewer`'s `ProjectionAnnotationModel`. This causes each paragraph to be decorated with the folding signs "+".

5.3.4 Bracket Matching

For programming languages, brackets are often matched in pairs. Thus, programmers would usually like to know which bracket a particular bracket matches syntactically so that they can easily check whether the brackets are matched correctly and thus correct them quickly if necessary. In most programming languages, a bracket is a single character. However, the LaTeX format of the Z language specifies some brackets which are composed of multiple characters. By following the convention adopted by most other programming editors, the CZT Editor marks the matched bracket that is a single character in the same way as other editors do when the cursor is positioned immediately after a bracket that is a single character. Figure 5.4 shows an example of the behavior.

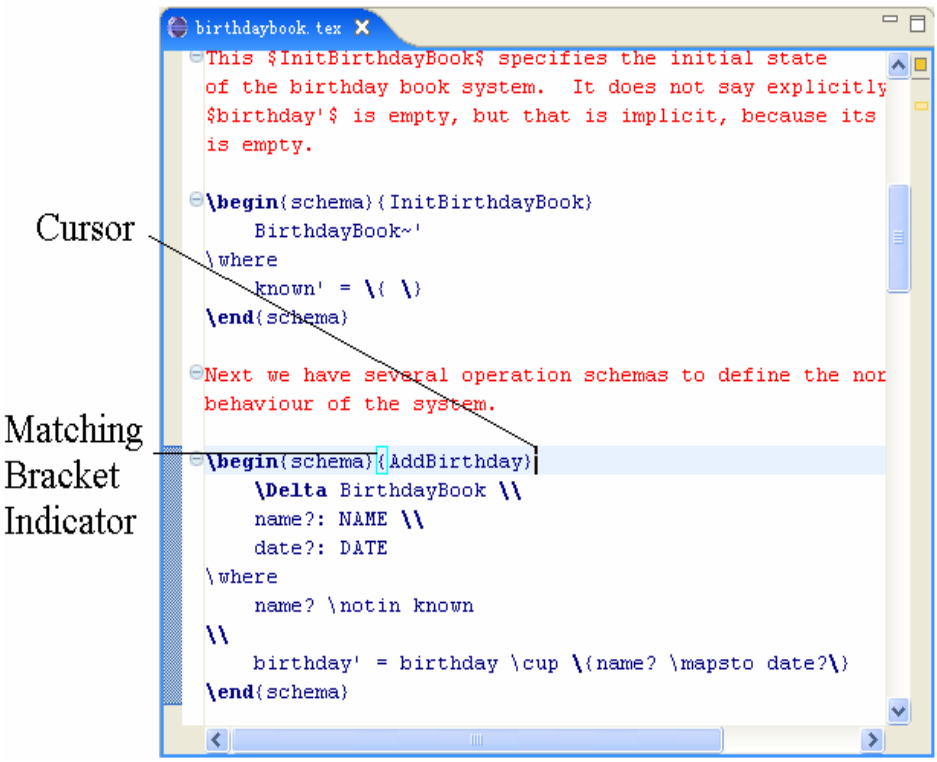


Figure 5.4 – Single-Character Bracket Matching

As shown Figure 5.4, the rectangle before "`AddBirthday`" indicates that the character it contains matches the character before the position of the cursor. For a bracket containing multiple characters, the CZT Editor marks the first character of the matching opening bracket when the cursor is positioned immediately after the last character of a closing bracket. Similarly, the CZT Editor marks the last character of the matching closing bracket when the cursor is positioned immediately after the last character of an opening bracket. The behavior is shown in Figure 5.5.

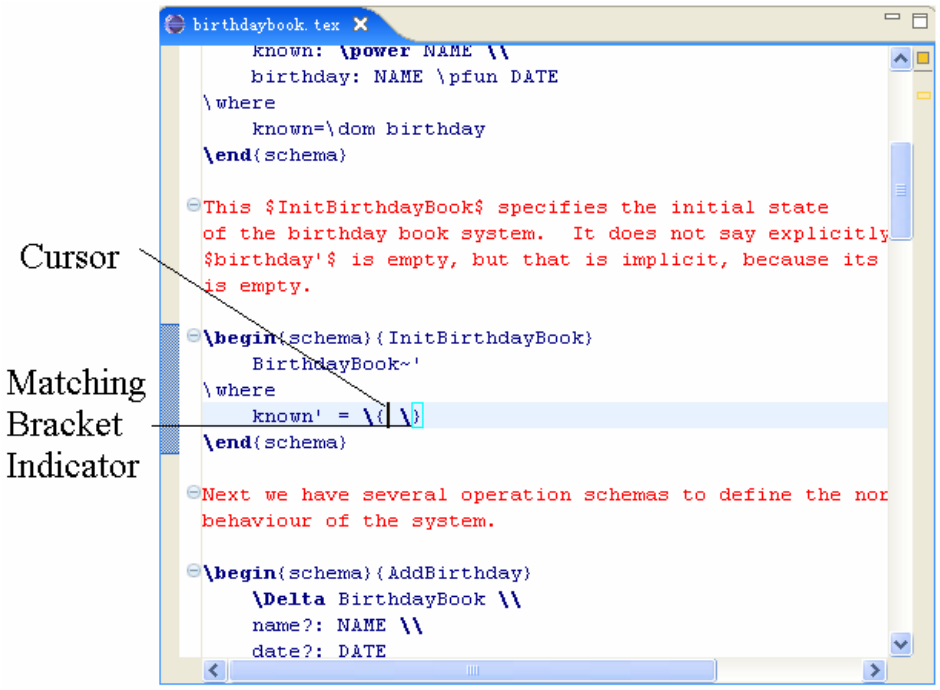


Figure 5.5 – Multiple-Character Bracket Matching

Figure 5.5 shows that the CZT Editor marks the last character, "`}`", of the matching Z-specific closing bracket, "`\}`", when the cursor is positioned immediately after the last character, "`{`", of the Z-specific opening bracket, "`\{`".

5.3.4.1 Design Issues and Alternatives

As the `TextEditor` class provides developers with an interface for adding custom implementation of the mechanism for bracket matching, it makes the work easier. However, it can only mark a single character at once. For Z brackets that are made up of multiple characters, it would be preferable to mark all of a matching bracket rather than just its first or last character. Unfortunately, this is not supported by the standard bracket matching provided by the `TextEditor` class. One alternative solution would be to implement a custom painter to decorate the multiple-character brackets. As the custom decoration may make the editor repaint more slowly and thus

reduce the performance of the editor, it is not worth doing this. As the Eclipse built-in support makes the job easier, this solution was adopted finally.

5.3.4.2 Implementation Techniques

To add the behavior, the editor was configured with an `ICharacterPairMatcher` instance. Firstly, the class which implements the `ICharacterPairMatcher` interface was created. Its "match" method returns the range of the text between a bracket and its matching bracket. Then, in the editor class, the `configureSourceViewerDecorationSupport(SourceViewerDecorationSupport support)` method was overridden to configure the support with the custom class just created. The class that implements the `ICharacterPairMatcher` Java interface defines keys that allow the bracket matching to be enabled or disabled, and allow the decoration color to be chosen. This allows users to configure these preferences in the corresponding preference page of the CZT Editor.

5.3.5 Selection by Double-Clicking

When browsing or editing a program in a text editor, programmers would also like to select a small segment of text precisely and quickly, particularly through mouse-clicks such as double-clicks. Most often, they want to select a language-specific word or a segment of text enclosed by a pair of brackets. Then they can perform some action on the selected text, such as the frequently used **copy**, **cut** and **paste** operations. The CZT Editor implements this behavior in the following way. By double-clicking after a bracket, users can select the text between the bracket and its matching bracket regardless of whether the

bracket is composed of a single character or multiple characters. Otherwise, the CZT Editor selects the syntactic word containing the cursor position, or selects nothing if the character at the position of the cursor is a white space character. Figure 5.6 illustrates its behavior when the user double-clicks at the position after the bracket "\{". The text between "\{" and "\}" is selected.

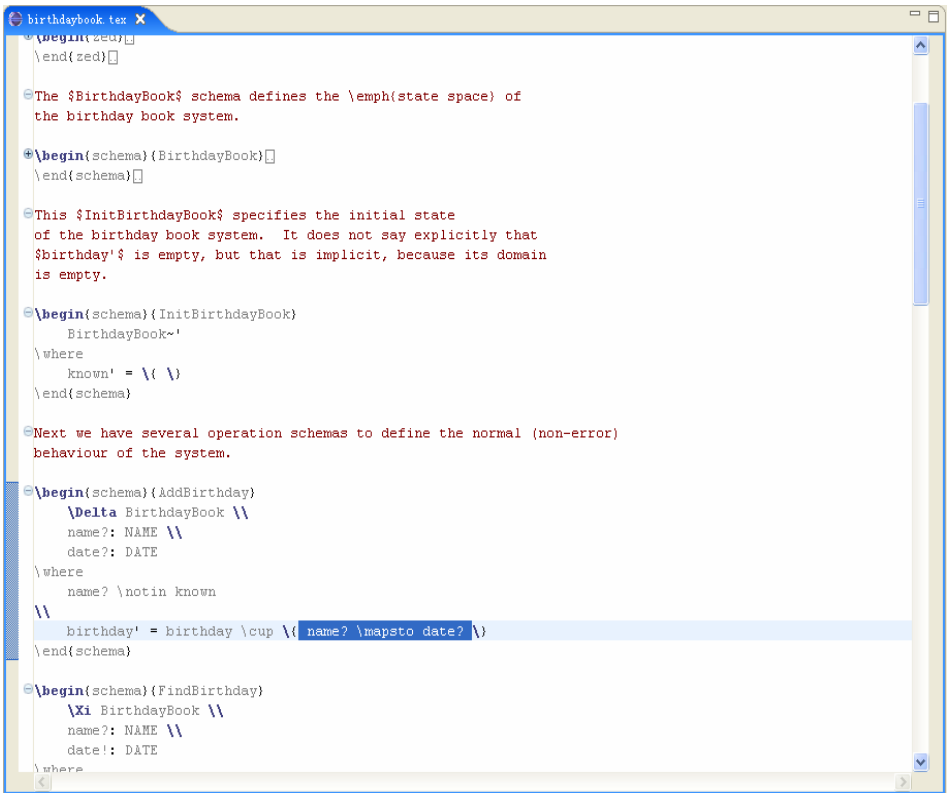


Figure 5.6 – Double-Click in the Editor

5.3.5.1 Implementation Techniques

To add this support, the `getDoubleClickStrategy` method in the custom `SourceViewerConfiguration` class was overridden so that it returns a custom strategy object for each kind of content type. The CZT Editor uses two different strategies for LaTeX and Unicode markups, because the specifications using LaTeX markup contain multiple-character brackets, while the ones using Unicode markup contain only the single-character brackets. These custome strategy

classes implement the `org.eclipse.jface.text.ITextDoubleClickStrategy` Java interface and specify their double-clicking behaviors by overriding `doubleClicked(ITextViewer textViewer)` method.

5.3.6 Marking References of A Z Name

Like the JDT plug-in for Eclipse, when the user moves the cursor onto a declaration of a Z name or a reference to a Z name, the name and all its references are marked in the editor. This is useful when the user wants to know where the name and its references are, to ensure that the name is used correctly. Figure 5.7 shows an example of the behavior.

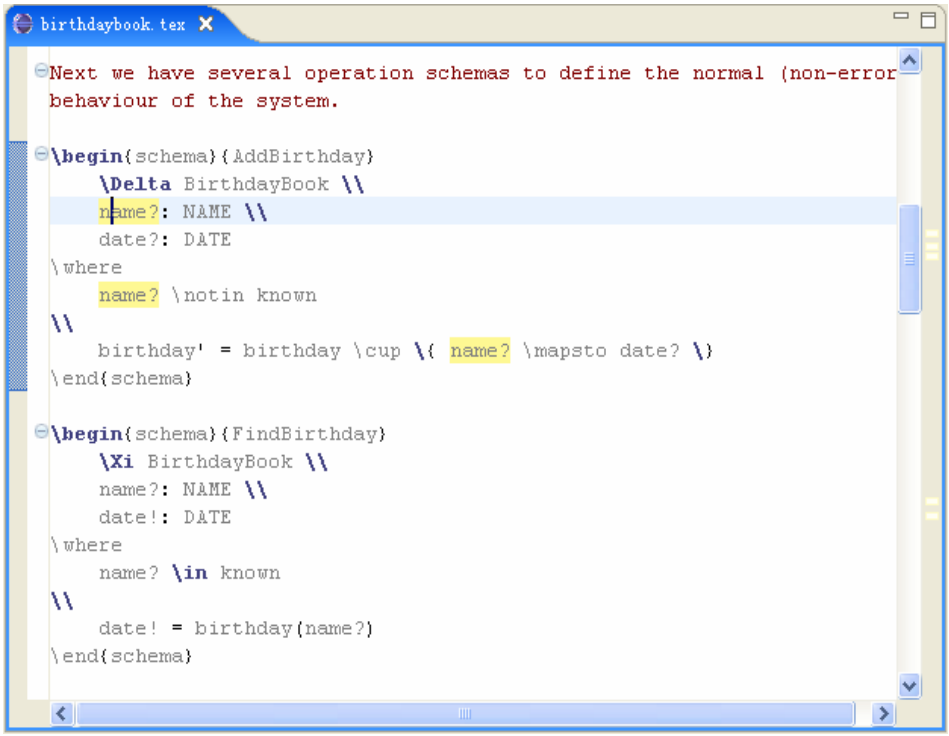


Figure 5.7 – Marking Occurrences of a Z Name

In Figure 5.7, the CZT Editor marks the name "`name?`" and all its references by changing their background color when the cursor position is inside the name.

5.3.6.1 Design Issues and Alternatives

As seen in Figure 5.7, the references of the name "`name?`" are decorated with shaded (yellow) background. Users can change the style of this decoration through the corresponding preference pages. This limits the choice of decoration styles to the styles provided by Eclipse, but this is a sufficient range of styles that there is no need to put more effort into developing alternative highlighting mechanisms.

If users change the decoration in the preference page, they will see that the JDT plug-in changes its behavior too. That is because the CZT Editor plug-in uses an annotation type of occurrences that the JDT plug-in contributed. Using the annotation type provided by the JDT plug-in gives the CZT Editor the same behavior for marking the references of names as the JDT plug-in. This is good because it makes the CZT Editor more familiar to existing JDT users.

5.3.6.2 Implementation Techniques

To make the positions of the occurrence annotations visible on the vertical bars beside the Eclipse editor, the type of annotation must be connected with a marker. Adding the new annotation type involved the following four steps.

1. The first step was to add a new type of marker to Eclipse. In the "Extensions" page of the plug-in manifest editor, a new marker extension was added to the extension point `org.eclipse.core.resource.markers`. The "id" and "name" fields were defined in the property panel on the right. Note that the actual id for the new marker will be the plug-in's id followed by a dot and the id specified here. If the id here is mistakenly used for creating markers, the markers may be shown in the editor,

but they will not be shown in the "**Problems**" view as other plug-ins do.

2. The right-click menu of the extension was used to add new "super", "persistent" and "attribute" entries with appropriate properties. The "super" extension specifies the id of the super type of the new marker. If the new marker needs more super types, they can be added in the same way. The "persistent" extension specifies whether this kind of markers will be persistent across Eclipse sessions. As there is no need to make the occurrence markers persistent across Eclipse sessions, this was specified to `False`. Similar to the "super" extension, the "attribute" extension is optional and can also be multiple. This extension, as its name suggests, is to add attributes to the new marker so that the plug-in can add information to this new kind of markers.
3. The third step was to add a new annotation type that extends the extension point `org.eclipse.ui.editors.annotationTypes`. This was added by right-clicking on the new entry and selecting the "**New/type**" menu entry. A unique name was specified for the new annotation as this name will be used when creating new annotations of this type. The id of the new type of marker entered here associates the annotation type with the type of marker. The "super" field plays the same role as the "super" extension field used for defining the new marker. The "severity" field is to indicate the severity of the marker. As the occurrence marker is not essential, there is no need to specify it – it was left blank.
4. As the new occurrence annotation type is always to be decorated in the same way as its super type, which is the occurrence annotation type provided by the JDT plug-in, this step was

omitted. But to allow it to use different decorations to those used by the super type, a new extension could be added to the extension point `org.eclipse.ui.editors.markerAnnotationSpecification`. The details about the properties for a specification will be introduced in Section 5.3.8.2.

After defining this new annotation type, code was written to add the add annotation objects of the new type whenever the text of a Z specification is changed or the cursor position is changed. Firstly, the editor removes all existing annotation objects of this type. Then the editor searches the parsing tree for the smallest Z term which contains the current cursor position. If the Z term found is a Z name, then the editor searches the parsing tree again for all Z terms identical to this name and adds annotation objects of this type for all terms found. Then these annotation objects are decorated in the editor.

5.3.7 Jumping To the Declaration of Variables

When editing a program in a text editor for a programming language, the programmer sometimes wants to quickly navigate to another part of the program. In particular, when referring to a variable whose declaration is invisible in the current screen or is even in another file, the programmer often wants to quickly browse to the declaration part of the variable. Figure 5.8 shows that the cursor is positioned on the word "`name?`".

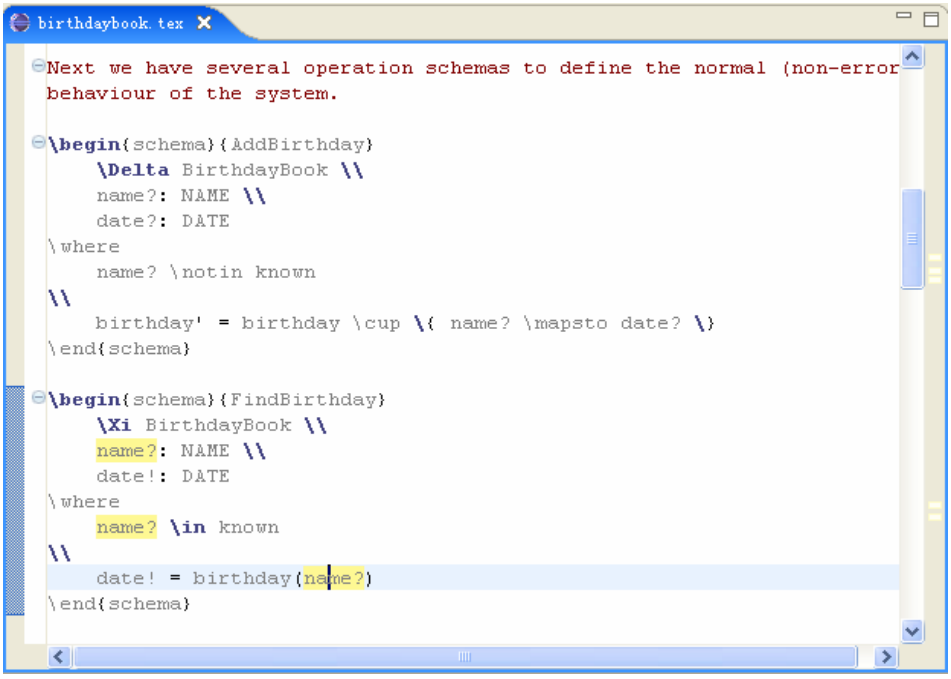


Figure 5.8 – Clicking Inside a Name

After the user executes the command by selecting the menu entry "**Edit/Go To Declaration**" in the workbench menu bar, the declaration of the Z name "`name?`" (the name itself) is selected as illustrated in Figure 5.9.

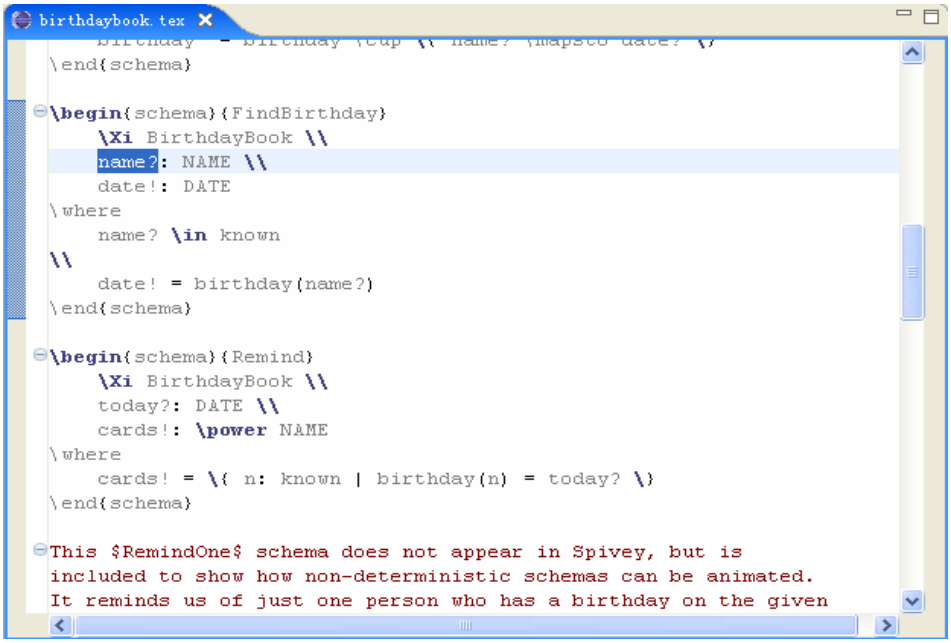


Figure 5.9 – Go to Declaration

5.3.7.1 Design Issues and Alternatives

When the user executes the "**Go To Declaration**" command on a name, only the name in its declaration part is selected. Another approach would be to select the whole declaration part so that the user will see the scope of the declaration of the name. However, in most other Eclipse editing tools, for example the Java Editor, this "**Go To Declaration**" command just selects the name in its declaration part. Therefore, the CZT Editor follows the same convention.

5.3.8 Highlighting the Enclosing Element

Like in CADiZ and the CZT jEdit plug-in, when browsing or editing a Z specification, the user can run a "**Highlight**" command to draw a box around the text associated with the underlying Z term which contains the current cursor position or is the enclosing term of the currently highlighted term. This helps users understand the logical structure of the Z specification and is useful for cutting and pasting expressions and predicates, selecting terms as arguments for Z-specific commands like rewriting or displaying type information, etc.

The CZT Eclipse editor plug-in implements this command by highlighting the smallest term that contains the currently selected text. In this way, the user does not have to start the highlight with a point, but can start it with an arbitrarily large text selection. When a term is already highlighted, further invocations of the highlight command expand the highlighted region to the enclosing term, thus making the highlight larger and larger. Figure 5.10 shows that, after clicking the menu entry "Edit/Highlight/Enclosing Term" in the workbench menu bar, the term underlying the text with yellow background color is highlighted.

As the user executes the command again and again, the highlighted area will get larger and larger until the largest term is highlighted.

There is also another highlighting command with the menu entry "Edit/Highlight/Restore Last Highlight". This command will do the highlight in the opposite way to the above command. That is, it will shrink the highlighted area until it highlights the smallest term containing the selected text.

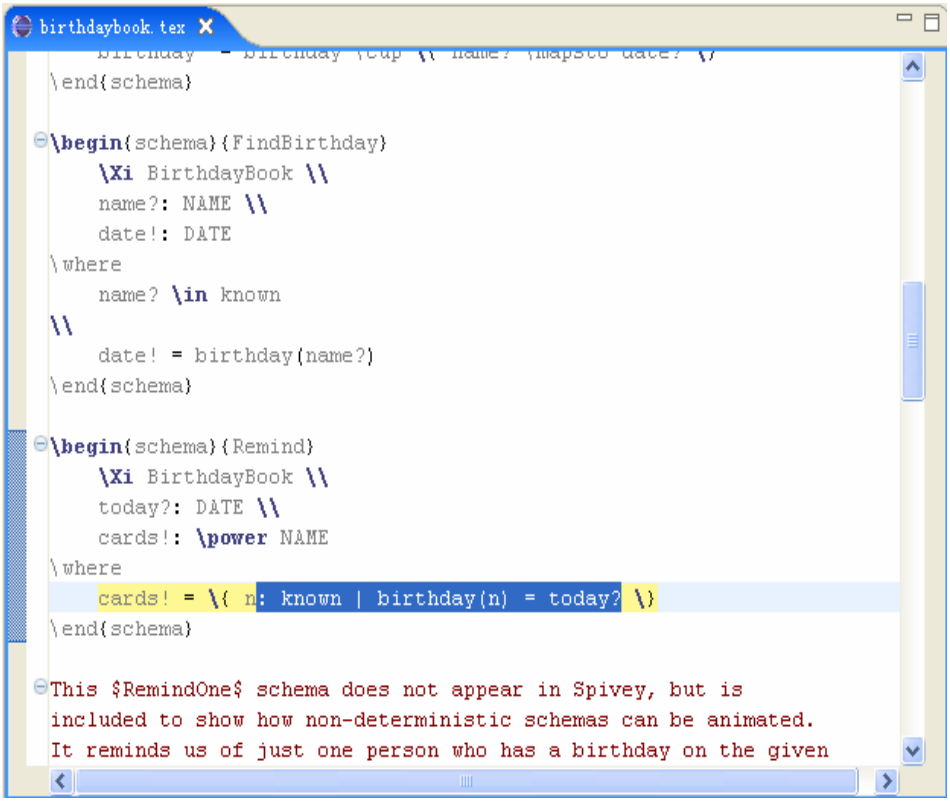


Figure 5.10 – Term Highlight

5.3.8.1 Design Issues and Alternatives

The CZT Eclipse plug-in could also highlight the term in the same way as CADiZ and the CZT jEdit plug-in do (based on just the cursor position, rather than the selected region). In fact, Z users who already know either of those two environments will obtain the same behavior in the CZT plug-in if they perform their usual actions.

However, the way that the CZT plug-in bases the initial highlighted term on the current selection has an advantages: it allows the user to save time by using the mouse to select most of the term they are interested in, then using the highlight command to expand that selection to a complete Z term.

5.3.8.2 Implementation Techniques

To implement the highlighting in the editor, the simplest way is to make it to be a type of annotation and make the editor draw the highlight decoration automatically when there is an annotation of this type. The new type of annotation was added to the Eclipse platform in a similar way as discussed in Section 5.3.6.2, except for a few differences.

As the editor can only contain at most one annotation of this type, there is no need to show markers on the vertical bar beside the editor. So in this case, a new type of marker did not have to be added.

In the property panel of the "Extensions" page, only the name field needed to be filled in ("`net.sourceforge.czt.eclipse.termhighlight`" in this case).

To make the new annotation type use its own settings, it was defined via a new specification rather than via inheritance from an existing super type. It involved the following two steps:

1. Firstly, a new extension was added to the extension point `org.eclipse.ui.editors.markerAnnotationSpecification`. This extension point is for plug-in developers to configure their own annotation type extensions.
2. The behavior of the new annotation type was configured in

the property page of the new specification extension. This configuration process is largely self-explanatory, since Eclipse displays help information to explain the purpose of each field. However, some of the major fields of the highlight annotation were configured as follows.

a*annotationType*: the type of the annotation this specification is used for.

label: the name to be shown in the preference page.

colorPreferenceKey: the preference key for the color used for decorating the annotation in the editor.

highlightPreferenceKey: the preference key for setting whether the type of annotation is to be highlighted in the editor.

textStylePreferenceKey: the preference key for setting the style of the decoration for the type of annotation if it is not to be highlighted.

includeOnPreferencePage: specify whether users change the decorations of the type of annotation in preference page. The decoration of the term highlight annotation is allowed to be changed in the preference page, so the field was set to `True`.

presentationLayer: the presentation layer on which the type of annotations will be decorated. As desired by Z users, the highlighting of a term should always have the highest priority over other kinds of decorations such as errors, so the

field is set to 6, which is the highest presentation layer.

In addition to these fields, some other fields are also provided, such as the default values for the above keys, to give users a reasonably good view of the decoration when they use the editor for the first time.

5.3.9 Problem Markers

During editing of a Z specification in the CZT Editor, a parser is constantly running in the background. Each time the user stops typing for about half a second after making some changes to the Z specification, the parser will start parsing the Z specification in the background without interrupting the editing. If there are any problems such as errors, the CZT Editor will report the problems in several ways. One way is to show error (red crosses) or warning markers (yellow "!") on the left-hand side of the editor. If a line of text contains problems, a marker would be drawn on the left-hand side of the line. Another similar way is to show markers on the right-hand side of the editor. One difference is that the markers are not drawn beside a particular row, but drawn at the relative position of the problem line to the whole specification. Another way is to show markers in the text and typically draw red squiggle line under the segment of text with problems. Figure 5.11 shows some sample problem markers reported to the user during editing of a specification.

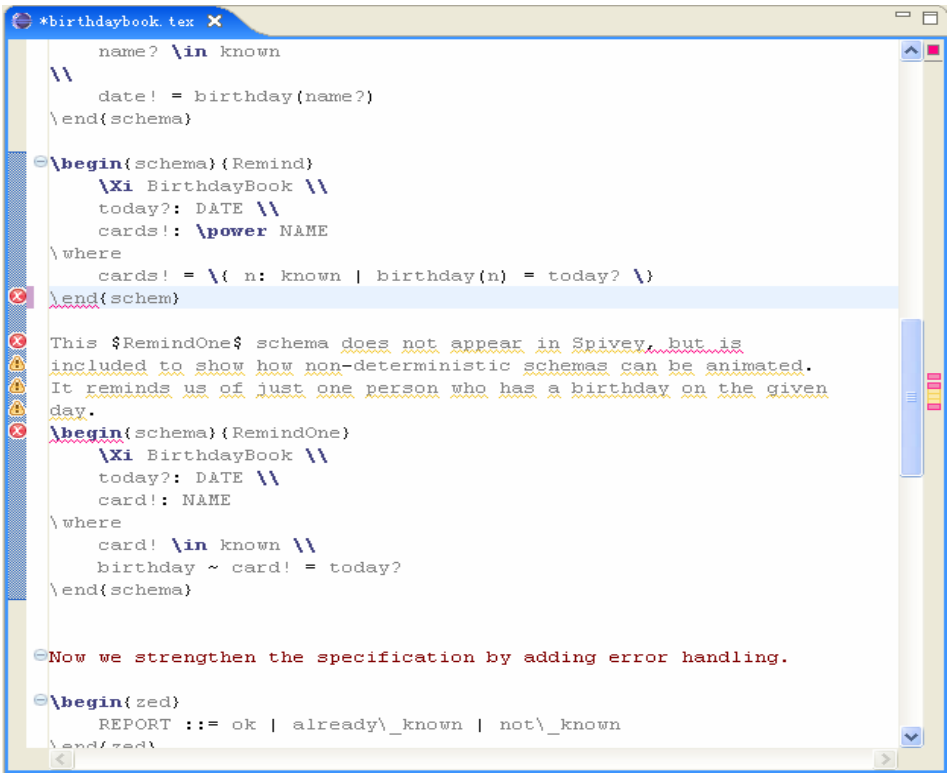


Figure 5.11 – Problem Markers

The background parser also reports problems to the user by adding entries into the "**Problems**" view that the Eclipse editor framework provides, which will be described in Chapter 8.

5.3.9.1 Design Issues and Alternatives

In the CZT jEdit plug-in, the errors or warnings are shown in the ZSidekick panel similar to the Eclipse "**Problems**" view. It can also output the information about the errors or warnings to the console.

The CZT Eclipse plug-in does not use the console output for the errors or warnings, but marks them directly where they occur. Doing so will give users a direct view of the problems and make them easily see where the problems are and thus fix the problems more quickly.

5.3.9.2 Implementation Techniques

Other Eclipse editors, such as the Java Editor, attach the problem markers to the files directly and persistently so that next time the file is opened, the problems can still be shown to users quickly. The CZT Eclipse editor does it in the same way.

Instead of adding a new annotation type, the CZT plug-in just contributes new marker extensions to Eclipse for problem reporting purposes.

The new marker extension was added in the same way as described earlier. But the "persistent" field was set to "`True`". In addition, since the problems will also be shown in the text, two marker types need to be set as its super types. These two super types are: `org.eclipse.core.resources.problemmarker` and `org.eclipse.core.resources.textmarker`.

When a specification file is opened or the text is changed in the editor, the background parser parses the file and returns the result to the editor. If any problems are found, the editor attaches the corresponding markers of the new marker type to the file. After that, the user will see the problems indicators around the editor and in the "**Problems**" view.

5.3.10 Hover Support

When the user moves the mouse around the CZT Editor and hovers above some Z text, the editor will display some information about the text in a floating panel next to the mouse position. This behavior is referred to as the hover support in Eclipse. Figure 5.12 shows that the editor displays the text "`Unknown latex command \end`" when the user

positions the mouse above the text "\end".

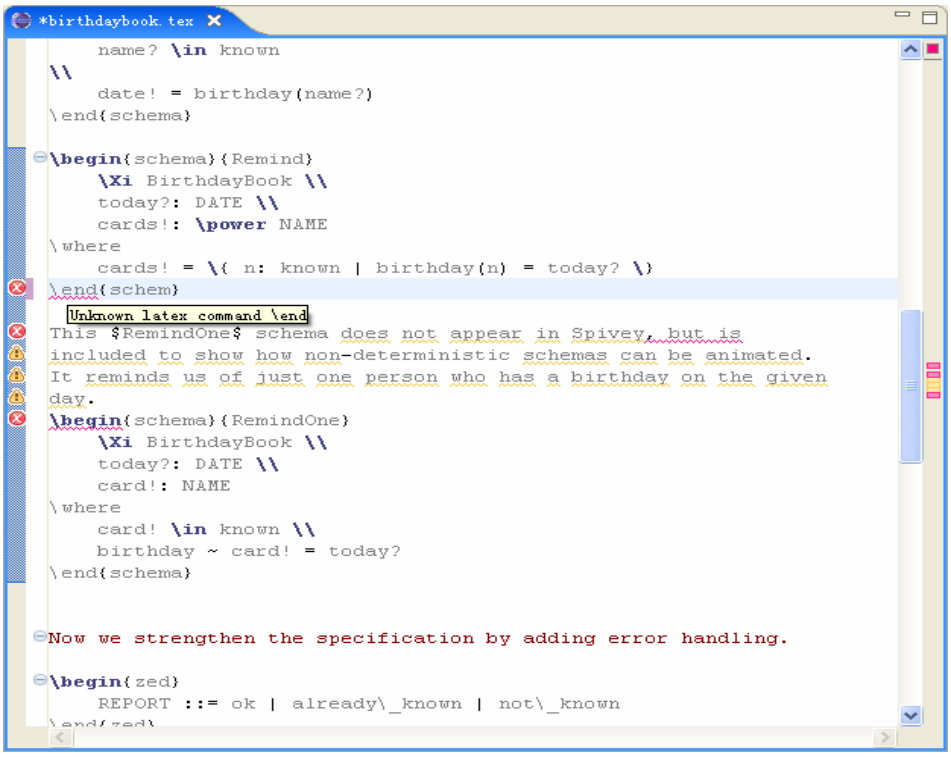


Figure 5.12 – Hover on Problems

The hover information displayed above is the error information for the text "\end". There are some other kinds of hover information provided by the CZT Eclipse editor. The editor determines the information to display in the following way. If the mouse is hovering above a marker on either vertical bar beside the CZT Editor, the information associated with the marker would be displayed. Otherwise, if the mouse is hovering above a position inside the text, the situation is more complicated. Firstly, if the mouse is hovering above the text that is being highlighted, the information about the highlighted term will be displayed because the information about the highlighted term always has highest priority. The second priority is given to the problems. That is, if there are any markers at the mouse position, the information associated with the marker would be displayed. Or if the mouse is hovering on a Z name, the type of the name will be displayed (see Figure 5.13). The

highlighted term is given the highest priority so that users can still use the term highlighting feature within an area that has a problem marker. For example, this can be useful to explore the types of the sub-terms that are causing the problem.

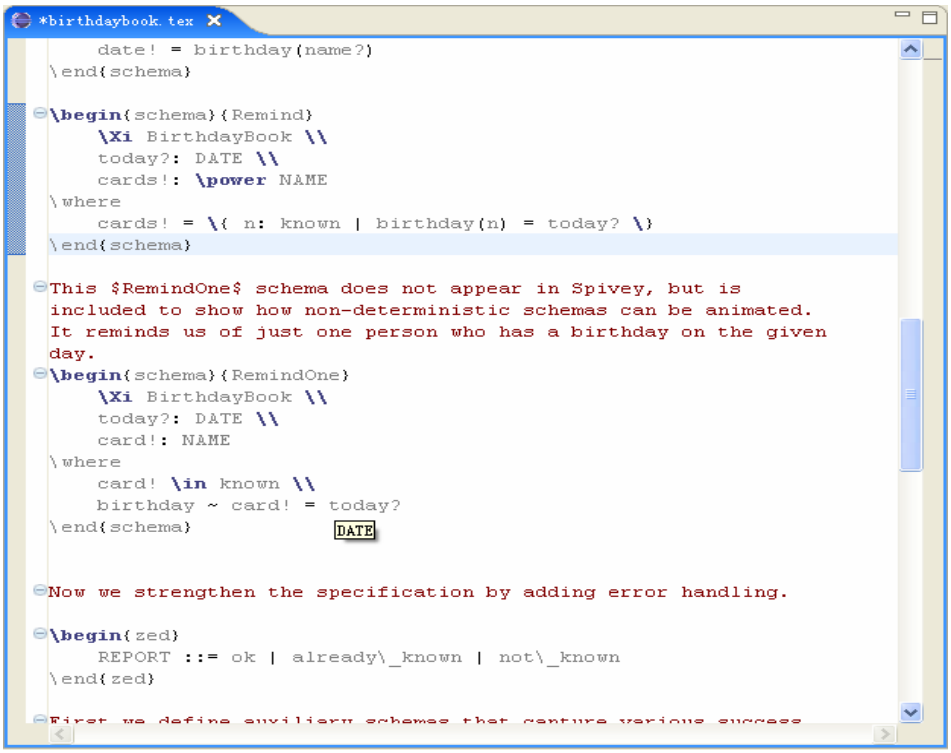


Figure 5.13 – Hover on a Z Name

5.3.10.1 Design Issues and Alternatives

As discussed in Chapter 4, some information can be displayed in the status bar at the bottom of the window. However, the status bar can not display multiple lines of information, so it does not provide enough display area for large amounts of information. Alternatively, some information, for example the information about a highlighted term, could be displayed on a special panel. But it is not a good practice to do so, because users who are not familiar with the tool will easily ignore the panel and miss seeing the information.

5.3.10.2 Implementation Techniques

To get the hover information on the vertical rulers beside the editor, the `getAnnotationHover` and `getOverviewRulerAnnotationHover` methods in the custom `SourceViewerConfiguration` class were overridden and made to return an instance of the `org.eclipse.jface.text.source.IAnnotationHover` Java interface. To get the hover information inside the text, the `getTextHover` method in the custom `SourceViewerConfiguration` class was overridden and made to return an instance of the `org.eclipse.jface.text.ITextHover` Java interface. The actual information to be displayed is determined by the `getHoverInfo` method inside the returned instances.

5.3.11 Viewing Specifications in Alternative Formats

As a Z specification can be written in different markups such as LaTeX and Unicode and be saved into a file using various character set encodings, the user may want to view a Z specification in different formats and/or encodings at the same time. The benefit is that the user can compare different markups and/or encodings of the same specification to ensure that the specification is being written correctly.

To meet this requirement, the CZT Editor plug-in provides the following set of commands in the "***Edit/Convert To***" menu, for the conversion of a specification.

- Convert to LaTeX: This converts the current editing specification written in Unicode markup into a new specification using LaTeX markup for comparison.
- Convert to Old LaTeX: This converts the current editing

specification written in Unicode markup into a new specification using old LaTeX markup (Spivey Z, which preceded the **ISO Z Standard**) for comparison.

- Convert to Unicode: This converts the current editing specification written in LaTeX markup into a new specification using Unicode markup for comparison.
- Convert to XML: This converts the current editing specification written in either LaTeX or Unicode markup into a new specification in XML format for comparison.

In order to view the specification using the new format, a panel such as an editor or a view must be used. As only one editor can be displayed in an Eclipse window, an editor is not suitable for this purpose. Instead, a view is used to display the output of the conversion, while the editor displays the original form of the specification. This allows the user to browse the specification in both formats at once, so that they can be compared. Another advantage of a view is that it can be moved around the editor window.

In Eclipse, the particular view can be implemented so as to allow multiple instances of the view to be opened at the same time, but the user may be confused by multiple instances if the view is not used properly. Due to this concern, the current implementation allows only one instance of the view to be created. This specific-purpose view is called the **Z Conversion View** and it will be described later.

5.3.11.1 Design Issues and Alternatives

The CZT jEdit plug-in provides this feature by putting the output of the conversion into the main editor window as an unsaved buffer. The advantage of that approach is that the converted specification

can be easily saved for future uses. But, as discussed above, only one editor can be shown to the user at once, so that approach would not be convenient for the user to compare two formats of the specification. But a view can be opened at the same time as the time an editor is opened. The user can then compare the specifications in the view and the editor easily. Although saving the contents in the view is not so convenient, it is a minor issue compared to the ability to compare specifications in the view and the editor.

5.4 Future Implementation

In the future, it would be better to make the editor provide users with a more comprehensive set of commands, such as the command for rewriting a selected formula and the command for proving a selected predicate in the editor provided by the CZT jEdit plug-in.

Chapter 6 Wizards

Most other editors provide a set of wizards to give their users a convenient way to create new documents. Especially in editors for programming languages, the wizards can help programmers to create and initialize new documents which are free of errors and contain some initial contents to help programmers to quickly start editing. The Eclipse platform also provides developers with wizards for creating new objects such as projects and files. A good example is obviously the JDT plug-in. The JDT plug-in provides Java programmers with a set of useful wizards for creating new Java elements. The most useful wizards are possibly the following three wizards:

- New Java project creation wizard
- New Java class creation wizard
- New Java interface creation wizard

To follow the same convention, the CZT plug-in provides its users with wizards for the creation of new CZT project and Z specifications.

The CZT project creation wizard is similar to the wizard for creating simple projects. However, since a Z specification can be written in different markups and/or encodings, the Z specification creation wizard has a special feature added. In the wizard page, the user can specify the markup and/or encoding for the new specification. Once the wizard is finished, the new specification file will be opened in a CZT Editor, using the markup and/or encoding that the user specified.

6.1 New CZT Project Creation Wizard

The wizard allows users to create a new CZT project, which is currently a simple general-purpose project. Figure 6.1 shows a screenshot of the main page of the wizard:

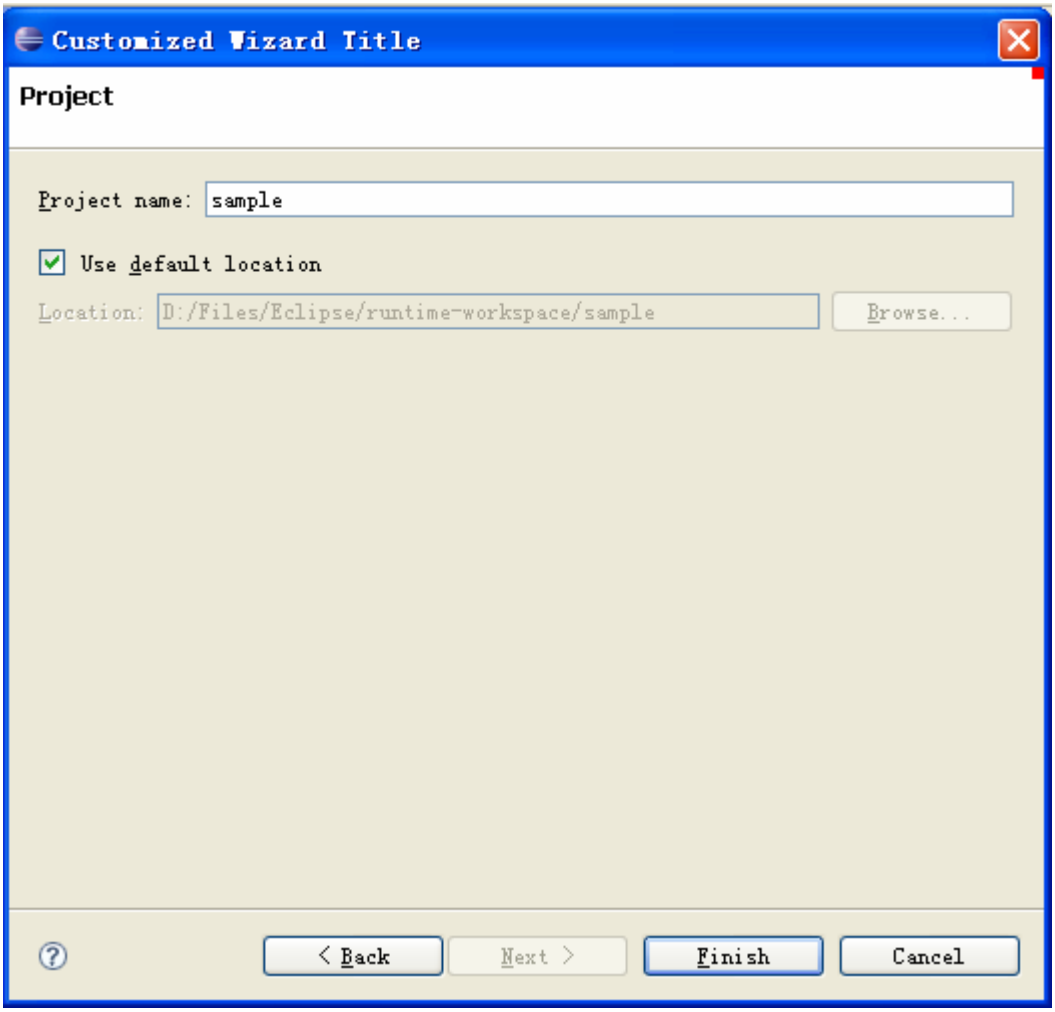


Figure 6.1 – New CZT Project Creation Wizard

To create a new CZT project, the user needs to enter a name of the new project. The name must not contain any invalid characters including '*', '/', '\', '?', '.', '|' and etc.

If the "Use default location" option is checked, the wizard creates the new project at the location the wizard chooses, which is a folder that has the same name as the project and is under the root folder of the current workspace. Otherwise, the new project will be created at the location the user specifies.

6.2 New Z Specification Creation Wizard

The wizard provides the plug-in users an easier way of creating a new Z specification. In this wizard, the users can specify the name and the enclosing element (a project, a folder, etc) of the new Z specification. As well, the users can set the markup and/or the encoding that the new

specification will use. Figure 6.2 shows a screenshot of the wizard.

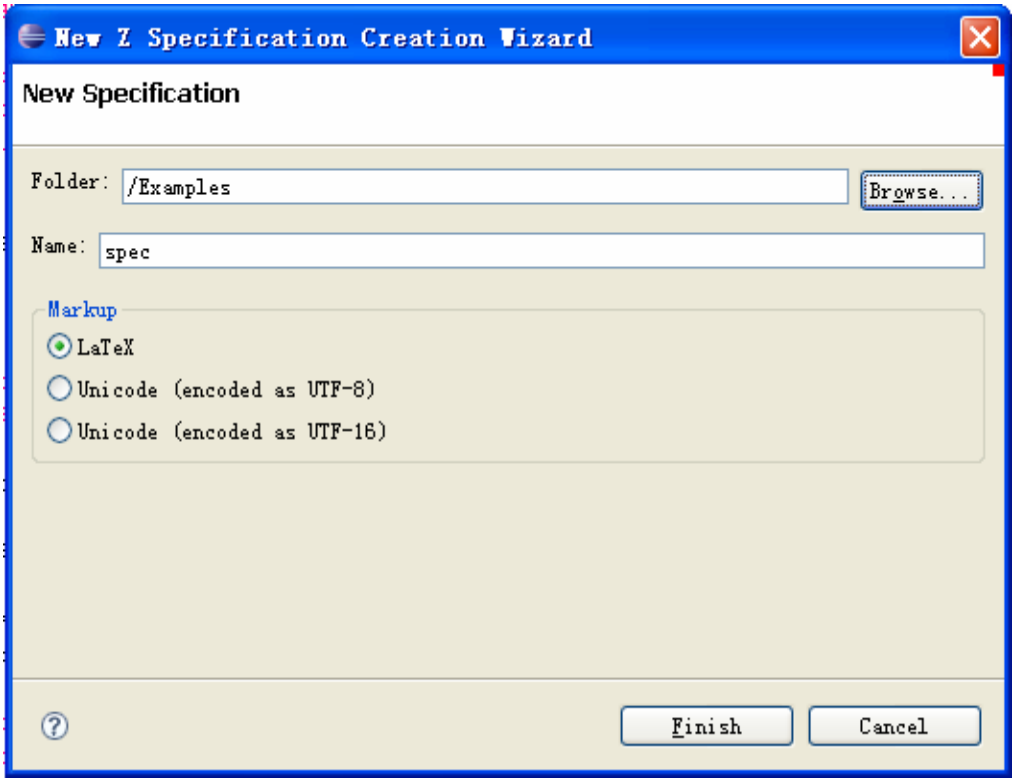


Figure 6.2 – New Z Specification Creation Wizard

The wizard gives users the following three options of the markups to use:

- Latex: the new specification will use LaTeX as its markup. No encoding type is specified here and thus the specification will be encoded as the default encoding type the Eclipse editor framework provides. The extension ".tex" is appended to the file name.
- Unicode (encoded as UTF-8): the new specification will use Unicode as its markup and be encoded as the UTF-8 encoding type. The extension ".utf8" is appended to the file name.
- Unicode (encodes as UTF-16): the new specification will use Unicode as its markup and be encoded as the UTF-16 encoding type. The extension ".utf16" is appended to the file name.

Since a Z specification file uses the extension of its name to imply its markup and encoding type, the user is not allowed to enter an extension into the name field. Its extension will be appended after the name that the user inputs according to the type of the markup and/or encoding that the user specifies.

6.3 Implementation Techniques

To create a wizard for new projects or new files, the following four steps are involved:

1. In the "Extensions" page of the plug-in's manifest editor, a new wizard extension point was added to the plug-in's extension list by using the extension point `org.eclipse.ui.newWizards`.
2. Next, a new category was added in the similar way as adding a category for views (see Section 4.3). The id and name of the wizard category used in the CZT Eclipse editor plug-in were `net.sourceforge.czt.eclipse.newwizard` and `CZT` respectively. The critical point here was also to try to specify a special id so as to make it unique. The name of the category is displayed in the entries of the new project/file wizard page, which is opened by using the menu entry "**New/Project…**" or "**New/Other…**".
3. Then, in the similar way, a new wizard extension. Selecting the new wizard entry in the "Extensions" page displays the settings for the wizard. The field named "`Final Perspective`" is for specifying an Eclipse perspective to be opened for the new project or file. If there is already one which is suitable to the new project or file, the id of it need be filled in the field. The creation of a new perspective will be discussed in Chapter 10. If the wizard is used for creating a new project, the "`Project`" field is also needed to be set to `True` to indicate that it is a new project creation wizard. Then the class file for the wizard was provided by clicking on the link "`class*`" which displays a new Java class wizard is displayed. After accepting the default setting the class file for the wizard was opened in the editor.
4. Finally, the Java class and its related classes defining the wizard

pages were created.

Chapter 7 The Outline View

As the Z language is inherently hierarchical, it is useful to display to the user the content outline of the currently editing specification. Like most other Eclipse editor plug-ins, The CZT plug-in reuses the outline view that the Eclipse platform provides to offer a content outline page, which is also similar to the Z Sidekick panel in the CZT jEdit plug-in. The outline page displays a simplified overview of the contents of the Z specification that the user is browsing or editing. Figure 7.1 shows a screenshot of a part of the outline view for a sample Z specification.

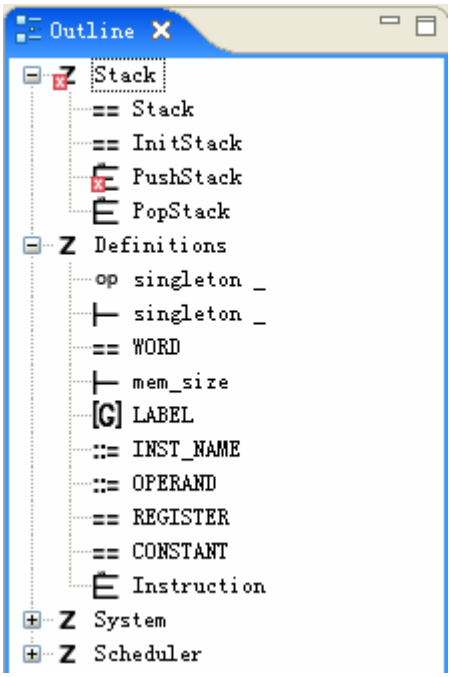


Figure 7.1 – Outline View

Unlike the Z Sidekick panel in the CZT jEdit plug-in, the outline view does not display the type of the term underlying an entry in the view, but it displays an icon before the name of an entry to indicate the type of the term.

In the tree structure of the content outline view, the first level elements give the description of each section in the specification. The second level elements outline each paragraph in its enclosing section. The third level elements show the information about the declaration inside each paragraph, but only if it contains more than one declaration.

From the outline view, the user can quickly grasp the hierarchical structure of the text in a specification being browsed or edited.

The outline view also offers users a quick way to browse to a specific segment of text in the currently active editor. When the user clicks on an element on the tree structure, the associated CZT Editor selects some text in it according to the selection in the outline view.

Additionally, if the user enables the outline view to synchronize its selection when the cursor moves in the CZT Editor, then the selection in the outline view will be changed as the cursor moves around the editor.

Furthermore, as seen in Figure 7.1, the icon for the section `Stack` and the one for the paragraph `PushStack` are decorated with red crosses. The red crosses indicate that the text related to the outline entries contain errors. This is another way of reporting problems to users.

7.1 Design Issues and Alternatives

Unlike the Z Sidekick panel in the CZT jEdit plug-in, the outline view does not display information about the Z narrative sections or paragraphs as the semantics of these is usually of less concern to the Z users. Another major difference is that the outline view does not display the type of the term before the name of an entry in the outline view, but displays an icon before the name instead. Different icons apply to different kinds of terms. Doing so makes the outline view more concise and visual (easy to recognize the different

types of paragraphs) and more like the one provided by the JDT plug-in as well as some other Eclipse editor plug-ins.

7.2 Implementation Techniques

Although each editor is linked to its own content outline page, the Outline view is not provided by the Eclipse editor framework, but by the `org.eclipse.ui.views` plug-in. When the Outline view is opened, an instance of `ContentOutline` is created. When the user changes editors, the instance will ask the active editor (if one exists) whether to support the `ContentOutline` view by invoking the editor's `getAdapter(Class required)` method to ask for an adapter of type `IContentOutlinePage`. If the editor supports it, then it is the editor's responsibility to provide an `IContentOutlinePage` adapter (Zoio, 2006). In the CZT Eclipse editor class, the `getAdapter` method looks like this :

```
public Object getAdapter(Class required){
    if (IContentOutlinePage.class.equals(required)) {
      if (fOutlinePage == null) {
        fOutlinePage = createOutlinePage(); // create one
      }
      updateOutlinePage(getParsedData()); // set its input
      return fOutlinePage;
    }
    …
    return super.getAdapter(required);
}
```

The class for the content outline page can be created by implementing the `IContentOutlinePage` interface. However, if the class is created by subclassing from the `org.eclipse.ui.views.contentoutline.ContentOutlinePage` class,

then the class is already set with a `TreeViewer`. As the CZT content outline page uses a tree to show the structure of a Z specification, it just subclassed the `ContentOutlinePage` class. The remaining implementation steps were to provide a content provider and a label provider for the `TreeViewer`.
Each time the text in the active CZT Editor is changed and the user stops editing for more than half a second, the background parser associated with the editor will parse the specification in the editor. Thereafter, the editor will prepare new input for its content outline page. If the **Outline** view is open at that time, the editor will pass the new input to its content outline page. Then the Outline view will update its layout using the editor's content outline page with new input.
To make the selection in the editor synchronize with the selection in the **Outline** view, the CZT Eclipse editor plug-in made use of the selection service provided by the Eclipse workbench. This service "allows efficient linking of different parts within the workbench window" and "gives plug-ins a clean design" (Hoffmann, 2006). The service adopts the selection provider-selection listener design pattern to link different parts. This pattern is a handy way of create a part (editor or view) that responds to changes in other parts (Pandit, 2005). To make the editor or the **Outline** view capable of publishing a selection, they added a selection provider to their respective workbench site. To make them capable of listening to selection changes in each other, they are registered as a selection listener to each other. In this way, the editor and the Outline view can consume the selection changes in each other so as to make them synchronize with each other.

Chapter 8 The Problems View

As described in Chapter 5, the CZT Editor makes use of a background parser to constantly parse the specification in the editor as the user edits it. If one or more problems (errors, warnings, etc) are found after parsing, the editor reports the problems to the user by adding markers on the vertical bars beside the editor, decorating the text with problems in the editor, and also adding problem entries into the **Problems** view that the Eclipse editor framework provides.

Figure 8.1 shows a screenshot of the **Problems** view when problems are found after parsing the specification.

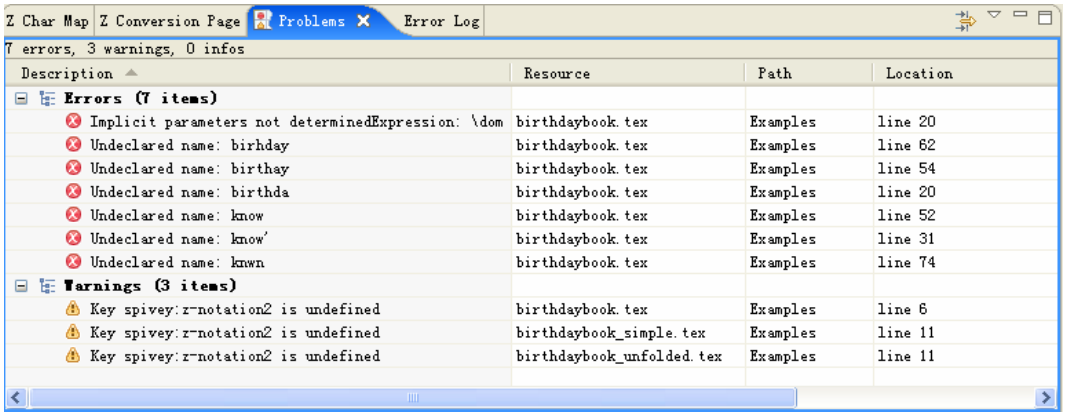


Figure 8.1 Problems View

In the **Problems** view, the problems are grouped together based on the resources in the current workspace. Each problem is displayed in its own row with the columns described below:

- Column 1: describes the problem. The description is the message that is attached to the problem marker corresponding to the problem. Additionally, as seen in Figure 8.1, an icon is also displayed before the

description to show the severity of the problem.

- Column 2: states the name of the resource in which the problem occurs.
- Column 3: states where the problem resource is.
- Column 4: states the number of the line in which the problem occurs.

In addition, the view groups all entries with the same severity together so as to make the user easier find a particular kind of problems. If the user clicks on an entry in the view, the view will also display the description (in the first column) on the status bar at the bottom of the editor window.

8.1 Implementation Techniques

Since the **Problems** view that Eclipse provides is sufficient for the purpose of displaying the problems found by the parser, there is no need to implementing a CZT-specific view. The CZT Eclipse editor plug-in simply reused the standard **Problems** view so as to follow the same convention used by most other Eclipse plug-ins.

As discussed in Section 5.3.9.2, the editor attaches problem markers to the specification file for the problems found by the background parser. Then, the user will see the problem indicators around the editor and in the **Problems** view. However, note again that the real ID of the custom marker type is not the one being specified in the plug-in manifest file. If a wrong ID is used for creating markers of this custom type, the problems found by the parser may still be visible around the editor, but will not be visible in the **Problems** view.

Chapter 9 The Z Conversion View

As discussed in Chapter 5, Z users often desire to view a Z specification in different formats. Particularly when they are editing a Z specification, they often want to view it in another format to ensure that it has been written correctly. For this reason, when the user executes a command for converting the current editing specification into another format for comparison, the new specification will be displayed in the specified format in the **Z Conversion view**.

Figure 9.1 shows a sample screenshot of displaying the specification named "`birthdaybook.tex`" using LaTeX markup in a CZT Editor and using Unicode markup in the Conversion View.

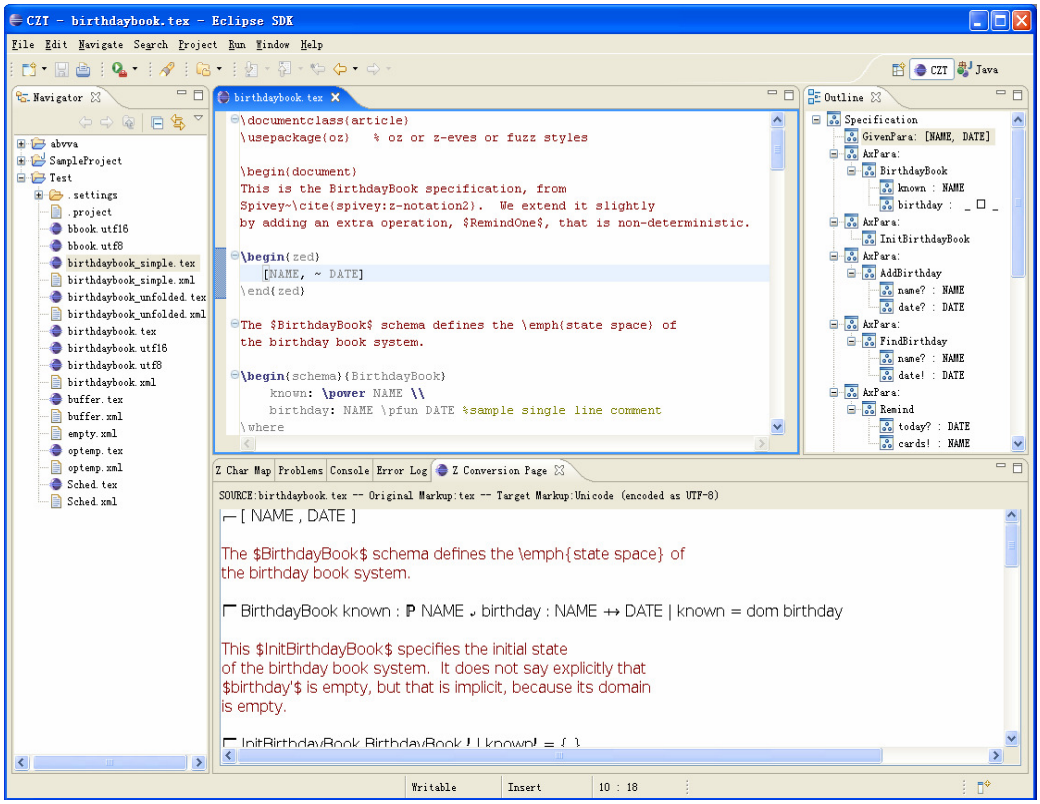


Figure 9.1 – Z Conversion View

As seen in Figure 9.1, the **Z Conversion view** also provides syntax coloring support for the new specification so that the user can see the internal relationship between those two formats and thus can easily make the comparison between the two formats of the same specification.

9.1 Implementation Techniques

The view is implemented using the similar technique as implementing the **Z Character Map** view. In addition, to make the view support syntax coloring, the view adopt another instance of the `SourceViewer` as used in the editor and configure it with a simple version of the `SourceViewerConfiguration` class used by the editor. The simple one only provides the syntax coloring feature as discussed in Section 5.3.1. The identical syntax coloring rules are also used here. Another difference from the editor is that the `SourceViewer` used in the view is set to be not editable because the view is used for viewing a specification only.

9.2 Design Issues and Alternatives

As discussed in Section 5.3.11, there are two major advantages of displaying a specification converted from another format in a view. One advantage over an editor is that a view can be displayed to the user at the same time that an editor containing the original specification is displayed. Another is that it can be moved around the editor window for ease of viewing and comparison.

The current implementation declares that only one instance of the view can be opened at the same time. Alternatively, it could be implemented so as to allow multiple instances of it to be opened at the same time so that user can see multiple format of the same specification for comparison at the same time. However, this would make the user confused about which view corresponds to the current editing specification if some views contain specifications converted from other specifications instead of the one in the active editor.

To avoid this issue, it is implemented so as to allow only one instance of the view to be opened at the same time.

For easy comparison, the view also supports syntax coloring using the same rules as used in the editor, so that the user can easily see the internal relationship between the two formats of the specification.

9.3 Future Implementation

In the future, it would be better to make the view support some browsing functionalities. For example, it would be better to make the selection in the view synchronize with the selection in the editor as long as the text in the editor is not changed after the conversion. The synchronization between the view and the editor would make the comparison much easier.

Chapter 10 The CZT Perspective

Eclipse provides users, as well as plug-in developers, with the ability to organize the appropriate editors, views and actions into a customized perspective.

"*A perspective in Eclipse defines a set of editors and views arranged in an initial layout for a particular role or task. For example, the Debug perspective is designed for the task of debugging source code. Eclipse can have one or more perspectives open at a time, though only one is visible. Perspectives also can be designed with a predefined set of functions available through the menu bar and toolbar that you can perform while in the perspective*" (D'Anjou et al., 2005, p.19).

The benefits of a perspective to user is that, once the perspective is opened, the user can get a clear view of what major functionalities the plug-in provides are supposed to be used frequently during the use of the plug-in and it can thus save the user's time for finding them.

Many Eclipse plug-ins provide their own sets of perspectives. For example, the JDT plug-in provides a perspective for programming Java (*Java Perspective*), a perspective for browsing Java code (*Java Browsing Perspective*) and a perspective for viewing a Java type hierarchy (*Java Type Hierarchy Perspective*).

As the CZT Eclipse editor plug-in also provides an editor and a set of useful views for editing and browsing Z specifications, it is a good practice to provide an appropriate CZT perspective for its users, so that it will give new users a good start of using the new interface. The CZT perspective can also help new users to understand the relationship among the editor, the views and the actions.

The CZT perspective predefines the following set of Eclipse elements:

- The Resource Navigator view provided by Eclipse. For Z users, they can use this view to manage their specification files.
- The Z Character Map view

- The Problems view
- The Z Conversion view
- The Outline view

The CZT perspective also adds shortcuts for a set of other perspectives, such as Java and Java Browsing perspectives, so that users can quickly change to other perspectives for other kinds of tasks. The shortcuts for these perspectives can be displayed by clicking on an icon on the top-right corner of the editor window.
It also adds shortcuts for the predefined set of elements listed above to the list of shortcuts of views. The shortcut list can be displayed using the menu entry "**Window/Show View**". This helps users to re-open these views quickly in case they close some of them.
In addition, it adds to the "**File/New**" menu the shortcuts for the new CZT project wizard, the new Z specification wizard, as well as some wizards that Eclipse provides, including the new file wizard, the new folder wizard and the new untitled file wizard.
The CZT perspective can help Z users to quickly start creating and editing Z specifications without having to search through deeply nested menus to find the CZT related Eclipse elements and commands.

10.1 Implementation Techniques

To contribute a perspective to Eclipse, the following three steps were followed:

1. In the "**Extensions**" page, the extension point `org.eclipse.ui.perspectives` was added to the plug-in's extension list.
2. Then, a new perspective extension was added by right-clicking on the new entry and selecting "**New/perspective**". In the property panel of the extension, the `id`, `name`, `class` and `icon` of the perspective were specified. The "`fixed`" field is to indicate whether the layout of the perspective is fixed. In the CZT case, the

layout of the perspective is allowed to be changed by the users when needed. In particular, the users may want to move the Z Conversion View around the editor, so that they can view two formats of the same specification easily for comparison.

3. Finally, the Java class specified in the "`class`" field was created. The class implemented `org.eclipse.ui.IPerspectiveFactory` interface, which must be implemented by perspectives. In the perspective factory class, the initial layout of the elements included in the perspective was specified. The CZT perspective also added some shortcuts for a set of elements for quickly finding them in the perspective.

10.2Design Issues and Alternatives

There is an alternative way to make users easily find the set of elements needed during the development of Z specifications. That is to add all the CZT elements (views, wizards, etc.) to an existing perspective, such as the Java perspective. However, during editing of Z specifications in Eclipse, users may need many views, including the Z Character Map view and the Z Conversion View. It is better to predefine the set of useful elements in a CZT-specific perspective so that users can be free of the concerns about finding them during editing Z specifications.

Chapter 11 Evaluation 1 – Design and Features

During the development of the CZT Eclipse editor plug-in, two evaluation exercises were conducted on it. This is the first evaluation. It aims to figure out some design issues for the plug-in and also to find some useful features desired by Z users.

The evaluation exercise was done among 18 people during the Second Annual CZT Workshop called "CZT: Past, Present and Future", which was held in York University, UK, on 24-25, August, 2006. The audience feedback on the Eclipse interface to CZT was generally positive and encouraging.

Here are some of the specific feedback points from the workshop (All text in italics is feedback from the people attending the evaluations – names have been removed to preserve anonymity):

- Several people said that, "*The Eclipse interface to CZT looks "very useful" and we prefer the Eclipse interface to the jEdit interface. I want to start using it as soon as it is easy to install. That is because Eclipse is more widely used by a lot of people for other purposes and because of the constant feedback about syntax and type errors during editing a Z specification without interruption"*. The person who wrote the parser and typechecker used in background parsing said that, "*I am very impressed that the background parsing works so well and so quickly"*. The discussion in the workshop regarding the parser showed that the parser would be possibly improved so as to work faster. As a result of the improvement, a Z specification would be parsed in less than one second in the background of the Eclipse interface. The concerns about the installation of the plug-in have been solved by a website used for storing the plug-in binaries so that the users can install the plug-in by

downloading the plug-in into the "**plugins**" directory of the Eclipse SDK. The users will not have to install them by manually building the plug-ins inside Eclipse. Hereafter, the installation will not be concerned by the users.

- Someone said that, "*The outline mode would be more useful if it was changed so that:*
    1. *It starts up with just showing the top-level items, rather than everything expanded. This would allow to get a good overview of the contents of the specification, rather than to see all the detail of just the first few paragraphs.*
    2. *The outline entries should show more information, rather than just a simple name of the type of a term.*
    3. *It uses a wider variety of different icons for different kinds of paragraphs (including different icons for schema definitions and constant definitions – these are currently just shown as axiomatic paragraphs, but they are important special cases of axiomatic paragraphs that need to be displayed differently)*".

  Regarding issue 1, the content outline page has now been changed in such a manner that it starts up with displaying to the users a reasonably short outline of the specification depending on the number of sections it contains. If it contains multiple sections, the outline page would initially show only the top-level items – those sections. Otherwise, it would start with showing the items in the top two levels, which are the single section and its paragraphs.

  Regarding issue 2, the outline page has been changed so that it displays the names of the paragraph. For example, it displays "`foo, bar`" or "`foo…`", rather than just "`AxPara`" for an axiomatic paragraph.

  Issue 3 has been solved by creating a set of icons and displaying an icon before the names of a section or a paragraph so that the users can figure out the types of the entries in the outline panel by the icons and the

interface can have the similar feature as JDT has.

- Someone said that, "*When a term has been highlighted and the mouse is hovering within that highlighted term, then the hover message should show the kind of the syntactic node that is highlighted, plus the type of the expression (if that expression is annotated with a type)*".

  As the highlight mode is different from the text selection in an editor, it would be nice to treat them separately. It is hard to decide what the hover message is when hovering within a highlighted term. The information about a highlighted term could be displayed in either of the following two ways. One is to display it in the hover message by allowing it to take the highest priority among all kinds of information regarding a hovering position. It is also possible to allow the users to specify the priority of each kind of information in the Preferences page. The other is to provide another specific panel to show the information related to the highlighted term, including the type, actions, or else. As no much information is concerned by the users at the moment, the former solution has been adopted. The preferences setting for the priorities of the information will be implemented if desired by the users.

- Someone said that, "*In the **ZCharMap** panel, it might be nice to provide keyboard shortcuts for entering each character, in addition to the entry by mouse clicks. This could be done by defining the keyboard shortcuts in the XML file that defines the **ZCharMap**. Then the same shortcuts would be available in Eclipse and jEdit*".

  For expert use, it would be easier and quicker to use keyboard shortcuts to enter into an editor a character in the **ZCharMap** panel and it would definitely be a more convenient way to enter some frequently used characters. However, the implementation would involve with the definition of tens of different keyboard shortcuts to meet all users' favorites and it would result in several issues. One is very high probability of conflicts between one another in Eclipse. Another is the

bad design involving with the “duplicate” definition of tens of very similar actions, whose differences are only the different characters to enter. In fact, no users would like to remember so many keyboard shortcuts and only a few of them could probably be used by a single user for a long period. The design is definitely not worth the implementation effort.

- Someone said that, “*It would be nice to show real boxes around the schemas, especially in the Unicode Eclipse editor*”.

  This could probably be done by having a custom editor in Eclipse, but it is not worth the implementation effort because the only advantage that the custom editor takes over the text editor would be to show real boxes around the schemas.

  Alternatively this could be done by using sequences of line-drawing Unicode characters, and stripping them out before saving. However, the background parser would give different feedback of errors and thus result in the annotations in the editor being not synchronized with the one associated within the physical file. Another issue resulting from this design is that the editor has to remember the positions of those inserted characters so that it can remove them from the text on the fly when saving the content back to the physical file.

  Another alternative is to implement it in the same way as problems indication. Firstly, add a new annotation type to the editor and associate it with a particular drawing algorithm, which is used to draw the boxes for the type of annotation. In the editor part, it just adds the kind of annotations after each parsing.

  As the last alternative is easier and follows the same conventions as Eclipse, it has been adopted in the implementation of real boxes around the schemas in the Unicode Eclipse editor. Chapter 12 will give the description of the feature in detail.

- People liked the facility to hide/show each paragraph using the +/-

controls. One of them said that, *"It would be very nice to provide a convenience command that hides all the Z paragraphs and opens all the narrative paragraphs, and another command that does the opposite (opens all the Z paragraphs and hides all the narrative paragraphs). Or perhaps just one 'toggle-these-paragraphs' command that toggles the display of all the paragraphs similar to the one that the cursor is within*". As the JDT plug-in implements similar commands for toggling all paragraphs on the left vertical bar beside its Java editor, the CZT Editor provides those commands in the same position so that Eclipse users can easily find them.

The feedback from this evaluation turned out to be very helpful. Most of the ideas described above were implemented before the usability evaluation was performed (see Chapter 13).

Chapter 12 Schema Box

The previous chapter states some design issues, as well as some features desired by Z users. One of them was to draw real boxes around the schemas. As discussed in Chapter 1, Z specifications are traditionally written on paper. When people write specifications on paper, they often use the Z mathematical symbols (like Unicode) and draw boxes around schemas as well. The **ISO Z Standard** suggests two frequently used box rendering styles for Z schemas (ISO/IEC 13568, 2002, p.38). As a result of this, it is a good practice to draw real boxes around schemas written in Unicode markup in the CZT Editor.

The CZT Eclipse editor plug-in provides users with the ability to show either of both set of rendering styles of boxes around a schema depending on the user's setting in the corresponding preference page.

Figure 12.1 shows a screenshot of the boxes, using the first box rendering style, in the editor.

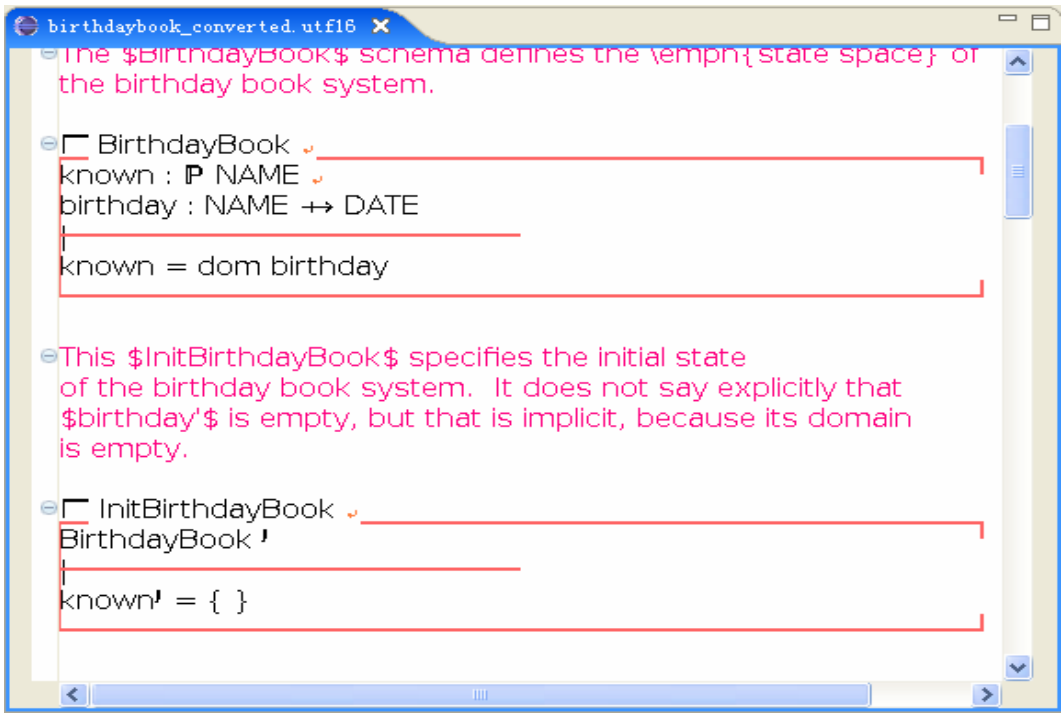


Figure 12.1 – The First Box Rendering Style

Figure 12.2 shows a screenshot of the boxes, using the second box rendering style, in the editor.

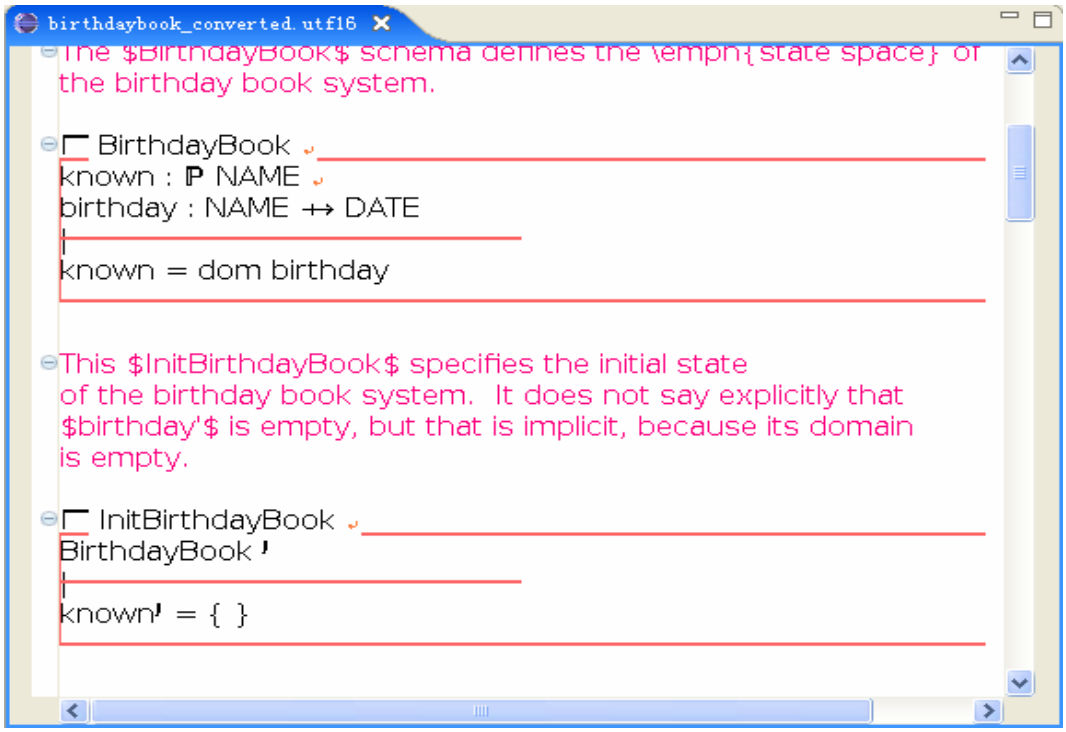


Figure 12.2 – The Second Box Rendering Style

12.1Design Issues and Alternatives

As discussed in the previous chapter, two other alternatives were also considered for drawing the real boxes during the development. The approach adopted is the easiest one and it follows the same convention as Eclipse.

An issue regarding the box rendering is that some Z users may be used to drawing boxes in one of the styles, while others are used to another style. The CZT Eclipse editor plug-in solves this by providing both box rendering styles, so that the Z users can choose the style they prefer.

12.2Implementation Techniques

As discussed in last chapter regarding the feature, drawing the box in the editor is adding certain annotations to each Z paragraph and then defining

special drawing strategies for those annotations. The ID of the annotation type used for drawing boxes is `net.sourceforge.czt.eclipse.schemabox`. The implementation of the box drawing involved the following steps:

1. After each parsing, the editor adds an annotation of the type `net.sourceforge.czt.eclipse.schemabox` to each Z paragraph via the editor's annotation model.
2. The annotations in Eclipse editors are displayed by a kind of painter called an *annotation painter*. A Java class extending `org.eclipse.jface.text.source.AnnotationPainter` may be created, but it is not required here. The CZT Editor simply creates an instance of this class in the `createControl` method so that the painter is created when the editor is created.
3. To control how the box annotations are displayed in the editor, a class implementing the interface `org.eclipse.jface.text.source.AnnotationPainter.IDrawingStrategy` was created. It is referred to as a drawing strategy here. As two styles of the boxes are needed, for each of the drawing styles, one subclass implementing the interface was created.
4. Then both drawing strategies were registered with the `AnnotationPainter` created in the first step by using the method called `AnnotationPainter.addDrawingStrategy(Object id, IDrawingStrategy strategy)`. Here each strategy was specified with a different id.
5. Since only one style is used at the same time, the annotation painter must know which style to be used as a particular time. This was done by setting the style when the user specifies it in the preference page and then updating the annotation painter to use the correct strategy by calling the `AnnotationPainter.addAnnotationType(Object`

`annotationType, Object drawingStrategyId)` method. If the annotation type is already registered with the annotation painter, then the new drawing strategy will replace the old one.

Chapter 13 Evaluation 2 – Usability

This is the second evaluation conducted during the development of the CZT Eclipse editor plug-in. The goal of the evaluation is to locate potential issues regarding the usability of the CZT Eclipse editor plug-in and also to find some new useful features desired by Z users.

Appendix A, B and C are the materials that were used during the evaluation. Appendix B is the question form for collecting the feedback from the participants. Table 13.1 shows the average responses to each numeric question.

No.	Description	Average	Range
4	Z familiarity	3.25	2-5
7	Eclipse familiarity	1	1-1
9.a	Issuing commands by menu selection	4	4-4
9.b	Issuing commands by command line input	3.5	2-4
9.c	Issuing commands by keyboard shortcuts	3.25	1-4
10.a	Error reporting by console output	2.75	2-4
10.b	Error reporting by error indicators in an editor	4.25	4-5
10.c	Error reporting by hover message in an editor	3.75	3-5
10.d	Error reporting by message boxes	4	3-5
10.e	Error reporting by specific panel	3.75	3-4
11.a	Easy to create LaTeX specifications	4.5	4-5
11.b	Helpful for browsing LaTeX specifications	4.5	4-5
11.c	Helpful for editing LaTeX specifications	4.25	3-5
11.d	Helpful for finding/fixing errors in LaTeX specifications	4	3-5
11.e	Understandability of LaTeX mode	4	3-5

12.a	Easy to create Unicode specifications	4	3-5
12.b	Helpful for browsing Unicode specifications	4.5	4-5
12.c	Helpful for editing Unicode specifications	4	3-5
12.d	Helpful for finding/fixing errors in Unicode specifications	4	4-4
12.e	Understandability of Unicode mode	3.25	2-4
14	Usefulness of constant background parsing	4.25	3-5

Table 13.1 – Statistics of the Evaluation Result

The evaluation exercise was done among 4 people (one female and three males in the 36-55 age ranges) from the computer science department, in School of Computing and Mathematical Sciences, The University of Waikato.

All of them have used most kinds of software available in the current computer world, such as word processors, databases, compilers and IDEs (Integrated Development Environment) for programming. They have all already known Z and one of them is expert at Z. All of them usually use **Emacs**, one of programmers' favorite editors, for editing Z specifications. The tool usually used by them for type-checking Z specifications is **Z/Eves** and **Jaza** is also occasionally used by one of them. Unfortunately, none of them already know Eclipse.

For issuing a command when programming, they all thought menu selection was useful. But, one of them disliked using the command line and another disliked using keyboard shortcuts.

Regarding the errors reported from a compiler, all people liked seeing them in a specific panel beside the editor, or inside the editor through indicators, or through hover messages or message boxes.

The participant feedback on the CZT Eclipse editor plug-in was generally positive and encouraging. They all agreed that the new interface makes it easy to create, browse and edit a Z specification written either in LaTeX or Unicode markup, and find and fix the errors in the specification. However, some of them did not

always understand what the editor is doing, because they were not familiar with the way Eclipse works.

Regarding the comparison of using the interface for between LaTeX specifications and Unicode specifications, the average response for question 11 (usability of the LaTeX mode) was 4.25, while the average response for question 12 (usability of the Unicode mode) was 3.95. However, in response to question 13, all of them said the Unicode mode of the interface was more helpful to use, 3/4 said the Unicode mode was more efficient to use and 3/4 said the Unicode mode was more enjoyable to use. The lower average for question 12 was mostly caused by question 12.e (understandability), which may be because all four people were familiar with the LaTeX markup for Z, whereas the Unicode markup was new to them. All of them think that the constant background parsing is very useful when editing a Z specification.

Here are some of the specific feedback points from the evaluation exercise (All text in italics is feedback from the people attending the evaluations – names have been removed to preserve anonymity):

Advantages:

- *The editor is very interactive for editing a Z specification.*
- *The constant background parsing and type-checking is very helpful when editing a Z specification. In particular, there is no need to use another program to parse and type-check a Z specification.*
- *The outline is very useful for browsing a Z specification. It can make the cursor jump to different sections. The "Go To Declaration" command is also very helpful because it makes the cursor be able to jump to the definition part of a name.*
- *The editor gives good feedback to us when editing a specification. The error highlighting support is particularly useful for seeing the errors that the editor contains.*
- *The conversion commands make us be able to see 2-dimentional layout (two different formats) of the same specification immediately.*

- *The Z Character Map view is very helpful if we can not remember the LaTeX word of a Unicode Z character.*

Disadvantages and suggestions:

- Someone said that, "*The constant parsing feature is useful if I have a well defined specification. But when starting from scratch, lots of errors will exist until I have defined enough, so the error highlighting will be annoying and encourage me to ignore the errors. It would be better to provide some way to disable the auto-parsing*".

  Regarding this issue, the CZT Eclipse editor plug-in now provides some preference settings related to the parsing support. One setting allows the user to completely disable the background parsing. Another two settings give the user two options for error reporting. They allow the user to choose whether the editor reports to the user constantly or only when the editor is saved.
- Someone said that, "*It would be really nice to make the editor integrate with a proof-tool*".

  This could probably be solved in the future by providing a set of commands for proving Z formulas.
- Someone said that, "*If one is very familiar with using LaTeX currently, it is quicker to type in the LaTeX word than to find and select the character symbol from the Z Character Map view*". Another person said that, "*It would be better to make the editor easier to enter Unicode characters quickly for novice and advanced users*".

  As the Z Character Map view contains lots of Unicode characters which are probably used in Unicode Z specifications, it is sometimes hard to find a particular character in the table, although the categories can help. However, it is hard to find some way to solve this issue. One possible option for advanced users would be to provide a set of commands with keyboard shortcuts for entering some frequently used characters, so that it can make entering these characters quickly.

- Someone said that, "*When editing a Unicode Z specification, the Z 'prime' character is confusing and it slows down typing*".

  This is a problem of the **ISO Z Standard** (ISO/IEC 13568, 2002), which requires a special prime character in the Unicode markup. This could probably be solved in the future by providing a "smart insert" function, which converts the usual prime character to Z-specific Unicode character when the user types the usual prime character into a Unicode Z specification being edited.

- Someone said that, "*It would be better to make us be able to see the Unicode character after its corresponding LaTeX word is typed in, so that we can simply type the LaTeX word to see its Unicode format if we are more familiar with the LaTeX word. It would be better to make the LaTeX mode of the editor display the Unicode characters just as in the Unicode mode*".

  For the Unicode mode, if the user types the latex word in the editor, the editor could be changed to automatically replace the LaTeX word with its corresponding Unicode character.

  For the LaTeX mode, it would be hard to display the Unicode character of a LaTeX word. If the LaTeX words are replaced with Unicode characters, the parser would not parse the LaTeX Z specification correctly as it assumes the editor is in LaTeX mode, so it would not recognize the Unicode character as a LaTeX word. In addition, the location information for the characters after a Unicode character converted from a LaTeX word would not be correct.

- Someone said that, "*It would be better to provide LaTeX and Unicode views so that the same document can be edited in both markups at the same time*".

  In Eclipse, this could probably be solved by implementing an editor with multiple pages with one page for LaTeX and one page for Unicode. Both pages will synchronize with each other to ensure they contain the

same contents. With the multi-page editor, the user could easily switch to different page to edit the same specification in both LaTeX and Unicode modes. However, the parser used in the plug-in can not convert a specification written in a format to another properly if the specification contains errors. Due to this limitation, this issue can not be solved at the moment, but could be solved in the future if the parser is changed to support the conversion.

The overall feedback from this evaluation is encouraging. The suggestions on future improvement are very helpful. The ability to turn off the parsing or error reporting has already been implemented and it could be useful to implement some of the other suggestions in the future.

Chapter 14 Preference Settings

Most Eclipse plug-ins contribute their own sets of preference pages to Eclipse. These pages provide the preference settings, which are to be configured by the end users to control the behaviors of the plug-in, for example, how the plug-in displays information. Eclipse provides plug-in developers with an extension point for contributing new preference pages. Once some pages are added, it can then display the pages to the user and save the values across Eclipse sessions. To give Z users some flexibility of using the CZT plug-in, the plug-in contributes a set of preference pages to Eclipse as described below.

1. Compiler Preference Page (see Figure 14.1)

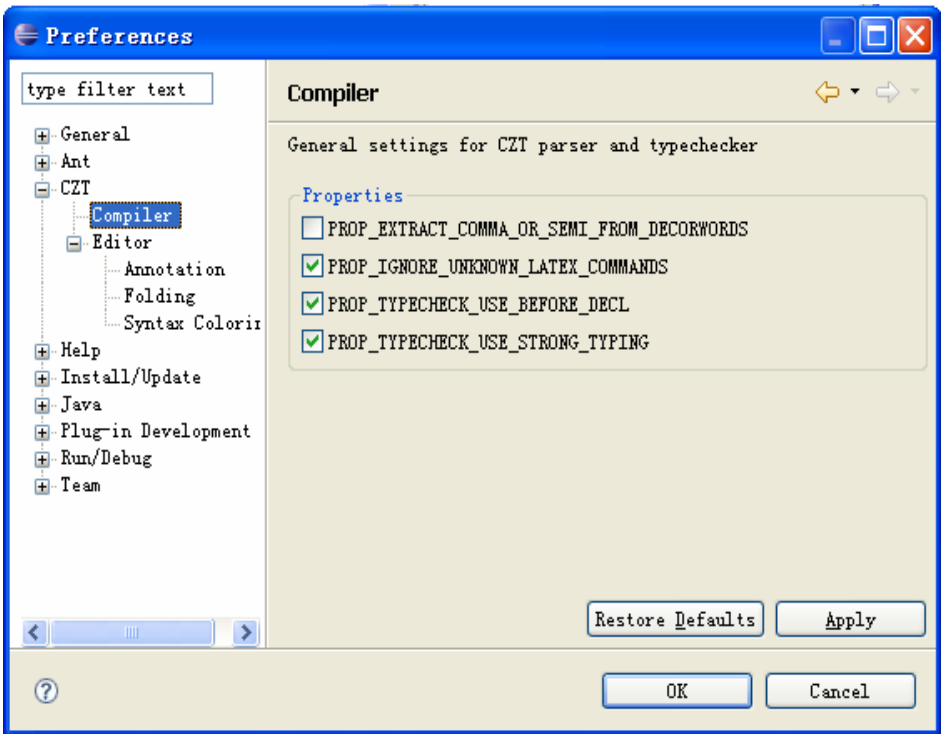


Figure 14.1 – Compiler Preference Page

This page offers the settings for the properties of the parser and type-checker.

2. Editor Preference Page (see Figure 14.2)

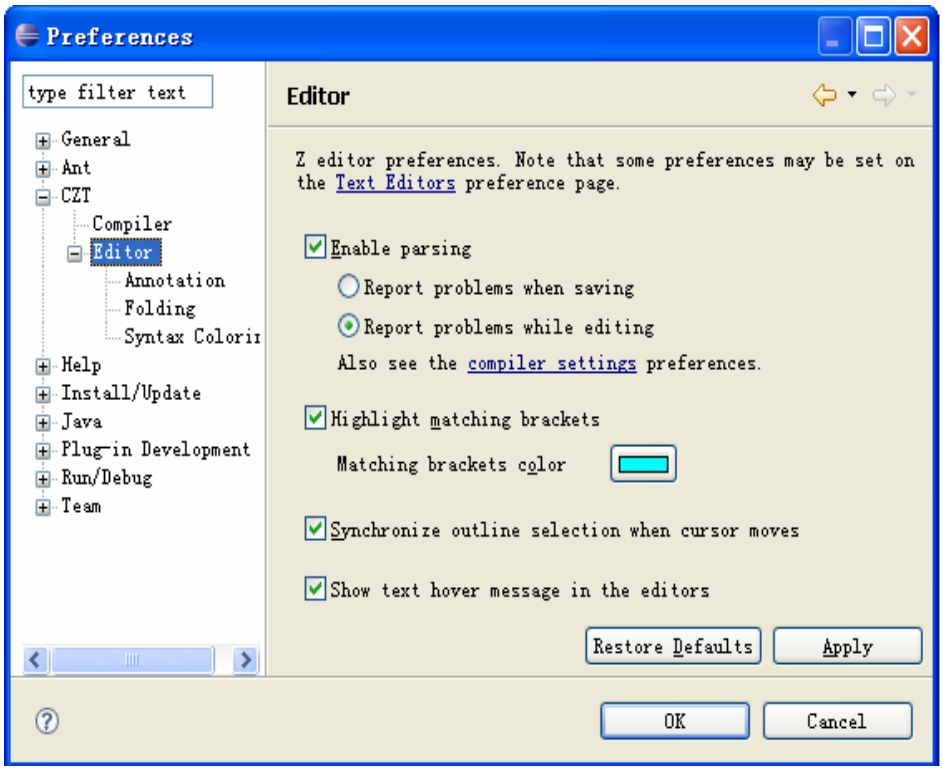


Figure 14.2 – Editor Preference Page

This is the main page for the settings of the CZT Editor. This page provides the following settings:

- Enable/disable the parser in the editor. If it is enabled, the parser runs constantly in the background each time a file is opened or the text in the editor is changed. If it is disabled, then those parser-dependent features, for example the content outline and hover messages, will be unavailable.
- When the parser reports problems to the user. If a specification is written from the start, it may usually contain many errors if the parser is enabled. The user may be annoyed by the error indicators. To avoid this, the user can make the parser report problems only when the file is saved. As the parser still runs in the background, the outline view can still be updated and the user can still get the hover information.
- Enable/disable the bracket matching support

- Setting the color used for drawing the annotation of a matching bracket.
- Whether the outline view synchronizes its selection with the selection in the editor.
- Whether to display relevant information to the user when the mouse is hovering in the editor.

3. Annotation Preference Page (see Figure 14.3)

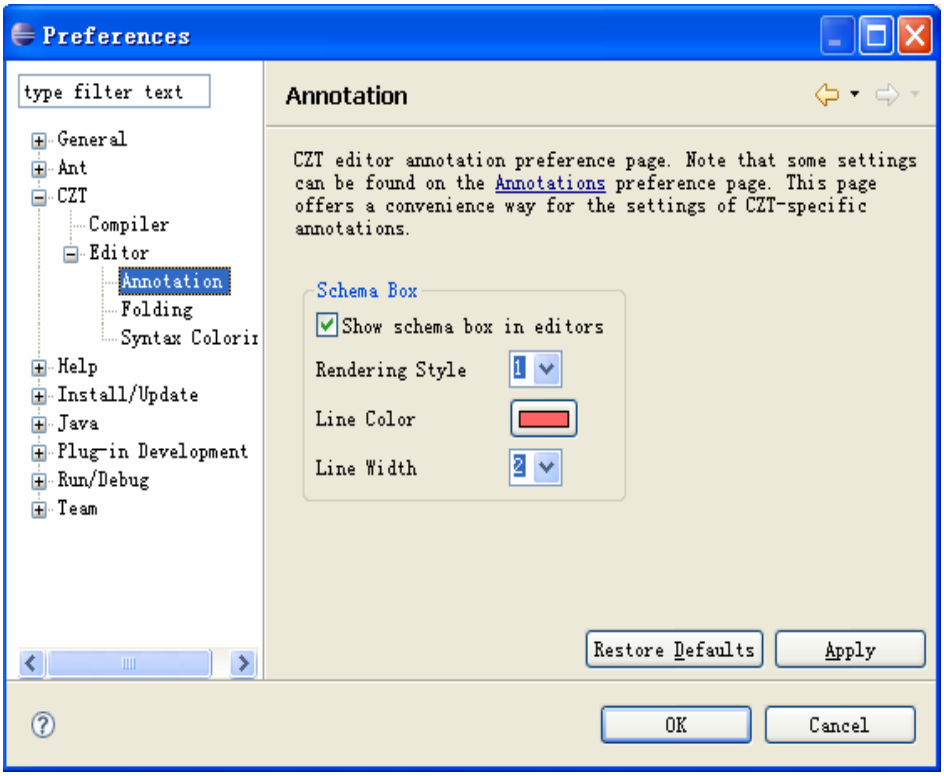


Figure 14.3 – Annotation Preference Page

This page provides the user with the ability to enable or disable the support of rendering boxes around schemas, as well as set the style of the box, the color and width of the line in the box.

4. Folding Preference Page (see Figure 14.4)

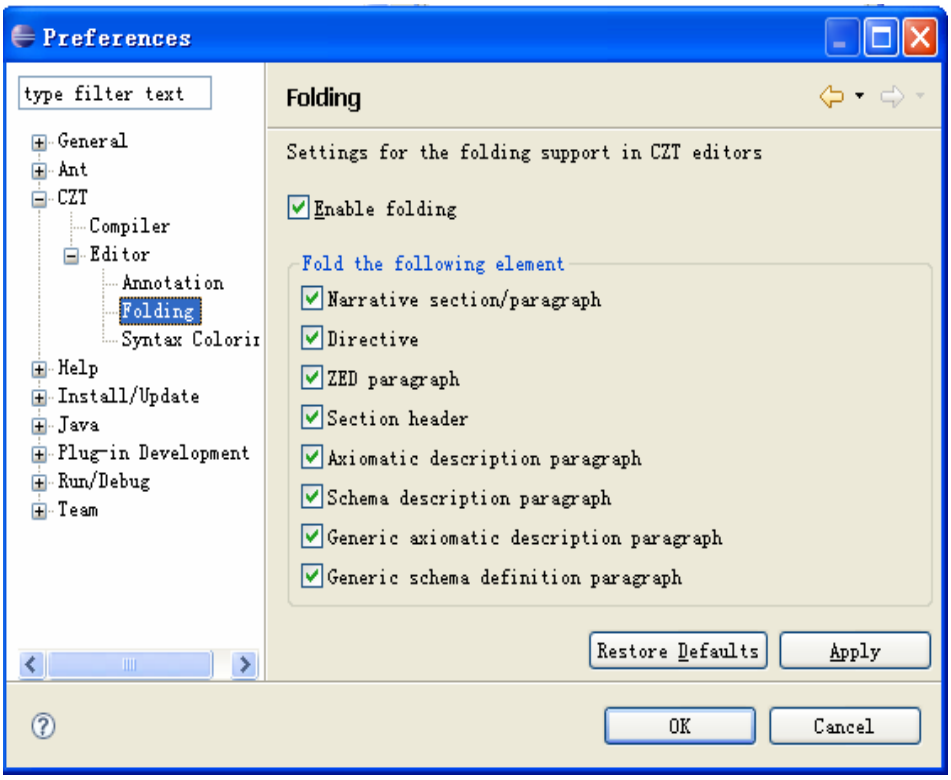


Figure 14.4 – Folding Preference Page

This page provides the users with the ability to enable or disable the folding support (see Chapter 5). In addition, it also allows the user to specify which kinds of element can be folded (hidden) in the editor.

5. Syntax Coloring Preference Page (see Figure 14.5)

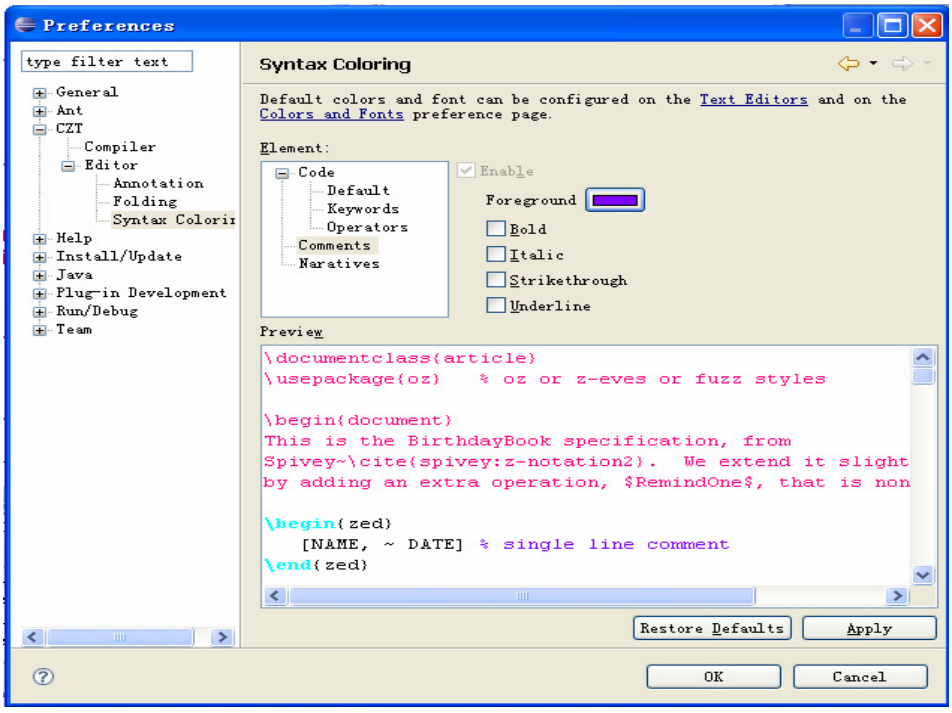


Figure 14.5 – Syntax Coloring Preference Page

This page provides the user with the ability to change the color and style for different elements in the editor, so that the user can feel comfortable with their preferred color and style.

Chapter 15 Conclusion

The CZT Eclipse editor plug-in integrates Z support into Eclipse, which is one of the favorite editors of programmers and is increasingly used as a general platform for end-user applications. It provides Z users with a visual WYSIWYG editor. This editor provides Z users with a set of useful features, such as the syntax coloring and folding support. It also runs a parser in the background for parsing and type-checking Z specifications without needing other programs or commands to parse and type-check Z specifications. In addition to the editor, the plug-in also provides a set of useful Eclipse views (panels) for browsing and editing Z specifications, such as the **Outline** view, the **Z Character Map** view and the **Z Conversion** view.

Currently, about 130 Java classes (excluding the CZT library) have been created for the CZT Eclipse editor plug-in. These Java classes contain about 950 public methods, 160 protected methods and 235 private methods. They contain about 24600 lines of code (including Java comments and blank lines). This is a large amount of code, but most of it seems necessary in order to obtain rich functionality that is similar to the JDT Eclipse environment for Java programming.

During the development, two evaluations were conducted. One is the design and feature evaluation, while the other is the usability evaluation. The feedback from both evaluations is very positive and encouraging. There are two major features which the participants like. One is the constant parsing support, because it constantly reports specification problems in the background without needing any manual operations. The other is the **Outline** view. It provides an overview of a Z specification with icons indicating the types of each section and paragraph. It also provides navigation support which helps the user to quickly jump to different

sections or paragraphs in the editor.

So far, the CZT Eclipse editor plug-in has met all the development goals, as stated in Section 1.3. Several additional features were also designed and implemented, including:

- The ability to highlight a term. This idea came from evaluating CADiZ. It is very useful because it allows Z users to explore the structure of Z expressions (for example, the precedence of operators) and show the type of each sub-expression.
- The conversion from one markup to another. This is a useful learning tool because it allows users who are familiar with one markup to see the same specification in another markup.

However, the CZT Eclipse editor plugin also has some limitations as described during the discussion of the evaluations. One of the major limitations is that it can be hard to find a particular Unicode Z character, although the **Z Character Map** view can help somewhat. As described in Section 1.3, this issue could be solved in the future by implementing commands with keyboard shortcuts for entering some of the frequently used characters, or by using LaTeX macro names as keyboard shortcuts for entering Unicode characters.

As stated in the sections regarding future implementations and the evaluation feedback, some improvements and new features have also been considered for possible implementation in the future. In addition, more commands for browsing and analyzing Z specifications will be added in the future, so that Z users can use the plug-in to do more semantic analysis of Z specifications. This will save users much time, since it will allow them to use the commands from within Eclipse, rather than switching to other tools.

However, the CZT Eclipse editor plug-in already provides a sophisticated modern environment for editing and analyzing Z specifications. It is the only Z environment that provides constant background parsing and typechecking, which gives users quick feedback about errors.

References

Arthorne, J., & Laffra, C. (2004). *Official eclipse 3.0 FAQs*. Boston, San Francisco, New York, London, Munich, Paris, Madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City: Addison-Wesley.

Bowen, J. (2005, September 15). *The Z notation*. UK: ZB2005. Retrieved April 15, 2006 from the World Wide Web: http://vl.zuser.org/#standards

Clayberg, E., & Rubel, D. (2004). *Eclipse: Building Commercial-Quality Plug-ins*. Boston, San Francisco, New York, Toronto, Montreal, London, Munich, Paris, Madrid, Capetown, Sydney, Tokyo, Singapore, Mexico City: Addison-Wesley.

D'Anjou, J., Fairbrother, S., Kehn, D., Kellerman, J., & McCarthy, P. (2005). *The Java™ Developer's Guide to Eclipse* (2nd ed.). Boston: Addison-Wesley.

Deva, P. (2005, March 11). *Folding in Eclipse Text Editors*. Retrieved June 12, 2006 from the World Wide Web: http://www.eclipse.org/articles/Article-Folding-in-Eclipse-Text-Editors/folding.html

"Eclipse Java Development (JDT) Subproject". (2006). Retrieved October 15, 2006 from the World Wide Web: http://www.eclipse.org/jdt/

Edgar, N., Haaland, K., Li, J., & Peter, K. (2004, February). *Eclipse User Interface Guidelines*. Retrieved December 26, 2005 from the World Wide Web: http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html

Ho, E. (2003, June). *Creating a text-based editor for Eclipse 2.1*. Retrieved February 21, 2006 from the World Wide Web:

http://devresource.hp.com/drc/technical_white_papers/eclipeditor/index.jsp

Hoffmann, M.R. (2006, April 14). *Eclipse Workbench: Using the Selection Service*. Retrieved July 3, 2006 from the World Wide Web: http://www.eclipse.org/articles/Article-WorkbenchSelections/article.html

"Introduction". (2006). Retrieved March 25, 2006 from the World Wide Web: http://texlipse.sourceforge.net/manual/intro.html

ISO/IEC 13568. (2002, July 1). *Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics* (1st ed.). Switzerland: ISO/IEC 2002.

"jEdit". (2006). Retrieved November 23, 2006 from the World Wide Web: http://sourceforge.net/projects/jedit/

"jEdit – Programmer's Text Editor". (2006, November 8). Retrieved November 23, 2006 from the World Wide Web: http://www.jedit.org/

Jacky, J. (1997). *The Way of Z: Practical Programming with Formal Methods*. UK: Cambridge University Press.

Malik, P., & Utting, M. (2005). *CZT: A Framework for Z Tools*. New Zealand. Retrieved June 21, 2006 from the World Wide Web: http://www.cs.waikato.ac.nz/~marku/papers/zb2005_czt.pdf

Pandit, C. (2005, November 15). *Make your Eclipse applications richer with view linking*. Retrieved July 3, 2006 from the World Wide Web: http://www-128.ibm.com/developerworks/library/os-ecllink/index.html

"TeXlipse". (2006). Retrieved March 25, 2006 from the World Wide Web: http://texlipse.sourceforge.net/

Toyn, I. (2005, Sep 2). *What does CADiZ do?*. UK. Retrieved April 18, 2006 from the World Wide Web: http://www-users.cs.york.ac.uk/~ian/cadiz/whatiscadiz.html

Toyn, I. (2006, April 11). *Home page – release 4.5*. UK. Retrieved April 18, 2006 from the World Wide Web: http://www-users.cs.york.ac.uk/~ian/cadiz

"Welcome to CZT". (2006, August 15). Retrieved September 10, 2006 from the World Wide Web: http://czt.sourceforge.net/

Zoio, P. (2006, April). *Building an Eclipse Text Editor with JFace Text*. Retrieved June 20, 2006 from the World Wide Web: http://www.realsolve.co.uk/site/tech/jface-text.php

Appendix A. Instructions for the Usability Experiment

The objective of the experiment is to find out how users use an editing tool to write a Z specification and to locate the issues in the new Z environment for Eclipse. It is strongly recommended that you can provide feedback containing both good and/or bad points of using the tool. Your suggestions for future improvements are always welcome.

These instructions will show you the steps of editing Z specifications using a modern IDE (Integrated Development Environment), Eclipse. Please use the reference article, "*Reference for the CZT Eclipse Interface*", provided with the instructions, or ask the interviewer, when you have questions about the interface.

Section A - Installation and Project Creation

1. Install Java 1.5 or higher (if required).
2. Download Eclipse SDK 3.2 from www.eclipse.org and install it by extracting it somewhere you prefer.
3. Install the CZT plug-in by downloading it from the web page http://www.cs.waikato.ac.nz/~marku/czt/eclipse.html into the "plugins" folder inside your Eclipse SDK.
4. Start Eclipse with the command "eclipse –clean".
5. Create a CZT project by using the menu entry "***File/New/CZT Project***" and open the CZT perspective if it is not opened yet.

Section B – Preparation

6. Look at the screenshot of the CZT interface to Eclipse in the "***Browsing and Editing Z Specifications***" section of the reference article and find the "Outline Panel" and the "Z Character Panel". You will be using these GUI

features later.

Section C – LaTeX Markup

7. Create a Z specification called "***bbook***" with **LaTeX** Markup using the menu entry "***File/New/Z Specification***".
8. For simplifying your experiment, several specifications are provided to you with the instruction. Copy and paste the contents of the file "bbook.zed" from the webpage into the ***bbook.tex*** specification you just created.
9. Browse the specification using the outline panel. Try to find the ***NotKnown*** schema using the outline panel.
10. Hide/show the ***NotKnown*** schema in the editor by using the Fold/Unfold icons (+/-).
11. Introduce an error in the editor: in the ***NotKnown*** schema, change ***\notin*** to ***\neq*** so that it will generate a "*Type Mismatch*" error. You can insert the ***\neq*** by using the **ZCharMap** panel or by typing it on the keyboard.
12. Inspect the error: in addition to the indicators and problems view introduced in the reference article, you can see the information about the error by hovering above the error in the editor.
13. Investigate the cause of the error by clicking inside a word and then using the term highlight commands "***Edit/Highlight/Enclosing Term***" to see the types of both the left and right hand side of the operator ***\neq***.
14. Correct the error by changing the operator ***\neq*** back to ***\notin***.
15. Save the file.
16. Add a new schema called ***DeleteBirthday*** after the ***FindBirthday*** schema. Use the ***ZCharMap*** panel to insert a schema and then change it as below:

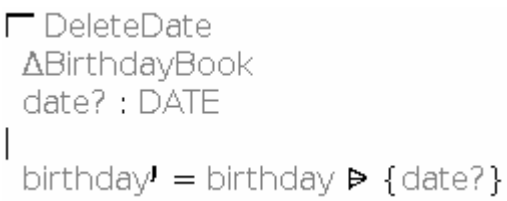
```
\begin{schema}{DeleteBirthday}
    \Delta BirthdayBook \\
    name?: NAME \\
\where
```

```
    birthday' = \{name?\} \ndres birthday
\end{schema}
```

17. Click on the name "***NAME***" will enable all the references of the name to be highlighted in the editor. Then use the menu entry "***Edit→Go To Declaration***" to jump to the definition part of the "***NAME***" variable.
18. Save the file

Section D - Unicode Markup

19. Create another specification using the name "***bbook2***" with **Unicode** markup and UTF16 encoding.
20. Copy and paste the contents of the file "bbook.utf16" from the webpage into the ***bbook.utf16*** specification you just created
21. Open the specification you just modified in section C. Execute the command "***Edit/Convert To/Unicode***". Then you can see the converted contents in the Conversion panel under the editor. Roughly compare it with the specification you just created. They should have nearly the same contents.
22. Browse the specification using the outline panel. Try to find the ***NotKnown*** schema using the outline panel.
23. Hide/show the ***NotKnown*** schema in the editor using the Fold/Unfold icons.
24. Introduce an error in the editor: in the ***NotKnown*** schema, change the name ***known*** to ***know*** so that it will generate an "*Undeclared Name*" error. Hover above the name "***BirthdayBook***" to see the type of this schema and the names that it declares.
25. Add the following schema using the **ZCharMap** panel:



Note that the Z standard prime character in Unicode, as seen in the above schema, is a special math prime character, not the usual ’.

26. Save the file.

Appendix B. CZT Eclipse Usability Evaluation Questions

Part I – General questions (regarding the participant's background)

1. Which age group do you belong to (circle one)?

   Under 25 **25-35** **36-45** **46-55** **Above 55**

2. What is your gender (circle one)? **Male / Female**

3. Which types of software have you used before (tick all that apply)?

   Word processor _____

   Database _____

   Spreadsheet _____

   Online help system _____

   Compiler _____

   IDE (Integrated Development Environment) for programming _____

4. To what extent are you familiar with Z language (circle one - 1 indicates unknown and 5 expert)?

   Unknown 1 2 3 4 5 Expert

5. What editor(s) do you usually use for editing Z?

   _____

   _____

   _____

6. What tool(s) do you usually use for type-checking/analyzing Z specifications?

   _____

   _____

   _____

7. To what extent are you familiar with Eclipse (circle one - 1 indicates unknown and 5 expert)?

   Unknown 1 2 3 4 5 Expert

8. What do you use Eclipse for?

_____

_____

_____

9. Please rank the usefulness of these methods of issuing a command when programming (circle one - 1 indicates not useful and 5 essential).

Menu selection	**Not Useful**	**1**	**2**	**3**	**4**	**5**	**Essential**
Command line	**Not Useful**	**1**	**2**	**3**	**4**	**5**	**Essential**
Control key accelerator	**Not Useful**	**1**	**2**	**3**	**4**	**5**	**Essential**

10. Please rank the usefulness of these methods of reporting an error when programming (circle one - 1 indicates not useful and 5 essential).

Command line	**Not Useful**	**1**	**2**	**3**	**4**	**5**	**Essential**
Error indicator in an editor	**Not Useful**	**1**	**2**	**3**	**4**	**5**	**Essential**
Hover message in an editor	**Not Useful**	**1**	**2**	**3**	**4**	**5**	**Essential**
Message Box from an editor	**Not Useful**	**1**	**2**	**3**	**4**	**5**	**Essential**
Specific panel beside an editor	**Not Useful**	**1**	**2**	**3**	**4**	**5**	**Essential**

Part II – Editing LaTeX Markup Specifications

11. Indicate how much you agree with the following statements. (1 indicates complete disagreement and 5 complete agreement.)

The interface makes it easy to create a new LaTeX Z specification.

Disagree 1 2 3 4 5 Agree

The interface helps me to browse a LaTeX Z specification.

Disagree 1 2 3 4 5 Agree

The interface helps me to edit a LaTeX Z specification.

Disagree 1 2 3 4 5 Agree

The interface helps me to quickly find and fix the errors in the specification.

Disagree 1 2 3 4 5 Agree

I always understand what the system is doing.

Disagree 1 2 3 4 5 Agree

Part III – Editing Unicode Markup Specifications

12. Indicate your agreement or disagreement with the following statements. (1 indicates complete disagreement and 5 complete agreement.)

The interface makes it easy to create a new Unicode Z specification.

Disagree 1 2 3 4 5 Agree

The interface helps me to browse a Unicode Z specification.

Disagree 1 2 3 4 5 Agree

The interface helps me to edit a Unicode Z specification.

Disagree 1 2 3 4 5 Agree

The interface helps me to find and fix the errors in the current specification in time.

Disagree 1 2 3 4 5 Agree

I always understand what the system is doing.

Disagree 1 2 3 4 5 Agree

Part IV – Comparison

13. Which part of the interface (circle one) was more:

Helpful to use	**LaTeX / Unicode**
Efficient to use	**LaTeX / Unicode**
Enjoyable to use	**LaTeX / Unicode**

14. Please rank the usefulness of the constant parsing in the background compared to other ways of parsing when editing Z specifications (1 indicates unsatisfied and 5 most useful).

    Unsatisfied 1 2 3 4 5 Most useful

15. What are the advantages and disadvantages of the CZT interface to Eclipse compared to your usual Z editor?

    Advantages:

    _____

    _____

    _____

    Disadvantages:

    _____

    _____

    _____

Part V - Suggestion

16. Can you suggest any improvements to the interface (specify below, on the back, or on other paper if more space is required)?

    _____

    _____

    _____

    _____

    _____

    _____

    _____

    _____

Thanks for your participation!

Appendix C. Reference for the CZT Eclipse Interface

The experiments require the participants to try the new Z GUI interface to Eclipse. As Eclipse is becoming more and more popular, a useful Z interface to Eclipse is desired for the widespread use of Z. One of the main issues for editing a Z specification is to insert a Z specific Unicode character into the editor. The new interface will be able to provide Z users with much ease of editing and thus enable them to concentrate on the semantics of Z specifications, rather than the editing effort.

This article aims to help the experiment participants understand the concepts involved in the instructions on the experiments. It is strongly recommended that all participants read it first before starting the experiments. However, this article will not strictly follow the experiment instructions. This article can be used as a reference manual during the experiments.

Here are some of the important concepts involved.

Installation

Installation of Eclipse SDK: download it from www.eclipse.org and install it by extracting it somewhere you prefer.

Installation of CZT plug-ins:

http://www.cs.waikato.ac.nz/~marku/czt/eclipse.html contains the link for downloading the CZT plug-in. For installing the CZT plug-in, you need download and copy it into the plug-ins folder called “plugins” inside your Eclipse SDK root directory.

Installation of CZT Font (optional): Ideally, you can have the CZT font installed on the computer if you have the privilege to do that. Otherwise, you may not have all Unicode characters recognized by the editor when you are editing a Unicode Z specification.

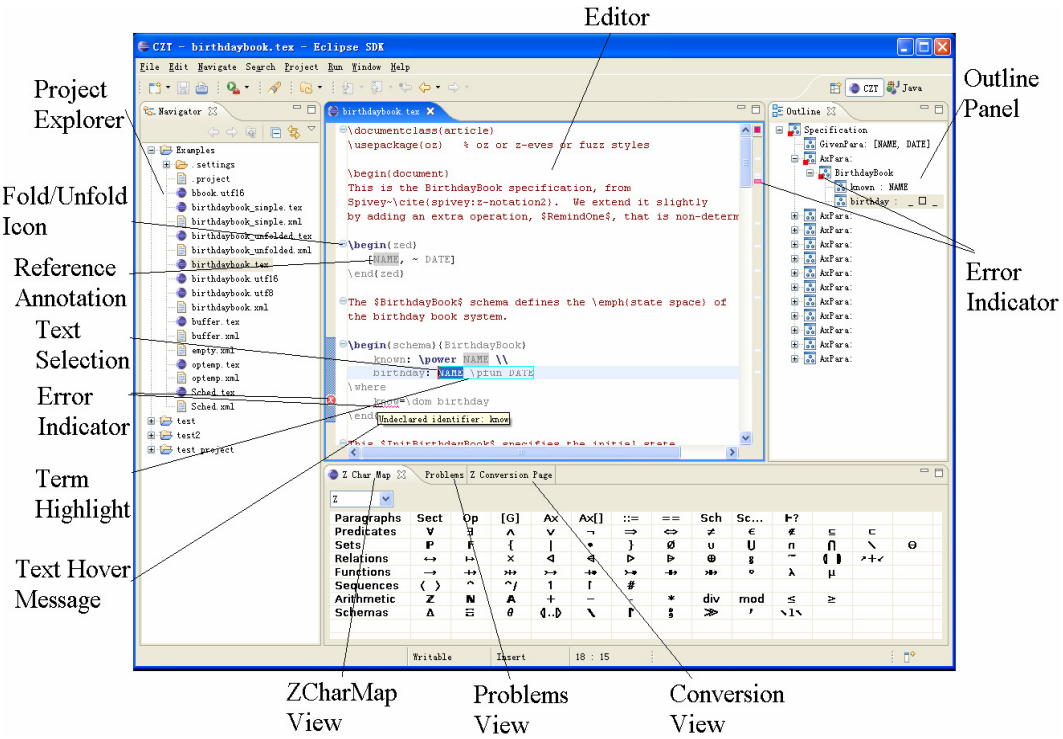
Running Eclipse: if you are running into some problems, try to start Eclipse with "-clean" parameter, that is "eclipse –clean". Due to the lazy start nature of Eclipse, the old configuration will remain unchanged inside Eclipse even you have remove some of the plug-ins you installed before. What the "-clean" parameter does is to clean all previous configuration before Eclipse start and to configure it with the installed plug-ins only.

Creating Z Specifications

CZT project creation wizard: the wizard can be started using the menu entry *File→New→CZT Project*. If you are asked whether you want to open the CZT perspective, just select "Yes". Opening the CZT perspective will help you start editing Z specifications with some useful views or panels opened automatically.
Z specification creation wizard: the wizard can be started using the menu entry *File→New→Z Specification.* Within the wizard, you can specify the markup and/or encoding that the new specification uses. Once it is finished, the new specification will be opened automatically in Eclipse. See the picture below for an overview of the Eclipse interface when a Z specification is opened.
Project/File import wizard: You can import an existing Eclipse project or file into an Eclipse project using the import wizard under the menu *File→Import*.

Browsing and Editing Z Specifications

Here is a screenshot of the CZT interface to Eclipse:



Content outline: when this panel is opened, it is initially positioned on the right of the editor part as shown in the above picture. The outline panel will show you the sections, the paragraphs and the variable declarations in the current specification. The outline page will synchronize with the editor in the following way. If the cursor position is changed, the corresponding item in the outline page will be selected. If an item is selected in the outline page, the name of the section, the paragraph or the variable will be selected in the editor.

Z Character Map panel: At the bottom of the window, you can see a panel containing a table of characters. Each click on a character in the table will result in inserting corresponding contents into currently active editor, but the left column is not clickable because it is used to display the category of the characters on the corresponding row. The panel will automatically choose the correct contents to insert depending on the markup and encoding of the specification in the editor. The panel can also show you a floating panel containing a description of a character when you are hovering above it.

Z partitions: a Z specification is partitioned based on its containing paragraphs. When the cursor position is changed, a vertical indicator on the left of the editor

(the blue vertical bar in the picture) shows you the range of the paragraph that the cursor is in.

Hiding/showing paragraphs: on the left-top corner of each paragraph is a "+" or "-" sign. Each click on it will result in hiding or showing the corresponding paragraph.

Syntax highlighting: The editor shows you the contents of the Z specification with syntax highlighted. You can change the colors under "CZT/Editor/Syntax Highlighting" category in the preferences page.

Variable references: When the cursor is positioned on a variable, all references to the variable will be highlighted. The menu bar command "*edit/Go To Declaration*" can lead you to select the declaration part of the variable.

Brackets matching: when the cursor is positioned after a bracket, the matched open or close bracket will be highlighted.

Selection by double click: As shown in the above picture, the selected text is presented with white foreground and blue background. Double click in the editor will enable a selection of a syntactical word at the current cursor position in the editor. Specifically, if double click after a bracket, you can select the contents inside the matched brackets.

Term highlight: you can execute the command "edit/Highlight/Enclosing Term" to highlight the text associated with the enclosing element containing the element at current position of the cursor. The highlight mode is different from normal text selection as shown in the above picture. The corresponding term is presented with yellow background and its normal foreground. The command "edit/Highlight/Restore Last Highlight" is used for highlighting the term which is highlighted previously.

Problems report: when a Z specification is being edited, it is parsed constantly in the background without interrupting the editing. Once any problems, such as errors or warnings, are found, they are immediately reported to the user in a variety of ways. Particularly, there is a Problems view at the bottom of the window, in which you can see a list of problems found. You can also see some

red crosses on both side of the editor. The crosses on the left indicate there are errors corresponding to the lines they are positioned on. The crosses on the right indicate the same errors, but corresponding to the relative position to the whole document. On the top-right corner is a special indicator which shows the number of problems found in the specifications.

Editor hover support: When the mouse is hovered above the editor, the editor will present some information to users if there is any useful information which could be concerned with by the users at current position of the mouse. If a term is highlighted there, the information about it will take the highest priority. The priority sequence for other information is error/warning, type, something else.

Z Conversion View: A specification can be browsed using another markup or encoding in a view at the bottom, so that users can make comparison of the specification between different markups and/or encodings. Their corresponding commands are in the “edit/Convert To” menu entry. The conversion view also show you the converted contents with syntax highlighted so that you can compare them easily.

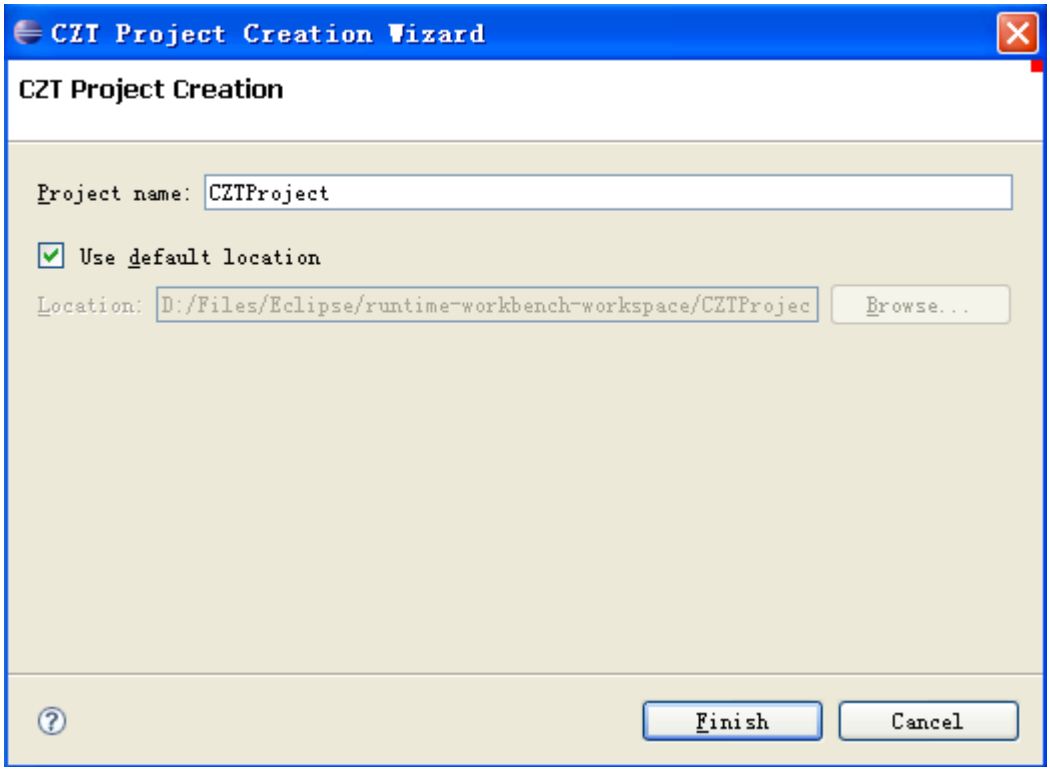
As the Z interface to Eclipse is still under development at the time of writing the instruction, it may be changed to behave in a slightly different manner.

Appendix D. User Documentation

This is a copy of the online documentation for the CZT Eclipse environment.

Create New CZT Projects

The CZT editor plug-in provides a convenient wizard to help create a new project for editing Z specifications. The wizard can be opened using the menu entry **File > New > Project... > CZT > CZT Project**.

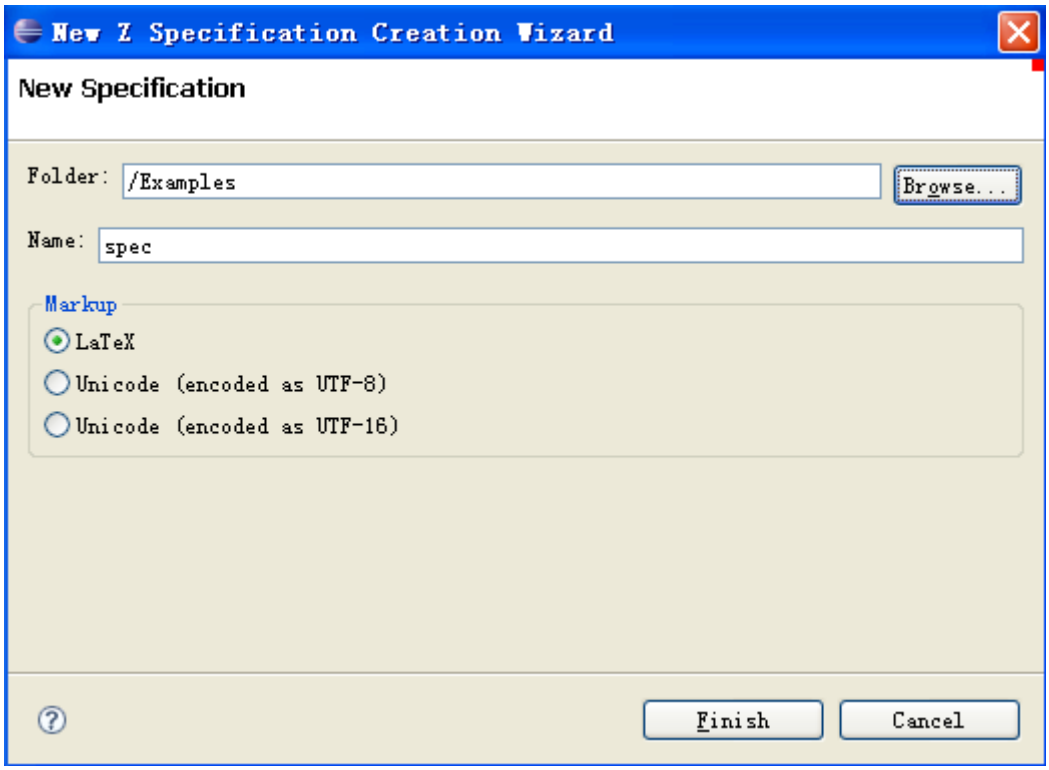


- In this wizard, enter the name of the new project.
- If you do not want to use the default location, specify one.

If the CZT perspective is not opened at the moment, the wizard will ask you whether you want to open the CZT perspective. It is highly recommended that you open it because it will automatically open a set of Eclipse elements useful for the CZT development. Then the new project will be created and selected in the workbench window.

Create New Z Specifications

The CZT editor plug-in provides a convenient wizard to help create a new Z specification. The wizard can be opened using the menu entry **File > New > Other... > CZT > Z Specification**.

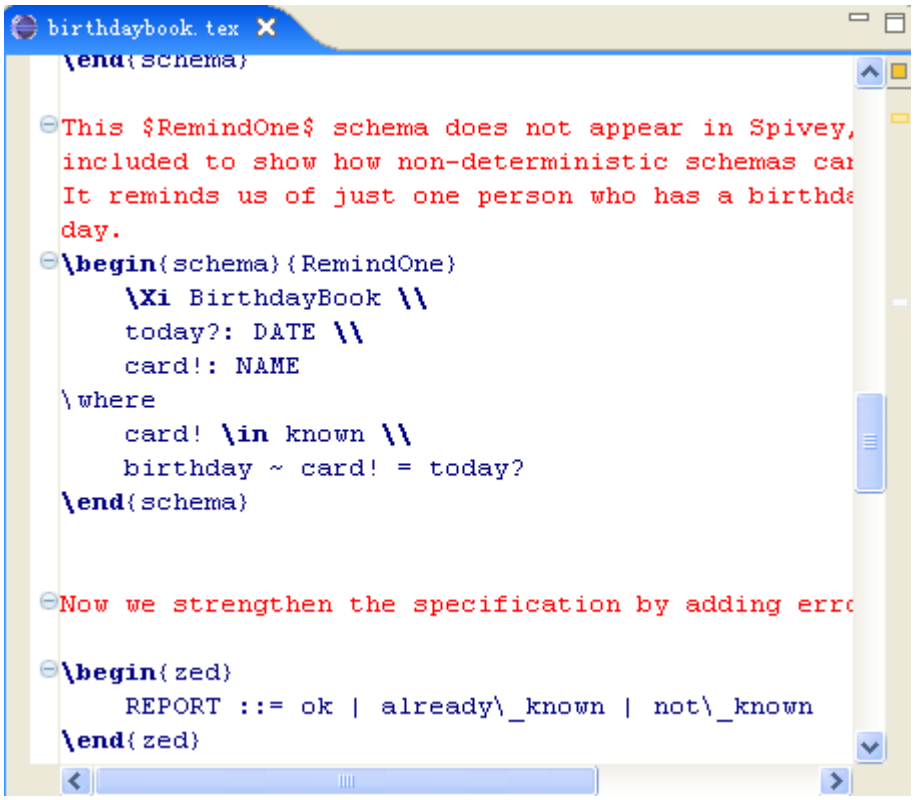


- In this wizard, enter the name of the new Z specification.
- If you selected a folder in the **Navigator** view, the default location will be the folder. Choose one folder for the new specification.
- As the Z specification can be written using LaTeX, Old LaTeX or Unicode markup, you need to specify a markup for the new specification. Then the name of the new specification will be the name you specified above appended by the extension corresponding to the markup you selected.

Then the new specification will be created with some initial contents and opened in the workbench window.

Syntax Coloring

The CZT Editor provides syntax coloring support. The Z elements, such as keywords, operators and narrative paragraphs, are displayed in different colors. The below figure shows a screenshot of the feature.

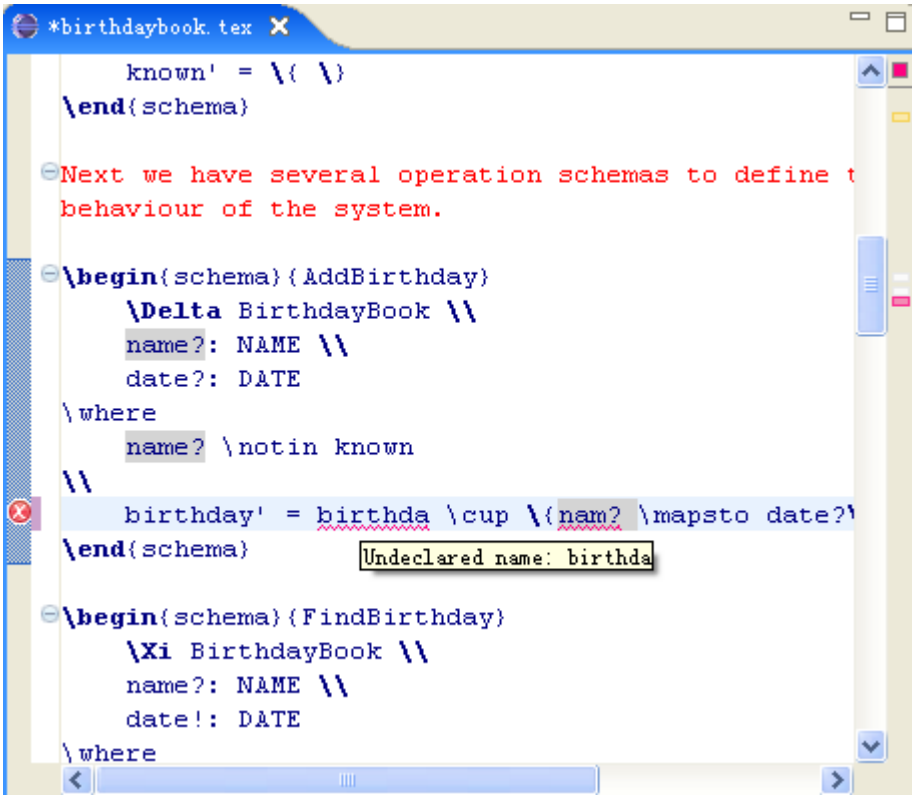


The users can change the colors and styles of these elements in the Syntax Coloring preference page.

Identifying Problems

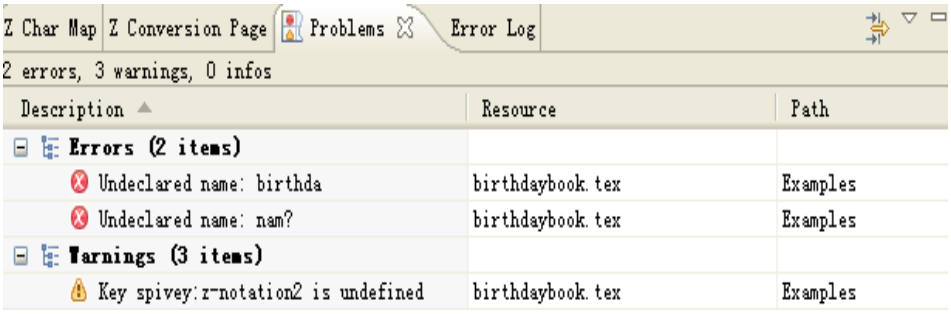
After the current editing specification is parsed and typechecked, the editor will report the problems to the user in several ways.

1. Display the problem markers within/around the editor



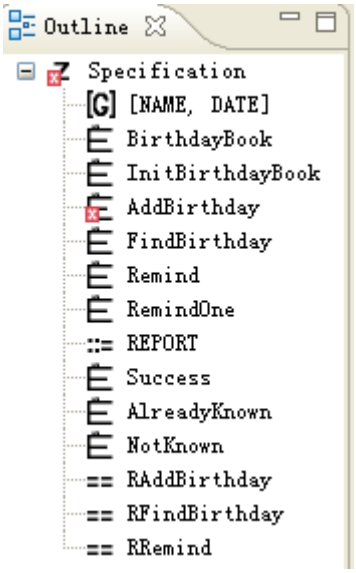
By hovering on the problem indicators, you can see the description about the problem.

2. Add problem entries to the **Problems** view



Each problem entry shows you the location, as well as the description, of the corresponding problem.

3. Display error indicators in the Outline view

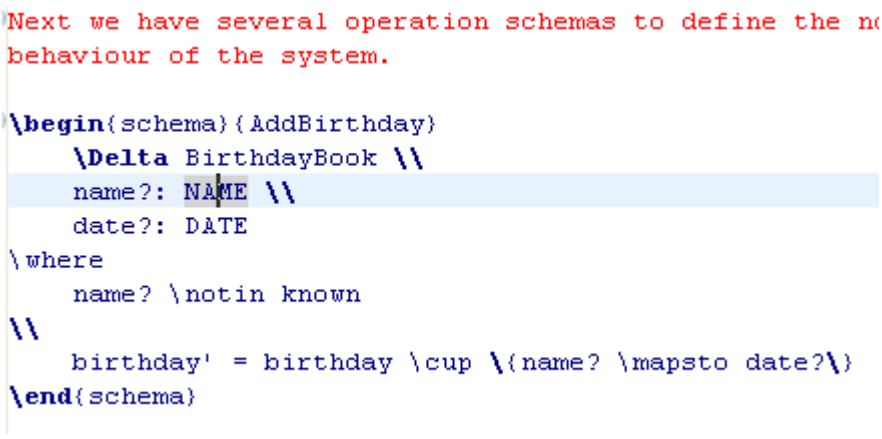


The problem marker on an Outline entry means that the underlying term of the entry contains problems.

Go To Declaration

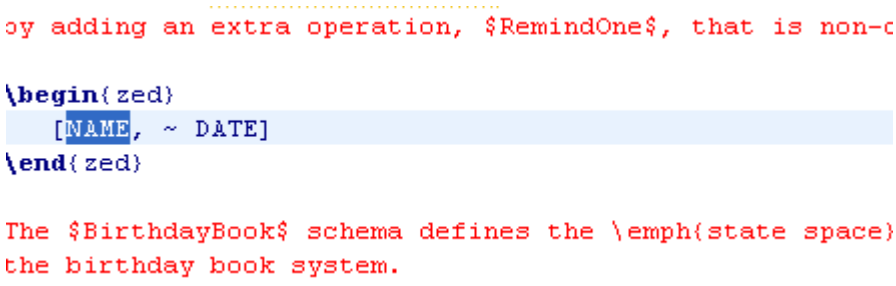
To quickly navigate to the declaration part of a Z name, you can follow the follow two steps:

1. Position the cursor within the name. For example, the below figure shows that the cursor is positioned within the name "NAME".



2. Select the menu entry **Edit > Go To Declaration**.

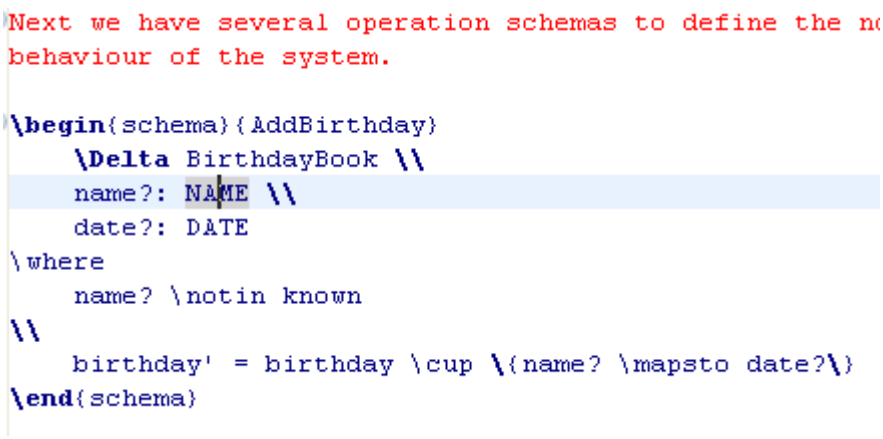
Then the name in its declaration part will be selected and visible in the editor as show below.



Highlight a Term

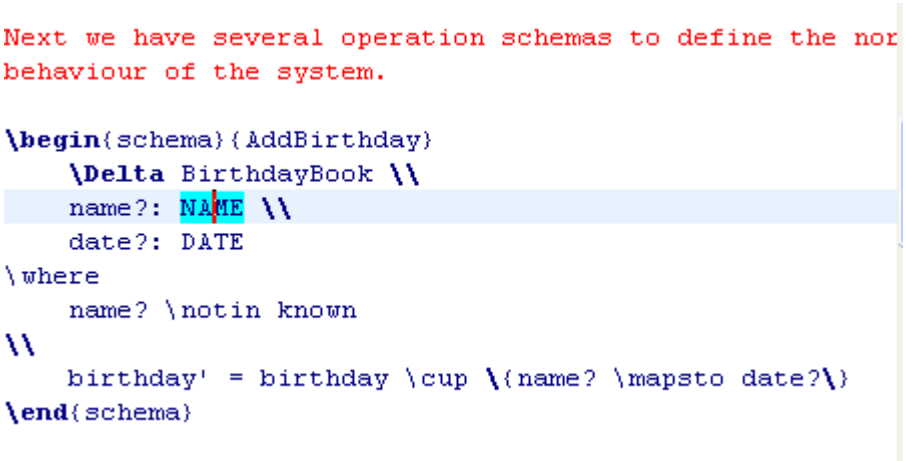
To highlight a Z term in the editor, you can follow the follow two steps:

1. Position the cursor within a term. For example, the below figure shows that the cursor is positioned within the name "NAME".



2. Select the menu entry **Edit > Highlight > Enclosing Term**.

Then the smallest term containing the cursor position is highlighted in the editor as show below.



If you want to highlight a bigger term, just perform the highlight command continually. The biggest term that can be highlighted is a Z paragraph. If a Z paragraph is already highlighted, then this command does nothing.

The **Edit > Highlight > Restore Last Highlight** does the reverse way of highlight. If the currently highlighted term is term that was highlighted at first, then this command does nothing.

You can also start the highlight procedure from a bigger term, rather than a point, by selecting a bigger segment of text before performing the highlight command.

CZT Editor

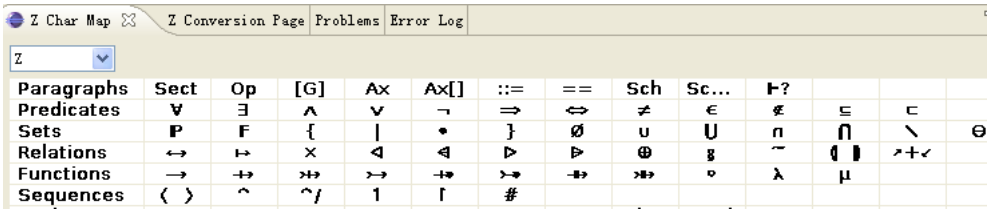
The CZT Editor is the major part that the CZT editor plug-in provides. It has the following features:

- Syntax coloring
- Highlighting of current partition
- Folding of partitions
- Bracket matching
- Selection by double-clicking
- Marking Occurrences of a Z name
- Jumping to the declaration of a Z name
- Highlighting the enclosing element
- Problem markers
- Hover support
- Viewing specification in alternative formats

CZT Views

In addition to the CZT Editor, the CZT editor plug-in provides a set of views.

- **Z Character Map**



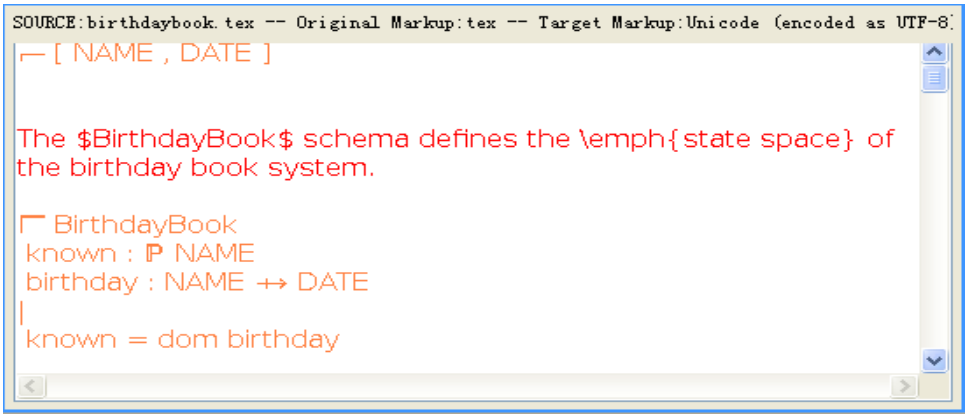
This view provides users a palette of Unicode character, including Z-specific characters, useful for writing Z specifications, particularly Using Unicode markup.

Users can insert a Z character or construct into the active editor by clicking on a character in the panel.

The actual text inserted follows the logic below:

1. If the active editor is not a CZT Editor, the description of the selected Z character is inserted in the editor.
2. If the specification is in LaTeX markup (the name of the specification file has the extension ".tex"), then the actual text inserted in the active editor is the LaTeX representation of the selected Z character.
3. If the specification is in Unicode markup (the name of the specification file has the extension of either ".utf8" or "utf16"), then the Unicode representation of the selected Z character is inserted into the active editor.

- **Z Conversion**



This view is used to receive the contents converted from the specification in the editor. It also support syntax coloring using the same rule as the editor.

- **Outline**

The view provides the overview of the Z specification in the active CZT editor. If enabled, the selection in the editor can synchronize with the selection in the **Outline** view.

CZT Perspective

The CZT perspective predefines the following set of Eclipse elements:

- **Resource Navigator view**
- **Z Character Map view**
- **Problems view**
- **Z Conversion view**
- **Outline view**

It adds shortcuts for the following perspectives. These shortcuts can be displayed by clicking on an icon on the top-right corner of the editor window.

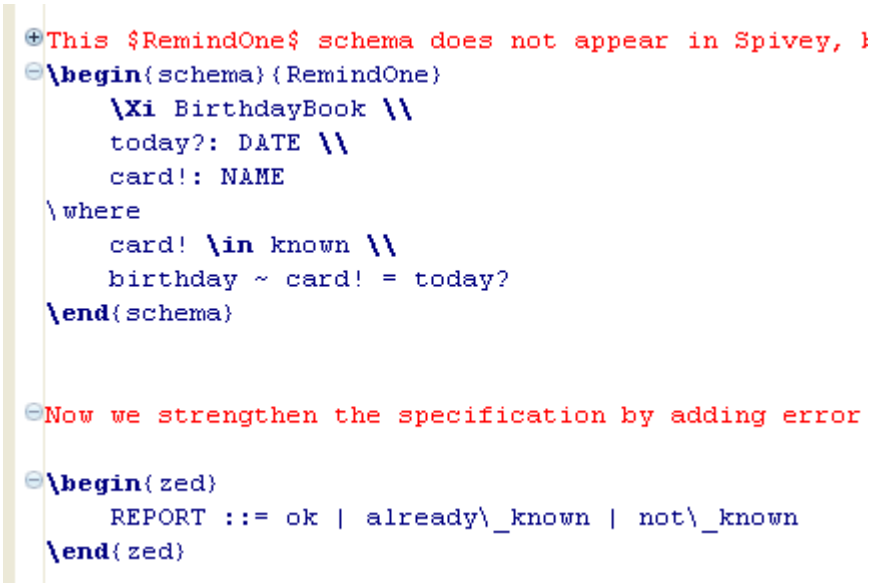
- **Resource Perspective**
- **Java Perspective**
- **Java Browsing Perspective**

It also adds shortcuts for the predefined set of element listed above to the list of shortcuts of views. The shortcut list can be displayed using the menu entry **Window > Show View**.

In addition, it adds to the **File > New** menu the shortcuts of the new CZT project creation wizard and the new Z specification creation wizard, as well as some wizard that Eclipse provides, including the new file wizard, the new folder wizard and the new untitled file wizard.

Folding

The Z paragraphs can be folded in the CZT editor as shown below.



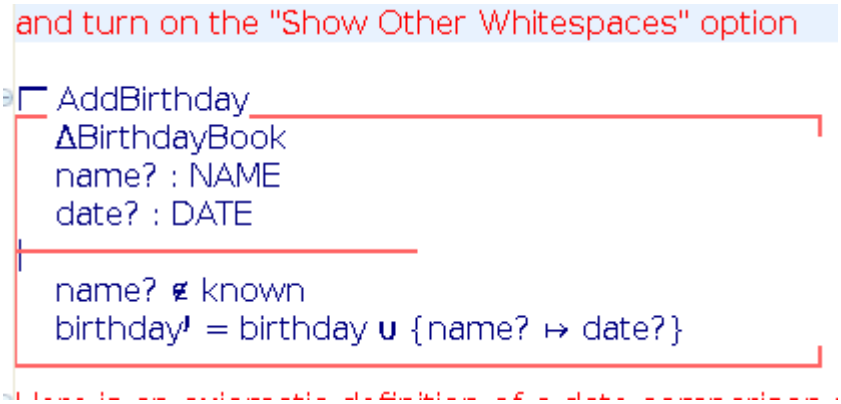
The "+" before a folded paragraph indicates that the paragraph can be expanded to show its contents in the editor, while the "-" indicates that the paragraph can be collapsed to show only the first line of the paragraph.

By default, all kinds of Z paragraphs are to be folded in the editor. Users can enable/disable the folding feature, choose a certain set of Z paragraph type to be folded, in the editor through the Folding preference page.

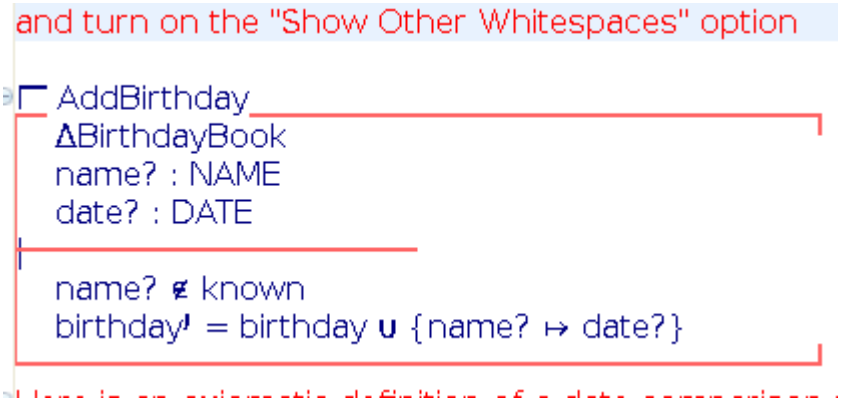
Schema Box

The ISO Z Standard suggests two commonly used styles for rendering Z schemas with boxes. To conform to the standard, the CZT editor plug-in implements the two box rendering style in the Unicode mode of its editor. Here are the sample views of those two styles in the Unicode mode of the editor.

- First box rendering style:



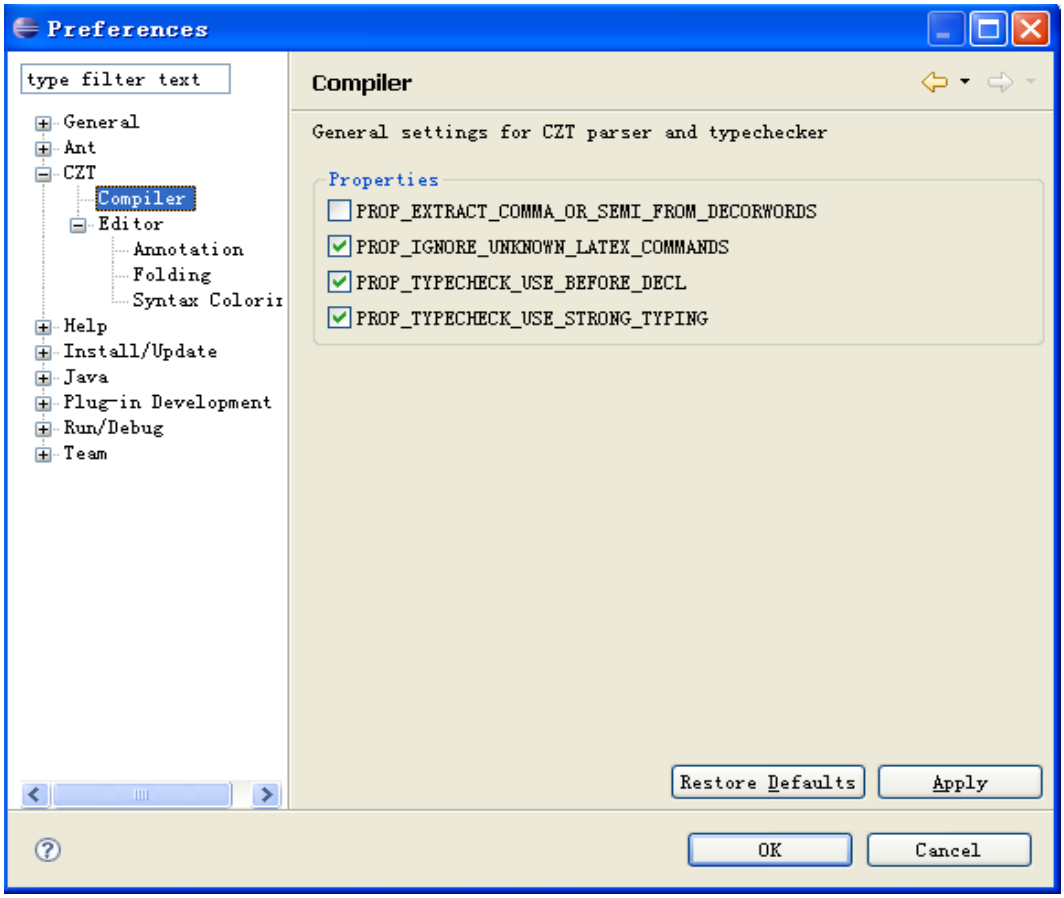
- Second box rendering style:



The style, line color and line width of the schema box can be changed in the Annotation preference page.

Compiler Preferences

The **CZT > Compiler** preference page allows to configure the various properties of the section manager for parsing and typechecking of Z specifications.



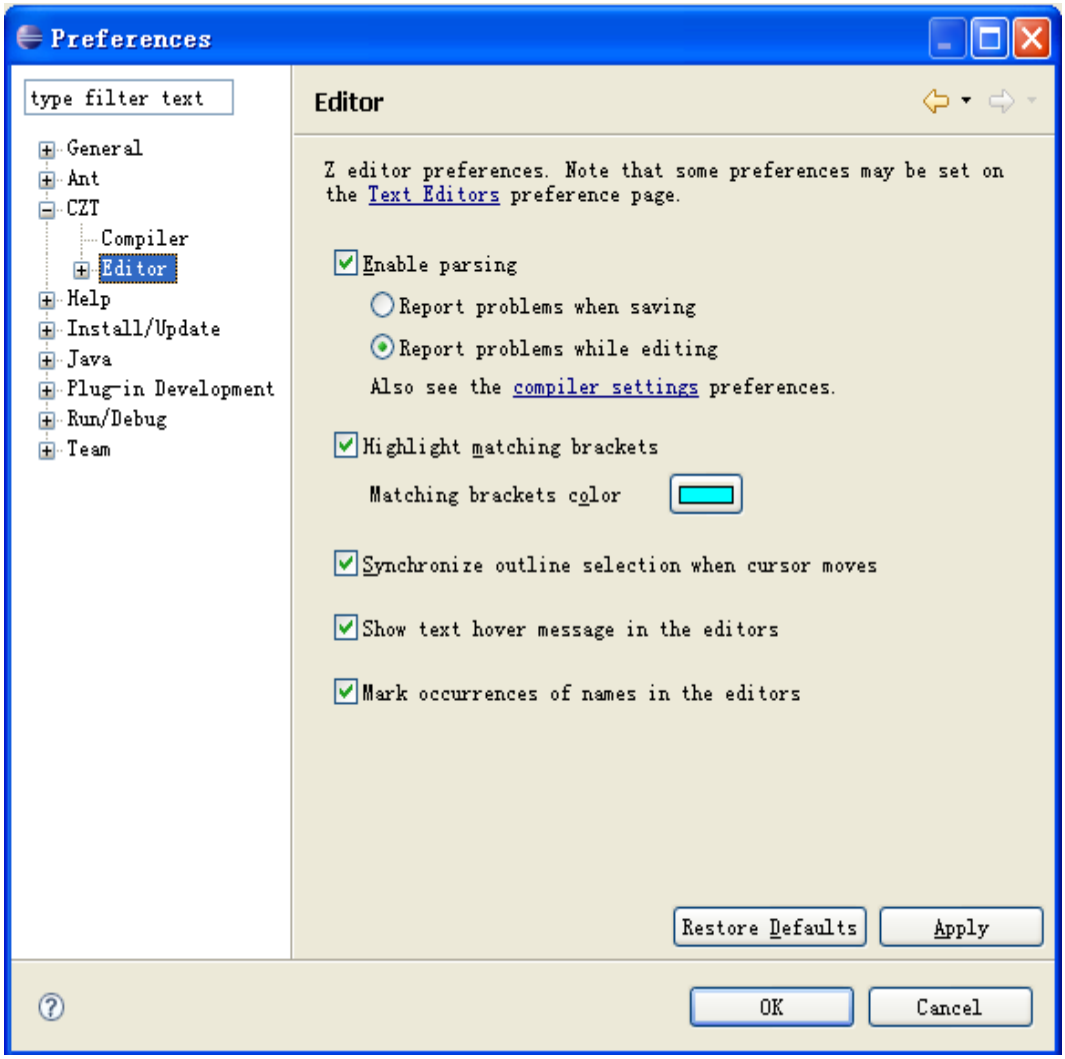
The following properties can be set in this page:

Option	Description	Default
PROP_EXTRACT_COMMA_OR_SEMI_FROM_DECORWORDS	When set to true, the parser tools extract symbol characters COMMA, SEMICOLON, and FULL STOP from the beginning and end of a WORD token to become WORDs themselves. This is a planned change to the Z Standard; see the Draft Technical Corrigendum 1: Corrections, including the use of Unicode, March 17th, 2006. As of now, this has yet to be submitted	On

	for official ballot.	
PROP_IGNORE_UNKNOWN_LATEX_COMMANDS	When set to true, the parser tools will ignore unknown LaTeX commands (that is, give a warning and use the name of the command) instead of reporting an error. Reporting an error is Standard conforming but ignoring those unknown commands is sometimes convenient.	On
PROP_TYPECHECK_USE_BEFORE_DECL	When this property is true, the typechecker will check that names are declared before they are used.	On
PROP_TYPECHECK_USE_STRONG_TYPING	Note: This property will affect object Z only. When this property is true, the typechecker will check the specification using strong typing.	On

Basic Editor Preferences

The **CZT > Editor** preference page offers the general settings for the CZT editor.



The following properties can be set in this page:

- **Parsing**

Option	Description	Default
Enable parsing	Enables the background parsing. If it is enabled, the parser runs constantly in the background each time a file is opened or the text in the editor is changed. If it is disabled, then those parser-dependent features, for example the	On

	content outline and hover messages, will be unavailable.	
Report problems when saving	The editor reports problems found by the background parser when the editor is saved	Off
Report problems while editing	The editor reports problems found by the background parser as the user is editing the specification	On

- **Matching Brackets**

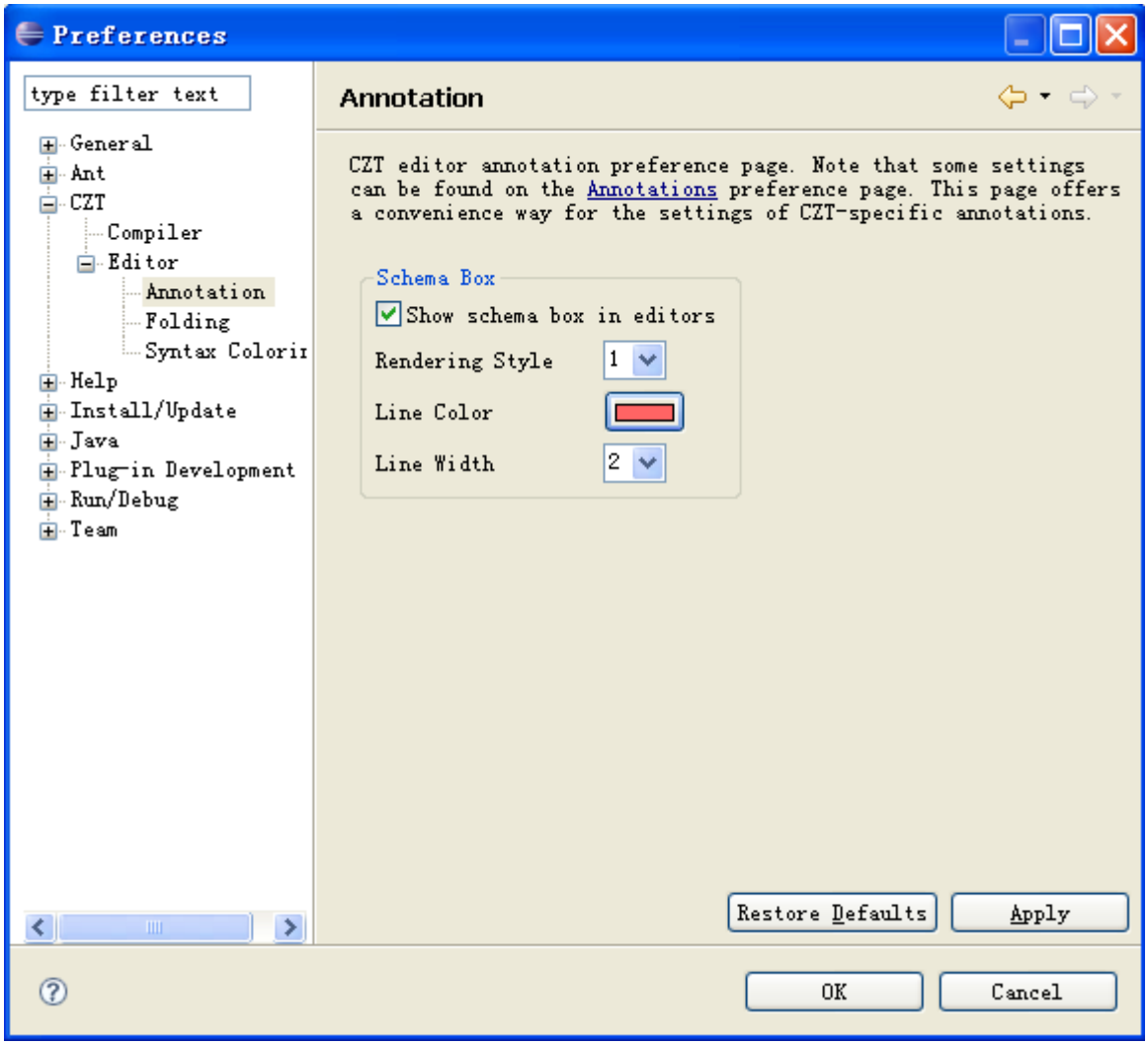
Option	Description	Default
Highlight Matching Brackets	Whether the matching brackets is highlighted in the editor	On
Matching Brackets Color	The color used for highlighting the matching bracket	RGB(192,192,192)

- **General Settings**

Option	Description	Default
Synchronize Outline selection when the cursor moves	the selection in the editor synchronizes with the selection in the Outline view	On
Show text hover	The editor displays hover message when the mouse is hovering inside the editor	On
Mark Occurrences	The editor highlights all occurrences of a Z name when the cursor within it	On

Annotation Preferences

The **CZT > Editor > Annotation** preference page offers a convenient way for the general settings of CZT-specific annotations in the CZT editors



The following properties can be set in this page:

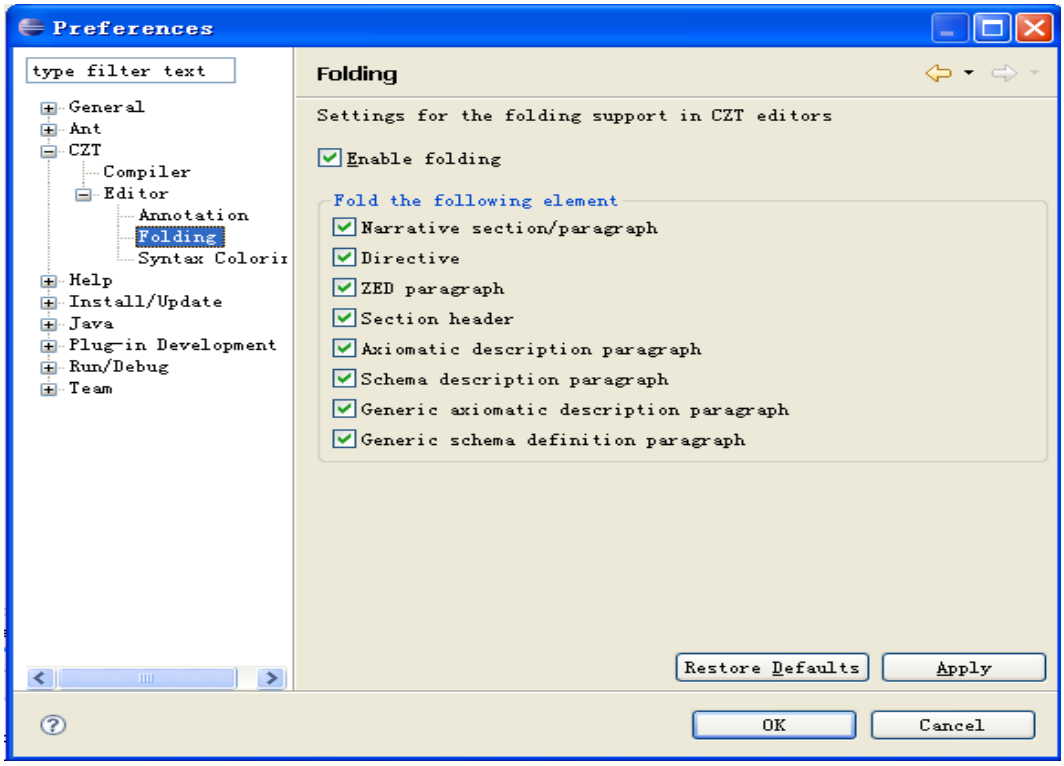
- **Schema Box Annotation**

Option	Description	Default
Show schema boxes	Show schema boxes in the CZT Unicode editors	On
Rendering Style	The style (1 or 2) used for rendering the schema boxes in the CZT editors.	2
Line Color	The color used for drawing schema boxes	RGB(255, 100, 100)

Line Width	The width of the line in a schema box. The available options are 0, 1, 2, 3, 4 and 5. If it is set to 0, the box will be drawn as fast as possible.	0

Folding Preferences

The **CZT > Editor > Folding** preference page allows to enable/disable the folding support in the CZT editors. It also allows to specify which kinds of element should be folded in the editor.

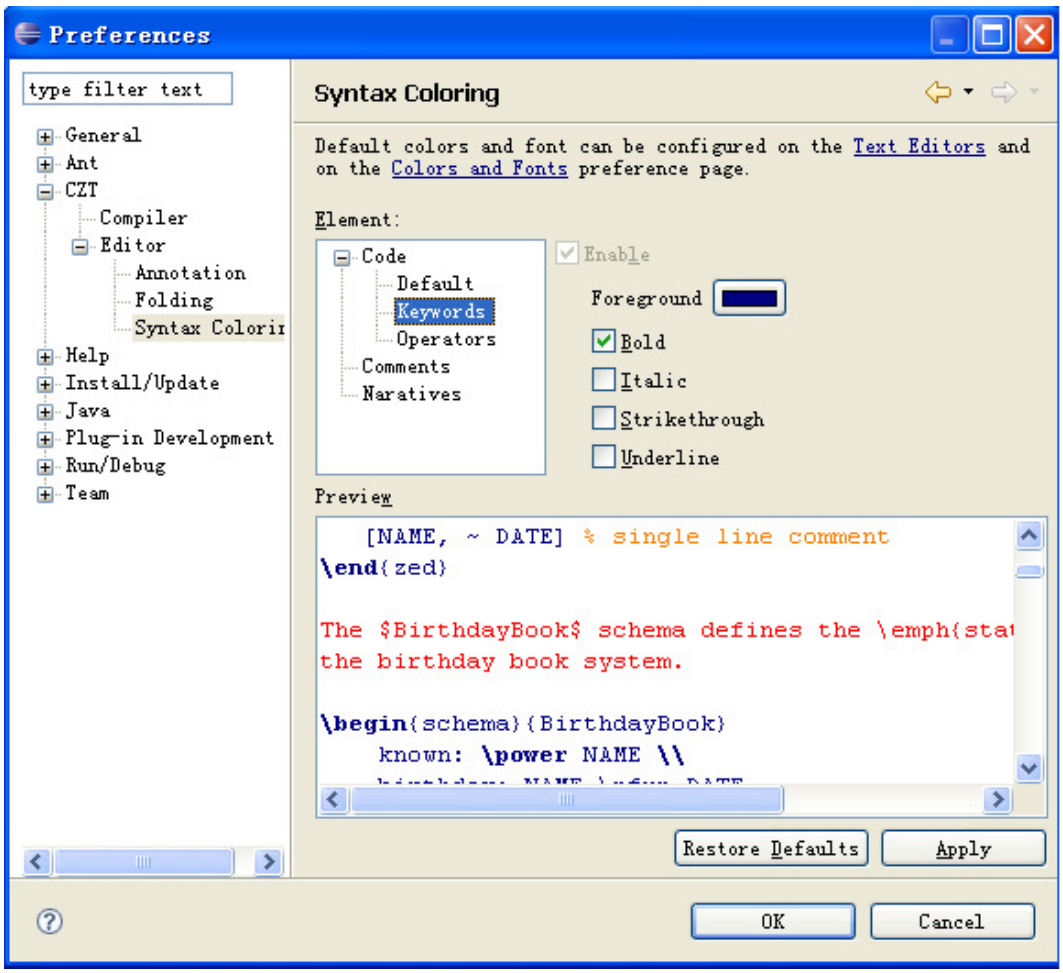


Here are the available settings in this page:

Option	Default
Enable folding	On
Fold narrative sections/paragraphs	On
Fold directive paragraphs	On
Fold general paragraphs (ZED in LaTeX mode)	On
Fold section header paragraphs	On
Fold axiomatic description paragraphs	On
Fold schema description paragraphs	On
Fold generic axiomatic description paragraphs	On
Fold generic schema definition paragraphs	On

Syntax Coloring Preferences

The **CZT > Editor > Syntax Coloring** preference page allows users to change to use their preferred colors and styles for different Z elements in the CZT editors



This page also provides a preview for the settings so that users can see whether the colors and styles meet their preferences before applying the changes.

The following table shows the available setting in this page along with their default values:

Element	Enable	Color	Style			
			Bold	Italic	Strikethrough	Underline
Default Code	On	RGB(0, 0, 0)	Off	Off	Off	Off
Z Keywords	On	RGB(64, 64, 1	On	Off	Off	Off

		28)				
Z Operators	On	RGB(64,64,128)	On	Off	Off	Off
LaTeX Comments	On	RGB(128,128,0)	Off	Off	Off	Off
Z Narrative Sections/Paragraphs	On	RGB(128,0,0)	Off	Off	Off	Off

CZT Editor Actions

This page describes the CZT-specific actions available in the CZT editors.

Edit Menu Actions

Name	Function	Keyboard Shortcut
Go To Declaration	Navigate to the declaration of a Z name	Ctrl+Shift+G
Highlight	• Enclosing Term - Highlight the term containing the current cursor position. If a term is already highlighted, it highlights the enclosing term of the highlighted term.(Arrow Up) • Restore Last Highlight - Highlight the term that is previously highlighted. If no term is highlighted or the highlighted term is the smallest term being highlighted, it does nothing.(Arrow Down)	Ctrl+Shift+Arrow Keys

Convert To	• LaTeX - Convert current Unicode specification to a LaTeX specification.(L) • Old LaTeX - Convert current Unicode specification to a Old LaTeX specification.(Spivey Z, which preceded the ISO Z Standard).(O) • Unicode - Convert current LaTeX specification to a Unicode specification.(U) • XML - Convert current specification to an XML specification.(X)	Ctrl+Shift+(L, O, U or X)