EasyChair Preprint

№ 622

Eclipse CDT code analysis and unit testing

Shaun D'Souza

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

November 10, 2018

Eclipse CDT code analysis and unit testing

Shaun C. D'Souza[1]

Wipro Limited
shaun.dsouza1@wipro.com

Abstract

In this paper we look at the Eclipse IDE and its support for CDT (C/C++ Development Tools). Eclipse is an open source IDE and supports a variety of programming languages including plugin functionality. Eclipse supports the standard GNU environment for compiling, building and debugging applications. The CDT is a plugin which enables development of C/C++ applications in eclipse. It enables functionality including code browsing, syntax highlighting and code completion. We verify a 50X improvement in LOC automation for Fake class .cpp / .h and class test .cpp code generation.

Keywords— Software Maintenance, Software Testing, Object-oriented, Regression Testing

1 Introduction

Eclipse supports a number of programming languages including C/C++ [8, 2], Java, PHP, XML, and HTML. It is an open source IDE and can be used on multiple platforms including Windows, Linux. It supports plugins to extend the functionality of the IDE for source code language modeling and analysis.

In our paper we study the use of Eclipse to enable an automation framework for generation of unit tests and fake classes for code debug [6]. Eclipse supports the parsing and compilation of code into an index file. The index file is used to store code binding information including identifiers bindings, source file location, macros and include files.

2 Stages of compilation parser

Source code	
Lexical analysis	Token stream
Syntax analysis	Abstract Syntax Tree
Semantic analysis	

Table 1: Stages of compilation parser

Scanning converts the input character stream into a stream of tokens. Eg. 'I' 'n' 't' is converted to a token object of type int. Preprocessing involves macro expansion, conditional compilation and inclusion of header files. Parsing converts the C++ language semantics into an abstract syntax tree structure [3]. The AST is an intermediate program representation for the code and captures all the semantic information for the source - Table 2. Source code is optimized for human readability.

2.1 Lexical analysis

Natural language: "I shot an elephant in my pajamas"

I	shot	an	elephant	in	my	pajamas

Programming language: "if (a == 0) a = b + 1"

if	(	a	==	0	)	a	=	b	+	1

2.2 Syntax analysis

Natural Language

The	cat	sat	on	the	mat
det	noun	verb	prep	det	noun
subject		predicate	prep	object	

Programming language

if (a == 0)	a = b + 1
test	assignment
if-statement	

2.3 Semantic analysis

Natural Language

The	green	*apple*	ate	a	juicy	bug
det	adj	noun	noun	det	noun	noun

Programming language

if (a == 0)	a = *foo*
test	assignment

Semantic analysis will report an error.

3 CDT Core

We use the CDT to function as a compiler frontend and use the AST to generate unit tests [4]. The CDT uses a translation unit to represent a source file cpp and h. The CDT core supports a Visitor API which is used to traverse the AST [1]. AST rewrite API is used to update the source code. We access the code AST using the Eclipse CDT API.

C-Model: ITranslationUnit for a workspace file

```
IPath path= new Path("project/folder/file.cpp");
IFile file= ResourcesPlugin.getWorkspace().getRoot().getFile(path);
// Create translation unit for file
ITranslationUnit tu= (ITranslationUnit) CoreModel.getDefault().create(file);
```

C-Model: ITranslationUnit for file in the editor

```
IEditorPart e= PlatformUI.getWorkbench().getActiveWorkbenchWindow().getActivePage().
    getActiveEditor();
// Access translation unit of the editor.
ITranslationUnit tu= (ITranslationUnit) CDTUITools.getEditorInputCElement(editor.
    getEditorInput());
```

C-Index: IIndex for project

```
ICProject project= CoreModel.getDefault().getCModel().getCProject("project");
IIndex index= CCorePlugin.getIndexManager().getIndex(project);
```

Eclipse supports the use of IBinding [5]. Binding completely represents the C/C++ entity. It contains information about the type of a variable, return type and parameters of a function. A compiler is used to translate one program representation to another. Most commonly the information is a program. We use the Eclipse CDT in our investigation to process the source tree and generate a set of Fake class and unit test files [7].

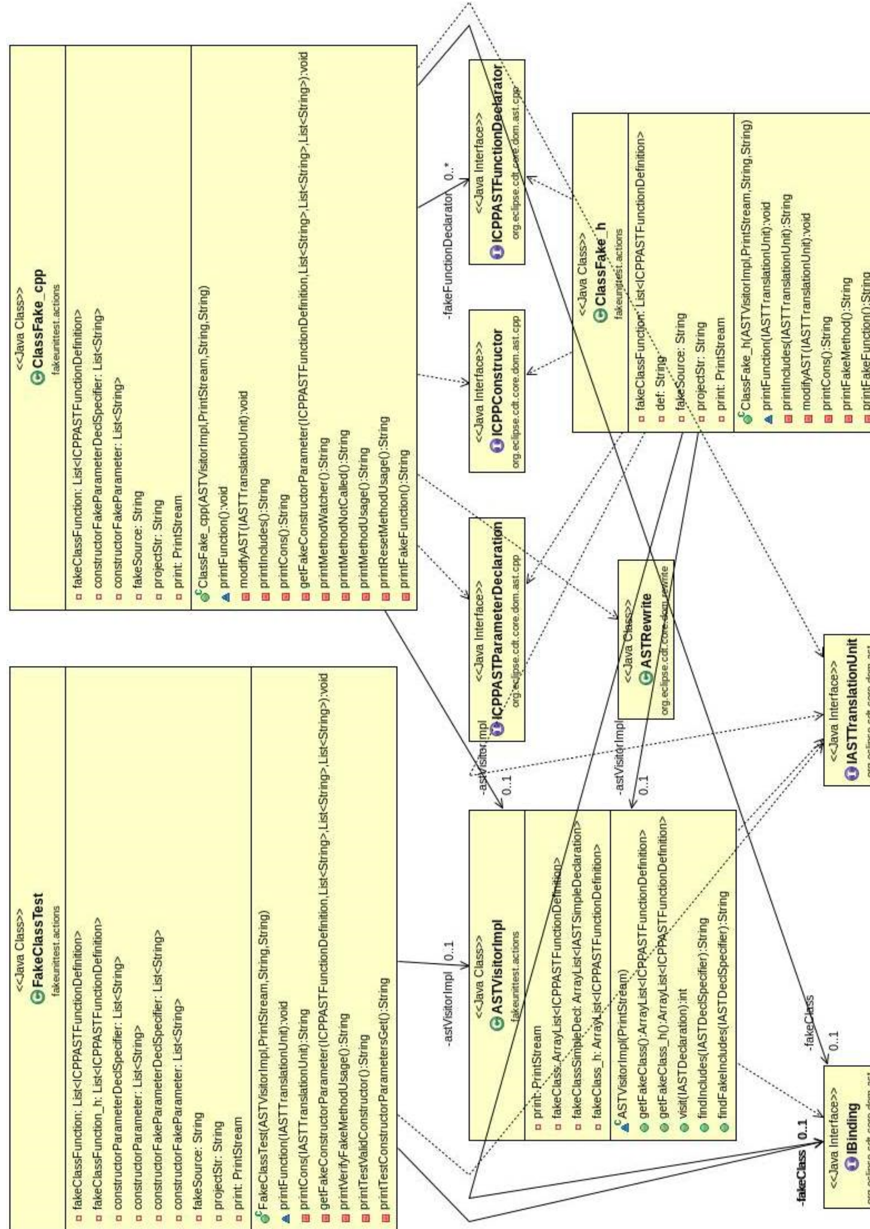
The CDT is not a compiler and is designed to support compiler frontend features. It is designed for performance and is able to parse code skipping included header files. The parsers do not perform any semantic analysis or type checking during the parse. The phases of parsing include scanning and preprocessing. During the scanning phase a stream of character inputs is converted into a stream of tokens. Preprocessing also involves macro expansion, conditional compilation and inclusion of header files. Parsing is used to convert the input token stream to an AST. The parser converts concrete syntax into an abstract syntax tree representation. The AST is used in semantic analysis of the code to implement type checking of the code definitions.

We implemented an ASTVisitorImpl class to traverse the code AST. This allows us to obtain all the declaration information for the functions in a class. We traverse all declarations in the code. Function and constructor information is used to construct the Fake Class .cpp and .h header files. This is then integrated into the unit testing framework. We store a list of function declarations and class information to create the Fake class files and unit tests.

4 Fake class plugin UML

```
discovery/storage/FakeStorageSCSI_DiscoveryAlgorithm.cpp

FakeStorageSCSI_DiscoveryAlgorithm::FakeStorageSCSI_DiscoveryAlgorithm()
: StorageSCSI_DiscoveryAlgorithm()
, fake_run( "FakeStorageSCSI_DiscoveryAlgorithm::run" )
, fake_associate( "FakeStorageSCSI_DiscoveryAlgorithm::associate" )
, fake_getDuplicatedHardDriveList( "FakeStorageSCSI_DiscoveryAlgorithm::
    getDuplicatedHardDriveList" )
, fake_addUniqueHardDrive( "FakeStorageSCSI_DiscoveryAlgorithm::addUniqueHardDrive" )
, fake_isDuplicateBackplane( "FakeStorageSCSI_DiscoveryAlgorithm::isDuplicateBackplane"
     )
}

void FakeStorageSCSI_DiscoveryAlgorithm::verifyFakeMethodUsage( const std::string&
    testCondition )
{
  TestUtility::verifyFakeMethodUsage( fake_run, testCondition );
  TestUtility::verifyFakeMethodUsage( fake_associate, testCondition );
  TestUtility::verifyFakeMethodUsage( fake_getDuplicatedHardDriveList, testCondition );
  TestUtility::verifyFakeMethodUsage( fake_addUniqueHardDrive, testCondition );
  TestUtility::verifyFakeMethodUsage( fake_isDuplicateBackplane, testCondition );
}

void FakeStorageSCSI_DiscoveryAlgorithm::run( UI_Facade& uiFacade )
{
  return fake_run( uiFacade );
}

discovery/storage/StorageSCSI_DiscoveryAlgorithmTest.cpp

  StorageSCSI_DiscoveryAlgorithm_data()
  : fakeDeviceReporter()
  , fakeDiscoveryRepository()
  , fakeIoConnectionOperations()
  , fakeTransportFactory()
  , fakeDiscoveryOperationsFactory()
```

Figure 1: Fake class plugin UML.



Parsing	Parsing
Lowering	Lowering
Interpreter	Analysis + Optimization
JIT compiler	Code gen

Table 2: V8 JS, g++ compiler

```
, fakeDiscoveredDeviceOperationsFactory()
, fakeFusionIO_AcceleratorFactory()
, fakePciOperationsFactoryPtr( new FakePCI_OperationsFactory() )
, fakeFileSystemOperations()
, fakeSmbiosOperationsPtr( new FakeSMBIOS_Operations() )
, fakeIloOperationsPtr( new iLO::Fake_iLO_Operations() )
, fakeTimeOperationsPtr( new FakeTimeOperations() )
, failureEventStatus( FakeEvt::failure )
, goodEventStatus()
{
}
```

Refactoring is changing restructuring existing code without changing its behaviour. We use the ASTRewrite class functionality to modify code dynamically by describing changes to the AST. Eclipse supports modification of specific code declarations in the source using the CDT.

```
IASTTranslationUnit tu = ...;
ASTRewrite r = ASTRewrite.create( tu );
IASTNode lit = r.createLiteralNode( String code );
r.replace( declaration, lit, null );
Change c = r.rewriteAST();
c.perform( new NullProgressMonitor() );
```

New AST nodes can be created using the getASTNodeFactory().

```
IASTBreakStatement breakStatement = tu.getASTNodeFactory().newBreakStatement();
```

However for our implementation in Fake class and unit test generation we use ASTRewrite createLiteralNode method. We however look at the use of getASTNodeFactory in implementing refactoring and extending the fake class and unit test functionality.

ASTRerwite uses the following functions to implement code refactoring.

```
void remove( IASTNode n, TextEditGroup eg )
ASTRewrite replace( IASTNode n, IASTNode repl, TextEditGroup eg )
ASTRewrite insertBefore( IASTNode p, IASTNode insPoint, IASTNode newN, TextEditGroup eg
    )
```

5 Conclusion

We enable an Eclipse CDT framework as a design for performance best practise. The developed unit test productivity accelerator, framework components facilitate source code integration. The plugin is developed for generation of unit test code and software engineering. Source files are input to the plugin in the project. We verify a 50X improvement in LOC automation for Fake class .cpp / .h and class test .cpp code. The open source plugin automates code analysis and unit test generation.

References

[1] Api for c/c++ ast. http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.cdt.doc.isv%2Fguide%2Fdom%2Findex.html.

[2] Gnu g++. http://gcc.gnu.org.

[3] Overview of parsing. http://wiki.eclipse.org/CDT/designs/Overview_of_Parsing.

[4] M. Dickheiser. *Game Programming Gems 6*, chapter 1. GAME DEVELOPMENT SERIES. Charles River Media, 2006.

[5] J. Gosling, B. Joy, G.L. Steele, G. Bracha, and A. Buckley. *The Java Language Specification, Java SE 8 Edition*. Java Series. Pearson Education, 2014.

[6] Paul Hamill. *Unit Test Frameworks*. O'Reilly, first edition, 2004.

[7] Guodong Li, Indradeep Ghosh, and Sreeranga P Rajan. Klover: A symbolic execution and automatic test generation tool for c++ programs. In *International Conference on Computer Aided Verification*, pages 609–615. Springer, 2011.

[8] Bjarne Stroustrup. *The C++ programming language*. Pearson Education, 2013.