INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

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

Extended Abstract

Bruno Dinis Ormonde Medeiros

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.

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 func-

[1]Eclipse: http://www.eclipse.org
[2]Visual Studio: http://msdn.microsoft.com/vstudio
[3]IntelliJ IDEA: http://www.jetbrains.com/idea/

tionalities.

- 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 semantic features, in a language-agnostic way. These intended features are: Project Management, Project Builder, a D Editor, syntax error reporting, a Navigator view, Outline view and pop-up, Type Hierarchy view and pop-up, Find Definition, Find References, Code Completion, Code Templates, Quick Fixes, and Refactoring.

1.1 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 is an Object Oriented language, with a single inheritance plus multiple interfaces object model. D features a proper module system (no header files), does not require a preprocessor, nor does it require forward declarations.

The main D compiler is DMD - Digital Mars D[7], 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[8], which is a port of the DMD front-end to the GNU Compiler Collection (GCC) back-end.

[4] Java Development Tools, http://www.eclipse.org/jdt
[5] C/C++ Development Tooling, http://www.eclipse.org/cdt
[6] D Programming Language: http://www.digitalmars.com/d/
[7] DMD: http://www.digitalmars.com/d/dcompiler.html
[8] GDC: http://dgcc.sourceforge.net/

2 Eclipse Platform

The Eclipse Platform is a comprehensive open-source framework for the development of IDEs, based on a plug-in architecture. 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].

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.

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, 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.

2.2 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.

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[9]. 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.

[9] Such as, for example, GNU Debugger (GDB), which is what CDT uses.

3 Advanced IDE Functionality

The IDE's data structures are the major point of support for most of the IDE functionality. 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. Some of the basic information that AST nodes should have are: the node type, the node's source range, and the parent node.

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.

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 (the ability for the parser to generate an incomplete, but still meaningful AST tree in the presence of syntax errors) and partial parsing (the ability to parse only a segment of the source file).

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.

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[8], 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, where each AST node type has an explicitly defined element structure. The benefits of a rich AST manipulation mechanism such as a DOM AST are mostly manifest in AST manipulation operations, particularly refactoring operations.

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. This means the IDE must update its model, but then the issue arises of how often should the IDE do that. Ideally the IDE would update the model as often as possible (such as with each new new keystroke), but in reality that is rarely feasible since updating (and parsing) is too costly in terms of performance. A balance must be reached, and most languages update a source's model on a fixed interval such as 500ms or 200 ms., on a background thread.

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[10].

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 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 addition, these hierarchical structures are lazy-loaded (with regards to their children and contents), and are managed in an unbounded Least Recently Used (LRU) cache. 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).

[10]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

Another technique employed by several IDEs (JDT, CDT, and others) to further decrease memory usage, and speed up certain semantic features, is indexing. Indexing consists of incrementally maintaining a structure of a large set of entries 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.

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[6]. Examples of refactoring operations are: renaming or moving a method or variable; extracting a method; extracting a superclass; etc.

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[7]: refactoring participants (supporting refactoring across different plug-ins), UI support (providing a refactoring wizard), and refactoring lifecycle (managing the phases of refactoring: check initial conditions, request additional input, check final conditions, describe textual changes, display a preview, and perform the refactoring).

4 Development and Implementation

4.1 Parser

The first major task of the thesis project was the development of a D language parser. Initially, this parser was developed from scratch, using the ANTLR parser generator, but later a third-party parser was adopted. This parser was a Java port of the DMD frontend parser, which was part of the Descent IDE, another Eclipse D IDE which had been released recently. Afterwards, an initial and experimental semantic functionality was developed with the AST generated by this parser, but the original DMD AST was heavily adapted and modified to better support the development of this new IDE functionality.

4.2 Eclipse Integration

The next major task was the integration of the parser with Eclipse, to create a full IDE prototype. This task involved a significant amount of work, especially in the required learning process, since the Eclipse Platform capabilities are quite extensive, and its various components each take some time to be learnt. 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 final prototype had the following features:

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

4.3 Reference Resolving

The first tier of semantic functionality to be implemented was reference resolving. The previous initial semantic work was expanded so as to allow the Find-Defintion functionality to resolve to all kinds of D definitions, to work with any kind of references and contexts, as well as be accurate according to the full D lookup rules. This work involved even more restruturing of DMD's original AST, by improving the structure of its OO type hierarchy. D's lookup rules were studied, and several test cases were built in order to firmly support the semantic functionality correctness.

This functionality was fully completed, with the exception of function overload resolution which was not implemented, as well as the resolving of the implicit references contained in expressions nodes (except for function calls, which were implemented). With the support of Reference Resolving the Open Definition UI operation was implemented, as well as editor hyperlinking, and DDoc [11] documention text hovers.

4.4 Code Completion

The next semantic feature to be developed was code completion (perhaps one of the most useful features in an IDE). This was implemented first by modifying the previous semantic find-reference functionality to search for multiple search results, according to a partial name of an incomplete reference. Since this feature is highly dependant on the parser's ability to recover from syntax errors in order to work satisfactorily, some additional code was written that attempted to do error recovery for a few common and simple cases of syntax errors occurring during code completion.

4.5 DLTK Integration

On the final phase of project development, the IDE was heavily modified to take advantage of the Dynamic Languages Toolkit (DLTK) Eclipse project. DLTK is a recent project that aims to further extend the capabilities of the Eclipse Platform with even more infrastructure that is common and reusable by other IDEs. Most of this infrastructure is taken and adapted from JDT. In particular DTLK integration offers the following: a JDT-like rich project model, model caching, model indexing, and a great amount of UI boilerplate code.

[11] the equivalent in D of JavaDoc

All the changes required for DLTK integration were perfomed gradually, until the IDE was able to benefit from most of available DLTK functionality. Altough DTLK was still in its early stages, and had little documentation, the integration was completed without much difficulty since DTLK also reused much of the concepts and architecture of JDT, which had already been studied and understood, making the adaptation easier.

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.

The first functionality an IDE 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 IDE 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, such as the ability to perform full error analysis, and an AST with a DOM-based manipulation mechanism.

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. 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.

References

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

[7] 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)

[8] 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/)

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

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

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