Universidade do Minho
Conselho de Cursos de Engenharia
Mestrado em Informática

Master Thesis
2007/2008

An Extension to the Eclipse IDE for Cryptographic Software Development

Miguel Ângelo Pinto Marques

Supervisores:
Manuel Bernardo Barbosa - Dept. de Informática da Universidade do Minho

Acknowledgments

First of all I would like to thank my supervisor Manuel Bernardo Barbosa for all opportunities, guidance and comprehension that he has provided to me.

I would also like to thank my friends, André, Pedro and Ronnie for the support given to me during this difficult year.

To Ana the kindest person in the world, who is always there for me.

And finally, to my mother to whom I dedicate this thesis.

Thank you all,

Miguel

"Prediction is very difficult, especially about the future."

Niels Bohr

Abstract

Modern software is becoming more and more, and the need for security and trust is determinant issue. Despite the need for sophisticated cryptographic techniques, the current development tools do not provide support for this specific domain. The CACE project aims to develop a toolbox that will provide support on the specific domain of cryptography. Several partners will focus their expertise in specific domains of cryptography providing tools or new languages for each domain. Thus, the use of a tool integration platform has become an obvious path to follow, the Eclipse Platform is defined as such. Eclipse has proven to be successfully used as an Integrated Development Environment. The adoption of this platform is becoming widely used due to the popular IDE for Java Development (Java Development Tools).

In this thesis we analyze the Eclipse platform as an integration platform for cryptographic software and provide a proof of concept editor for the CAO language using Eclipse.

Application Areas: Cryptography

Keywords: Eclipse, IDE, Integration Platform

Resumo

A evolução do software atingiu um nível de progresso em que a confiança e segurança se tornaram factores determinantes a ter em conta. Apesar de a maioria das aplicações depender de técnicas criptográficas avançadas, as actuais ferramentas de desenvolvimento não fornecem suporte para este domínio em concreto. O projecto CACE visa colmatar esta lacuna através do desenvolvimento de um conjunto de ferramentas que irá suportar o domínio especifico da criptografia. Vários parceiros irão focar o seu conhecimento em domínios específicos da criptografia e fornecer novas ferramentas e linguagens de programação. Dadas as várias contribuições em ferramentas e linguagens torna-se óbvia a necessidade de usar uma plataforma de integração de ferramentas como é o caso da Eclipse Platform. O Eclipse já demonstrou ser utilizado com sucesso na criação de um ambiente de desenvolvimento através das Java Development Tools.

Nesta tese é feita uma análise ao Eclipse como uma plataforma de integração de software criptográfico e é fornecido um proof-of-concept através do desenvolvimento de um plugin que implementa um editor para a linguagem CAO.

Áreas de Aplicação: Criptografia

Palavras Chave: Eclipse, Ambiente Integrado de Desenvolvimento, Plataforma de Integração de Software

Acronyms

AWT	*Abstract Window Toolkit*
CACE	*Computer Aided Cryptography Engineering Project*
DSL	*Domain Specific Language*
WP	*Workpackage*
AST	*Abstract Syntax Tree*
API	*Application Program Interface*
SWT	*Standard Widget Toolkit*
DSL	*Domain Specific Language*
JDT	*Java Development Tools*
IDE	*Integrated Development Environment*
UI	*User Interface*
JAR	*Java Archive*
CVS	*Concurrent Version Support*
DSL	*Domain Specific Language*
EBNF	*Extended Backus-Naur Form*
XML	*Extended Markup Language*
PDE	*Plugin Development Environment*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Contextualization

This dissertation results from a collaboration with the Cryptography and Information Security group at Computer Science and Technology Center (CCTC) from the University of Minho in the FP7 European Project titled Computer Aided Cryptography Engineering (CACE).

1.2 Motivation

Software development typically follows a design flow and uses a common set of tools (e.g., compilers, debuggers) that automate the tasks performed by developers. Although these tools are always evolving and becoming very complex they usually do not provide support for any specific domain. The CACE project aims at providing a toolbox that supports the production of high quality cryptographic software. Quoting [1]:

"To enable verifiable secure cryptographic software engineering to non-experts by developing a toolbox which automatically produces high-performance solutions from natural specifications"

The project is separated into five workpackages/partners (WP) where each of them will develop different tools from specific cryptographic domains. Figure 1.1 represents a typical architecture of a system using advanced cryptographic techniques as well as the fields where each workpackage (WP) will focus on.

WP1 (CAO): WP1 will focus on the development of a language, named CAO, that allows to specify efficient and secure implementations of low-level cryptographic primitives (such as encryption schemes and digital signatures, including symmetric cryptographic primi-

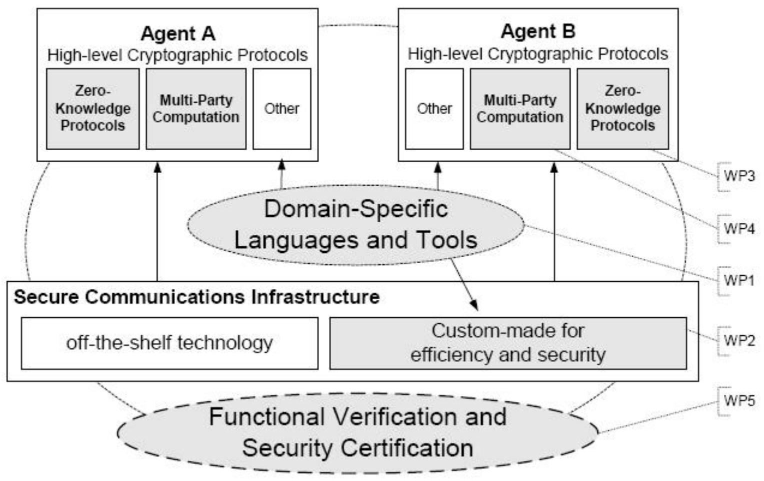

Agent A
High-level Cryptographic Protocols
Zero-Knowledge Protocols
Multi-Party Computation
Other
Agent B
High-level Cryptographic Protocols
Other
Multi-Party Computation
Zero-Knowledge Protocols
Domain-Specific Languages and Tools
Secure Communications Infrastructure
off-the-shelf technology
Custom-made for efficiency and security
Functional Verification and Security Certification
WP3
WP4
WP1
WP2
WP5

Figure 1.1: Typical architecture of a system using advanced cryptographic techniques

tives) which are then transformed into executable code by an optimizing, security-aware compiler.

WP2 (NaCl): Accelerating Secure Networking will provide all of the core operations needed to build high-level cryptographic tools such as zero-knowledge compilers (WP3) and systems based on multiparty computation (WP4). It will contain a new easy-to-use high-speed software library for network communication and basic cryptographic primitives such as encryption, signatures, etc.

WP3 (ZKC): WP3 will develop a compiler that, given a high-level specification of the goals of a Zero Knowledge Proof, automatically finds and generates a protocol that meets specification along with code that implements the protocol.

WP4 (SDMI): Securing Distributed Management of Information focuses on facilitating the practical use of secure multiparty computation protocols in real-world applications. To this end, WP4 will develop a language that allows software engineers to specify secure multiparty computations and a compiler which transforms these specifications into executable code.

WP5 (VERIF): Formal Verification and Validation aims to globally address both functional and security requirements in resulting cryptographic software implementations, analyzing the transformations between different levels of abstraction, and resorting to previous scientific results in the formal validation, verification and certification of secure software.

The resulting tools will be combined into a toolbox that will allow non-experts to develop high-level cryptographic applications and business models.

All of the languages, compilers and tools mentioned will have a vast set of different technologies and their integration into one toolbox can be very difficult. Thus, the use of a tool integration platform has become an obvious path to follow, the Eclipse Platform is defined as such. Eclipse has proven to be successfully used as an Integrated Development Environment. The adoption of this platform is becoming widely used due to the popular IDE for Java Development (Java Development Tools). In fact, there are several recently developed tools based on Eclipse, such as the Bioclipse, an Eclipse distribution for biochemists and bioinformatics [6]. This dissertation will focus on the integration of the CACE tools into the Eclipse Platform.

1.3 Aims and Contributions

The main objectives of this dissertation are:

- To explore the Eclipse architecture and plugin development process.
- To perform the requirements analysis for an Eclipse plugin to support the CAO domain specific language (based on the outputs of the initial discussion stages in the CACE project). Features to look at should include:
  - syntax highlighting;
  - code-completion;
  - compilation support;
- To design and implement the above referred plugin.

1.4 Document Structure

This document is organized as it follows:

Chapter 1 contains the introduction: motivation, objectives and document structure. Chapter 2 provides a general introduction to the Eclipse Platform, more specifically the Java Development tools (JDT). Chapter 3 introduces the Eclipse architecture, plugins and the facilities it provides for IDE development. Chapter 4 refers to the CAO language and focuses on parsing and abstract syntax tree generation. Chapter 5 explains the developed plugin. Chapter 6 is devoted to the implementation details of the mentioned plugin. The final chapter 7 provides conclusions and presents a small introduction to future work in the field.

Chapter 2

What is the Eclipse Platform?

"Eclipse is a platform that has been designed from the ground up for building integrated web and application development tooling. By design, the platform does not provide a great deal of end user functionality by itself. The value of the platform is what it encourages: rapid development of integrated features based on a plugin model" [2].

Eclipse development began in the late 90's when IBM had in mind the establishment of a common platform for all IBM developed products. The main idea was to avoid the duplication of the most common elements of the products infrastructure and to allow customers to have a more integrated experience while switching from one product to another. The result was an extensible tools integration platform which was later named Eclipse. This platform had a common user interface model for all tools and was fully compatible with all operating systems (OS). Not only did it become compatible with many OS's but it was also optimized to make the most of their native user interface (UI) application development interfaces (API) in order to provide the same *look and feel* as that of the OS. Extensibility was achieved through an adopted plugin architecture that allowed the deployment of any plugin/tool in all of the platforms without the need for any customization. Eclipse began mainly as a Java IDE and it evolved into a set of tools and runtime libraries for building applications from plugin components. The most widespread uses of Eclipse are still programming IDE's for languages such as Java and C++; nevertheless there are others such as NASA's Mars rover mission planning software, or even the bit-torrent client Azureus.

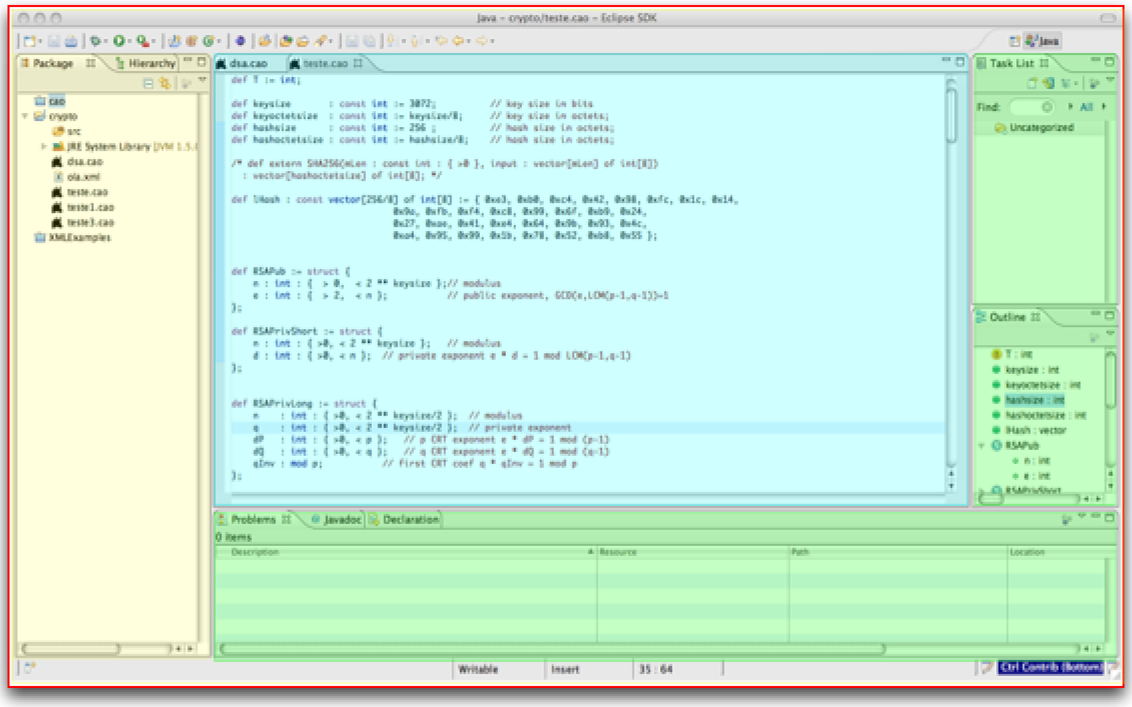
behavior behaviour

2.1 JDT: Java Development Tools

A typical Eclipse IDE is built to provide to programmers comprehensive facilities for software development. Usually they consist of several components - a source code editor, compiler/interpreter support, build tools, a debugger, etc. Each component functionality is usually connected through a *main window/core* interface that provides all the between components. The most popular Eclipse distribution contains the Java Development Tools (JDT) IDE. The JDT is a *set* of plugins that provides tools to help in the development life-cycle of a Java software product, from the creation of the project to its deployment.

2.1.1 Eclipse Main Features

The JDT uses several of the main Eclipse infrastructures for building IDE's: *workspaces*, *wizards*, *editors* and *markers*. The *main window* is named the *workbench* (Figure 2.1).



```
def T := int;

def keysize       : const int := 3072;          // key size in bits
def keyoctetsize  : const int := keysize/8;     // key size in octets;
def hashsize      : const int := 256 ;          // hash size in octets;
def hashoctetsize : const int := hashsize/8;    // hash size in octets;

/* def extern SHA256(mLen : const int : { >0 }, input : vector[mLen] of int[8])
  : vector[hashoctetsize] of int[8]; */

def IHash : const vector[256/8] of int[8] := { 0xe3, 0xb0, 0xc4, 0x42, 0x98, 0xfc, 0x1c, 0x14,
                             0x9a, 0xfb, 0xf4, 0xc8, 0x99, 0x6f, 0xb9, 0x24,
                             0x27, 0xae, 0x41, 0xe4, 0x64, 0x9b, 0x93, 0x4c,
                             0xa4, 0x95, 0x99, 0x1b, 0x78, 0x52, 0xb8, 0x55 };


def RSAPub := struct {
    n : int : {  > 0,  < 2 ** keysize };// modulus
    e : int : {  > 2,  < n };            // public exponent, GCD(e,LCM(p-1,q-1))=1
};

def RSAPrivShort := struct {
    n : int : { >0, < 2 ** keysize };   // modulus
    d : int : { >0, < n };  // private exponent e * d = 1 mod LCM(p-1,q-1)
};


def RSAPrivLong := struct {
    n    : int : { >0, < 2 ** keysize/2 };  // modulus
    q    : int : { >0, < 2 ** keysize/2 };  // private exponent
    dP   : int : { >0, < p };   // p CRT exponent e * dP = 1 mod (p-1)
    dQ   : int : { >0, < q };   // q CRT exponent e * dQ = 1 mod (q-1)
    qInv : mod p;            // first CRT coef q * qInv = 1 mod p
};
```

Figure 2.1: Eclipse Workbench

Projects and Workspaces

Eclipse projects help structuring a software project. They serve as a container where all the project resources are stored (packages, images, source code files, etc.) and provide support for specific project preferences (for instance the version of the compiler to use). Workspace is the name that Eclipse gives to the directory hierarchy that contains all of the existing projects and plugin preferences.

Typically, workspaces are used to group several projects that somehow relate with to each other. For instance, it is common to have two separate workspaces, one for C projects and one for Java.

Editors

Editors are one of the most important and more frequently used components in Eclipse. An Eclipse editor provides all the basic operations performed when editing a document, such as opening, saving, undoing. By extending a basic editor one can produce a full-blown editor such as the Java editor. Eclipse contains several editors that sometimes are not interpreted as one, such as the *XML*, *HTML*, simple text, etc. One common misconception is to associate the Java editor as the plugin for Java development, but in reality the JDT is made of several plugins and one of them is the editor.

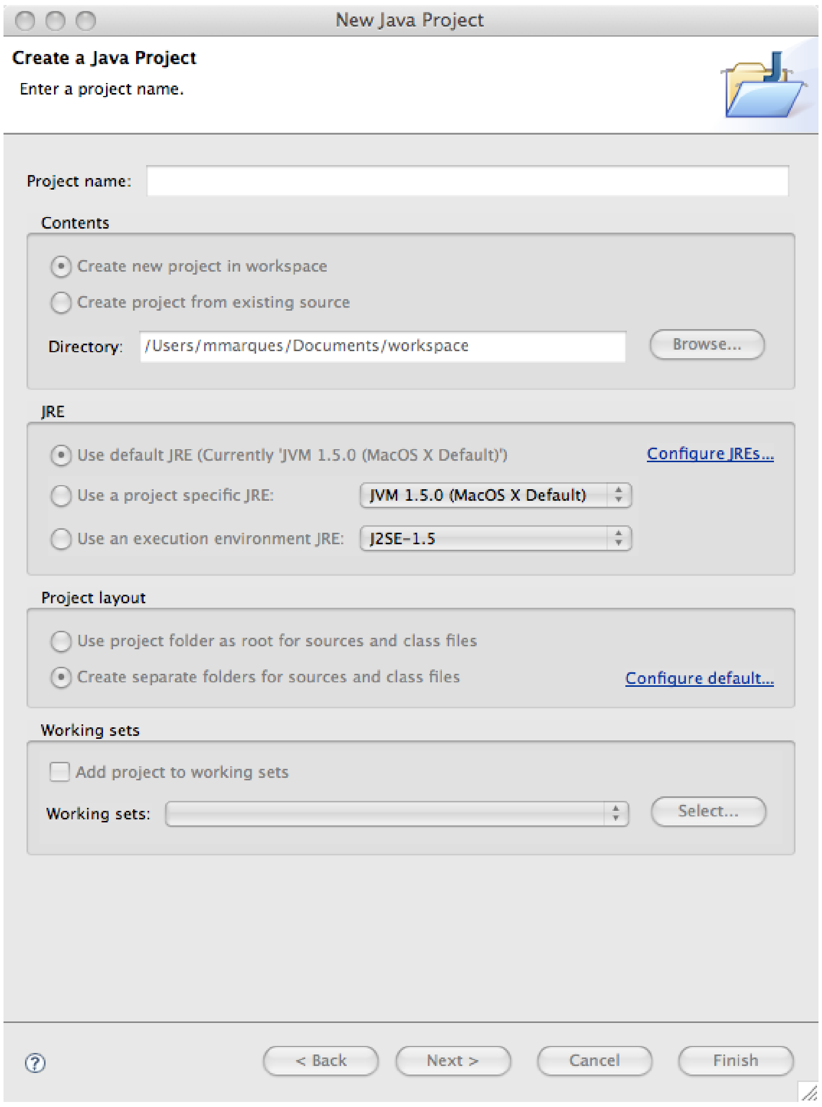
Wizards

Wizards are a main part of the JDT, and they are used to guide the user through a sequenced set of tasks. The most common one is the *new java project wizard* (2.2) that helps setting up a new java project (java runtime, classpath, etc).

Views and Perspectives

Views provides information about one object in the workbench. For example, content outlines normally show the information about the content of the editor in a structured way (Figure 2.1). Eclipse provides many standard views; others such as content outline for specific languages are provided by plugins.

The workbench can have many different perspectives. Each perspective can contain different editors and viewers at the same time, but only one opened perspective visible at a time. Perspectives are always associated with the domain of the task one is performing on Eclipse. While on language development such as Java it is convenient to have the content outline on a view, Java documentation on another, notes, etc., when committing the code to a version control system such as CVS a new perspective is needed containing appropriate viewers to this task. Figure 2.3 shows the selection of different perspectives. Perspectives are also fully configurable, which leaves space space for a full customization by the user.



New Java Project

Create a Java Project

Enter a project name.

Project name:

Contents

[✓ Create new project in workspace]

Create project from existing source

Directory: /Users/mmarques/Documents/workspace

Browse...

JRE

[✓ Use default JRE (Currently 'JVM 1.5.0 (MacOS X Default)')]

Configure JREs...

Use a project specific JRE: JVM 1.5.0 (MacOS X Default)

Use an execution environment JRE: J2SE-1.5

Project layout

Use project folder as root for sources and class files

[✓ Create separate folders for sources and class files]

Configure default...

Working sets

Add project to working sets

Working sets:

Select...

< Back Next > Cancel Finish

Figure 2.2: New Java Project Wizard

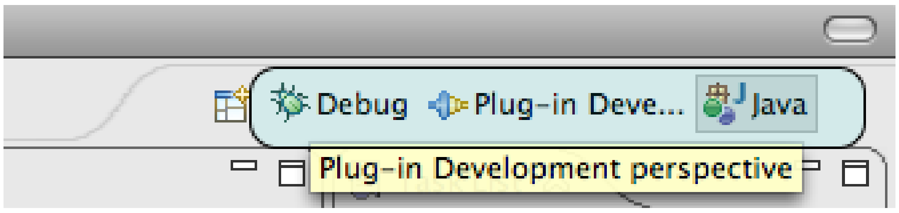
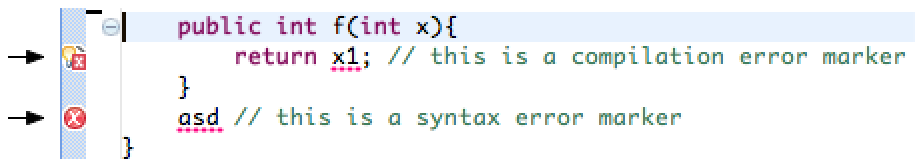


Figure 2.3: Choosing Perspectives

Markers

Markers are used to provide annotations on a specific resource: compiler error messages, to-do items, debugger breakpoints, etc. Eclipse allows to extend the base marker to create markers for each different situation. The most common markers are present in the majority of editors and provide information about compilation errors. In figure 2.4 one can see at least two types of markers.



```
public int f(int x){
    return x1; // this is a compilation error marker
}
asd // this is a syntax error marker
}
```

Figure 2.4: Different Markers

2.1.2 Debugger

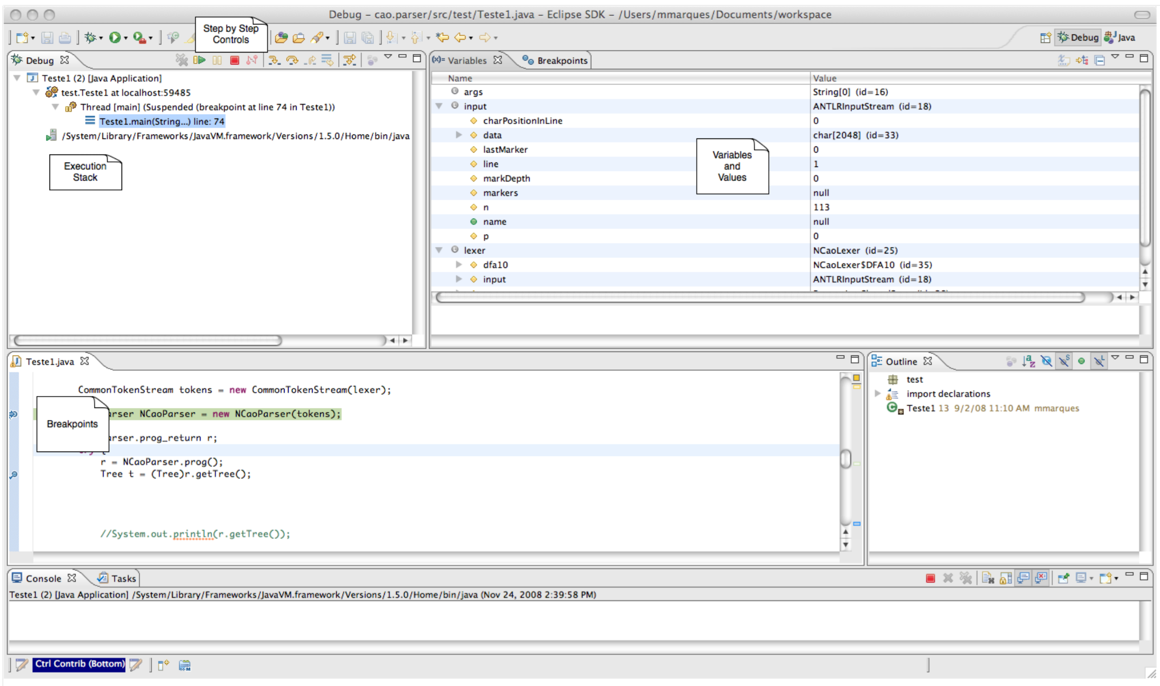
The JDT features a built-in Java debugger that provides all standard debugging functionality. Figure 2.5 shows the overall debug perspective. The debugger includes the ability to perform step by step execution, to set breakpoints and values, to inspect variables and values, and to suspend and resume threads.

2.2 JDT Editor

There is a common misconception that an editor plugin for Eclipse contains all the well-known functionalities provided by the JDT. The Java editor is only a small part of the whole set of plugins that form the JDT. In Eclipse's terminology an editor is the component that is used to edit documents (see chapter 2.1.1), other concepts such as projects, natures, wizards, builders and debuggers are *not* part of an editor. In general, many features of an editor can be classified according to their behavior ([19]) :

Simple or Text-based Features

Simple features rely neither on any underlying model nor on the current input of the editor are classified as simple or text model-based features. Such features include line numbering, quick differences, block commenting, lexical (keyword based) syntax highlighting (Figure 2.6), etc.



```
Teste1 (2) [Java Application]
  test.Teste1 at localhost:59485
    Thread [main] (Suspended (breakpoint at line 74 in Teste1))
      Teste1.main(String...) line: 74
  /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/bin/java
```

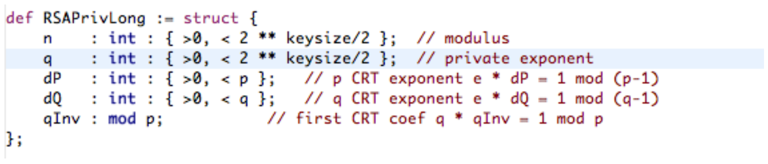
Name	Value
args	String[0] (id=16)
input	ANTLRInputStream (id=18)
charPositionInLine	0
data	char[2048] (id=33)
lastMarker	0
line	1
markDepth	0
markers	null
n	113
name	null
p	0
lexer	NCaoLexer (id=25)
dfa10	NCaoLexer$DFA10 (id=35)
input	ANTLRInputStream (id=18)

```
        CommonTokenStream tokens = new CommonTokenStream(lexer);

        rser NCaoParser = new NCaoParser(tokens);

        rser.prog_return r;

            r = NCaoParser.prog();
            Tree t = (Tree)r.getTree();



            //System.out.println(r.getTree());
```

```
test
import declarations
Teste1 13 9/2/08 11:10 AM mmarques
```

```
Teste1 (2) [Java Application] /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home/bin/java (Nov 24, 2008 2:39:58 PM)
```

Figure 2.5: Overview of the Debug Perspective

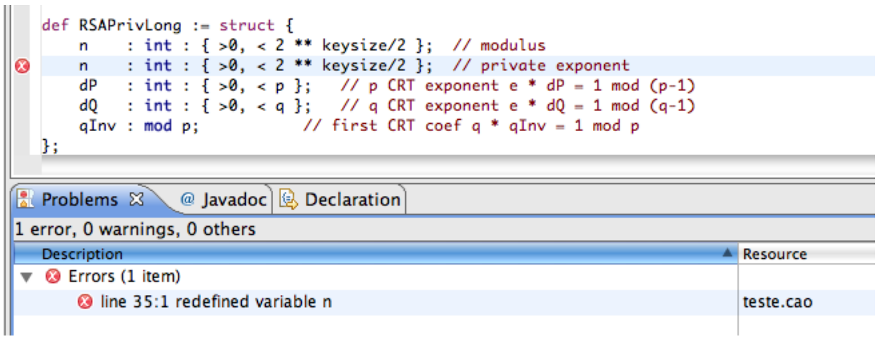


```
def RSAPrivLong := struct {
    n    : int : { >0, < 2 ** keysize/2 };  // modulus
    q    : int : { >0, < 2 ** keysize/2 };  // private exponent
    dP   : int : { >0, < p };    // p CRT exponent e * dP = 1 mod (p-1)
    dQ   : int : { >0, < q };    // q CRT exponent e * dQ = 1 mod (q-1)
    qInv : mod p;            // first CRT coef q * qInv = 1 mod p
};
```

Figure 2.6: Syntax highlight in an eclipse editor

Advanced AST-based Features

There are there are two common types of programming errors: syntactic and semantic. Sometimes while programming one makes naive syntax errors such as forgetting to close a parenthesis or a curly bracket, in other situations uses non-declared variables, etc. Knowing the current state of the program in the text editor in almost real-time reduces these common mistakes that can be time-consuming, specially when changing between programming languages. With the support of the compiler, the advanced Abstract Syntax Tree (AST) based features perform syntactic and semantic analysis using a parser-generated AST and provide the developer with a notion of the current state of the program in the text editor. Figure 2.7 shows an error generated due to the re-declaration of a variable.



```
def RSAPrivLong := struct {
    n    : int : { >0, < 2 ** keysize/2 };  // modulus
    n    : int : { >0, < 2 ** keysize/2 };  // private exponent
    dP   : int : { >0, < p };   // p CRT exponent e * dP = 1 mod (p-1)
    dQ   : int : { >0, < q };   // q CRT exponent e * dQ = 1 mod (q-1)
    qInv : mod p;           // first CRT coef q * qInv = 1 mod p
};
```

1 error, 0 warnings, 0 others

Description	Resource
Errors (1 item)	
line 35:1 redefined variable n	teste.cao

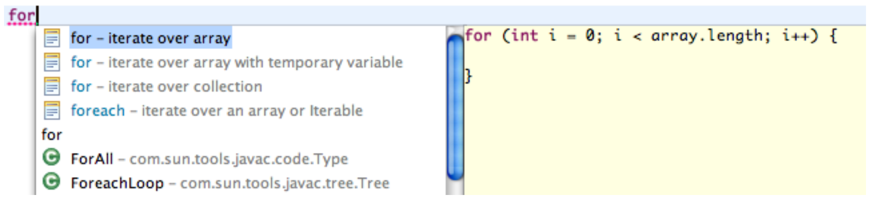
Figure 2.7: Programming error marker.

Coding Assists

Coding assists infers actions from a developer's current context of interaction with the source code. Eclipse distinguishes three types of coding assists: content assist, quick assist, and quick fix.

Content assist is responsible for code completion. Depending on the implementation it, can provide static content completion based on templates or dynamic content completion based on the current status of the code (using AST and parser) - figure 2.8 shows content assist in action.

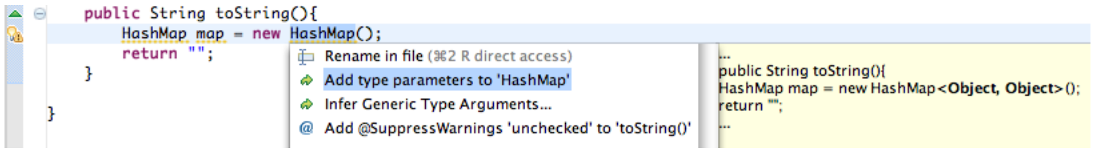
Quick assists suggests and performs local code transformations based on the content of the text editor. There can be several suggestion engines and each one provides suggestions regarding one specific domain. For instance, one engine may suggest alterations in a compilable that but it can be further improved. In Figure 2.9 content assist's engine suggests a change on the `HashMap` declaration to include type parameters, a new feature


for
for - iterate over array
for - iterate over array with temporary variable
for - iterate over collection
foreach - iterate over an array or Iterable
for
ForAll - com.sun.tools.javac.code.Type
ForeachLoop - com.sun.tools.javac.tree.Tree

```
for (int i = 0; i < array.length; i++) {

}
```

Figure 2.8: Content Assist

in Java 6.



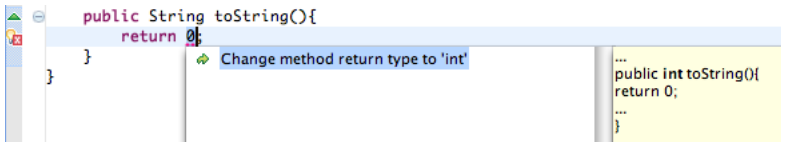
```
public String toString(){
    HashMap map = new HashMap();
    return "";
}
}
```

Rename in file (⌘2 R direct access)
Add type parameters to 'HashMap'
Infer Generic Type Arguments...
Add @SuppressWarnings 'unchecked' to 'toString()'

```
...
public String toString(){
HashMap map = new HashMap<Object, Object>();
return "";
...
```

Figure 2.9: Quick Assist

Finally, *quick fixes* shows syntactic and semantic errors found while editing (Figure 2.10) and offers suggestions for corrections.



```
public String toString(){
    return 0;
}
}
```

```
...
public int toString(){
return 0;
...
}
```

Figure 2.10: Quick Fix

2.3 Language and Feature Support

Providing a language support can be either an easy or daunting task, which always depends on the required features. The simplest editor may have only one purpose, recognizing file extension and supporting all the basic document editing operations - this is very easy to implement as Eclipse provides all the facilities for it. Adding a simple feature, such as syntax highlighting can make the editor more user friendly. This simple improvement, fosters the need to know the whole language syntax and provides means and mechanisms to distinguish between all the different language elements that can be found. With the help of lexical patterns and rules a simple syntax highlighter can be developed. However, if the language supports user defined data types, the lexical rules wont suffice.

Regarding coding assists only content assist can be provided without a having support from a more complex model due to the possibility of providing code completion based on templates. Other features such as *quick fix* or *quick assist* (Figure 2.2), that provide assistance based on the current status of the program, need parser and AST support.

Quoting [19]: *"Features that are based on AST support dominate the feature space. Overall, 21 out of 35 features require AST support. There include three coding assists and 18 advanced features..."*. It becomes obvious that a parser based AST is the pillar needed in order to provide a state of the art editor for a given programming language.

2.4 Compiler Support

To provide compiler integration into Eclipse, two different approaches can be pursued. The first one, is taken by the JDT [3]: implementing an incremental[1] compiler in Java as an Eclipse builder; the second is to add the existing compiler as an external tool builder.

The first one is very difficult to achieve as it has two problematic issues: implementing a new compiler in Java and making it work incrementally. Compilers are very complex programs and the question arises: Is it possible to implement a new compiler for the language in Java, and is it worth it? This is the one of the main reasons that makes the JDT the most complete set of language development tools integrated into Eclipse. They have available an incremental compiler that is the most of a powerful core that an Eclipse plugin can have.

The second alternative, despite being of easy achievement, also has its setbacks. If a compiler is added as an external tool there will be no access its own internal data structures and therefore, there will be no way to report the errors generated by the compiler to Eclipse. The same happens for the AST-based features, without an AST implemented in Java the is no way to obtain advanced features. A possible solution to this problem would be having the parser and an AST implemented in Java working together with a compiler as an external builder. The AST and parser could be used to generate on the fly assistance and the builder for the final compilation step. A more challenging solution would be to implement a custom middleware that would extend the compiler and allow it to communicate with Eclipse.

[1]Incremental compilers are used for systems that allow modifications or additions to a to a program by extending the previously compiled programs or replacing some parts of the program. [14]

Chapter 3

Eclipse Overview

In this chapter we present an overview of the Eclipse architecture, its evolution and the frameworks that it provides for IDE development.

3.1 Eclipse Architecture

Eclipse's history is an important part of the explanation for how the architecture structure was chosen, and why. Eclipse's history began in the late 1990's when the internet boom occurred. This boom affected the way companies thought about their products making them aware that online applications were part of the future. With this, the structure of applications started to change, leaving the concept of core standalone applications and opening way to more heterogeneous applications composed by several different technologies - Java, Php, Perl, Asp, Web services. With each of these different technologies came the need for new client tools that supported their development and maintenance, fact that contributed to the appearance of several new IDE's. Although there were several IDE's and support for all the different technologies, one problem appeared, the interoperability between them was not so good. Also, at the same time, the open source community gained more credibility with the appearance of emacs, mozilla browser, apache web server which companies started to use and more importantly, provide support and even extend. Within this panorama IBM decided to create a set of application development tools that worked together with the possibility to extend and integrate new tools. Initially the Eclipse architecture was designed to provide support for integration of development tools and it has later evolved into a platform that can host any kind of client application: the Rich Client Platform.

3.1.1 Plug-ins

"A plugin is the smallest unit of Eclipse Platform function that can be developed and delivered separately" [5].

Eclipse is made of a massive set of plugins that interconnect with each other through the use of extensions and extension points. Everything in Eclipse is a plugin, there are plugins on top of plugins that use other plugins. Plugins are developed with the Plugin Development Environment (PDE).

Plug-in Anatomy

A plugin consists of a Java Archive (JAR) containing all the source code and resources it relies on - images, code libraries, icons, etc. Along with the resources there are two important files: *manifest.mf* and *plugin.xml* (Figure 3.1). The first one describes the plugin to the eclipse runtime, it contains the plugin identifier, version, name, code location, dependencies and exports. The second one contains all existing plugins extensions and the extension points it declares.

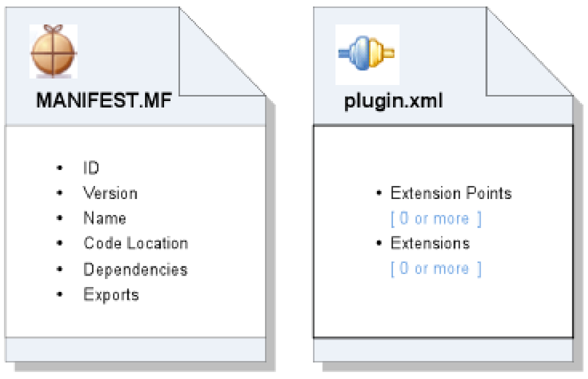


Figure 3.1: Manifest and plugin.xml files
retrieved from [8].

Extensions and Extension Points

Plugins interconnect with other plugins through the use of *extension points* and *extensions.* They are best described by the electrical socket and plug analogy: an *extension point* is a socket and an *extension* the plug that connects. As with electrical outlets, *extension points*, may have many shapes and sizes, and will only work with the appropriate *extension.* Each *extension point* defines a contract, typically a combination of XML schema and Java interfaces, that *extensions* must comply to. This way, the extended plugin needs to know nothing about implementation of the extending plugin. If a plugin B extends a plugin A, A doesn't know anything about B, only that B complies with the declared *extension point*'s schema.

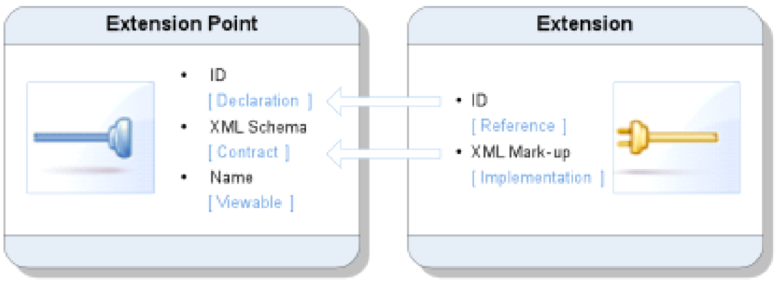


Figure 3.2: Eclipse Mechanism for Extensibility
retrieved from [8].

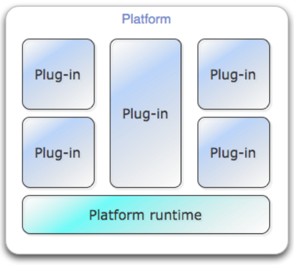


Figure 3.3: Layered Platform

Listing 3.1 shows an *extension point* and listing 3.2 shows one possible corresponding *extension* [1].

Listing 3.1: Extension Point

```
<plugin>
  <extension-point id="cace.example.languages" name="CACE␣Languages"
      schema="schema/languages.exsd"/>
</plugin>
```

Listing 3.2: Extension

```
<extension point="cace.example.languages" >
  <extension="cao" name="Cryptography␣Aware␣Language␣and␣Compiler"/>
  <compiler name="cao␣compiler" class="cace.org.parser" />
</extension>
```

3.1.2 Architecture

The main idea behind Eclipse was the creation of an IDE-related tool integration platform while maintaining the same operating paradigm between tools that is, an extensible platform for development tools. Eclipse architecture was designed with extensibility and component reuse in mind, resulting in a layered platform extensible through plugins (Figure 3.3) with a runtime core. This platform, know as the Eclipse Platform, provides a set of common services and *frameworks* required to support a tool integration platform. Each of these services and frameworks represent common facilities required by any IDE related tool. This includes support for user interface, widgets, project model, resource management, compilers, builders, language independent debug model, version support management, etc. Figure 3.4 shows the overall first Eclipse architecture, as described in [5], it includes the frameworks for the referred facilities: platform runtime, workbench (contains JFace and Standard Widget Toolkit (SWT)), Help, Team and Workspace. Adding new features is always done by extending some existent framework or plugin. For example, if one wants to provide SVN support it will extend the Team framework or in case of an editor, the workspace framework.

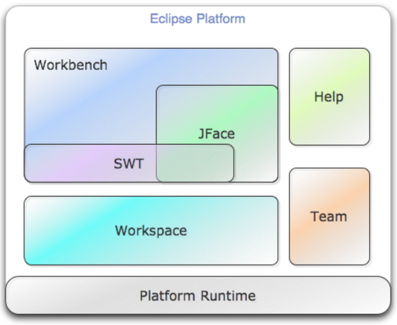


Figure 3.4: Eclipse Initial Architecture

This architecture proved to be very useful, making developers and contributers realize that the core of the Eclipse technology was powerful enough to support non-IDE oriented applications, due to the fact that many components where not particularly specific to IDE's - plugins, workbench, help system, etc. With the adoption of the OSGi [18] architecture came the Eclipse Rich Client Platform that is defined as: *"The minimal set of plugins needed to build a rich client application (...) "* [2]. With it, non-IDE applications can be developed using a chosen subset of the platform, thus, maintaining the same plugin based philosophy. The common components were separated from the initial IDE oriented frameworks resulting, in new frameworks or abstractions of the existing ones. Subsequently,

[1]This example assumes that the language schema contains the information for the `extension` and `name` elements.

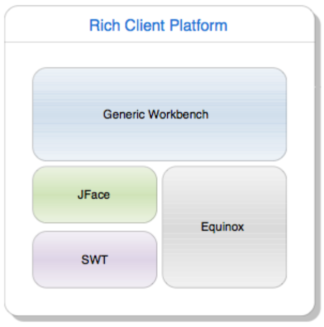


Figure 3.5: Rich Client Platform
retrieved from [15].

frameworks comprising this new architecture where classified into three groups/layers: Platform Runtime, Rich Client Platform and Workbench IDE or Eclipse IDE Platform (Figure 3.1.2). Each layers extends the functionality of the previous one.

3.2 Eclipse IDE Platform

Eclipse provides a powerful platform for IDE development. It can be used for development of all the key components in an IDE such as (Figure 3.7): resource management, text editor framework, language independent debug model, builder integration, version control repository integration, help system and update manager.

3.2.1 IDE Workbench

"The Eclipse User Interface (UI) is built around a workbench that provides the overall structure and presents and extensible UI to the user" [5].

The workbench API and implementation is built with the help of two toolkits: SWT and JFace. SWT is responsible for the native *look and feel* on each operating system and it's a known alternative to the Java Abstract Window Toolkit (AWT) and Java Swing toolkits. The native *look and feel* is accomplished by using the Java Native Interface (JNI) to access the native GUI libraries of the operating system. JFace is a UI toolkit that provides helper classes for developing monotonous programming tasks. It provides components, images, dialogs, preferences, wizard frameworks, progress reporting, and, most importantly views and editors.

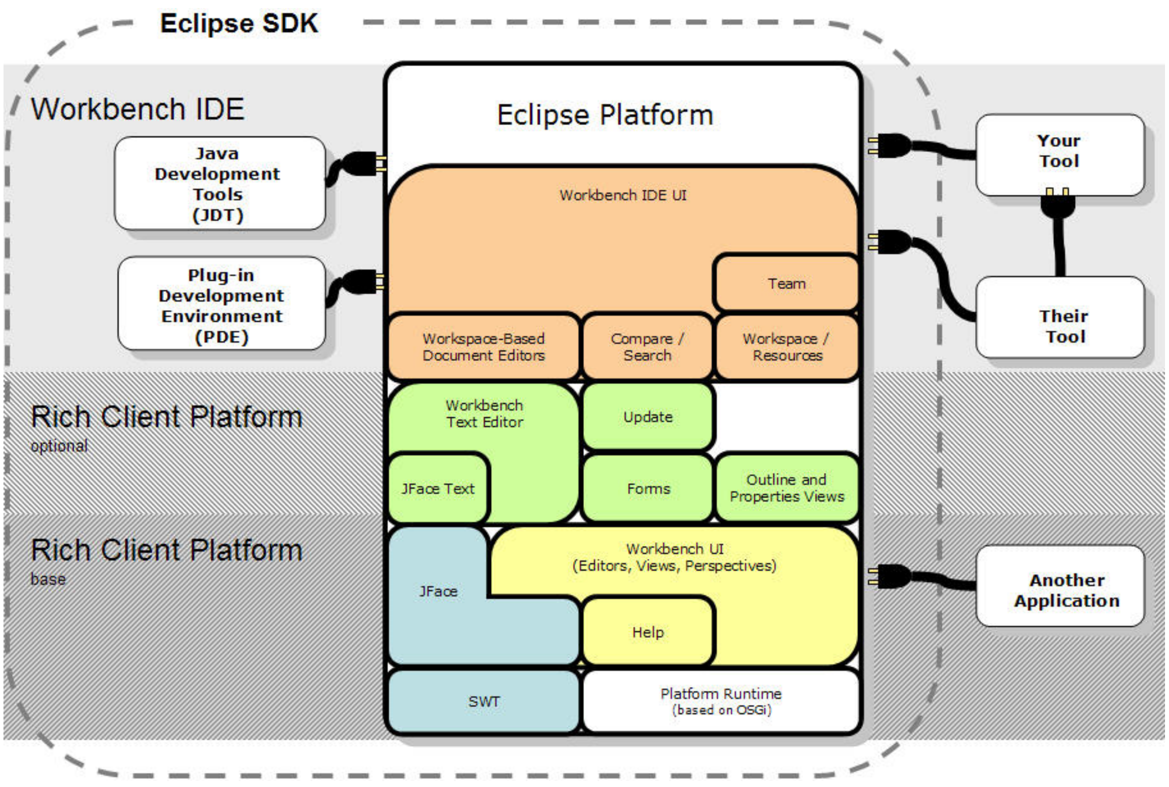


Figure 3.6: Eclipse Platform Architecture
retrieved from [15].

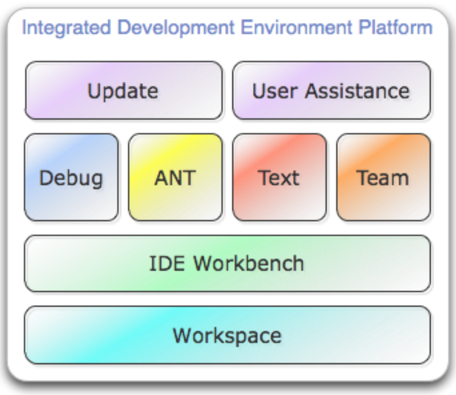


Figure 3.7: Integrated Development Environment

3.2.2 Help and Team

The help mechanism allows to define tools to integrate help facilities such as the Java documentation that is available while developing Java programs. Eclipse also allows all projects in the workspace to be stored under Concurrent Versioning System (CVS), and it supplies extension points for new providers that allow new kinds of team repositories to be plugged in such as the Subclipse plugin that adds support for Subversion (SVN).

3.2.3 Update Manager

The update manager provides a convenient way to add new features into the Eclipse IDE. These features are automatically installed and when possible, updated. Since plugins are too small for distribution they are grouped into features. For instance, instead of installing all of the plugins that form the JDT, Eclipse installs the JDT feature that contains all of those plugins.

3.2.4 Debug

Eclipse provides a framework for building and integrating debuggers known as the debug platform. The debug platform defines a set of Java interfaces modeling a set of common actions and artifacts to many debuggers, known as the debug model. Artifacts are threads, stack frames, variables, and breakpoints; and actions are suspending, stepping, resuming, and terminating. The platform does not provide an implementation of a debugger, but only the model. However, it does provide a basic debugger user interface/perspective that can be extended with features specific to a particular debugger. The debug platform also provides a framework for launching applications from within the Eclipse IDE and a framework to perform source lookup.

3.2.5 Workspace API

The most important features workspace API provides are: resource management, project natures and markers[2].

Resource Management

The workspace API allows to manipulate resources in any project using resource handles. A resource handle is a pointer to any folder, file or project that exists in the *workspace*. It allows to create, delete or move resources.

[2]For markers refer to: 2.1.1

Project Natures and Builders

Project natures are responsible for the association between a project and specific plugin or tool, they give projects their "personality". When a project is associated/tagged with a nature, it means that the plugin is configured to use with that project. For example, the Java nature tags a project that contains source code for a Java program and configures the builder[3]. Each project can always have as many natures as required. Natures are often used to associate a builder and an image/icon to a project (Figure 3.8).

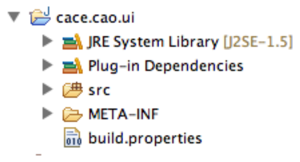


Figure 3.8: Java Project Nature Icon

Builders are very important components associated to the project that are invoked periodically. Although builders are typically used to integrate compilers, they can perform any kind of processing. A project can have as many builders as it needs. Each project has the notion of a resource delta that knows the variation of all of the current opened resources (e.g, source files). When a builder is called it has access to the resource delta and builds all or only the new/changed resources, according with its own implementation.

3.2.6 Text Framework

Eclipse includes a text framework that is the foundation for text related tools. This framework provides a model (`IDocument`) for text manipulation and all the convenient manipulation methods; it also provides a text editor framework which contains a basic text editor that can be extended to implement new editors.

The text framework includes several components for editor development thus only the fundamental ones will be mentioned:

- Document Infrastructure
- Source Viewers
- Text Editor Framework

[3]In fact, there is no such thing as a Java Project, Eclipse provides a "new java project" wizard that creates a new project and loads the Java nature.

Document Infrastructure

All of the editor input is managed through a document model. The document model (`IDocument`) provides text content manipulation, text position management, document partitioning, searching and change notifications.

Each editor window is connected to a `DocumentProvider` class. A document provider class is a translator between the editor text input and the corresponding `IDocument`. Given an editor input, the document provider returns the corresponding `IDocument`. It also communicates changes that were made in the `IDocument` to the text editor. The document provider also manages the current state (checks if the document has changed) and manages character encodings.

The text editing framework also provides support for partitioning a document. Partitioning a document divides it into non-overlapping partitions that have a specific content type. In text editors, partitioning is typically used to separate language comments from the rest of the source code. Features such as syntax highlighting and code completion are implemented regarding a partition type. For instance, a partition of comments will have different syntax highlight than a partition with source code.

In the class diagram on Figure 3.10 its possible to see how the documents, providers and partitioners partitioners relate with each other.

Source Viewers

Each editor contains a reference to a source viewer. A source viewer is directly connected to a text viewer that provides the default behavior of a text editor such as undo, redo, copy/paste; features that are common in all text editors and are very hard to implement. A source viewer extends a text viewer introducing more capabilities such as syntax highlighting, error markers, amongst others.

Eclipse provides a default Source Viewer implementation. Nevertheless it needs to be configured in order to add custom implemented syntax highlighters and content assist processors to the editor. This is done by extending the `SourceViewerConfiguration` class.

Text Editor Framework

Adding a text editor for a plugin can be done by using the existing basic text editor or by extending it. To provide specific domain features such as syntax highlight, reformatting, code assist one just needs to customize the new extended editor. Text editor framework contains a hierarchy of classes for editor development (Figure 3.9) where each class is responsible for specific editor features:

- `AbstractTextEditor` - Abstract base implementation of a text editor;
- `AbstractDecoratedTextEditor` - An intermediate editor that contains functionality not present in the leaner `AbstractTextEditor` class but used in many editors (especially source editing), such as line numbers, change ruler, overview ruler, print margins, current line highlighting, etc;
- `TextEditor` - The standard text editor for file resources. This is the common text editor that Eclipse uses when the file extension isn't associated to a custom editor;

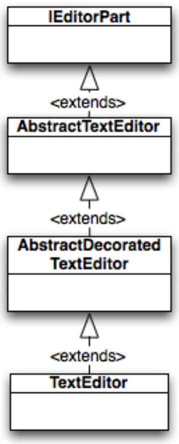


Figure 3.9: TextEditor class diagram

When implementing a new text editor it is common to extend the `TextEditor` and inherit all the functionalities and only in very rare situations extend one of parent classes.

Figure 3.10 shows how all of the previously referred components interact with each other within an editor. A text editor implementation is always dependent on a document provider, the source viewer and its respective configuration.

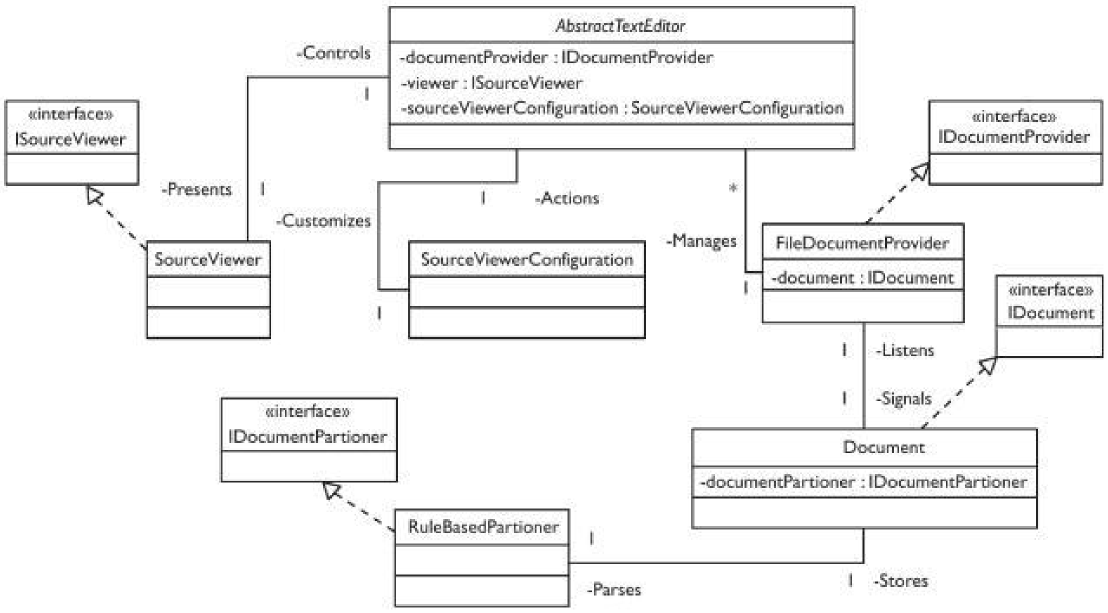


Figure 3.10: TextEditor Common Architecture
retrieved from [15].

Chapter 4

CAO Language

This chapter introduces the CAO Language and focuses on the parsing tools and AST construction considerations.

4.1 Introducing the CAO Language

CAO is a Domain Specific Language (DSL) for describing cryptographic software. The CAO language allows to specify efficient and secure implementations of low-level cryptographic primitives (such as encryption schemes and digital signatures, including symmetric cryptographic primitives) which are then transformed into executable code by an optimizing, security-aware compiler. CAO can be described as a mix of small features of C, Ocaml and Haskell where the first two are the most present. Listing 4.1 shows a snippet of CAO source code to provide a quick look into the language syntax.

Listing 4.1: CAO Example

```
def x : matrix[10][10] of int;
def y : vector[20] of int;

def point := struct[
                x : int;
                y : int;
                z : int;
                   ];

def max( x : int, y : int ) : int {
    def t : int;
    if( x > y )
        t := x;
    else
        t := y;
    return t;
}

def min( x : int, y : int ) : int {
    def t : int;
    if( x > y )
        t := y;
    else
        t := x;
    return t;
}

key_public : int : { public };
key_secret : int : { secret };
modulus : int : { public };

rsa_enc( x : int ) : int : { encrypt } {
    return ( x ** key_public ) \% modulus;
}

rsa_dec( x : int ) : int : { decrypt } {
    return ( x ** key_secret ) \% modulus;
}
```

4.2 ANTLR Parser Generator

ANother Tool for Language Recognition (ANTLR), is a *LL(*)* parser generator. It can generate lexers, parsers, recognizers and source to source translations from grammatical descriptions using the standard Extended Backus-Naur Form (EBNF) format to write the parsing rules. ANTLR supports parser generation into several languages such as *Java, C, C++, Ruby, Python, etc.* It is most commonly used to build translators and interpreters for Domain Specific Languages (DSL) and in particular it is used in the prototype CAO compiler developed at the University of Bristol.

4.2.1 Parser Generation

ANTLR generates parsers in several languages, however in this case only the Java generated parsers will be addressed. From a grammar file `Grammar.g`, ANTLR generates a java parser and lexer, `GrammarParser.java` and `GramarLexer.java`. ANTLR also supports embedding custom code inside the grammar rules in the target language. These actions will be executed when the parser is matching input thus providing means to build more powerful parsers or even translators and interpreters.

4.2.2 Abstract Syntax Trees

An abstract syntax tree, or syntax tree, captures the essential structure of a language source code. They are more abstract than parse trees, that represent the full syntax of the language according to a grammar specification by leaving out unnecessary syntactic details such as: commas, used to separate variable declarations or semi-colons to terminate a declaration. As opposed to parse trees, ASTs represent the structure of the language, not the grammar [22]. Typically referred as *intermediate representations* of programs [9] [7], they are often build by parsers while compiling source code.

Figure 4.1 shows a parse tree while Figure 4.2 shows one possible AST representation of code Listing 4.2. Is is noticeable the difference between them, mainly in the presence of syntactic tokens on the parse tree and the lack of them on the AST. These tree are typically traversed using a Visitor pattern.

Listing 4.2: Code Example for AST

```
def x : int;
```

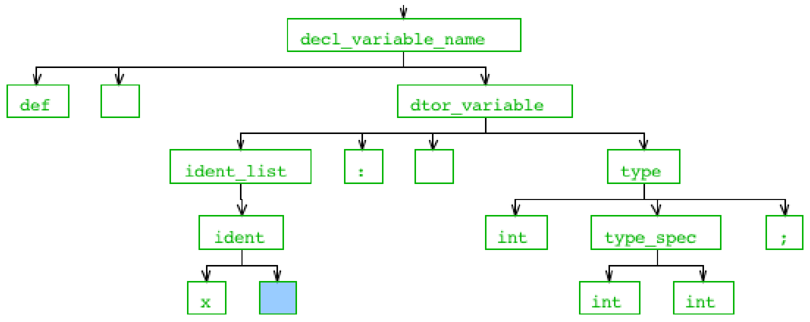


Figure 4.1: Parsing Tree of Listing 4.2

Visitor Pattern

The Visitor pattern allows to perform an operation on elements of a data structure without changing the classes of the elements where it will operate. It is typically used to add new operations on data structures that weren't thought of in the design process. The Visitor pattern is also recommended when there is the need to perform an operation on the data contained in a number of objects that have different interfaces [13].

Implementing a visitor consists of three steps:

- Adding an `accept` method to each class that will be visited;

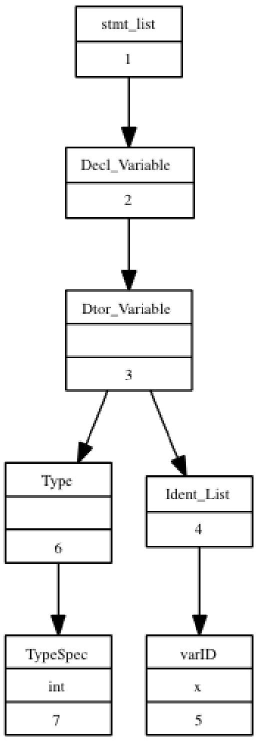


Figure 4.2: Example of one possible AST for Listing 4.2

- Creating a Visitor interface or abstract class;
- Implementing the Visitor interface;

The first step is the most simple, adding an `accept` method to each class that will be visited. Listing 4.3 shows the typical structure of an `accept` method. This method has one argument: the instance of a Visitor. When invoked, it calls the `visit` method and passes the class object as the argument.

Listing 4.3: Visitor Accept Method

```
public void accept(Visitor v){
        v.visit(this);
    }
```

The Visitor interface will contain all of the `visit` methods for each class element that will be visited (Listing 4.4).

Listing 4.4: Visitor Interface

```
public interface Visitor {
    public void visit(Dtor_Variable dtor);
    public void visit(Type t);
    public void visit(Ident_List idList);
    public void visit(TypeSpec tSpec);
}
```

And finally, implementing the `Visitor` interface will implement the operations that are needed to do at each element. Listing 4.4 shows an example implementation that prints to standard output information of each class.

Listing 4.5: Visitor Interface Implementation

```
public class PrintVisitor implements Visitor {
    public void visit(Dtor_Variable dtor){
        System.out.println(dtor.toString());
    }

    public void visit(Type t){
        System.out.println(t.getType.toString());
    }

    public void visit(Ident_List idList){ ... }
    public void visit(TypeSpec tSpec){ ... };
}
```

Listing 4.6 shows how to use the visitor supposing that the `Dtor_Variable`, `Type` ,`Ident_List` and `TypeSpec` all implement the same interface `ASTMembers`. The behavior is as it follows:

- Perform a loop on a list of `ASTMembers`;
- The Visitor calls each `ASTMember` `accept` method.
- That instance of the `ASTMember` class calls the Visitor's `visit` method;
- The Visitor prints to the standard output the information of the `ASTMember`;

Listing 4.6: Using the Visitor Patter

```

    Visitor v = new PrintVisitor();
    for(int i=0; i< list.length; i++){
        list[i].accept(v); // list of ASTMembers
    }
```

4.2.3 ANTLR Grammars

An ANTLR grammar is a common text that combines both the parser and lexer rules. These rules specify the grammatical and lexical structure (tokens) of a custom text. For example, a small language subset that only supports variable declarations such as Listing 4.7 would be defined as Listing 4.8.

Listing 4.7: Language Example

```
def x,y,z : int;
def h : bool;
```

Listing 4.8: ANTLR Grammar Example

```
1      prog : decl_variable_name*;
2      decl_variable_name  : 'def' ident_list ':' type ';';
3      ident_list          : ident (',' ident)*;
4      type     : 'int',
5               | 'bool'
6               ;
7      ident               : IDENT;
8      // TOKENS
9      IDENT                 : ( '0'..'9' | 'a'..'f' | 'A'..'F' );
```

A grammar rule is a named list with one or more alternatives such as `prog`, `decl_variable`, `ident_list`, `type`, and `ident`. The `prog` rule is the starting matching point for an ANTLR grammar. The second line reads as: start with the token `def`, followed by the rule (`ident_list`), followed by the token **':'**, followed by the rule `type` and finally the token `';'`. The list of identifiers (line 3) is defined as a regular expression and reads as: follow rule `ident` one time, read the token `','` and expect another `ident` zero or more times. A type (line 4) can be either a token `int` or `bool`. Finally the last line represents an identifier defined as a regular expression.

Grammar Actions

From a standard grammar, ANTLR generates a parser and lexer or a recognizer that can only validate the input. To build an interpreter or a translator, grammars can be extended with actions coded in the target language. Actions allow to execute computations on input symbols or emit new output symbols. Using actions it is possible to execute code besides recognizing the language. Listing 4.9 exemplifies a grammar with embedded actions.

Listing 4.9: Grammar with actions

```
decl: type ID ';'
    {System.out.println(+$ID.text+":"+$type.text+";");}
    | t=ID id=ID ';'
    {System.out.println(+$id.text+":"+$t.text+";");}
    ;
```

Variable Scoping

Using actions ANTLR also allows to perform variable scoping when parsing programming languages. Since it is a complex mechanism, it won't be detailed in this document, and we refer to [23] for more information.

Token Attributes

Tokens matched by the parser and lexer have a set of read-only attributes that can be accessed through *$label.attribute* or *$rule.attribute*. For example, these properties provide useful token information regarding the text that was matched such as the line number, column number, etc. Table 4.2.3 represents the most relevant attributes. ANTLR also allows to define new attributes by extending the `CommonToken` class.

Table 4.1: Token Predefined Attributes
adapted from [23].

Attribute	**Type**	**Description**
text	String	The text matched for the token; translates to a call to getText().
type	int	The token type (nonzero positive integer) of the token such as INT; translates to a call to getType().
line	int	The line number on which the token occurs, counting from 1; translates to a call to getLine().
pos	int	The character position within the line at which the token's first character occurs counting from zero; translates to a call to getCharPositionInLine().
index	int	The overall index of this token in the token stream, counting from zero; translates to a call to getTokenIndex().
tree	Object	When building trees, this attribute points to the tree node created for the token;

Rule Attributes

Rules also have similar types of attributes. Whenever the recognizer matches a rule it set up several attributes such as the text matched for the whole rule, the first matched symbol, the last symbol recognized by the rule, etc. More attributes can be defined by using parameters (Listing 4.10 line 1) and return values on grammar rules (Listing 4.11 line 1). Table 4.2.3 shows the most relevant rule attributes.

Listing 4.10: ANTLR Grammar Example - Rule with parameters

```
1 decl: type declarator[\$type.text] ';' ;
2 declarator[String typeText]
3     : '*' ID {System.out.println($ID.text+":^"+$typeText+";");}
4     | ID {System.out.println($ID.text+":"+$typeText+";");}
5     ;
```

Listing 4.11: ANTLR Grammar Example - Rule with return values

```
1 decl returns [String type, List vars]
2     : t=type ids+=ID (',' ids+=ID)* {\$type = \$t.text; \$vars = \$ids;}
3     ;
```

Abstract Syntax Trees and Rewrite Rules

ANTLR can automatically generate AST's with a list structure or rewrite rules can be added to the grammar to have more customization power. Rewrite rules allow to specify the structure of the tree that one wants to build, based on the input tokens. It provides support for node reordering, node omission and tree manipulation for root and child elements.

Listing 4.12: ANTLR Grammar Example - Rewrite Rules

```
1
2 dtor_function : ident_function '(' dtor_list? ')' ':' R2=type_list
3          -> ^( TOKN_DTOR_FUNCTION ident_function dtor_list? type_list)
4 ;
```

Listing 4.12 shows all of the mentioned properties. After matching the input with the `dtor_function` rule a new node of the AST will be built (line 3). This new tree will have the virtual[1] token `TOKN_DTOR_FUNCTION` as root with three child nodes (resulting trees from the other rules). The literal tokens were omitted as they are useless on an AST.

[1]A Token only used in the AST that doesn't belong to the input tokens of the lexer.

Table 4.2: Rule Predefined Attributes
adapted from [23].

Attribute	Type	Description
text	String	The text matched from the start of the rule up until the point of the $text expression evaluation;
start	Token	The first token to be potentially matched by the rule;
stop	Token	The last non-hidden channel token to be matched by the rule;
tree	Object	The AST computed for this rule, normally the result of a **-¿** rewrite rule;
text	String	The text matched thus far from the start of the token at the outer most rule nesting level;
type	int	The token type of the surrounding rule;
line	int	The line number, counting from 1, of this rule's first character;
pos	int	The character position in the line, counting from 0 of this rule's first character;

4.3 Parsing the CAO Language

Having in mind the goal of integrating the parser with an Eclipse plugin there was an obvious need to build an AST for the CAO language. Although the AST is the foundation for advanced Eclipse features, having a good syntax checker is probably the most important one. The implemented parsed provides AST generation, syntax checking and a variable scope checker.

The first step was to implement a recognizer, followed by extending the grammar with actions to perform scope checking. From there the grammar was extended with rewrite rules to build an AST for the language.

4.3.1 Parser Implementation

The prototype CAO compiler developed by the University of Bristol uses ANTLR for the internal parser generation. The initial idea was to try and re-use as many infrastructures from the compiler as possible, being the most important the AST. Since the compiler is written in Python programming language, none of the infrastructures could be re-used, only the grammar.

Although the provided grammar contained rewrite rules and actions to build the AST (Listing 4.13 shows the grammar actions and rewrite rules for building the AST of a type declaration), reusing these rewrite rules was not possible due to the fact that they are written in an unconventional way: there are embedded actions inside the rewrite rules, written in Python, (Listing 4.13 line 7) that alter the default behavior of ANTLR tree building process.

Listing 4.13: Original CAO Grammar Excerpt

```
1  decl_typename : T0=TOKN_DEF R0=dtor_typename
2      {
3       for symbol in R0.dtor.unwind_symbs( symb.SCOPE_LOCAL ) :
4                     self.get_scope().def_symb( symbol )
5       }
6       -> ^(
7           {self.ASTify(decl_factory.TYPENAME(),TOKN_DECL_TYPENAME,T0)} \$R0
8          )
9       ;
```

That being said, only the grammar was used to generate a simple parser/recognizer for the language. This grammar was then provided with rewrite rules to create the AST and finally extended with actions to provide variable scoping while parsing. The variable scoping was introduced as an option to provide further functionality when integrating with Eclipse.

4.3.2 Building the AST

The CAO AST was built by extending the grammar with rewrite rules (listing 4.14). Through rewrite rules one may choose the actual structure of the tree, but not how it is implemented in Java. ANTLR allows two different approaches: generating an homogenous tree or an heterogenous one. The first one is fully automatic and there is no need to implement the classes for the AST: it generates a tree using only one class (`CommonTree`) from ANTLR standard libraries. The second approach is the exact opposite of the first one, each node of the AST has a custom type and thus a different Java class that needs to be manually implemented.

Listing 4.14: ANTLR Grammar Example - Rewrite Rules

```

decl_function :
    TOKN_DEF R0=dtor_function R1=stmt_block_basic
```

```
      -> ^(TOKN_DECL_FUNCTION<Decl_Function> $R0 $R1)
    ;

dtor_function  :
 R3=function_qual? R0=ident_function '('R1=dtor_list?')' ':' R2=type_list
 -> ^( TOKN_DTOR_FUNCTION<Dtor_Function> $R3? $R0 $R1 $R2)
 ;

stmt_block_basic :
    T3='{' R3=stmt_list T4='}'
      -> ^(TOKN_SEQ<STMT_BLOCK_SEQ>[STMT_BLOCK_SEQ.OPER_LIST] \$R3)
    ;
```

An homogenous tree isn't useful for Eclipse integration. To provide the any advanced language feature there is a need to have a rich tree with information of each node type.[2]

AST implementation

Extending the CAO grammar with rewrite rules to build the AST proved to be one of the most time consuming tasks of the whole work. The rewrite rules system that ANTLR provides are very simple and practical to use when using them the *way* that the ANTLR developer thinks it is the most appropriate form to do so. In ANTLR's philosophy the AST tree generated should be parsed again by a tree grammar that ultimately feeds any other computations. This way the tree grammar recognizes all of the nodes accordingly to some specific logic and to do so, the Java generated AST is built using only one class (`CommonTree`), resulting in an homogenous tree. A `CommonTree` is a tree structure that in each node contains a `Token` object, or in other words, the node contains all the information described in table 4.2.3.

To implement heterogeneous AST's ANTLR introduces new syntax to the grammars that allows define how the each node will be built. The CAO AST was implemented as an heterogenous tree by extending the `CommonTree` class that ANTLR provides meaning that all of tree infrastructure is inherited. Figure 4.3 shows a subset of the tree class diagram. Note that for each node in the tree there is an implemented class and a Visitor implementation, all written by hand.

The chosen tree structure captures the majority of the language syntax removing non-relevant literals such as commas and parenthesis. Since the whole tree structure is very large, only a part of it will be presented by means of a small example. Listing 4.15 shows the maximum function coded in CAO and figure 4.4 the respective generated AST.

[2]This will become clearer in chapter 6.

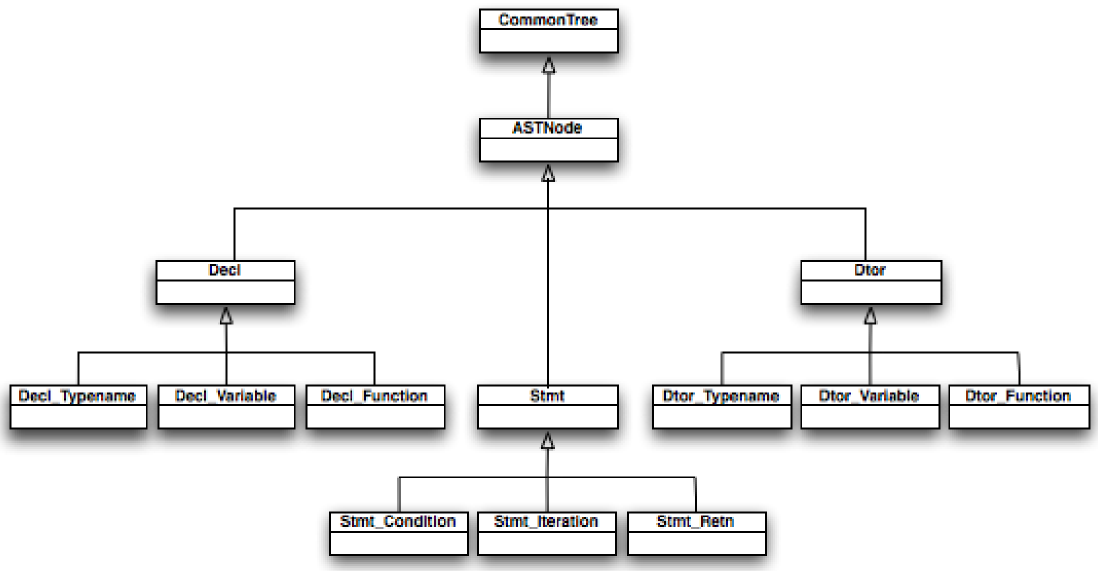


Figure 4.3: AST Class Diagram (stripped-down version).

Listing 4.15: Maximum number function coded in CAO

```
def max(x:int, y:int): int{
    if(x>y)
        return x;
    else
        return y;
}
```

Each node on the tree is directly related to the grammar rule that was used to parse the input. Having a look at figure 4.4, grammar listing 4.14 and code listing 4.15 one may notice that when parsing a function the `decl_function` rule is chosen which reflects on the AST structure with a node that belongs to the `Decl_Function` class. Following the grammar, a function is composed of two parts, the header `dtor_function` and the body `stmt_block_basic`, which in their turn will generate sub-trees that are appended to `Decl_Function` node.

4.4 Implementation issues

Implementing an AST representation in any programming language is not a simple task and it becomes harder if the language is still in development stage and constantly changing. The AST construction complexity depends essentially on the size of the parsed language, the more complex the language, the more classes or modules (depending on the language

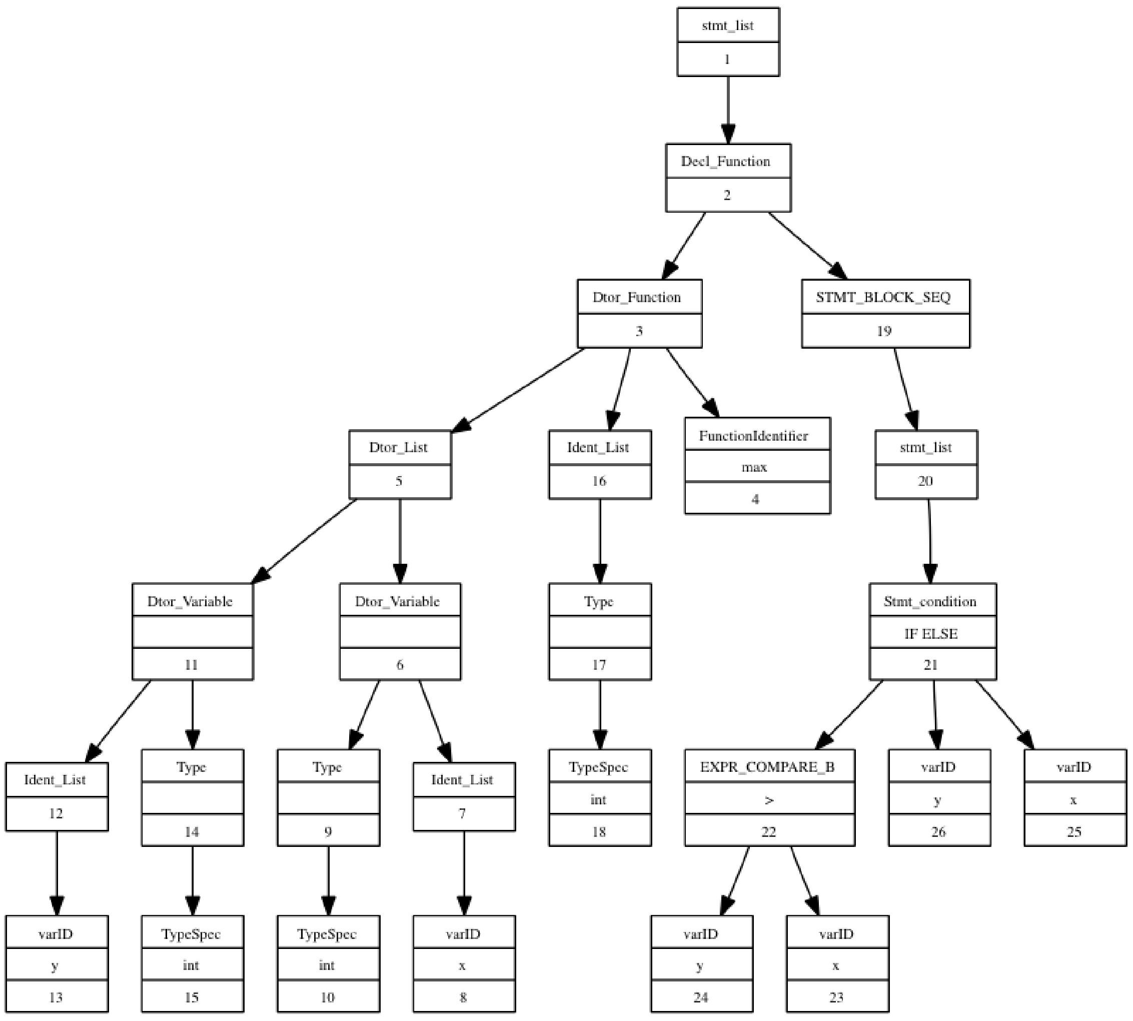


Figure 4.4: AST representing the maximum function from code listing 4.15.

in which the AST is being implemented) will be needed. Also, when introducing any new features to the language, the AST structure will also change and thus its implementation. The related `Visitor` classes would also need changes. This workflow, changing the grammar, generating the parser with AST, implementing AST classes and changing visitors is very error prone.

Choosing the right structure for an AST a somewhat difficult process. Literature typically refers to AST's as *"intermediate structures between the parsing tree and the compiler"* that represent the whole language. In this specific case, having the AST as an auxiliary tool for an Eclipse plugin, perhaps there isn't the need to have such a rich AST, similar to the one the compiler will have. Nevertheless it is important to notice that the more complex features the Eclipse plugin will have the richer the AST will need to be.

Chapter 5

CAO Editor

In this chapter the developed plugin will be presented.

5.1 CAO Editor Plugin

The CACE project will provide several tools and languages from each workpackage. Since all of these tools are still in early development stages, there still isn't a good notion on how they will function. Their functionality will tell how all them will interact with each other and how they depend between them. It is important to know how they will work to be able to integrate them in Eclipse. Knowing that, the result can be a single CACE eclipse distribution, a plugin providing all of the CACE tools into the Eclipse IDE that can work together, or a set of independent plugins compatible with non-CACE plugins. The last two options are not exclusive, plugins can be configured to work together or separate. For example, in the JDT one can edit XML files without the need of being in a XML project.

The developed plugin provides support for the CAO language and provides the most common features in any editor: syntax highlight, integration with the CAO parser for syntax error checking, static code completion, content outline, wizards for new project and files. There is also an online update site that allows to install and update the plugin automatically. Independently on how the CACE tools will integrate with each other, similar features for the other tools will also need to developed making the CAO Plugin a proof of concept for other tools.

Due to the early stage of the CACE project, the development of the language and tools is still suffering from many crucial changes that do not allow the evolution of the plugin. This plugin can evolve by introducing richer features such as a debug model, compilation support, compilation problem solving suggestions. However, this should be postponed to a stage where the language and compiler development is more stable.

5.1.1 Wizards

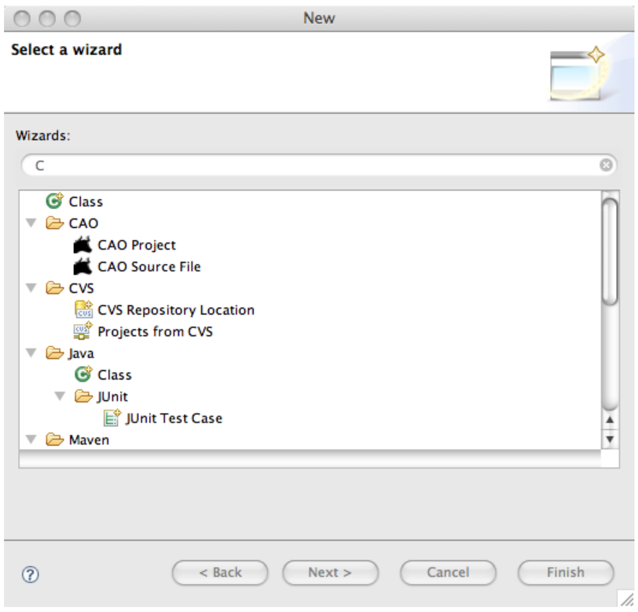
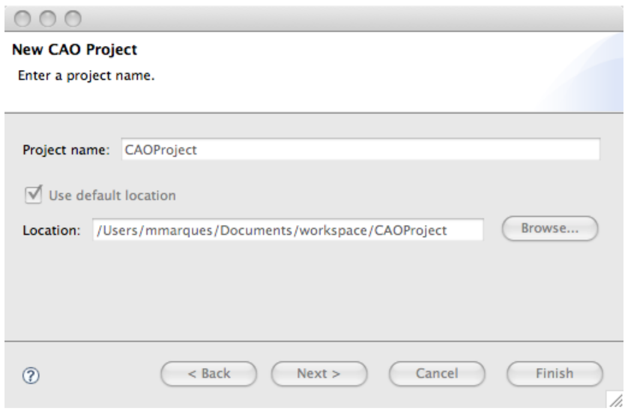


Figure 5.1: Wizard Selection

The plugin provides wizards, one for new projects and another for new source code files (Figure 5.1). The new project wizard creates a new project in Eclipse's workspace for CAO. Figure 5.2 shows the actual wizard menu. Currently, there are only two available options: project name and the location of the project in the workspace; the same happens for the new source code file wizard (Figure 5.3). Both of them can be further extended to support new parameters (once the CAO language development becomes more stable) such as imports, templates, compiler configuration, etc.


New CAO Project

Enter a project name.

Project name: CAOProject

☑ Use default location

Location: /Users/mmarques/Documents/workspace/CAOProject

Figure 5.2: New CAO Project Wizard

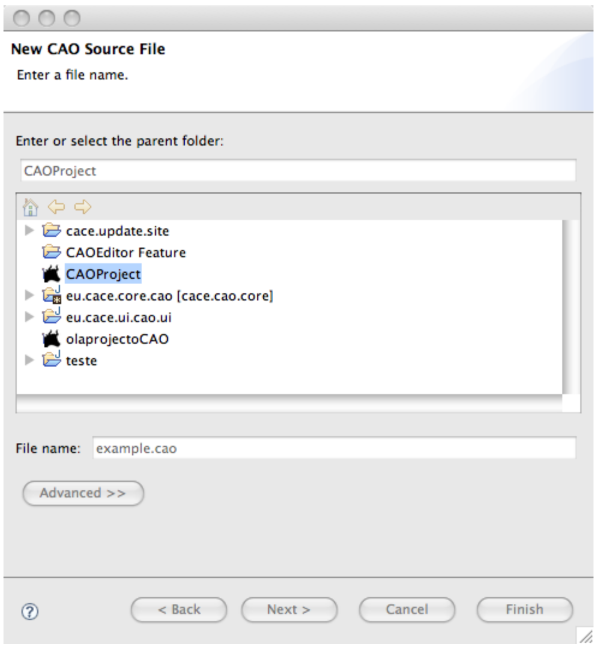


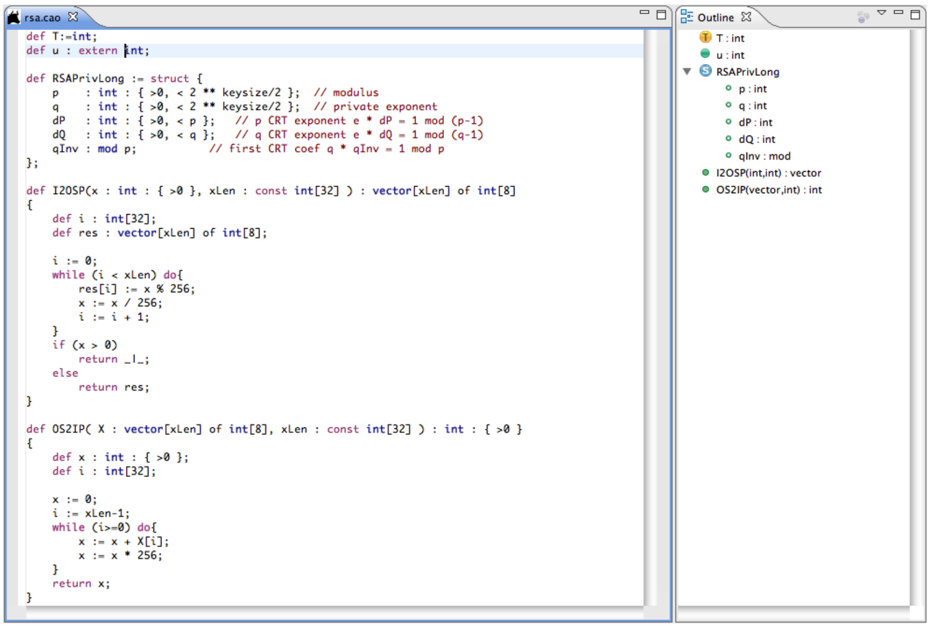
Figure 5.3: New CAO Source File Wizard

5.1.2 Editor

The most noticeable feature of the editor is the syntax highlight capability. It displays the source code in different colors, according to their category. The content outline viewer shows a tree-like structure based on the contents of the editor, with clickable behavior. That is, clicking on an element from content outline will automatically select the corresponding element in the text editor. Figure 5.4 shows the CAO editor and the outline view.

Content Assist

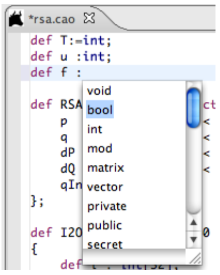
The available content assist provides code completion based on the input of the text editor. When requesting for code completion the editor proposes completions based on the first characters of the input. For instance, typing "i" and requesting for code completion will result in two proposals: "int" and "if". Although it is based on static keywords, the code completion (Figure 5.5), proves to be very useful for fast programming. Again, in the future, this feature can take benefits from the AST and provide dynamic content assist, or even the definition of templates for cycles, function declarations, etc.



```
def T:=int;
def u : extern int;

def RSAPrivLong := struct {
    p    : int : { >0, < 2 ** keysize/2 };  // modulus
    q    : int : { >0, < 2 ** keysize/2 };  // private exponent
    dP   : int : { >0, < p };   // p CRT exponent e * dP = 1 mod (p-1)
    dQ   : int : { >0, < q };   // q CRT exponent e * dQ = 1 mod (q-1)
    qInv : mod p;            // first CRT coef q * qInv = 1 mod p
};

def I2OSP(x : int : { >0 }, xLen : const int[32] ) : vector[xLen] of int[8]
{
    def i : int[32];
    def res : vector[xLen] of int[8];

    i := 0;
    while (i < xLen) do{
        res[i] := x % 256;
        x := x / 256;
        i := i + 1;
    }
    if (x > 0)
        return _|_;
    else
        return res;
}

def OS2IP( X : vector[xLen] of int[8], xLen : const int[32] ) : int : { >0 }
{
    def x : int : { >0 };
    def i : int[32];

    x := 0;
    i := xLen-1;
    while (i>=0) do{
        x := x + X[i];
        x := x * 256;
    }
    return x;
}
```

```
T : int
u : int
RSAPrivLong
    p : int
    q : int
    dP : int
    dQ : int
    qInv : mod
I2OSP(int,int) : vector
OS2IP(vector,int) : int
```

Figure 5.4: Text Editor and Outline View

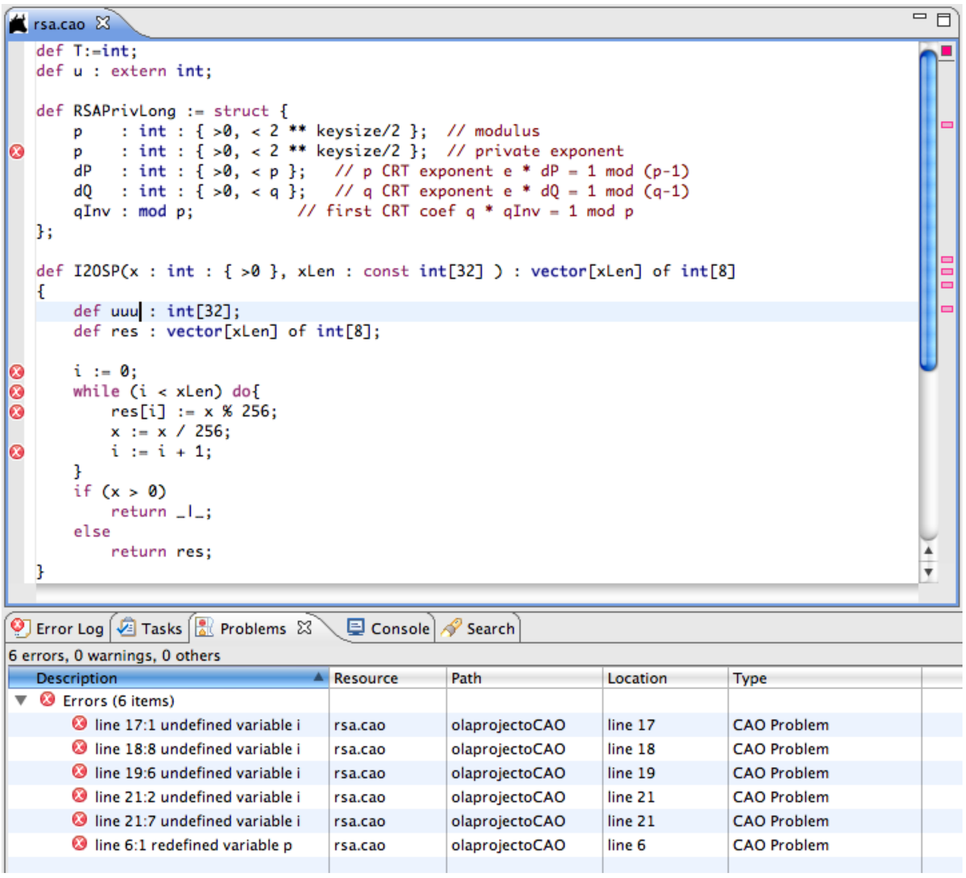


```
def T:=int;
def u :int;
def f :
```

Figure 5.5: Code completion

Problems

While programming there are common errors that appear often, such as forting to close a parenthesis, missing a semi-colon or using an non declared variable. The provided plugin detects common errors like those on-the-fly and reports them in the problems view. Simultaneously the line where those errors appear is marked in the text editor. Figure 5.6 shows the text editor window and the problems view to exemplify these features.



```
def T:=int;
def u : extern int;

def RSAPrivLong := struct {
    p    : int : { >0, < 2 ** keysize/2 };  // modulus
    p    : int : { >0, < 2 ** keysize/2 };  // private exponent
    dP   : int : { >0, < p };   // p CRT exponent e * dP = 1 mod (p-1)
    dQ   : int : { >0, < q };   // q CRT exponent e * dQ = 1 mod (q-1)
    qInv : mod p;           // first CRT coef q * qInv = 1 mod p
};

def I2OSP(x : int : { >0 }, xLen : const int[32] ) : vector[xLen] of int[8]
{
    def uuu : int[32];
    def res : vector[xLen] of int[8];

    i := 0;
    while (i < xLen) do{
        res[i] := x % 256;
        x := x / 256;
        i := i + 1;
    }
    if (x > 0)
        return _|_;
    else
        return res;
}
```

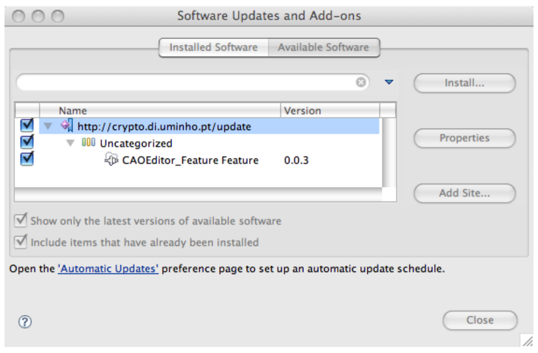
6 errors, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (6 items)				
line 17:1 undefined variable i	rsa.cao	olaprojectoCAO	line 17	CAO Problem
line 18:8 undefined variable i	rsa.cao	olaprojectoCAO	line 18	CAO Problem
line 19:6 undefined variable i	rsa.cao	olaprojectoCAO	line 19	CAO Problem
line 21:2 undefined variable i	rsa.cao	olaprojectoCAO	line 21	CAO Problem
line 21:7 undefined variable i	rsa.cao	olaprojectoCAO	line 21	CAO Problem
line 6:1 redefined variable p	rsa.cao	olaprojectoCAO	line 6	CAO Problem

Figure 5.6: Problems

5.1.3 Update Site

There is also an update site where the plugin can be automatically installed (Figure 5.8) and updated (Figure 5.7).



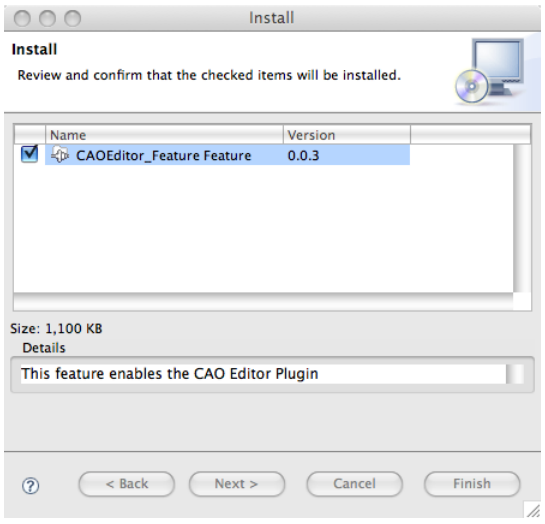
	Name	Version
✓	http://crypto.di.uminho.pt/update	
✓	Uncategorized	
✓	CAOEditor_Feature Feature	0.0.3

✓ Show only the latest versions of available software

✓ Include items that have already been installed

Open the 'Automatic Updates' preference page to set up an automatic update schedule.

Figure 5.7: Update Site for the CAO Editor



Install

Review and confirm that the checked items will be installed.

	Name	Version
✓	CAOEditor_Feature Feature	0.0.3

Size: 1,100 KB

Details

This feature enables the CAO Editor Plugin

Figure 5.8: Installing the CAO Editor

Chapter 6

Plugin Implementation

The CAO Editor is separated into two different plugins, the core plugin and the user interface plugin (UI).

6.1 Core Plugin

This plugin contains the core components of any plugin that are fully independent from other plugins. It contains the CAO builder, nature, code assistant and parser/AST support.

6.1.1 Builder

The builder was implemented by extending the `org.eclipse.core.resources.builders`. To implement builders Eclipse, provides the following classes and interfaces:

- `IncrementalProjectBuilder` - This class must be extended by any custom builder. It provides the framework for implementing builders;
- `IResourceDelta` - A resource delta represents changes in the state of a resource tree between two points in time. It can be processed using the Visitor pattern;
- `IResourceDeltaVisitor` - Any resource delta processor must to implement this interface. The visitor pattern will be used to process each entry in the delta;
- `IResourceVisitor` - Same as above for all resources;

Builders are directly invoked by Eclipse or by the user. When requesting a build Eclipse provides a resource delta (an object containing all the changes) for the project. There are four types of builds defined in the `IncrementalProjectBuilder` class:

- `FULL_BUILD` - is processed by request and forces a new build of all resources;

- `AUTO_BUILD` - is automatically triggered by Eclipse based on the preferences;
- `INCREMENTAL_BUILD` - is triggered by request when auto-build is off;
- `CLEAN_BUILD` - is triggered by request, clears all the current build state information and is followed by a full build;

Listing 6.1 shows build methods from the CAO builder. The main build method is executed by Eclipse when a build is triggered. Accordingly with the type of build it calls `fullBuild` or `incrementalBuild`.

Listing 6.1: Builders

```
protected IProject[] build(int kind, Map args, IProgressMonitor
    monitor)
     throws CoreException {
         if (kind == FULL_BUILD) {
             fullBuild(monitor);
         } else {
             IResourceDelta delta = getDelta(getProject());
             if (delta == null) {
                 fullBuild(monitor);
             } else {
                 incrementalBuild(delta, monitor);
             }
         }
         return null;
     }

protected void fullBuild(final IProgressMonitor monitor)
throws CoreException {
    try {
        getProject().accept(new CAOResourceVisitor());
    } catch (CoreException e) {
    }
}

protected void incrementalBuild(IResourceDelta delta,
            IProgressMonitor monitor) throws CoreException {
        // the visitor does the work.
        delta.accept(new CAODeltaVisitor());
}
```

Full Build

A full build as the name says results in building all of the resources in the workspace. To do so **IResourceVisitor** interface was implemented as listing 6.2 shows.

Listing 6.2: CAO Resource Visitor

```
class CAOResourceVisitor implements IResourceVisitor {
        public boolean visit(IResource resource) {
            checkCAO(resource); // building logic function
            //return true to continue visiting children.
            return true;
        }
    }
```

Incremental Build

As an opposite to a full build incremental builds have notion the notion of resource deltas. Three types of deltas are defined in the `IResourceDelta`:

- `ADDED` - this resource is new;
- `REMOVED` - this resource was removed;
- `CHANGED` - this resource was changed;

To walk through the resource delta **IResourceVisitor** interface was implemented as listing 6.3 shows. When a resource is added or changed, it should be built; when it is removed there is no need to build.

Listing 6.3: CAO Delta Resource Visitor

```
class CAODeltaVisitor implements IResourceDeltaVisitor {

        public boolean visit(IResourceDelta delta) throws
           CoreException {
            IResource resource = delta.getResource();
            switch (delta.getKind()) {
            case IResourceDelta.ADDED:
                // handle added resource
                checkCAO(resource); // call the builder
                break;
            case IResourceDelta.REMOVED:
                // handle removed resource
                break;
            case IResourceDelta.CHANGED:
                // handle changed resource
```

```
                    checkCAO(resource); // call the builder
                    break;
               }
               //return true to continue visiting children.
               return true;
          }
     }
```

Builder Behavior

Until now, only the building infrastructure was detailed but not the actual builder behavior. The implemented builder is responsible for generating the error listing in the problems window and the problem markers in the text editor window. The builder calls the parser with an error handler that reports all the errors to Eclipse through the `addMarker` method (listing 6.4).

Listing 6.4: CAO Error reporting

```
public void reportError(RecognitionException e, String[] tokenNames
    ) {
               String hdr = getErrorHeader(e); // retrieve error
                   header;
               String msg = getErrorMessage(e, tokenNames); //retrieve
                    error message;
               CAOBuilder.this.addMarker(file, hdr+"␣"+msg, e.line,2);
                    // add a new marker to the text editor window;
}
```

6.1.2 CAO Nature

The nature implementation is responsible for informing the Eclipse platform of the CAO Nature. And also, associated with it nature is the CAO builder. Whenever a new CAO project is created this nature is automatically loaded and thus, the builder. It was implemented by extending the `org.eclipse.core.resources.natures` extension point (Listing 6.5) and implementing the `IProjectNature`.

Listing 6.5: CAO Project Nature Extension Declaration

```
<extension
     id="caoNature"
     name="CAO␣Project␣Nature"
     point="org.eclipse.core.resources.natures" >
  <runtime>
   <run
```

```
      class="eu.cace.core.cao.nature.CAONature" > <!-- class that implements
           IProjectNature -->
    </run>
  </runtime>
  <builder
        id="eu.cace.core.cao.CAOBuilder" >
  </builder>
 </extension>
```

6.1.3 Code Assistant

The `CAOCompletionProcessor` is very simple. It only proposes keywords as completion candidates. The keywords are defined in a field, `allWords`. When requesting for assistance the `CAOCompletionProcessor` implementation checks if the first characters of the input word match any of the keywords and proposes them as a suggestion.

6.2 UI Plugin

The UI plugin only contains components that extend the Eclipse UI, thus the selected package namespace always starts with `eu.cace.ui`. All the wizard classes are within the `eu.cace.ui.wizards` namespace and the all the editor's related classes and packages start with the `eu.cace.ui.editors.cao`.

6.2.1 Wizards

As it was previously mentioned Eclipse wizards typically guide the user through a set of steps. Wizards in Eclipse consist of one wizard class and several wizard page classes. A wizard page class represents a step in the wizard or in other terms each menu that appears between clicking on the next button. The wizard class manages the page set and provides user interface controls that enable page navigation and task invocations.

Wizards are implemented by extending the Eclipse's `Wizard` and implementing the suited interface for the wizard. There are three specific interfaces for wizards:

- `INewWizard` - support for wizards that create new resources (projects, files, etc.);
- `IImportWizard` - support for wizards that import resources to the workspace;
- `IExportWizards` - support for wizards that export resources to the filesystem;

Wizard page are implemented by extending the `WizardPage` class and overriding the `createControl` method that controls the page interface. Eclipse also provides reusable page classes for common wizards such as a new project (`WizardNewProjectCreation`) or new file (`WizardNewFileCreationPage`).

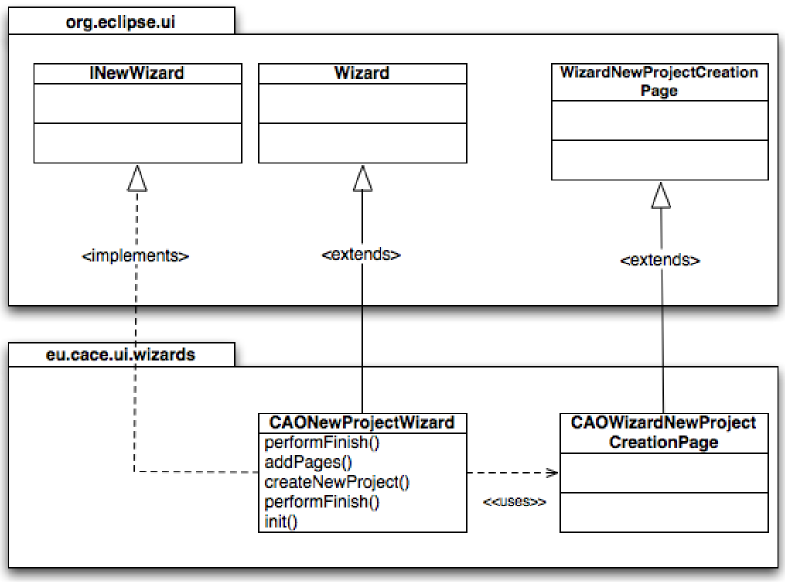


Figure 6.1: Class Diagram for the CAO project wizard

The UI plugin provides two wizards: new CAO project and new CAO source file. Figure 6.1 shows the class diagram of the new project wizard. The `CAONewProjectWizard` class implements `INewWizard` interface and extends the `Wizard` class. This wizard uses the `CAONewProjectCreationPage` that extends the `WizardNewProjectCreationPage` class. The same principle applies to the new source file wizard whereas it implements the same `INewWizard` interface and extends a different page class, `WizardNewFileCreationPage`.

6.2.2 Text Editor

CAO text editor was implemented accordingly with the architecture mentioned in chapter 3.2.6, Figure 3.10. The overall class diagram of the editor can be seen in Figure 6.2.

The `CAOEditor` extends the `TextEditor` class from the text framework thus inheriting all the key features of a text editor. To provide syntax highlight and code completion the inherited `TextEditor SourceViewConfiguration` was overridden using the convenience method `setSourceViewerConfiguration` with the `CAOSourceViewerConfiguration`, listing 6.6 line 4. The same procedure was used to set the up the document provider, listing 6.6 line 6.

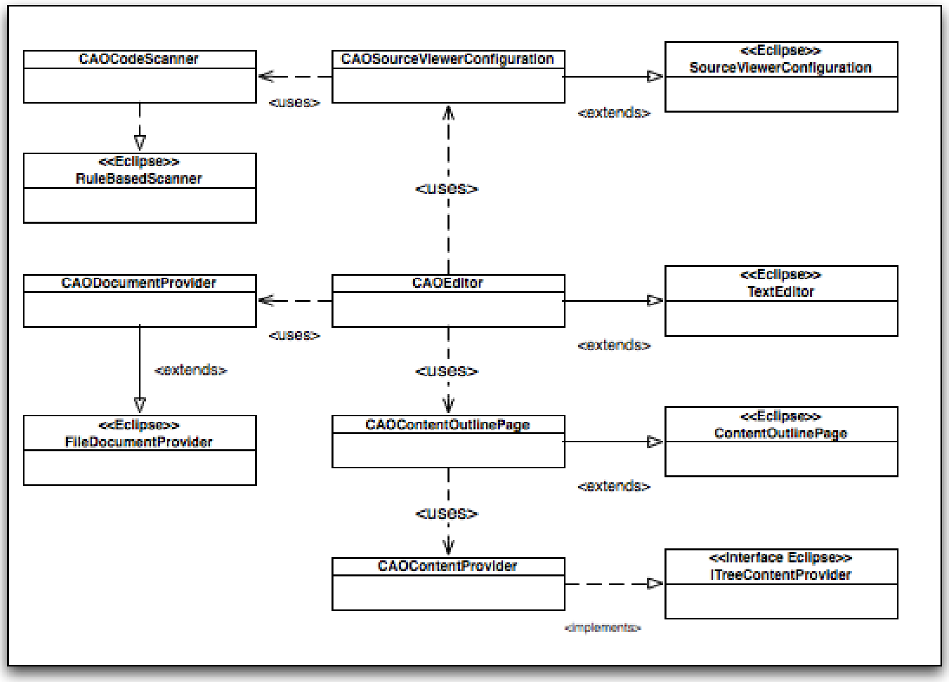

CAOCodeScanner
<uses>
CAOSourceViewerConfiguration
<<Eclipse>> SourceViewerConfiguration
<extends>
<<Eclipse>> RuleBasedScanner
<uses>
CAODocumentProvider
<uses>
CAOEditor
<<Eclipse>> TextEditor
<extends>
<extends>
<uses>
<<Eclipse>> FileDocumentProvider
CAOContentOutlinePage
<<Eclipse>> ContentOutlinePage
<extends>
<uses>
CAOContentProvider
<<Interface Eclipse>> ITreeContentProvider
<implements>

Figure 6.2: Class Diagram for the CAO Editor

Listing 6.6: CAO Example

```
public CAOEditor() {
  ...
  caoEditorSourceViewerConfiguration = new
     CAOEditorSourceViewerConfiguration(colorManager);
  setSourceViewerConfiguration(caoEditorSourceViewerConfiguration
     );

  caoDocumentProvider= new CAODocumentProvider();
  setDocumentProvider(caoDocumentProvider);


  ...
}
```

Document Provider

Since all of the editing within the CAO editor is based on files located in the local filesystem, the `CAODocumentProvider` class extends the Eclipse based `FileDocumentProvider` to defined a document partitioner for the CAO language.

Listing 6.7: CAODocumentProvider

```
public class CAODocumentProvider extends FileDocumentProvider {

  protected IDocument createDocument(Object element) throws
      CoreException {
    IDocument document = super.createDocument(element);
    if (document != null) {
      IDocumentPartitioner partitioner =
        new FastPartitioner(
          new CAOPartitionScanner(),
          new String[] {
            CAOPartitionScanner.CAO_COMMENT });
      partitioner.connect(document);
      document.setDocumentPartitioner(partitioner);
    }
    return document;
  }
}
```

Partitioning and Scanners

The CAO source code is partitioned into two different sections: source code and comments. Partitioning is implemented with scanners. The implementation focuses on rule based scanners that Eclipse provides, it is straightforward to implement so it wont be detailed here.

Two different scanners were implemented: `CAOPartitionScanner` for text partitioning, `CAOCodeScanner` for finding CAO keywords and giving them attributes such as colors.

Syntax Highlight

To provide syntax highlight Eclipse uses a computational model of damage, repair and reconciling. When the text in an editor is modified, parts of the editor must be redisplayed to show the changes. With syntax highlight a simple char can change the whole text (e.g. a comment).

Damagers determine the region of a document's presentation which must be rebuilt due to a change and return a damage region that serves as input for a repairer. A repairer processes all the damage region and generates descriptions of the repairs that are need to be made. Finally the reconciler controls the overall process. It monitors changes in the editor and notifies the damager to compute the damaged region and then passes it to the repairer to perform the needed repairs. Typically there is one repairer and damager for each partition type.

Eclipse provides a default damage repairer implementation (`DefaultDamagerRepairer`) that is configured with scanners. These scanners are the ones responsible for the highlighting since they scan the code into regions and then provides them with colored attributes. In listing 6.8 the reconciler is configured with two damagers/repairers, one for each content type.

Listing 6.8: Presentation Reconciler Configuration

```

public IPresentationReconciler getPresentationReconciler(
    ISourceViewer sourceViewer) {
        PresentationReconciler reconciler = new
            PresentationReconciler();

        // DamagerRepairer configured with the CAOCodeScanner for
            non-comment content (DEFAULT_CONTENT_TYPE)
        DefaultDamagerRepairer dr = new DefaultDamagerRepairer(
            getCAOCodeScanner());
        reconciler.setDamager(dr, IDocument.DEFAULT_CONTENT_TYPE);
        reconciler.setRepairer(dr, IDocument.DEFAULT_CONTENT_TYPE);

        // DamagerRepairer configured with a single token scanner
            for comment content (CAO_COMMENT)
        dr = new DefaultDamagerRepairer(
                new SingleTokenScanner(
                        new TextAttribute(
                            colorManager.getColor(CAOColorConstants
                                .CAO_COMMENT))));
        reconciler.setDamager(dr, CAOPartitionScanner.CAO_COMMENT);
        reconciler.setRepairer(dr, CAOPartitionScanner.CAO_COMMENT)
            ;

        return reconciler;
    }
```

6.2.3 Content Outline

Implementing a content outline consists of implementing three interfaces:

- `ITreeContentProvider`;
- `ILabelProvider`;
- `ContentOutlinePage`;

Content Provider

Eclipse provides an abstract implementation for content outlines. Internally content outlines are managed as tree structures and all the means to traverse them, such as a hierarchical visitor pattern, are already implemented. Content outlines are based on a hierarchical model that represents the information inside an editor, typically an AST. Since the AST implementation may vary from language to language there is the need to adapt models to the internal eclipse tree representation. This is done by implementing the `ITreeContentProvider` that consists in implementing common tree navigation methods (see 6.9)

Listing 6.9: ITreeContentProvider implementation

```
public class CaoContentProvider implements ITreeContentProvider{

 public Object[] getChildren(Object parentElement) {
         ASTNode a;
         if(parentElement instanceof ASTNode){
             a = (ASTNode) parentElement;
             if (a.getChildCount()>0)
                 return a.getChildren().toArray();
         }

         return new Object[0];
     }

     public Object getParent(Object element) {
         if(element instanceof ASTNode)
             return ((ASTNode) element).parent;

         return null;
     }

     public boolean hasChildren(Object element) {

         if(element instanceof ASTNode)
             return (((ASTNode) element).getChildCount() > 0);

         return false;
     }
}
```

Label Provider

Having the content provider Eclipse needs to know how to handle the tree contents to retrieve information. The label provider is responsible for the text labels and icons that will appear on the outline. Listing 6.10 a small part of the implementation.

Listing 6.10: ILabel implementation

```
public class CaoLabelProvider implements ILabelProvider{

    public String getText(Object element) {
            ...
            if(element instanceof StructElement){
                StructElement var = (StructElement) element;
                return var.getText();
            }
    }

    public Image getImage(Object element) {
            ImageDescriptor descriptor = null;
            Image image = null;
            if (element instanceof StructElement) {
                URL installUrl = Activator.getDefault().getBundle()
                   .getEntry("/icons/field_public_obj.gif");
                descriptor = ImageDescriptor.createFromURL(
                   installUrl);
            }
            ...
    }
}
```

Content Outline Page

The content outline class controls the content outline life-cycle. It is responsible for configuring the label provider, content provider and the clickable behavior.

Chapter 7

Conclusions and Further Work

In this chapter, the conclusions of this work are presented, as well the suggestions for further work.

7.1 Conclusions

Despite having powerful frameworks for IDE related tools provided by Eclipse, implementing an editor for the CAO language turned out to be a very difficult task. When integrating languages into Eclipse there are two main issues that need to be addressed: understanding how the frameworks that Eclipse provides work, and more importantly how they interact with each other; since in Eclipse Java is the *lingua franca*, there is always the need to provide a model for the language to be integrated implemented in Java (if there is interest on features more advanced than syntax highlighting).

On one hand, Eclipse is a state of the art technological monster and has a very steep learning curve. The available documentation is poor and example oriented. Sometimes in order to explain a feature or how a framework works, the documentation forwards to JDT source code, which is very complicated to understand. On the other hand, after overcoming the technological challenge, features that Eclipse provides for creating editors are easy to use, but the majority of them still relies on a model for that language. These models that Eclipse relies on are usually AST's implemented in Java.

The typical AST's structure contains all the information from the parsed input, except for some syntactic tokens. An AST developed for Eclipse doesn't need have the same structure as of compiler generated AST's; they can be trimmed down depending on what kind of features the plugin will have. Implementing the AST is a time consuming process as it involves creating a new parser and implementing the tree classes and visitors. If possible, an automatic mechanism of generation of the AST should be used.

That being said, Eclipse has proven that it provides all the required facilities to integrate new language editors for the CACE project.

Overall, a proof-of-concept editor was implemented with features such as: wizards, syntax highlighting, code completion, syntax check and content outline;

7.2 Further Work

The next big step would be to integrate the prototype compiler, not only has an external tool but also by trying to use its facilities to retrieve more information for the plugin. For instance, using the compiler to retrieve semantic errors that are not provided by the parser. To do so there would be the need for a communication platform between the compiler and the Eclipse plugin. One possibility would be trough the use of XML files or by the exploration of the Jython platform [4] that allows to run Python code on the Java Platform.

The implemented plugin is just the basis for what is possible to do with Eclipse thus, as future work, several new functionalities can be implemented:

- Debugger integration;
- Provide quick fixes for syntax errors;
- Use the Eclipse template engine for source code templates;
- Dynamic Content Assist (based on the current state of the whole source code using the AST);
- Code refactoring support;

The AST model can also be target of further improvements:

- Automatic generation of the tree;
- Automatic generation of visitors;
- Automatic generation of Test Units;

Bibliography

[1] *CACE Project Home Page* - http://www.cace-project.eu. [Online; accessed 1-May-2008].

[2] *Eclipse FAQ* - http://www.eclipse.org/home/newcomers.php. [Online; accessed 23-May-2008].

[3] *JDT Core Component* - http://www.eclipse.org/jdt/core/index.php. [Online; accessed 29-October-2008].

[4] *The Jython Project* - http://www.jython.org/Project/. [Online; accessed 1-February-2008].

[5] Eclipse platform technical overview. Technical report, Object Technology International, February 2003.

[6] Bioclipse: An open source workbench for chemo- and bioinformatics. *BMC Bioinformatics*, 8(1), 2007.

[7] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[8] C. Aniszczyk, B. Bauman, W. Melhem, and M. Pawlowski. Fundamentals of plugin and rcp development. Eclipse Con 2007, March 2007.

[9] A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, December 1997.

[10] J. Arthorne. Project builders and natures. Eclipse Corner Articles, November 2004.

[11] M. Barbosa, R. Noad, D. Page, and N. Smart. First steps toward a cryptography-aware language and compiler. Cryptology ePrint Archive, Report 2005/160, 2005. http://eprint.iacr.org/.

[12] E. Clayberg and D. Rubel. *Eclipse: Building Commercial-Quality Plug-Ins*. Pearson Higher Education, 2004.

[13] J. W. Cooper. *Java design patterns: a tutorial.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[14] M. K. Crowe. An incremental compiler. *SIGPLAN Not.*, 17(10):13–22, 1982.

[15] J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *Java(TM) Developer's Guide to Eclipse, The (2nd Edition).* Addison-Wesley Professional, 2004.

[16] T. Eicher. Text editor recipes. Eclipse Con 2006, 2006.

[17] M. Grand. *Patterns in Java, volume 1: a catalog of reusable design patterns illustrated with UML.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[18] O. Gruber, B. J. Hargrave, J. McAffer, P. Rapicault, and T. Watson. The eclipse 3.0 platform: adopting osgi technology. *IBM Syst. J.*, 44(2):289–299, 2005.

[19] D. Hou. Studying the evolution of the eclipse java editor. In *eclipse '07: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange*, pages 65–69, New York, NY, USA, 2007. ACM.

[20] W. Melhem and D. Glozic. Pde does plugins. Eclipse Corner Article, September 2003.

[21] D. Page. *CAO: A Cryptography Aware Language and Compiler.* University of Bristol, 2008.

[22] T. Parr. *What's the difference between a parse tree and an abstract syntax tree (AST)?* - http://www.jguru.com/faq/view.jsp?EID=814505.

[23] T. Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages.* Pragmatic Programmers. Pragmatic Bookshelf, first edition, May 2007.