INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

# Creation of an Eclipse-based IDE for the D programming language

**Bruno Dinis Ormonde Medeiros**

Dissertação para obtenção do Grau de Mestre em

## Engenharia Informática e de Computadores

**Júri**

Presidente: Prof. Pedro Nuno Ferreira da Rosa da Cruz Diniz

Orientador: Prof. António Paulo Teles de Menezes Correia Leitão

Vogal 1: Prof. António Manuel Ferreira Rito da Silva

Vogal 2: Prof. David Martins de Matos

**Setembro de 2007**

## Abstract

Modern IDEs support a set of impressive semantic features, such as code navigation, code assistance, and code refactoring, which greatly enhance the productivity of IDE users. Of these, Eclipse JDT stands out as one of the most advanced open-source IDEs available, and is one of several IDEs based on the Eclipse Platform, an extensible framework for the creation of custom IDEs.

This document explores the issues and techniques concerning the creation of language IDEs with rich semantic features, based on the Eclipse Platform, while at the same time describing the development of one such IDE implementation for the D programming language. The architecture, and the various components of an IDE are examined, with particular focus given to the concepts and data structures that provide support for IDE semantic functionality. The application of those concepts to the creation of the D IDE implementation is then described, illustrating how it is possible, with the current state of the art, to use Eclipse and related projects to create a feature-rich IDE for a new language, with functionality such as an advanced code editor, code completion, rich project model, and others.



**Keywords:** Integrated Development Environments, IDEs, Eclipse, Eclipse Platform, JDT, semantic analysis, D programming language.

# Table of Contents

# List of Figures

# 1 Introduction

## 1.1 Motivation

The success of modern programming languages nowadays is influenced not only by the characteristics of the language itself, but also by the availability and quality of the whole set of language tools: compilers, build tools, debuggers, Integrated Development Environment(IDEs), profilers, etc. Of all these tools, the IDE is a most crucial one: not only it is responsible for integrating with the other development tools, but it also provides an editing environment where a software project and its source files are manipulated. Indeed, the IDE is where a developer's workflow is centered, and where he spends the majority of his time[1], illustrating its importance.

IDEs have seen a large amount of development since their dawn, having greatly increased in power. As an example, consider modern IDEs such as Eclipse[1], Microsoft Visual Studio[2], or IntelliJ IDEA[3]. For the Java and C# languages, these IDEs have a rich set of features, including semantic code navigation (go to declaration, code outline, etc.), full tool integration (compiler, builder, debugger), code assistance and completion, and, more recently, various automated refactorings. This level of IDE functionality greatly enhances developer productivity[2], and for this reason aspiring new languages with IDEs having little more than basic text editing features may quickly find themselves at a serious disadvantage versus the full-featured IDEs available for these other languages.

Indeed, the existence of high-quality IDEs for a given language may play a role as important as the quality of the language itself. But as much as it may be important, implementing an IDE from scratch is also a complex and laborious task. For this reason, Eclipse presents a very attractive option: the Eclipse Platform. The Eclipse Platform is an extensible IDE development platform that offers[3]:

- A comprehensive framework for the development of custom IDEs, providing support for generic IDE functionalities.
- Integration with complementary development tools, such as the build tool Apache Ant, source control systems such as CVS or Subversion, or any other tools available as extensions.
- A common User Interface environment and behavior across different languages and tools.
- More recently, the possibility of inter-language integration.

The Eclipse Platform is host to two IDEs that are officially part of Eclipse: JDT[4] and CDT[5]. Both of these have become very popular, particularly JDT, which is recognized as one of the most powerful and feature rich IDEs in existence[4], rivaled only by IntelliJ IDEA (a commercial Java IDE), and, to a lesser extent, NetBeans.

The success and continuous growth of JDT and CDT, as well as of the underlying Eclipse Platform, is notorious, and has attracted many to the development of Eclipse-based applications. Other projects have started with the goal of creating Eclipse IDEs for newer languages, such as PHP, AspectJ, Ruby, C#, and others. One such new and aspiring language is the D Programming Language, and it is with this motivation that this thesis project will consist in the creation of an Eclipse-based IDE for the D Programming Language, while at the same time, examining the state of the art in creating IDEs with rich code manipulation features, in a language-agnostic way.

[1] Eclipse: http://www.eclipse.org
[2] Visual Studio: http://msdn.microsoft.com/vstudio
[3] IntelliJ IDEA: http://www.jetbrains.com/idea/
[4] Java Development Tools, http://www.eclipse.org/jdt
[5] C/C++ Development Tooling, http://www.eclipse.org/cdt

## 1.2 The D Programming Language

The D Programming Language[6] is a systems programming language, created by Walter Bright in 2001 and under development since then. D was created with the intention of being a successor to C++, and aims to bridge the capabilities of low-level systems programming (such as native code generation, hardware access and manipulation, and efficient code generation), with modern high-level language features (such as garbage collection, contract programming, or functional constructs to name a few), while at the same time maintaining a simple and clean design (something considered quite lacking in C++).

D is a C-family, natively-compiled language. It is not a superset of either C++ or C, and as such it is not source-compatible with them (as indeed many aspects of C++ syntax were intended to be redesigned), but it has binary compatibility with C. D features a proper module system (no header files), does not require a preprocessor, does not require forward declarations, has well-defined primitive types, supports structs, unions and enums, supports Unicode both in language data types and in language source code, and has strong string and array support. D is an Object Oriented language, with a single inheritance plus multiple interfaces object model. D supports anonymous classes, function pointers and delegates, function literals (known as lambda expressions in functional languages), and has very extensive meta-programming capabilities[7].

The main D compiler is DMD - Digital Mars D[8], developed solely by Walter Bright, and DMD's front-end is released under an open-source license (GPL), whereas the back-end is proprietary. DMD is closely followed by its sibling GDC - Gnu D Compiler[9], which is a port of the DMD front-end to the GNU Compiler Collection (GCC) back-end.

The D language, although under continuous development, already has a very powerful set of core features, and is already an attractive alternative to C++, but only in terms of language qualities: the overall toolchain of the D language is poor comparatively to C++ and to other languages. D has good compiler and build tools available, but is lacking support in terms of IDE and debugger tools. Currently existing IDEs[10] go barely (if at all) beyond the features of syntax highlighting and project building, and no fully-functional specialized debuggers exist for D (C debuggers can be used on D programs, but with no knowledge of D's object file internals).

## 1.3 Project Goals

The goal of this thesis project is to develop a feature rich IDE for the D language, built upon the Eclipse Platform. As much as possible, it will aim for the following set of user features:

**Project Management -** Functionality for managing a project's settings and its member resources and source files.

**D Editor -** A source code editor for D, with various editor features such as syntax highlighting, code folding, etc.

**Syntax Error Reporting -** The ability to report syntax errors.

**Outline View -** An outline view showing the top-level structural elements of a D source file.

**Project Builder -** A builder capable of managing, building, and launching a D project.

[6] D Programming Language: http://www.digitalmars.com/d/

[7] For a more extensive overview of D, see: http://www.digitalmars.com/d/overview.html

[8] DMD: http://www.digitalmars.com/d/dcompiler.html

[9] GDC: http://dgcc.sourceforge.net/

[10] As of late 2006.

**Navigator View -** An Eclipse navigator view customized for D language projects.

**Code Navigation Assistance** :

**Go to Definition -** An operation that finds and opens the definition of a selected language element, possibly in another file.

**Find References** - An operation that finds all references to the selected element.

**Go to Super Class/Method -** An operation that finds and opens the definition of the parent of the selected language element.

**Outline Pop-up -** An editor pop-up, containing an outline view with a simple textual query for filtering names (also called Quick Outline).

**Type Hierarchy Pop-up -** An editor pop-up, containing a type hierarchy view (also called Quick Type Hierarchy).

**Code Editing Assist** :

**Code Completion -** An operation that proposes completions for the elements currently being typed.

**Code Templates -** An operation that proposes code templates (in the same pop-up as code completion), whenever a certain predefined text key is entered .

**Quick fixes -** An operation that proposes changes (fixes) for errors in the source code.

**Code Formatting -** Functionality for adjusting indentation and spacing according to document structure.

**Refactoring** :

**Refactor-Rename -** An operation that changes the name of a given element, updating all references to that element.

Debugger integration is not a planned project feature.

Given that this is a practical, real-world project, it is intended to be released as an open-source project in the D developer community as soon as it becomes usable. This will allow the project to have an user base, enabling feedback and also a certain degree of testing. Additionally, as far as it is worthwhile, the code produced should be well structured and documented, so that the work done here can potentially be extended or reused by others.

This document is structured in two major parts: The first part (Chapters 2 and 3) explores the various issues surrounding the implementation of IDEs with rich semantic features, in a language-agnostic way. Two particular terms are used here:

**Target Language -** The language for which the custom IDE is developed for, which is D.

**Host Language -** The language in which the custom IDE is developed. In the case of Eclipse that is Java.

The second part of this document (Chapter 4) details the development effort to produce the Eclipse-based IDE implementation for the D language.

# 2 Eclipse Integration

## 2.1 Eclipse Platform

The Eclipse Platform is a comprehensive open-source framework for the development of IDEs. It offers a rich foundation of building blocks where custom IDEs (as well as other kinds of applications) can be built upon and integrated together[3].

### 2.1.1 Eclipse Architecture

Eclipse is designed and built using a plug-in architecture. A project developed for the Eclipse Platform consists of a set of one or more plug-ins, where each plug-in represents a logical module of that project, which in turn can depend on other plug-ins. Each plug-in contributes to Eclipse by providing implementations (called extensions) of well defined extension points, and many of Eclipse's components are plug-ins themselves (this is represented in Fig. 2.1). A group of inter-related plug-ins can be grouped in what is called a feature, which allows them to be installed and updated together.

### 2.1.2 IDE Architecture

Most Eclipse IDEs, regardless of the target language, follow a similar general architecture. They are divided into separate major components where each component is packaged into a plug-in of its own, with a well defined API of interaction. There are usually at least two main components: the Core, and User Interface (UI). Additionally, there may be components for debug, the build system, or documentation, if their size merit a separate plug-in. Fig. 2.2 shows one such basic IDE architecture, with three main components.

#### 2.1.2.1 The core:

The core is the brain of the IDE and it is responsible for managing the domain logic of the projects of the target language. The domain logic, which in its whole can be called the language model, is composed primarily of two parts: the project model, which specifies what are the project's files, configuration options, etc., and the source model, a structured, semantic representation of the source code of the project's files. This structure is usually an Abstract Syntax Tree (AST), or some derivate of it. It is generated by a language parser (See 2.2.1), and is a very important data structure, used extensively throughout the IDE.

The core is also responsible for building and launching the project, which usually consists in calling an external compiler with the various project configuration options. The build system may also collect compiler messages from this process (particularly if the compiler is external) and report them back to the user. If this system is complex enough it may sometimes be placed into its own plug-in.

#### 2.1.2.2 UI:

The UI is responsible for user interaction and controlling the operations on domain data, such as the source code, file system resources, or other language elements. The UI consists of several components such as the editor, views, actions, wizards, preference pages, etc. It is responsible for these components, as well as the interactions between them and the Eclipse workbench.

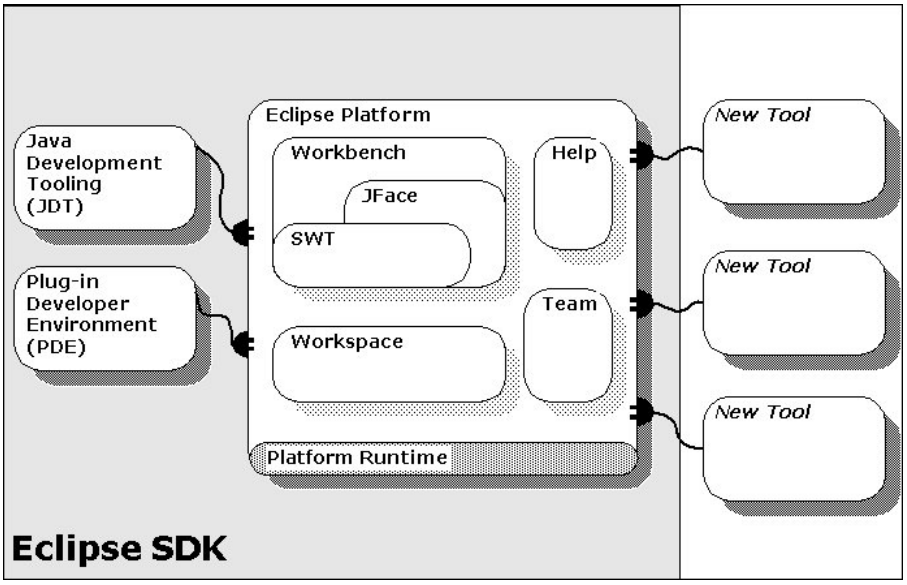


Figure 2.1: Eclipse Platform Plug-in Architecture

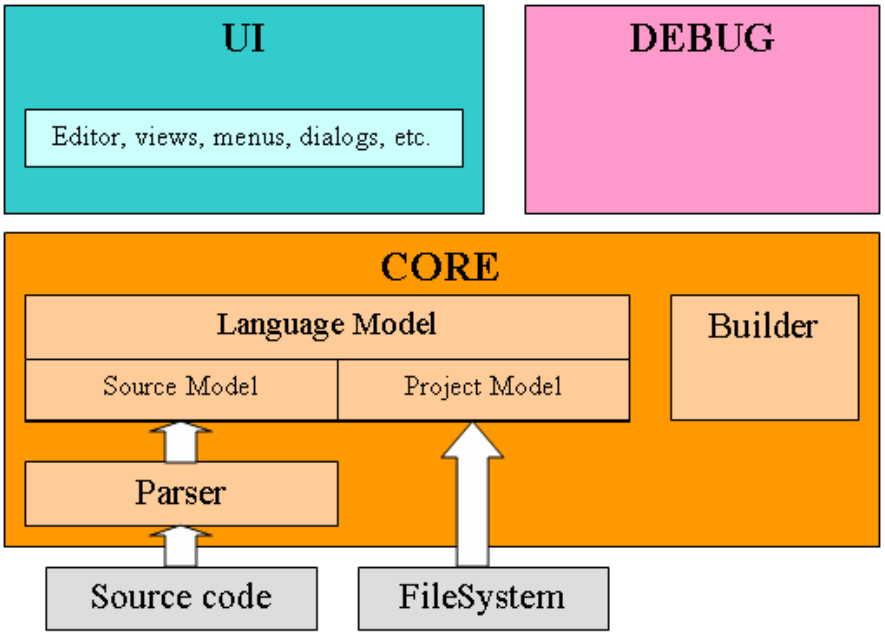


Figure 2.2: Basic IDE Architecture

#### 2.1.2.3 Debug:

The debug component is responsible for launching and running the target language's programs in a debug environment, and provide the functionality required for interactive debugging. Debug may implement a debugger of its own, or it may interface with an external one[1]. The Debug component may have a UI component of its own, to host debug-specific UI features.

The topic of implementation of the Debug component is not examined in this document.

## 2.2 Core Component

The core is, like the name says, the center of the IDE. In this section the basic subcomponents of an IDE core are examined. In chapter 3 the more advanced features of an IDE are analyzed.

### 2.2.1 Parser

One of the primary subcomponents of the core is the language parser. The parser's job is to recognize the language from a source file, while creating a in-memory structured representation of the source code. Aho, Sethi, and Ullman [5, 15] distinguish between two kinds of representations: an abstract syntax tree (AST) and a parse tree[2]. The AST differs from the parse tree in that the nodes that are redundant or irrelevant to the program semantic structure are removed or simplified, whereas the parse tree is a structure directly or very closely related to the parser's grammar productions.

Creating a parser is a task fairly outside the scope of the Eclipse Platform, and as such, Eclipse does not offer direct support for it. A parser is coded independently of Eclipse, either as an external application, or as Java code used directly by the Eclipse IDE. There are several options to create the parser: the parser can be developed from scratch; pre-existing external compiler tools can be used to parse an AST (if such tools exists for the target language); or the parser can be created using a parser generator.

Here is a list (by no means extensive) of some parser generators that generate code in the Java language:

- ANTLR[3] - Parser generator developed by Terrence Parr with over 15 years of development. Very well supported and documented. Generates LL(*) parsers as of version 3.
- JikesPG[4] - Parser generator that is part of the Jikes compiler. Fast but not well documented and no longer actively developed. Still, it is the one used by JDT.
- JavaCC[5] - Java-based LL(1) parser.
- SableCC[6] - Java-based LALR(1) parser generator.

### 2.2.2 AST

One of the central data structures that the core must manage is the AST. The AST, as mentioned before, is created by the parser from a source file, and is a structured, semantic representation of the source code, in tree form. An

[1] Such as, for example, GNU Debugger (GDB), which is what CDT uses.
[2] Also called Concrete Syntax Tree (CST).
[3] ANTLR URL: http://www.antlr.org/
[4] JikesPG URL: http://jikes.sourceforge.net/
[5] JavaCC URL: http://javacc.dev.java.net
[6] SableCC URL: http://www.sablecc.org/

IDE's semantic functionality is driven directly or indirectly from the AST.

At the very minimum, the AST's nodes contain information such as the node type, children nodes, and the position in the source code where the node originates. But many of an IDE's advanced features (like open definition, code completion, code refactoring, etc.) gradually require a more complex and detailed AST structure. In section 3 the design issues and functional requisites required for such advanced features are examined in greater detail.

### 2.2.3 Project Model

Another important data structure that the core manages is the project model. This structure consists of the information necessary to describe a project in the whole, such as the project's source folders, source files and required libraries (generically called the "build path"[7]), compiler and build options, compile-time conditionals and variables, etc. This information is used not only to build a project, but it is also necessary in most core IDE features, to describe the global semantic structure of the project. For example, in target languages with structured module systems[8], the build path information is necessary to have a full notion of what source files and language entities and symbols are available for code completion, semantic search, refactoring, etc.

### 2.2.4 Project Builder

Another core subcomponent is the project builder (or simply "builder"). Although not strictly required, this component is usually present in most advanced IDEs, as it provides a better user experience when the builder is partially or fully integrated with the IDE, not requiring external action to invoke it.

The builder is responsible for executing whatever steps are necessary for building and compiling the target artifact, such as an executable or a library. Additionally, it may also be in charge of processing the output of the building process and report any notable messages or events back to the UI. One such example are compilation errors and warnings, that are reported back to the Eclipse workbench in the form of problem markers[9]. This may be useful if the IDE itself does not have a full semantic analysis engine available, and thus relies on a builder or compiler for such information.

Eclipse builders can work in two ways: full or incremental. Full builders produce the build output from scratch, on each invocation. Incremental builders are builders that take into consideration only the filesystem resources or model elements that have changed since the last build, and are able to produce the build output without doing a full build. This is capacitated in Eclipse by a mechanism that tracks and collects resource change deltas and inputs them to the builder when requested.

### 2.2.5 Project Nature

Another task of the IDE core is to define a project nature for the target language. An Eclipse project nature is a mechanism that identifies a workbench project with a particular characteristic (such as the project's language). Recall that the Eclipse platform can host several IDEs, each responsible for their own language, and as such a

[7] Or "classpath" in Java-specific terms.

[8] like Java or D, but not like C/C++.

[9] Eclipse markers are a mechanism to annotate resources with various informations, such as compiler messages, to-do list, bookmarks, breakpoints, etc.

project can belong to any of these languages. A project nature identifies which language the project actually belongs to. This allows Eclipse to know for which projects it should enable a particular IDE plug-in and its respective features (such as UI elements). Also, a project can have multiple natures, as is the case, for example, of a PDE[10] project, which has a PDE nature as well as a Java nature.

A project nature is also the standard way to attach and configure (and de-configure as well) a builder for that project[6]. When a nature is added to a project (which usually happens during project creation), then that nature configures an appropriate builder for the project (and does the inverse in the case of nature removal).

## 2.3 UI Component

The next major component of the IDE is the UI. The UI consists of elements such as views, menus, actions, preference pages, wizards, and of course the editor itself. Figure 2.3 illustrates the various UI elements of the Eclipse workbench. Most of these UI subcomponents are simple and straightforward to implement, and one can easily learn how to create UI extensions by looking at implementation examples or at the respective documentation. However, some of these subcomponents are a bit more intricate, particularly the source code editor and its underlying text framework, so an overview of this framework and its customization mechanism is given next.

### 2.3.1 The editor

An Eclipse editor is a contribution to the `org.eclipse.ui.editors` extension point, mainly consisting of a class implementation of the `org.eclipse.ui.IEditorPart` interface, which in turn specifies the required interface for any kind of editor, be it a text editor, a visual editor, a multi-page editor, etc. To create a custom editor, one could implement this interface from scratch, but that is not the recommend approach for source code editors, since Eclipse already provides a pre-existing abstract implementations for this kind of editor. Namely, there is the class `org.eclipse.ui.texteditor.AbstractTextEditor` for generic text editors, and its subclass `org.eclipse.ui.texteditor.AbstractDecoratedTextEditor`, specialized for source code editors.

An `AbstractTextEditor` has two main components of interest, the source viewer (an instance of the `org.eclipse.jface.text.source.SourceViewer class`) and the document (an instance of `org.eclipse.jface.text.IDocument`). The `SourceViewer` is a JFace[11] adapter that wraps the raw SWT[12] widget `StyledText` into a more high-level abstraction. The `IDocument` instance represents the document which is the input to the editor and source viewer. These components are structured in a Model-View-Controller (MVC) design: the `IDocument` instance represents the Model's textual information, whereas the `SourceViewer` is the Controller (despite its name), and the View is the underlying `StyledText widget`. The `SourceViewer` is then responsible for updating the view's presentation according to changes in the input document and vice-versa. The editor is integrated tightly with these MVC components, and provides a layer of interaction between them and the Eclipse workbench context, meaning it controls the contributions and interactions with the Eclipse workbench. Eclipse editors follow an open-save-close lifecycle.

[10] Plug-in Development Environment (PDE) - The Eclipse SDK plug-in for the development of Eclipse plug-ins.

[11] JFace - a GUI application framework built on top of SWT, which provides richer functionality to it.

[12] Simple Widget Toolkit (SWT) - the GUI widget library used by Eclipse.

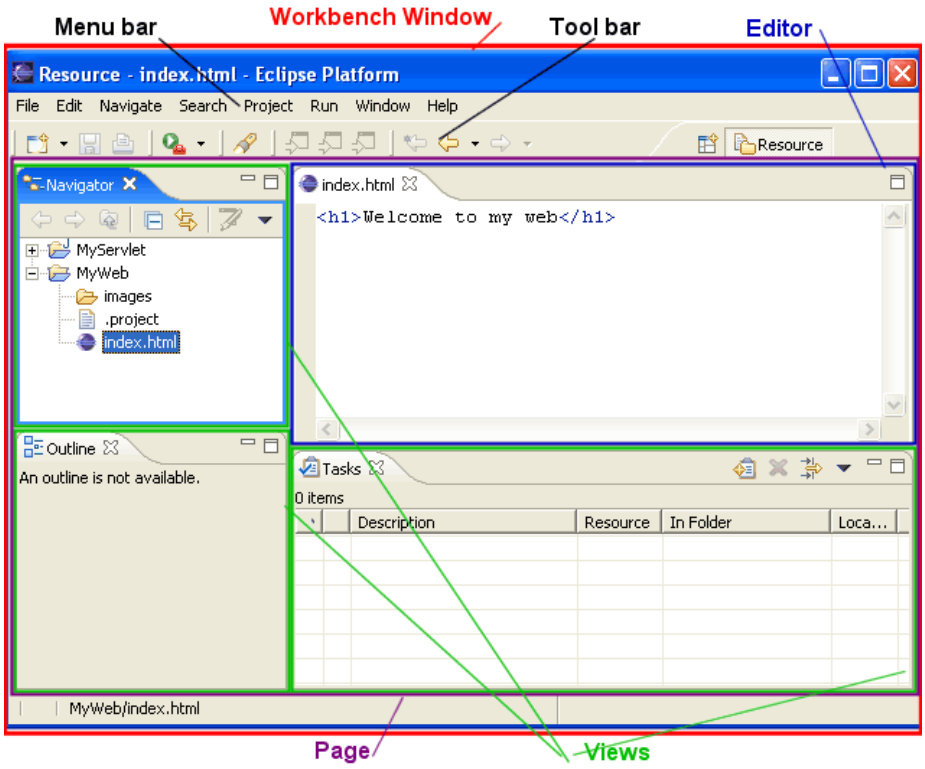


Figure 2.3: Eclipse Workbench UI

### 2.3.2 IDocument

An `IDocument` is a JFace component which holds the textual data of its underlying input source (usually a file). It provides support for:

- Text Manipulation - Modifying the underlying source text.
- Line Information - Providing the line number for a given offset.
- Positions - Maintaining information about several positions of interest in the document, and updating them during document changes (these positions are actually ranges).
- Partitions - Maintaining information about document partitions.
- Document change listeners and document partition change listeners - Keeping a list of document and partition change listeners, and notifying them when such changes occur.

An `IDocument` allows itself to be divided into several non-overlapping regions called partitions, where each partition has an associated partition type (also called content type). These partitions represent logical divisions in the document's text (such as code, comments, or strings in a source file), and are used in the configuration of several features of the editor and `SourceViewer`. For example, features such as syntax highlighting and code completion can be configured on a per partition type basis. An `IDocument` is set up with an instance of a partitioner, a class that knows how to calculate the document's partitions, as well as update them when there are document changes.

An `IDocument` instance is not directly provided to an editor. Instead the editor uses an `org.eclipse.jface.text.IDocumentProvider` implementation to create an `IDocument` instance from the given editor input. For example, when the input is a file (as is the usual case), the `IDocumentProvider` acts

as a persistence manager, and is responsible for loading and saving the `IDocument` from the the filesystem (the storage medium), keeping track of already created documents, and notifying interested listeners of changes in the file's contents. Eclipse provides a default implementation for this kind of document provider, which is `org.eclipse.ui.editors.text.TextFileDocumentProvider`. An editor's `IDocumentProvider` is also responsible for creating the annotation model from the editor input, as well as setting up the partitioner for the created editor document.

### 2.3.3 Source Viewer

The `SourceViewer` is the editor's main component, as it is responsible for the editor's document presentation and editing features. The `SourceViewer` abstracts away many of the base functionality common to source code editors, thus offering an extensive base for customization of common editor features (such as syntax highlighting, code completion, hovering, etc.). To implement custom editor features, one subclasses not the `SourceViewer` itself, but the `org.eclipse.jface.text.source.SourceViewerConfiguration` class, a component of a `SourceViewer`, which is the central point for editor customizations. The editor's `SourceViewer` is configured with the custom `SourceViewerConfiguration`, when the editor is initialized.

`SourceViewerConfiguration` contains a series of getter methods, each responsible for one particular editor feature. To implement such a feature, one overrides the respective getter method, making it return a custom class that will further control how the feature will operate. The available features to customize are:

**Syntax Highlighting -** Highlights regions of the text according to a damage-repair mechanism and a token scanner.

Method: `IPresentationReconciler getPresentationReconciler()`

**Text Hovers -** Displays small tooltips in the editor area presenting information about the current selection or the element the mouse is pointing to. Usually this information is the documentation comments of methods, classes, etc.

Method: `ITextHover getTextHover()`

**Auto Edits -** Configures various auto edits, which are automatic textual changes that the editor performs, like adding a closing brace or quotes when the opening one is typed.

Method: `IAutoEditStrategy[] getAutoEditStrategies()`

**Code Completion -** Displays a list of possible completions for a method, variable or class that the user is typing, according to the partial name already entered. This same extension option is responsible for code templates, which are pre-defined pieces of code that are inserted into the source, when requested to be inserted during code completion (code templates appear in the same pop-up list as code completion proposals).

Method: `IContentAssistant getContentAssistant()`

**Double-Click Selection -** Enables language-aware selections when double-clicking in the editor. This feature can select identifiers (according to what the language considers identifiers), the text range between braces or parenthesis, or any other selection deemed useful.

Method: `ITextDoubleClickStrategy getDoubleClickStrategy()`

**Hyperlink detection -** Detects if what the mouse is pointing to in the text editor is a hyperlink or not, and if so, what to do if the user requests to follow the link. The link can be language neutral links like URL addresses inside strings or comments, or language specific links such as a link from an entity reference to its definition (an alternative way to invoke the Open Definition operation).

Method: `IHyperlinkDetector[] getHyperlinkDetectors()`

**Code Formatting -** Defines a formatter for the editor. The formatter reorganizes certain textual elements of the source file, such as indentation, spaces, newlines, etc., in order to make the code prettier and more presentable to the user.

Method: `IContentFormatter getContentFormatter()`

**Quick Fixes -** Quick fixes are a feature, similar to code completion, where the user selects a problem in the project (such as a compilation error, name mismatch, type conflict, etc.), and is presented with a list of fix proposals for that given problem. There can be multiple fix proposals for each problem, being up to the user to choose one to be applied.

Method: `IQuickAssistAssistant getQuickAssistAssistant()`

**Model Reconciler -** This configuration option allows one to specify a class that will be in charge of reconciling the language model (such as the AST or related structures) whenever textual changes occurs (such as the user typing new characters). The reconciler runs asynchronously, and can be configured to operate in either incremental or non-incremental mode. In incremental mode the reconciler collects the textual changes since the last update and creates a textual delta that is used to update the model. In non-incremental mode, the update is simply performed with the whole source file. Section 3.6 examines this further.

Method: `IReconciler getReconciler()`

# 3 Advanced IDE Functionality

The core's data structures are the major point of support for most IDE functionalities. This section describes various design issues and functional requirements necessary for advanced IDE features. Most of these considerations are Eclipse-independent.

## 3.1 Basic AST Design

The first thing to note about the AST (especially if one is trying to reuse existing compiler tools to parse the AST), is that the AST design must preserve all source code information relevant to the user, whereas a compiler only needs to work with the information necessary to generate compiled code. This means that language constructs such as comments or preprocessor directives, which can safely be ignored in a compiler-oriented AST, should be recorded in some way in the data structures generated for IDE usage. That is, the AST and overall source model of an IDE should be at the same conceptual level of that which is the user view of the source code. Code formatting is an example of one such feature which is difficult or impractical to implement without an adequate level of information in the IDE's AST.

Here is some of the basic information an AST node should have:

**Parent node:** A link to the parent node. This allows traversing the tree in an upward as well as downward direction.

**Source range:** The source range is the start position and end position in the source code where the node appears, usually coded as a character offset and length. This information allows navigating from an element back into the original source text, as well as selecting an element from the source text. This range should start with the first significant character of the node, and end with the last significant character. For maximum usefulness, the range should also be present in an all nodes, and be recursively consistent, such that the ranges of a node's children should be non-overlapping, and contained in their parent node's range.

**Node type:** A value indicating the type of this node. This may sometimes be preferable to use instead of class runtime type identification (i.e., mechanisms such as Java's `instanceof`).

**Source file:** To which source file (if any) this AST node belongs to.

## 3.2 AST - Homogenous vs. Heterogenous tree

One design aspect of the AST is whether the tree should be a homogenous or heterogenous tree. A homogenous tree is one where there is only one class for all the nodes. In a heterogenous tree there are different classes for each node type, although they still share a common parent class (commonly named `ASTNode`).

Homogenous trees are simpler and faster to implement, and are easier to traverse, but heterogenous trees make it easier to work with the particularities and semantics of each type of tree node, and so, ultimately they are considered more adequate than homogenous trees (both JDT, CDT, and nearly any other IDE use heterogenous trees).

To make traversing heterogenous tree nodes simpler, a Visitor design pattern is usually employed, allowing concrete AST visitors to automatically dispatch to different methods according to the type of a node. This saves the hassle of having to write, for each visitor, the code that would manually check the type of the node, and dispatch

accordingly. JDT makes use of some helpful additions to the basic visitor pattern ([7], chapter 33): First, the visitor actually has two visit methods, `visit()` and `endVisit()`. `visit()` is called when descending into a node, and `endVisit()` is called after the node's children are visited. `visit()` returns a boolean controlling whether the visitor should actually descend into the node's children or not. Second, there are also two generic visit methods, `preVisit()` and `postVisit()`, which are used to traverse the node in a non-type specific way, as opposed to the normal visitor pattern. Thus, the overall invocation order becomes[8]:

1. `preVisit(ASTNode node)`
2. `visit(`*`<some subclass of ASTNode>`* `node)`
3. Now visit the node's children if `visit()` returns true.
4. `endVisit(`*`<some subclass of ASTNode>`* `node)`
5. `postVisit(ASTNode node)`

## 3.3 Parser

Common language parsers are able to recognize a language from a given input, and derive a structure from it (such as an AST). This is sufficient for most uses of parsers, such as code generation in compilers, but in IDEs there are certain features that require special parser capabilities not present in an ordinary parser. Two such capabilities are error-recovery and partial parsing.

Partial parsing is the ability to parse only a segment of the source file, where this segment corresponds to a complete syntactic element, such as a statement or declaration. This ability becomes useful when one wants to obtain information about a given language element (like a function declaration for example), but does not want to re-parse the whole compilation unit. Note that doing partial parsing requires knowing beforehand the source range of the element one wants to parse.

Error-recovery is the ability for the parser to generate an incomplete, but still meaningful AST tree in the presence of syntax errors. This is useful in order to keep the IDE functionality working even in the moments when the user is editing a source file and has an incomplete, syntactically invalid file. This is particularly important for the code completion feature, which, by its very nature, is invoked most of the time in a syntactically incorrect file, and will operate badly (if at all), if the parser is not able to do error recovery well. With this capability, AST nodes may be annotated with information on whether they where created from syntactically valid source or not.

## 3.4 Entity References

In an AST, many of the AST nodes represent references to other named language elements. As such, a key aspect of the language model is the ability to, given a reference in the code, find the referenced entity. These references are called bindings in JDT, and they are used by many other IDE features. The most basic example is the "Open Definition" operation, where given a selected type or variable reference, the cursor and editor focus is set to the location where the referenced entity is defined. A more advanced example is the inverse operation: given an entity definition, find all references to that entity.

In most languages, a reference is usually restricted in the kind of elements it can refer to, such as a type kind or a variable kind. For example, a name reference appearing in a mathematical expression should refer to a variable

entity, and not a type entity (it usually does not make sense to "add" two type names). It is useful to specify these restrictions in the reference structures of the language model. Also, the references may be categorized in more fine-grained types, according to the roles they play in the nodes they appear[8]. Here are some possible roles, taken from JDT's bindings:

1. Name reference - An explicit name reference in an AST to a type, function, variable, etc.
2. Type reference - the type of an AST expression.
3. SuperType reference - the super type of a given type.
4. Thrown exceptions - the expressions thrown by an expression or statement.
5. Members reference - the members of a given type.

An AST node can have more than one such reference. For instance, an expression node can have a reference to the type of the expression, as well as a name reference to a variable present in the expression. A reference data structure may also hold in itself the target entity to which it resolves to (a sort of caching). Typically, in such cases the validity of resolved references ends when there are model changes, as is the case in JDT[9], which requires the reference to be resolved again, should it be reused after changes.

## 3.5 DOM AST

An advanced AST technique is to design the AST with capabilities similar to a Document Object Model (DOM). The term DOM comes from the XML/HTML DOM[16], and is defined in a general way as a *platform and language-neutral interface, that defines a standard model of the logical structure of a document, as well as a standard interface for accessing and manipulating that structure*[5]. The term DOM has been generalized from XML to any hierarchical structure that fits that definition, such as a DOM AST. The benefits of a rich AST manipulation mechanism such as a DOM AST are mostly manifest in AST manipulation operations, particularly refactoring operations.

Designing an AST with DOM capabilities means the following in terms of code: Instead of just coding the children of an AST node as class fields of that node, such as this:

```
class WhileStatement extends ASTNode {
    Expression condition;
    Statement body;
    ...
```

one also defines in the node class certain structural information (called *structural properties* in JDT[8], just as in general XML DOM parlance), that for each of the node's elements describe certain attributes such as the name of the element, whether the element is mandatory or not, the kind of element (single children, list of children, or simple property), etc. Figure 3.1 shows an example of the structural properties of a Java method declaration, in JDT's DOM AST. This structural information allows the IDE to conveniently and dynamically inspect, as well as manipulate, the structure of an AST. This functionality is similar to language reflection, and in fact the reflection capabilities of Java (the host language) would allow to do this in a certain degree, but since the DOM mechanism is specifically designed for this purpose, it is much more adequate. Namely, it is simpler to code and to understand, and it is also more efficient.

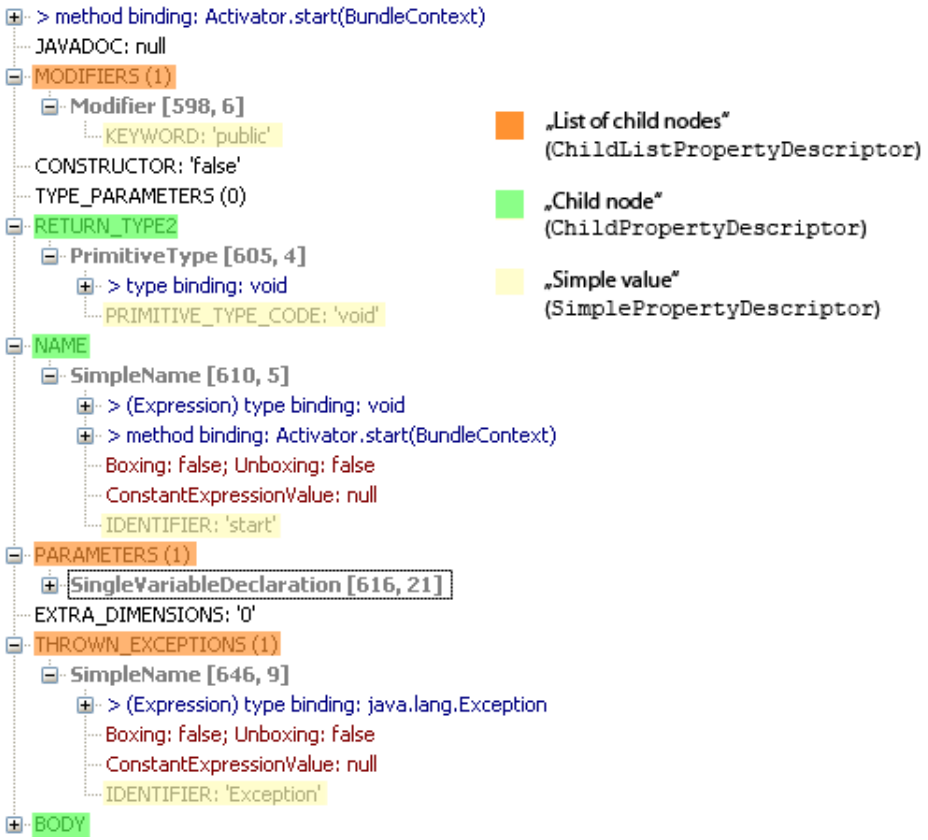


Figure 3.1: Structural properties of a Java method declaration

## 3.6 Model Updating

One important aspect of the IDE architecture is the model update behavior. Since an IDE is an interactive program, where the user will often be modifying the source code, this will cause the underlying language model to become outdated. The model will then need to be re-calculated to reflect the latest textual changes in one or more source files. This can be done in several ways:

- The whole model is updated on each project build or file save.
- The file's model is updated on the fly, as the user alters the file's text.
- The file's model is updated periodically on a fixed interval, like 500 ms., or 200 ms..

The first alternative is the simpler behavior, and the easiest to implement, but it is also the least satisfactory one, as the user must manually invoke a build or file save for the model to be updated (making some IDE functionality out-of-date in the meanwhile).

In the second alternative the model is updated constantly, with each new keystroke. This may seem the ideal behavior, but it has a notable problem: parsing is a costly operation, performance-wise, and doing a parse with each new keystroke will likely tax the Eclipse UI to the point where it will become sluggish or unresponsive. For this behavior to work satisfactorily, a very fast and scalable parser is necessary, which usually means the parser needs to be incremental[10] (i.e., able to create the new model without re-parsing the whole file). However, building an incremental parser is a very complicated task, other than for very simple languages. Not even JDT updates the model in this way. Instead, the third alternative, which is to update the model periodically on a small interval, is

the most common and adequate method for most languages. The model is updated on a background, low-priority thread, so that even in the moments where it is activated the UI won't be slowed down.

Eclipse provides support for any of these behaviors, in the form of what is called the source Reconciler (see the `SourceViewer` Model Reconciler option, Section 2.3.3).

## 3.7 Model Scalability

Another important issue with the language model, is scalability. Given that the AST is a relatively large data structure, building a complete model for a project, where ASTs would be created for all source files, would entail a very large memory footprint in IDE execution, especially in large projects with many source files. This is quite undesirable, even more in IDES based on Garbage-Collected languages such as Java, where memory usage plays quite an important role in application performance[1].

Furthermore, most IDE functionality, like the outline view, navigator view, type hierarchy view, etc., revolve around the usage and manipulation of top-level source code elements only (such as classes, methods, variables, etc.), and do not descend into more precise AST elements such as statements, expressions, and their various children. This makes the AST a too fine-grained structure for many of the IDE's functionality.

For this reason, one possible strategy (which JDT employs) is to have a separate data structure for such top-level source elements, which is used instead of the AST nodes for many IDE features. These data structures are lightweight: they have minimal info and most don't contain children, and although they are created from a file's AST, after the source file's elements are created, the whole AST can be discarded, keeping memory usage low. In JDT, these source elements are also part of the project model's elements (i.e., the source folders and libraries, package fragments, compilation units, etc.), which together form what is called the Java Model (basically a sort of lightweight front-end of JDT's language model).

But even being much lighter than the AST structure, computing and maintaining the light-weight model for the whole project might still be somewhat expensive. As such, an additional strategy that can be employed is the use of lazy-loading[10] (which again JDT employs). Instead of computing a model for the whole project, initially only the files or other elements in which the user is working on have their element information computed. If during the course of IDE interaction some operations (such as opening a new compilation unit, or requesting completion assistance) require information of another language element, then (but only then) will the full element information be retrieved for the new element. This is called "opening" the element, and the language elements that support this capability are called handles.

This lazy loading mechanism is then complemented with caching of the open elements, so that when a certain limit of open elements is reached, a cache manager tries to close some of the open ones, so as to keep the overall memory footprint from growing during the course of time. In JDT this cache is an unbounded Least Recently Used (LRU) cache.

[1] For example, certain garbage collectors, such as copying collectors, require twice as much memory than the one in use in order to run effectively.

## 3.8 Model Indexing

It was discussed in the previous section how the IDE can be optimized by keeping in memory only the elements necessary for the work the user is doing at the moment. This is appropriate for most IDE features, however, other features such as semantic search, where for example the user requests searching for declarations which start with a given name, or requests searching for all references to a given element, these kinds of operations will require processing information from all of the language model.

As mentioned before, keeping the whole language model (including ASTs) in memory is not desirable, so a naive solution would be to recreate the language model whenever search-oriented features required it. This is better, but still far from ideal: recreating the whole model (which would imply reparsing all the project's files) is a time-consuming operation, and on large projects it might take several seconds for the operation to complete (or even more, depending on the target language[2]). And the more semantic features the IDE has, the more this problem would be accentuated, since many semantic features would likely depend on some sort of search mechanism (for example, the rename refactoring requires searching for all references of the entity to rename).

The technique employed by several IDEs (JDT, CDT, and others) to address this is called indexing. Indexing consists of maintaining a structure of a large set of keys which map names to locations. The names are usually names of language entities, such as types or variables, and the locations are the location (such as file and source range) where the indexed entity is located. Unlike the lightweight handle elements, which maintain information only about definition elements, the index also maintains information about other interesting language constructs (such as references or method calls), making the index a larger structure overall. The key requirement of indexing, it that it must manage its information in an incremental way. That is, the index must be updated according to model changes, but must be able to do so considering only the changes that have occurred since the last update. Otherwise, the performance penalty would be the same as not using indexing at all (reparsing all files from scratch). Another useful addition, given that the index is a large structure, is the possibility to store the index, or parts of it, in disk instead of memory, and swap whenever necessary. This also has the benefit that the index can be preserved across IDE sessions, such that it doesn't have to be recalculated whenever the IDE is restarted.

## 3.9 Refactoring

Refactoring is the process of altering existing code in order to improve its readability or internal structure, but without the explicit intent of altering its external behavior[11]. Examples of refactoring operations are: renaming or moving a method or variable; extracting a method; extracting a superclass; etc.

Refactoring is regarded as a highly valuable software development technique, but its benefits are hard to realize without the support of automated refactoring tools[12]. Performing a refactoring manually requires various tedious (and error-prone) code alterations, as well as the creation of a suite of tests to ensure that the refactoring change was performed correctly and did not introduce a bug[11]. One of the first languages to support automated refactoring was Smalltalk, with its Refactoring Browser tool. According to Opdyke (in [11], Chapter 14), this tool, which initially was just a stand-alone tool, was actually mostly ignored as such, and only when it was integrated into the Smalltalk development IDE did it become popular. This illustrates the importance of having semantic features

[2] C++ for example is known to have relatively large parsing and compilation times.

built into the IDE itself.

### 3.9.1 Language Toolkit

The vast support in JDT for various automated refactorings of Java code, has made this technique popular among Java developers, and has significantly raised the bar for the code manipulation support of other IDEs, Eclipse-based or not. Recognizing the importance of refactoring, the JDT team has abstracted the generic functionalities for refactoring into a language neutral layer of the Eclipse Platform, called the Language Toolkit (LTK), which can be found in the `org.eclipse.ltk.core.refactoring` and `org.eclipse.ltk.ui.refactoring` plug-ins. These plug-ins offer generic refactoring support in three areas: refactoring lifecycle, refactoring participants, and UI support[13]:

**Refactoring lifecycle -** LTK provides a framework which controls the lifecycle of refactorings, and where each phase of the lifecycle must be explicitly defined and implemented by a refactoring operation. The phases are:

1. Checking the validity of initial conditions, such as if the selected element supports the requested refactoring operation, or if the source file is writable.
2. Requesting additional user input, if necessary.
3. Checking for validity of final conditions, so as to ensure the refactoring can be applied correctly (i.e., without altering program behavior).
4. Create the set of textual changes of the refactoring.
5. Display a preview of the changes to the user so that he may review them and confirm the refactoring application.

**Refactoring participants -** Sometimes a code entity can be referred in disparate language domains, such as a Java variable being used in a JSP file, or a Java JNI[3] method which is linked to a C function. Usually an Eclipse IDE is only aware of one of these domains, and consequently only knows how to perform refactoring on that context. Elements in other contexts must be updated separately. LTK provides support for what it calls the *refactoring participants*, a mechanism that allows other plug-ins to selectively take part in refactoring operations via plug-in extensions, thus effectively enabling automated refactoring across language domains.

**UI component for refactoring -** Finally, LTK presents UI support for refactoring operations, support which consists of an UI wizard dialog that controls the flow of each step of the refactoring, provides user input, and presents a preview dialogue that displays the textual changes that the refactoring will perform (see Fig. 3.2).

Of course, the actual logic of each of the refactoring operations must be implemented by the IDE developer, according to the target language's semantics. Fowler details in his book[11] the necessary steps to perform common refactoring operations in Object Oriented code.

[3] Java Native Interface, a protocol which allows for Java methods to be implemented by C functions.

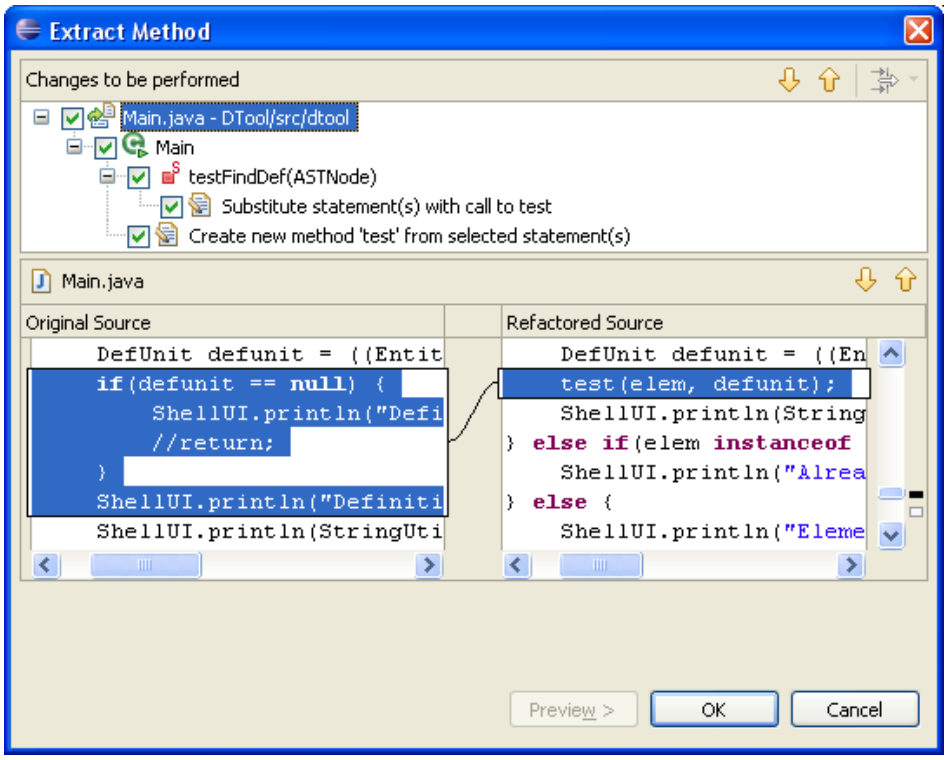


Figure 3.2: Refactoring Preview Dialog

# 4 Development and Implementation

## 4.1 Overview

The project began, first of all, by getting a general overview of the Eclipse Platform capabilities, and of the steps required to create an Eclipse-based IDE. This learning effort would continue from here on throughout the duration of the project, and consisted in studying relevant literature on the topic of IDE construction, as well as in great part in examining JDT's source code, particularly JDT's core component. It was on the knowledge gained here that the first part of this document is based. Simultaneously, this knowledge was applied to the development work to create an IDE implementation for the D language.

The fist task done here was to examine any existing Eclipse-based D IDEs or related tools that might provide for some useful code base. At the time, there were two open-source D IDE projects (DDT[1] and Blackbird[2]), but they were both abandoned projects, and had very basic and limited functionality. Still, these projects were kept in mind should they be able to provide any code re-use in the future.

From here on, the actual programming could be started on either Eclipse integration or creating a D parser. But as there isn't much that can be done in Eclipse without having a parser for the language, creating the parser is generally considered the first step in creating an Eclipse-based IDE [5], so this was the first implementation task.

## 4.2 Parser and AST

This task consisted in designing and implementing an initial (and prototypal) AST, together with a parser that would create such AST from source code.

Existing tools were examined to see if they could be re-used. The two existing Eclipse D IDEs had D parsers, both using ANTLR v2 as a parser generator, but those parsers were poorly documented and mostly incomplete (they did not generate an AST). There were also some other potential (non-IDE) D tools that were examined, but it was shown that they all had very limited parsing capabilities. There was really only one tool capable of fully parsing D code: the hand-written[3] parser from the DMD compiler itself, which is written in C++, and whose source is available.

There was then the option of re-using DMD's parser, which would require porting it to Java, or creating a new parser using parser generator tools, possibly re-using some of the previous D IDEs grammar code. Some of the current parser generator tools, such ANTLR, JikesPG, and others (See 2.2.1), were examined to get an idea of the work involved in creating a new parser, and then to assess which approach would be more beneficial.

Porting DMD's parser (a fairly large code base) would be a somewhat complex and lengthy process (as well as risky). Besides porting, the parser would also have to be adapted for IDE needs (See 3.1), since that parser and its AST were designed for code-generation. Namely, in DMD's case, the lexer had to produce tokens for whitespace and comments (instead of ignoring them), source ranges would need to be added to AST nodes, but most importantly, the overall AST structure created would have to be altered (especially because DMD's AST hierarchy was very awkwardly designed, to say the least). In contrast, the existing modern parser generators

[1] DDT: http://www.dsource.org/projects/ddt

[2] Blackbird: http://www.prowiki.org/wiki4d/wiki.cgi?EditorSupport/BlackbirdForEclipse

[3] Meaning it wasn't parser generated.

seemed robust and powerful enough to allow developing a minimal parser without expending too much time. In addition, using a parser generator would allow, if necessary, to develop a functional parser for a simple but relevant subset of the D language, so as to not spend an excessive amount of time in this task.

So the approach of using a parser generator was chosen. And of those examined, ANTLR seemed the better developed, documented, and supported parser generator, so this one was selected. It was also the parser generator used in the two previous D IDEs. This required learning ANTLR v3 (then the latest ANTLR version which redesigned some of the ANTLR language and engine capabilities) as well as generic parsing concepts, and then implementing a D lexer and parser.

### 4.2.1 Descent IDE and the DMD Parser

However, during the course of parser development, where the lexer and a small part of the parser's grammar implementation had been finished, a new Eclipse D IDE project, called Descent[4], was announced and released to the D community. This IDE, a volunteer open-source project with one developer, already had some interesting features in its initial release, namely:

- Minimal Project Model.
- Syntax Highlighting.
- Syntax Error Reporting.
- Outline View.

Also, this IDE had at its core an AST parser that was a Java port of the DMD front-end parser (the option that was previously discarded). This was a significant amount of work that prompted a re-evaluation of the initial project planning. Not only this IDE was a somewhat more advanced than the previous Eclipse IDEs, but its creator was interested in continuing its development. So, in order to avoid redundant work (and as stated in the project goals), this raised the possibility of future cooperation between these projects, and possibly even integration into a single project, so long as that wouldn't compromise this project's goals and essence as a thesis work.

An overall look at Descent showed that its Eclipse component was not very extensive. Instead, it was the ported DMD parser that so far comprised the most significant amount of code (and work). This port was examined to see if indeed it was stable and solid enough to be used, and, since it was deemed to be so, the decision was made to discontinue the previous work with the ANTLR parser and start using the Descent parser. But since it was a direct port from the DMD compiler parser, the generated AST would need to be adapted, because of the issues mentioned in the beginning of this section.

So, in order to further understand how an IDE-oriented AST should be structured like, a new and experimental AST design was developed, where a small part of DMD's most relevant AST classes where being converted to the new AST model. This was done until there was a small code base that could perform the find-definition functionality. It only worked for the small subset of converted AST nodes, but it was enough to show how that feature could be implement with the full AST hierarchy. Now, instead of finishing the AST functionality to work in all nodes, it was decided to move next to Eclipse integration, so as to produce an early IDE prototype, instead of delaying this phase to much later in the project timeline.

[4]Descent: http://www.dsource.org/projects/descent

## 4.3 Eclipse Integration

This task would consist in creating an initial Eclipse IDE implementation, so as to have available a full IDE prototype, and not just the core AST features, which cannot be used just by themselves. No code from Descent ended up being re-used in this early prototype (other than the parser), so as to be able to learn the most out of the relevant Eclipse concepts and components, and also because there wasn't a significant amount of Descent IDE code.

This task proved to take quite some time (overall consuming a sizable amount of the total project development time), because the Eclipse Platform capabilities are quite extensive, and its various components each took a while to be learnt. Indeed, even the code produced in this task was not extensive, as most of the work consisted in the learning process of the Eclipse Platform. For reference, some of the most significant components which were studied and used were: Eclipse editors and the JFace text framework; SWT (the graphics toolkit used to build UI components) and JFace viewers; UI actions, commands and menus; IDE preferences and persistence; Workspace resource management and change tracking; Jobs API and Eclipse concurrency;

The initial prototype had the following features:

- A basic D editor with syntax highlighting. Syntax highlighting was achieved through the use of DMD's lexer.
- A basic outline view, that showed the AST of the current source file.
- UI commands to find the definition of the editor's selected element. (This plugged into the previously existing AST semantic functionality.)

As the Eclipse development continued, the following features were gradually added:

- A D project nature, and project creation wizard.
- A syntax highlighting preference page, as well as persistence of these IDE preferences.
- An proper editor outline, now showing only the top level elements of a source file.
- A initial project model similar to JDT (with source folders, flat package fragments, compilation units), persistence of the project model settings, and basic UI to manage the project.

## 4.4 Reference Resolving

Now having a basic functional IDE, it was time to go back to the work on the semantic features, namely finishing the first tier of semantic functionality: resolving references (i.e, find the definition a reference resolves to). This meant completing the previous find-definition functionality for all kinds of nodes and reference targets, and considering all D name lookup rules. It should be noted that in this aspect D is somewhat more complex that most other languages, such as Java. For example, Java has two basic import constructs: Importing the name of a class or method in a given module, and importing all names in a given scope. D has four kinds of imports: an import of fully qualified names, a namespace content import, a selective import, and an aliasing import. D also has several kinds of definable entities, such as modules, aggregates (classes, interfaces, structs, unions), variables, function parameters, catch-clause parameters, enums, enum members, aliases, typedefs, templates, template parameters (type parameters, value parameters, alias parameters, tuple parameters), named template mixins, import aliases, foreach

loop variables, static if-type identifier definitions, and post-condition result variables. All these entities have been named in the D IDE implementation as Definition Units, or "DefUnits" for short (to avoid the ambiguous term "declaration", often used in C family languages). The find-definition functionality was intended to be designed and implemented in a way where it would be able to resolve any kind of reference to any of these definition units, in whatever context the reference might appear.

Recognizing the algorithmical and somewhat complex nature of this functionality, it was decided to heavily support it with test unit suites. As such, with each iteration of the implementation of the feature, a set of test cases was added (trying to encompass as most situations as possible), so as to ensure the quality and correctness of the feature, as well of future additions, refactorings, or other modifications. In some cases, the tests cases were created even before the sub-functionality was implemented, so as to have a clear picture of D's lookup and semantic rules (specially in more complicated cases), and subsequent functional requirements.

### 4.4.1 AST Redesign

As the work in this feature progressed, more changes were necessary in the DMD AST hierarchy, making it ever more clear that the original DMD AST was not very adequate for this kind of functionality. It was not well structured and overall its code was not really intended for public reuse (since the code is developed and maintained by a single person, this is not surprising). For example, there wasn't a single class that consistently represented a definition. The same was true for references. Instead, in the case of definitions (called DSymbols in the DMD hierarchy) there where some nodes that did not represent true source code definitions, but were under the DSymbol class, and yet there were other nodes that did represent definitions, but weren't correctly subclassing DSymbol. The case was similar for references, with the addition that there was also some duplicated nodes that were semantically very similar, but differed only in syntactic constraints. It gave the impression that in some aspects these nodes were structured more like a CST than an AST.

Thus, this explains the motivation for redesigning the AST hierarchy. The new AST (called Neo AST) had a single base class for all references (called `Reference`) and a single base class for all definition units (called `DefUnit`). Additionally, the new AST also reformulated the semantics regarding scopes, such as which nodes represent a lexical scope, which nodes represent an unscoped block, which do not, etc. A Scope was defined as: a block of code containing member nodes, where some of them might be definitions units. A scope has one outer scope and several super scopes. The other scope is the nearest lexical outer scope. The super scopes are other scopes whose members are available in the base scope, by means of semantic inheritance. For example, when a base class inherits from a parent class, the parent class scope becomes available in the base class scope (and transitively, all scopes in the super chain). These semantics are similar to Java and other OO languages, but in D there are also constructs other that inheritance that introduce non-lexical scopes in a base scope (such as the `with` statement).

### 4.4.2 Lookup Rules

With these concepts in place it was now relatively straightforward to define and implement the scope name lookup rules. The search simply starts at the reference, and using the AST node's parent links, it finds the nearest scope, looks for DefUnits in that scope, and then iteratively navigates the chain of scopes (first the base scope, then the

super scopes, then the outer scope, and so on) to find the requested name. This general algorithm works for non-qualified references, where the starting scope (also called the lookup scope) is the nearest lexical scope. But for qualified references the starting scope is not the nearest scope. So, for the algorithm to be fully correct, it must determine if the requested reference is qualified or not, and if it is, the qualified reference node will be responsible for providing the starting scope to the name search, according to the kind of the qualified reference (again, in D there are several kinds of qualified references, other than just a simple qualified references such as "Foo.bar".

Any given search is parameterized with a search context, that provides not only the name of the definition unit to find, but also any other relevant information, such as function arguments in the case of a function call reference. This allows the search to disambiguate between several different homonym definition units (for example, function overloads). However, although the search has this capability, function overload resolution was not implemented at the moment. Also not implemented was the resolving of the implicit references contained in expressions nodes (with the exception of function calls, which was implemented). There wasn't a significant technical difficulty in doing this, it was simply preferable to advance to other functionality which would likely be more useful, since the majority of cases for reference resolving were already covered.

With this core feature in place, all that was left now was to simply add some UI actions and code, that given a selected reference in the D source editor, would find and open its definition. But the find-definition functionality also allowed implementing two additional features on top of it: editor hyperlinking, which highlights references under the mouse cursor when Ctrl is pressed, and opens them when the mouse button is pressed; and text hovers, in which whenever the mouse cursor passes over a reference, a pop-up will be shown with the signature of the target definition, as well as its DDoc (the equivalent in D of JavaDoc). See figure 4.1.

## 4.5 Code Completion

With the basic semantic functionality of find-definition in place, it was time to tackle the next semantic feature, which is perhaps one of the most useful features in an IDE: code completion. An initial analysis on this task showed that performing code completion would consist in three major steps:

1. Determine, often with a syntactically invalid source, what is the reference and semantic context under the cursor where completion is requested.
2. From the determined reference and context, find all the elements that match the reference.
3. Present the results to the user, and change the source according to the completion proposal chosen by the user.

Step 3 is basically an UI and interaction task, and the Eclipse framework (more specifically, the JFace `SourceViewer` component) already provides full support for this, since this sub-functionality is pretty much independent of the target language (See 2.3.3).

Step 2 was also simple enough to do, given that we already had the find-reference functionality. To implement this step, it was only necessary to modify the basic find-reference code in two ways: One, the search would now look for partial names matches instead of exact name matches (which would often give multiple search results). And second, names that were occluded by identical names in nearer scopes (name shadowing), should not be considered as search results. The remaining scope lookup rules would remain basically the same, so the necessary

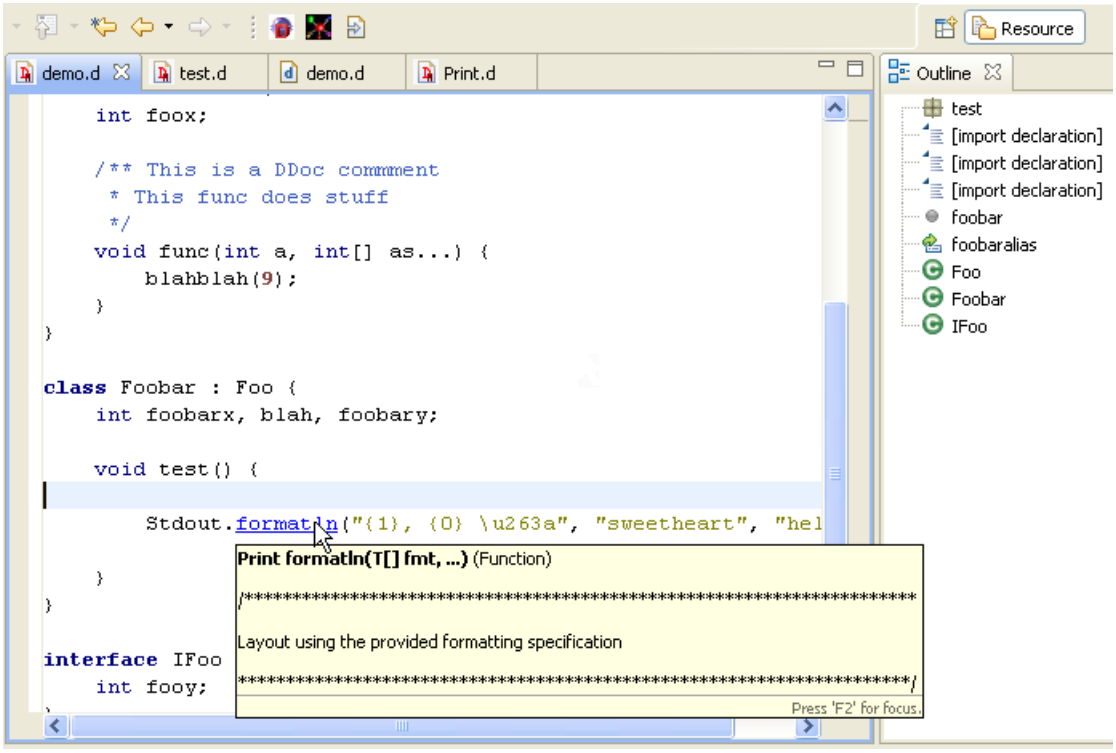


Figure 4.1: Text hovers and hyperlinking in the D IDE implementation

changes were quickly implemented, taking only a fraction of the time that was necessary for the find-reference functionality.

Step 1 would undoubtedly be the most challenging one. First, it would be necessary to determine the semantic context where the completion is requested. What this means is, that according to the target language semantics, different places in the source may have different results available, or have the relevance of each result change. For example, completion results of variables do not make sense in a place where only types are semantically allowed. But second, and most importantly, it would be very important to the user experience that the parser be able to do the most error recovery as it can (just as described in 3.3).

Examining DMD's capabilities in this regard, it was shown that it would still create an AST when the source file had syntax errors, but it would only contain elements until the point where the error occurred. This was not entirely bad, but it was still far from optimal: the desired was to have a full AST when the file had syntax errors. In order to achieve this, additional code was written that attempted to do recovery for a few common and simple cases of syntax errors occurring during completion:

1. When a statement or declaration is not complete, requiring a semicolon (';')
2. When the dot ('.') of qualified reference is typed, but the following name is not yet typed.

Basically what the recovery code does is, if it sees the source has syntax errors, it tries to determine if it might be because of one of those two cases, and if so, creates a temporary, modified source with either one (or both) of those corrections applied, and parses an AST from that. This is only a basic and partial solution (compared to JDT's very intelligent error recovery), but it covers a fair amount of cases, and it would be quite adequate for the time being.

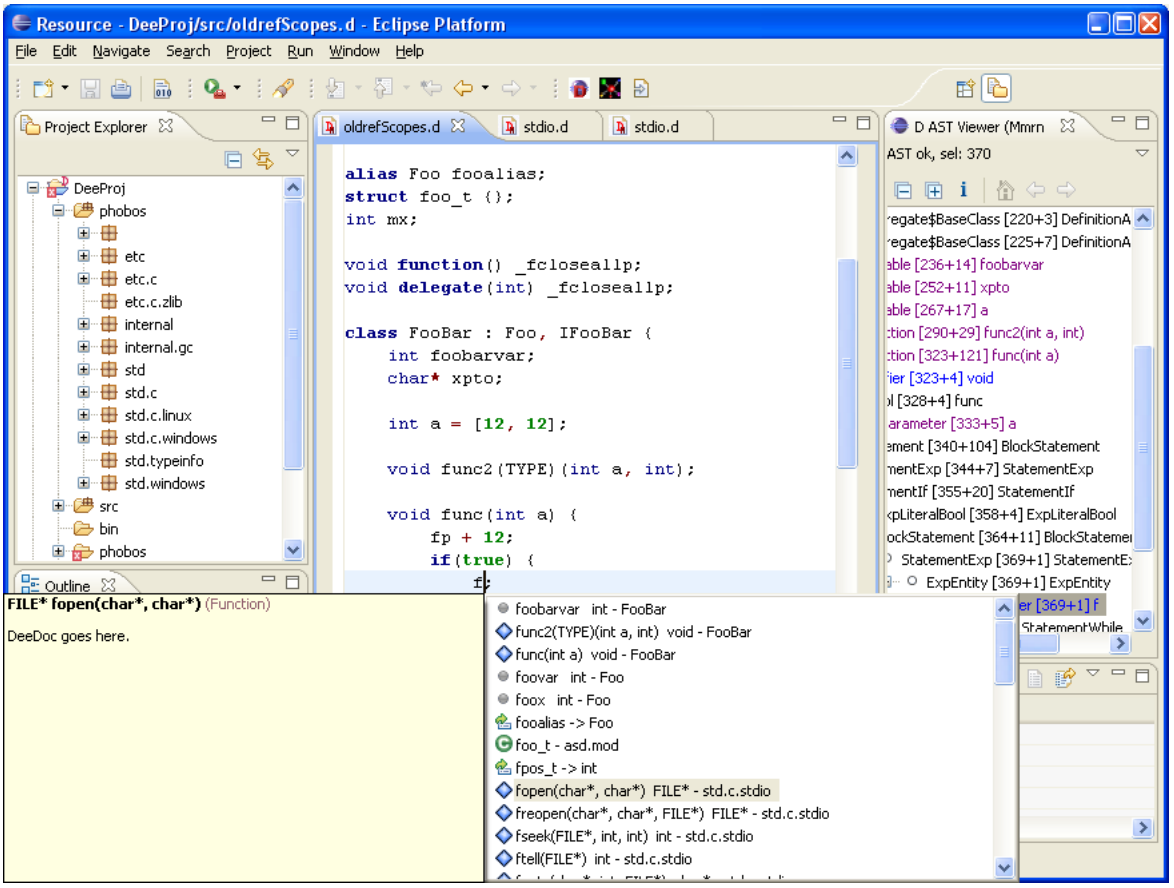


Figure 4.2: Code completion in the D IDE implementation

## 4.6 Public Release

Given that the D IDE implementation now had an interesting and useful basic set of features for D development, it was desirable to do a public release of this work. Not only it would potentially be useful for D developers, but it would also be beneficial for the project itself to have an user base, who eventually could provide feedback and bug reports (as mentioned in the project goals).

But there were some aspects that needed to be fixed and prepared before the IDE would be ready for release. One such aspect was the language model. Up until now all of the project files were parsed and kept in memory, which is quite problematic for large projects (as seen in 3.7). This was improved by implementing lazy-loading of the contents of compilation units, such that the compilation unit would only be parsed when necessary. Once loaded, the compilation unit contents would not be evicted from memory, so this was not an actual caching system, like what JDT has, but it would be sufficient for now.

However, the most important aspect in need of revision was the parser: the version of the currently used parser was a source copy of Descent's parser from its initial release in late 2006. In the Descent project, the parser had been continuously updated, but the D IDE was still using the initial version. This was problematic because not only the early Descent parser had some bugs (which had been fixed meanwhile), but the D language itself had evolved and gained several new features in the passing time period (particularly with the release of D 2.0). As such, to prepare the IDE for actual use, it was necessary to update the parser to the latest one available (which already supported D 2.0). This required adapting some of the newly introduced DMD AST classes to the AST hierarchy used by the D IDE, but the conversion was accomplished without much delay. After this, version 0.1 of the IDE was publicly released, and also, the D IDE started sharing (under SVN source control) the same parser code base than Descent, allowing the parser to stay updated from now on.

## 4.7 DLTK Integration

A major upheaval in the project implementation and planning was prompted by the discovery of the Dynamic Languages Toolkit (DLTK) Eclipse project. DLTK[5] is a very recent Eclipse project (started in early 2007), that aimed to extend the Eclipse Platform to provide a very comprehensive framework for the creation of IDEs for dynamic languages. This framework would augment the Platform's capabilities even further with an additional set of common language features, many of which taken from JDT itself[14]. Some of these capabilities were tailored for dynamic programming languages, but many of them, (particularly those taken from JDT infrastructure) seemed applicable to most programming languages, dynamic or not. Namely, some of those features that DLTK was able to provide were:

- A Java-like rich project model, with build path support for source and zip folders, local and external libraries, build path variables and containers, build path access rules, as well as UI boilerplate code for build path management, and model element navigation and manipulation.
- Model caching (as seen in 3.7), automatic resource delta processing, and model element delta generation[6].
- Model element indexing (as seen in 3.8).

[5] DLTK: http://www.eclipse.org/dltk/

[6] This later feature is of use to clients of the IDE itself.

- A great amount of UI boilerplate functionality such as: language search dialogs; a rich source code editor with bracket matching, code folding, hovers, etc.; UI preference pages for the editor, editor code templates, editor highlighting, installed compilers (the equivalent of Installed JREs in JDT), and others.

The value of these infrastructure contributions was immediately recognized. In the D IDE's current status it was the case that the model was lazy-loaded, but not cached. It also did not automatically respond to workspace resource changes, such that whenever a user added a file, the user would have to manually invoke an action to refresh the IDE model, so that the new file would become part of the model and subsequently considered in semantic functionality. To fully and correctly implement even just one of these features (such as caching) is something that would be impractically costly for the one-person manpower available in this thesis project. But having all of this functionality already implemented would be extremely useful, and so the possibility of using DLTK for the D IDE was immediately put into consideration.

Such integration had two main aspects that would need to be taken into account: Whether such framework would really be adaptable for a non-dynamic language such as D, and whether DLTK was mature enough to be used. Regarding maturity, DLTK was then at the 0.9 version, and together with the DLTK framework, it also provided reference IDE implementations for Ruby and Tcl, which successfully demonstrated the abilities DLTK so far aimed to provide (and work was beginning on a Javascript IDE as well). As for using a non-dynamic language, an initial look into the DLTK API and architecture, as well as its Eclipse project proposal slides[14] and developer posts in its newsgroups, showed that indeed there was no reason o think that some, if not the majority, of the DLTK functionality would not be adaptable for a language like D.

Given this, the development effort soon shifted to an experimental integration of the D IDE with the DLTK framework. It was shown that DLTK had very scarce documentation, and the API was still in its early stages, however, given the knowledge previously gained in studying JDT architecture, as well examining the Ruby DLTK-based IDE, an initial integration was able to be completed quickly (in less than a week). The primary integration tasks consisted of specifying the project nature for the custom DLTK-based IDE (as in 2.2.5), specifying a language parser, adding a source element provider (which creates light-weight source elements from the AST created from the parser), and adding some basic UI components, such as the editor. There was also the option of using some AST abstract classes provided by DLTK, which would ease the implementation of AST related features of DLTK, but weren't strictly required for DLTK integration. As such, it was decided not to use these abstract classes, at least for now, since it would imply several non-trivial (and possibly incompatible) structural changes on the existing D IDE AST hierarchy and design.

This new version of the D IDE with the initial DLTK integration was able to preserve all features it had prior to the integration, while gaining some of the new ones offered by DLTK (such as a rich project model, and model caching). As such, at this point this was already enough to show the feasibility and worth of integrating with DLTK. In addition, the fact that DLTK had been accepted as an Eclipse Foundation project [7], showed that we could expect continuous support and evolution in DLTK. Given all this, the decision to fully adopt DLTK integration was firmly consolidated, even if as a result, a lot of the D IDE's existing code would be scrapped (such as the code that created and managed the project model, the project explorer view extensions, most UI dialogs, and a few other elements). Figure 4.3 illustrates an architectural overview of the D IDE, after the main DTLK integration process.

[7] That is, a project managed and overseen by the Eclipse Foundation itself, just as JDT, CDT, the Platform, and many others.

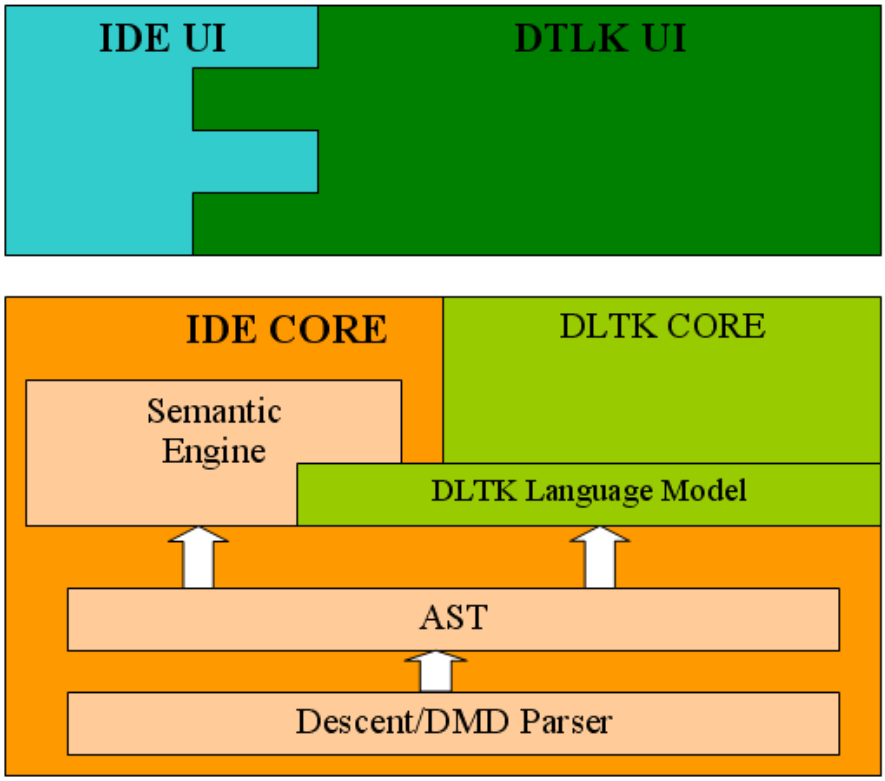


Figure 4.3: D IDE overview.

The integration development continued with several minor tasks, such as adding more UI actions and dialogs, adding code folding to the editor, adding and modifying preference pages, adding support for UI label decorations of element modifiers (such as `final`, `static`, `public`, `private`, and others of the same kind). Also added where UI actions from the editor to invoke the Quick Outline, Open Type[8], and Type Hierarchy navigation dialogs. Since these features are based on the source elements created by the source element provider, nothing more was necessary to implement them, since DLTK would take care of the rest. Nonetheless the Type Hierarchy feature was a work in progress, and was not yet implemented in DLTK 0.9.

The next development task, and the final major feature of the D IDE implementation, was to integrate with the search features of DLTK. Again, DLTK provided base functionality both in terms of UI and Core components, but since the D IDE was not using the DLTK AST hierarchy abstract classes, some modifications and some extra work would be necessary in the core implementation of the search features. The first type of search, Search Definitions, was implemented easily enough. The main body of work was implementing an AST visitor that would traverse an AST and report the definitions that matched the search query (which was parameterized both in terms of text patterns, as well as the archetype of the definition - type[9], function, or variable).

The second type of search, Search References, was a bit more complex. First, it also involved the creation of an AST visitor to traverse the AST and report the correct matches, but it was also necessary to extend the editor in order to be able to detect references in current selection, and since not using DLTK's AST hierarchy, this required some extra work, and even some DLTK class duplication, due to some limitations in DLTK. Second, it

[8]Display a dialog with a list (filtered by a text pattern) of all types available in the project, where the user can select one to navigate to.

[9]All of D's aggregate types, was well as templates, are included in this archetype

was necessary to correctly report named references to the indexer. The reason is that, although the code responsible for reporting matches works on the AST and not on another type of structure, the indexer is used by DLTK before the actual matching to determine the set of possible compilation units eligible to have references that match the search parameters. This works by checking the name stored in the index key to see if the name is a possible match, and if so, the compilation unit is later processed by the AST visitor responsible for reporting matches. This later step is still necessary because the reference may not actually refer to the chosen target definition, but rather to another definition of the same name, located someplace else. Finally, some UI modifications were introduced in the context menu creation code, and the semantic search feature was completed.

# 5 Conclusions

Nowadays the development tools that a language has available, and not just the language itself, are of vital importance to the productivity and success of that language. One such crucial tool is the IDE, which provides a rich editing environment, and integrates with the other development tools. A modern IDE is expected to have several features, such as a file/project explorer, a project builder, an integrated debugger, as well as a code editor, with syntax highlighting, source outline, code formatting, code completion, and in more advanced IDEs, advanced navigation features, semantic search, and refactoring functionality.

The Eclipse Platform is an extensible IDE and tool development framework, that by abstracting away common IDE functionality, offers great potential to those interested in creating an IDE or related tools for a new language. All of the above mentioned features have some degree of support in the Eclipse Platform framework. To make use of this support, it is necessary to extend the framework with implementations of language-specific functionality of the custom IDE. Doing so will require learning about many aspects and components of the Eclipse Platform, a worksome task which presents a steep learning curve, but which nonetheless, will surely compensate in terms of the development work saved, and the richness of functionality provided by Eclipse.

An Eclipse-based IDE has two main components, the Core and the UI. The UI component has most of its base functionality provided by the Eclipse Platform (as in Eclipse most UI infrastructure is common to any IDE or tool), and in general is simple enough to extend (although some components are more intricate, like the JFace text framework). The Core component on the other hand, is more extensive and complex, since the majority of it will be composed of language-specific functionality. As such, the implementation of the Core is a central task in the development of an IDE.

The first functionality the Core will require is a parser - a component responsible for parsing a source file of the target language and creating an AST from it. Another fundamental aspect for developing the Core is the design and creation of the language model: the data structures that represent the target language's source code and project structure on a semantic level (of which the AST is a part of). For example, navigation features such as reference resolving, or editing assistance features such as code completion, will require a model capable of understanding the language's semantic constructs (such as definitions, references, scopes, etc.) as well as having a semantic engine capable of navigating and searching among those constructs, according to the language's rules. More advanced semantic features, such as refactoring, will in turn require even more advanced model functionality: The semantic engine must able to perform full error analysis and precondition checking, which requires comprehensive semantic analysis capabilites in the engine (as comprehensive as, or even moreso, than a compiler). Also, the AST structures should be fitted with an adequate mechanism for describing and performing AST manipulations, such as a DOM-based AST, which applies the general concept of the XML/HTML DOM API to the language AST structure.

Performance and scalability concerns, especially when dealing with large projects, will also impose certain non-functional requirements on the language model. These concerns are addressed with techniques such as model indexing and model caching, which, in a general way, both aim to reduce unnecessary memory usage in the IDE. Model indexing consists of maintaining a name-based index of language elements (such as definitions or references) of interest to global semantic operations, allowing these operations to be able to either use the index directly, or preemptively determine which files the operation will need to work on, instead of examining and calculating ASTs for all of the project's files. Model caching on the other hand aims to reduce overall memory

usage by having light-weight (but incomplete) versions of the complete language elements and allowing a cache manager to automatically discard the content information of old elements when the cache gets full.

In the D IDE implementation, most of the project goals and target features were able to be successfully implemented, thus achieving a fairly interesting and useful IDE. This was possible not only due to the Eclipse Platform capabilities, but also in great part to the possibility to reuse existing code and tools, such as Descent's DMD-based parser, and the DLTK framework, which, although in both cases required dropping existing code, were quite worthwhile in the end, allowing to concentrate development effort on other parts of the IDE, and thus increasing its overall quality.

In summary, the main contributions for the project were: designing and building a basic semantic engine, learning the various aspects of the Eclipse Platform, and extending them in order to build IDE components integrated with the semantic functionality. In the early stages of the project there was also some work done in learning parser concepts and ANTLR v4 syntax, but, as mentioned before, that work was dropped.

The Eclipse learning effort was a substancial but quite important part of the project's work, and in it several components were covered such as: Eclipse editors and the JFace text framework; SWT (the graphics toolkit used to build UI components) and JFace viewers; UI actions, commands and menus; IDE preferences and persistence; Workspace resource management and change tracking; Jobs API and Eclipse concurrency. JDT's source code and implementation was also examined often.

In the final stages of the project the following features had been implemented:

- A rich JDT-like project model, provided by DLTK and customized with D's language-specific model element providers. This functionality provides:
  - Build path support for source and zip folders, external libraries, build path variables and containers, build path access rules.
  - Extensive UI functionality for project management, namely a project view with a structural view of D's packages and modules, and various UI commands and dialogs to create or modify project elements or build path.
  - Full model indexing and caching (as described in 3.7, 3.8, and 4.7).
  - Model resource delta processing, and model element delta generation.
- A D source code editor with:
  - Syntax highlighting. Implemented by defining simplified lexers (using JFace text functionality) that scan the document and color it according to syntax rules. The syntax highlighting is configurable through a preference page, allowing to choose the color and bold/italic/underline text attributes for each coloring unit. These settings are persisted using Eclipse's preference store mechanism.
  - Syntax error reporting. Implemented by plugging the parser's error reporting mechanism to DLTK's error requestor, which then handles the error's lifecycle, resource markers and UI.
  - Content outline and Quick-Outline. Content outline is provided by traversing and reporting the top-level elements of an AST, or other elements that are of interested to appear in the outline. Quick-Outline works in the same way, but DLTK provides the base functionality for the pop-up UI control, as well as the textual filtering mechanism.

  - Folding, bracket matching. Folding simply queries the AST for the kinds of elements to fold, and its respective source ranges. Bracket matching is implemented with a simple textual search pattern.
  - Open definition (with hyperlinking and DDoc text hovers). The find-definition functionality is one of the major contributions to the project. It required adapting the majority of DMD's AST to a new AST format, and then implementing the reference resolve mechanism according to D's various language constructs and lookup rules. The current functionality is able to correctly determine the target definitions in nearly all situations, except for function and template overloads, and the references implicitly contained in expression nodes (except for function call expressions).
  - Code completion. This functionality is built upon the previous find-definition functionality, which is extended and modified to allow the type of search required for code completion. Code completion is implemented using the same lookup rules as find-definition, but instead of looking for exact matches, all matches with the same search prefix will be collected. Some special code is also necessary for UI and text interaction, although JFace's Source Viewer will take care of most of the remaining UI functionality. Some minor parser error recovery was also implemented, as the parser's ability to do error recovery is very important to code completion.

- Open-Type dialog. This feature is also provided by DLTK and to be implemented it only required the previously specified model element providers (however, there seemed to be some bugs with the Open-Type dialog in the 0.9 version of DLTK).
- Semantic search. Again based on DLTK, this has the ability to:
  - Search for definitions based on a text pattern. This functionality was implemented by querying the index for the definitions that matched the search pattern. This required that the model element providers be made so as to process and report all existing definitions in D source files.
  - Search for all references to a given definition. This functionality works similarly to the definition search, but besides the textual search pattern, it is also necessary to resolve the candidate references, so as to check if they actually match the given target definition. Due to some DLTK limitations it was also necessary to implement some work-arounds to deal with some UI issues, and to enable specifying the target definition of the search pattern.
- Basic project builder. A simple project builder was also implemented, which collects all the modules and libraries of a project, places them in an output folder, and feeds them to a command line external D build tool that takes care of the rest (invoking the compiler to create an executable).

The D IDE project was named "Mmrnmhrm´´, and released online at this location:
http://www.dsource.org/projects/descent/wiki/Mmrnmhrm

## 5.1 Future Work

Given the grass-roots nature of the community of D developers, where there is no corporate backing and most of D's tools and libraries are developed in a volunteer, open-source way, enabling extension and reuse of the D IDE implementation was an important and explicit goal from the start. So, these are the main points to take into consideration for possible future work on the D IDE:

**Continuous DLTK integration -** The DLTK version upon which the D IDE is based (DLTK 0.9) is still in its early stages, and although it was already quite featured, there were minor bugs present, as well as some DLTK limitations that had to be worked around by means of code duplication or other temporary solutions. But most importantly, the upcoming versions of DLTK plan to introduce more features (such as Type Hierarchy view) that were not present the in the latest version.

Transitioning from DLTK 0.9 to 1.0 will involve several API changes (as can already be seen in the 1.0 alpha builds), but these should be mostly minor changes, and the overall transition work is expected to be straightforward.

**Descent integration -** Just as the work on the D IDE implementation continued, so did Descent's. Although the two projects were not integrated and were developed separately (other than the D IDE using the DMD parser), it was agreed to avoid developing overlapping features. As such, while the D IDE developed basic semantic functionality, Descent developed other features, such as a code formatter, debugger integration (with a command-line D debugger that had meanwhile been released), and some other minor features. As such, it would be desirable to combine the functionality of each IDE together. This is particularly true with regards to debugger integration, since, although this project's focus was on semantic features, and debugging functionality was not part of the project's plan, it is nonetheless a very important feature of an IDE, and quite influential to developer productivity.

**Parser improvements -** As mentioned before, the parser's ability to do error recovery is very important for code completion and other IDE functionality. In the D IDE implementation some code was developed to perform error recovery in a few common cases, but it would be quite useful to further increase the cases that the IDE is able to recover from. Currently, if a source file has syntax errors, the IDE tries to provide to the parser (using simple heuristics) some variations of the source code that may now be correct source. However, in the future the best approach to do error recovery would be to implement it directly in the parser, instead of on an upper layer of abstraction. Only in this way will the parser be able to better recover from more complicated error situations.

**Semantic engine improvements -** The functionality of the semantic engine implemented in the D IDE offers a fair degree of support for semantic features, but there is of course room for many improvements. The engine, particularly the find-definition functionality, can be improved by more accurately dealing with function overloads or template meta-programming mechanisms (such as D's implicit instantiation of function templates). This is a welcome improvement in functionality, but it will require more comprehensive semantic analysis capabilities, as the engine would need to understand D's rules for function calling, overloading and parameter passing, as well as D's template instanciation rules, which are somewhat complex.

Besides the find-definition functionality, error analysis is another avenue to pursue, although a more complex one. It would allow interactively reporting semantic errors in the code editor, just as syntactic errors are reported, and it would be a prerequisite for other features such as refactoring. Implementing such full-blown semantic engine would be a complex and long task (even more in D, due to its meta-programming capabilities), but eventually, the bulk of the IDE development work would have be to focused on this task, as it would be central for the more advanced semantic features.

**Refactoring -** Eventually, when the semantic engine became powerful enough, it would be possible to tackle refac-

toring functionality. This document described some of the considerations to take for this functionality, most of them based on the experience with JDT, which would provide a starting point for refactoring in the D IDE. However, given the advanced language constructs that D has, most significantly its meta-programming capabilities and (more recently) arbitrary compile-time code generation, new issues and considerations would certainly arise when implementing refactoring for a language with such capabilities. Given that D takes a structured approach to meta-programming and code generation (unlike raw methods such as pre-processor text macros), with enough work, it should be possible to deal with those issues satisfactorily.

# Bibliography


[1] G. Booch, A. Brown: Collaborative Development Environments. *Advances in Computers*, Vol. 59, Academic Press, (2003). (http://www.booch.com/architecture/blog/artifacts/CDE.pdf)

[2] J. des Rivieres, J. Wiegand: Eclipse: A platform for integrating development tools. *IBM Systems Journal*, Vol. 43, No. 2, pp. 371 - 383, (2004). (http://researchweb.watson.ibm.com/journal/sj/432/desrivieres.html)

[3] *Eclipse Platform Technical Overview*. Eclipse Corner Whitepaper (2006). (http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html)

[4] G. Goth: Beware the March of This IDE - Eclipse Is Overshadowing Other Tool Technologies: *IEEE Software*, Vol. 22, No. 4, pp. 108 - 111, (2005). (http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1463218)

[5] J. Arthorne, C. Laffra: *Official Eclipse 3.0 FAQs*. Addison-Wesley, (2004).

[6] E. Clayberg, D. Rubel: *Eclipse: Building Commercial-Quality Plug-Ins*. Addison-Wesley, (2004) (http://www.qualityeclipse.com)

[7] E. Gamma, K. Beck: *Contributing to Eclipse: Principles, Patterns, and Plug-Ins*. Addison-Wesley, (2003)

[8] Kuhn T., Thomann O. : *Abstract Syntax Tree*. Eclipse Corner Articles, (2006). (http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/index.html)

[9] M. Aeschlimann, D. Bäumer,J. Lanneluc: Java Tool Smithing - Extending the Eclipse Java Development Tools. In *EclipseCon 2005 presentation*, (2005). (http://eclipsecon.org/2005/presentations/EclipseCON2005_Tutorial29.pdf)

[10] P. Deva: *Create a commercial-quality Eclipse IDE, Part 1, 2 and 3*. IBM developerWorks, (2006). (http://www.ibm.com/developerworks/edu/os-dw-os-ecl-commplgin1.html)

[11] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts: *Refactoring - Improving the design of existing code*. Addison-Wesley, (2004).

[12] W. Opdyke: *Refactoring Object-Oriented Frameworks*. Ph.D. thesis, University of Illinois, (1992). (ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z)

[13] Frenzel L. : *The Language Toolkit: An API for Automated Refactorings in Eclipse-based IDEs*. Eclipse Corner Articles (originally in *Eclipse Magazin*, **Vol. 5**, January 2006), (2006). (http://www.eclipse.org/articles/Article-LTK/ltk.html)

[14] Dynamic Languages Toolkit - Creation Review Slides. (2006). (http://www.eclipse.org/proposals/dltk/Dynamic%20Languages%20Toolkit%20-%20Creation%20Review%20Slides.pdf)

[15] A. Aho,R. Sathi, J. Ullman : *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, (1986).

[16] W3C (World Wide Web Consortium): *Document Object Model (DOM) Level 1 Specification*. W3C Technical report, (1998) .(http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/)

[17] The Eclipse website at www.eclipse.org (2006).

[18] The Eclipse JDT source code at CVS://:pserver:anonymous@dev.eclipse.org:/cvsroot/eclipse (2006).

[19] Eclipse Help - Platform Plug-in Developers Guide. http://help.eclipse.org/help31/index.jsp (2006).