Lappeenranta University of Technology
Department of Information Technology

Integrating test case generator into Eclipse environment

The topic of Master's thesis has been approved in the Departmental Council of Department of Information Technology on 11th October 2006.

Examiners:
Professor Jari Porras, D.Sc. (Tech)
Kimmo Hoikka, M.Sc. (Eng)

Instructor:
Professor Jari Porras, D.Sc. (Tech)

Vesa Jääskeläinen
Lappeenranta, 1st December 2006.

ABSTRACT

Lappeenranta University of Technology
Department of Information Technology

Vesa Jääskeläinen

Integrating test case generator into Eclipse environment

Thesis for the Degree of Master of Science in Technology

2006

61 pages, 25 figures and 1 appendix.

Examiners: Professor Jari Porras, D.Sc. (Tech)
Kimmo Hoikka, M.Sc. (Eng)

Keywords: software testing, source code parsing, automatic test case generation, Eclipse, C++, Symbian OS

This thesis studies how Eclipse environment can be used for test case generation. One of the major topics in thesis is, can existing Eclipse components be used to increase symbol knowledge to get more information to test case generator.

In this thesis first a quick introduction to software testing will be given to introduce reader to software engineering area being handled. More details will be given about the actual process of the test case generation. When basics are covered, an introduction to Eclipse environment will be given. More details will be provided for Eclipse components related to test case generation. Next an example case of test case generator and how it was integrated to the Eclipse environment will be provided. Lastly Eclipse based solution will be compared with existing solution to give some details of the differences in symbol knowledge and in execution time.

As a result of this thesis a prototype application was built that demonstrates that Eclipse environment can be used for test case generation and it can increase symbol knowledge. This increase comes also with an increase in execution time, in some cases even with a major increase. It is also noted that there are ongoing projects to improve Eclipse components, and those changes can improve results in the future.

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Tietotekniikan osasto

Vesa Jääskeläinen

Testitapausgeneraattorin integrointi Eclipse -ympäristöön

Diplomityö

2006

61 sivua, 25 kuvaa ja 1 liite.

Tarkastajat: Professori Jari Porras, TkT
Kimmo Hoikka, DI

Hakusanat: ohjelmistojen testaus, lähdekoodin parsinta, automaattisten testitapausten generointi, Eclipse, C++, Symbian OS
Keywords: software testing, source code parsing, automatic test case generation, Eclipse, C++, Symbian OS

Tämä diplomityö tutkii kuinka Eclipse -ympäristöä voidaan käyttää testitapausten generoinnissa. Eräs diplomityön pääaiheista on tutkia voidaanko olemassa olevilla Eclipsen komponenteilla parantaa symboolitietoutta, jotta testitapausten generointiin saataisiin lisää tietoa.

Aluksi diplomityö antaa lyhyen katsauksen ohjelmistojentestaukseen, jotta lukija ymmärtää mitä ohjelmistotekniikan osa-aluetta diplomityö käsittelee. Tämän jälkeen kerrotaan lisää tietoa itse testitapausten generointiprosessista. Kun perusteet on käsitelty, tutustetaan lukija Eclipse -ympäristöön, mikä se on, mistä se koostuu ja mitä sillä voidaan tehdä. Tarkempaa tietoa kerrotaan Eclipsen komponenteista joita voidaan käyttää apuna testitapausten generoinnissa. Integrointi esimerkkinä diplomityössä esitellään valmiin testitapausgeneraattorin integrointi Eclipse -ympäristöön. Lopuksi Eclipse -pohjaista ratkaisua verrataan symboolitietouden sekä ajoajan kannalta aikaisempaan ratkaisuun.

Diplomityön tuloksena syntyi prototyyppi jonka avulla todistettiin, että Eclipse -ympäristöön on mahdollista integroida testitapausgeneraattori ja että se voi lisätä symboolitietoutta. Tämä tietouden lisäys kuitenkin lisäsi myös tarvittavaa ajoaikaa, joissakin tapauksissa jopa merkittävästi. Samalla todettiin, että tällä hetkellä on menossa projekteja joiden tarkoituksena on parantaa käytettyjen Eclipse komponenttien suorituskykyä ja että tämä voi parantaa tuloksia tulevaisuudessa.

FOREWORD

I would like to thank my Master's thesis instructor and examiner professor Jari Porras, my thesis examiner Kimmo Hoikka and my co-worker Jussi Vestman for their comments on how to improve my thesis. Project for which this thesis was related was interesting and educating project for me. It taught me a lots of new things for which I am grateful and I want to thank Lappeenranta University of Technology's Laboratory of Communication Engineering for opportunity to work on this project.

Vesa Jääskeläinen
Lappeenranta, 1st December 2006.

TABLE OF CONTENTS

TABLE OF FIGURES

ABBREVIATIONS

API	Application programming interface; a term in software development that specifies how other software components can interact with component being externalized.
ARM	Advanced RISC Machines, popular 32 bit microprocessor architecture within mobile device manufacturers.
AST	Abstract syntax tree, a term in software engineering used to describe source code features in tree form.
CDT	C/C++ Development Tools, a software component developed for Eclipse environment to aid in C/C++ software development.
DOM	Dynamic Object Model, a term in software engineering meaning object model being used can be modified dynamically during runtime.
EPL	Eclipse Public License, a software license what is used to license most of the Eclipse Foundation's software.
IA32	Intel Architecture 32, most popular 32 bit computer architecture in personal computers.
IDE	Integrated Development Environment, a software development term meaning software where development is occurring. This software usually contains integration of source code editor, compiler and debugger.
PDOM	Persisted DOM, a software component within CDT used for symbol resolving and processing of C/C++ source code to extract symbols from it.
SDK	Software Development Kit, a set of software libraries, tools and documentation aimed to help on developing software for vendor's software platform or using specific component from vendor in developed software.
SWT	Standard Widgets Toolkit, a software component in Eclipse environment used to handle user interface.
TPTP	Test & Performance Tools Platform, a software component developed for Eclipse environment to aid in testing of software and to aid in improving performance.

1 Introduction

In software engineering area of software testing is getting more and more attention and its importance has been realized as the end users are demanding more stable products. As software testing is a time consuming process and needs human resources, many different ideas about how software testing could be automated has been researched for a long time and new techniques are being studied all around the globe to find even better solutions. One of the branches in automatic software testing techniques uses application source code as a basis of their operations. In this thesis it is being studied how Eclipse environment could be used for automatic test case generation by analyzing Eclipse environment for what features does it provide to aid in test case generation, and then study functionality of existing product doing automatic test case generation to recognize common elements from these both.

This work was conducted in co-operation between Lappeenranta University of Technology's Information Technology Department and SysOpen Digia's Smartphone Business Division. SysOpen Digia has recently released a new version of their EUnit Professional Edition software aimed to help on testing of Symbian OS C++ software. This product includes automatic test case generation technology and it will be used in this thesis as a basis for analyzing existing product doing this functionality.

Additional interest to Eclipse environment comes from fact that Nokia Corporation has recently released Eclipse based Carbide software suite to aid in development of Symbian OS C++ software for mobile devices. Carbide uses many components from Eclipse environment, adds Symbian OS C++ specific features and application wizards to fasten development. In order to streamline software developer's process of testing their software, there has been an idea to integrate EUnit Professional as an extension to Carbide software suite. In order to accomplish this task; EUnit Professional, Carbide software suite and Eclipse environment will be analyzed to determine what functionality could be used from these products to minimize maintainable components from EUnit Professional and to improve its existing features.

In order to limit scope of this thesis this study limits to C++ source code analysis in Eclipse environment and to describe some of the issues and risks related to porting existing Java based software to Eclipse environment. Prototype application will be presented to demonstrate the process and problems discovered during development of the prototype application will be described.

First an introduction to software testing will be presented in chapter 2. Then test case generation will be described in chapter 3. Introduction to Eclipse environment will be given and features provided by it will be described in chapter 4. Quickly introduction is given to Carbide software in chapter 4.4. Case study of the EUnit Professional Edition and demonstration prototype is detailed in chapter 5. Lastly results of the study will be revealed in chapter 6 and conclusions will be presented in chapter 7.

2 Software testing

Software testing is one activity in the process of software development, dedicated to verify correctness of the software being developed. Additional reason for verifying developed software is to minimize costs of the software development by detecting product defects early in development cycle. If defects are detected later in the cycle of development it might even require changes to product requirements. This causes lots of additional work to fix the problems. Additional work means that expenses of the software development grow by requiring additional development time and human resources; it might even delay release of the product and thus cause additional profit loss for the company.

Software testing has been researched for a long time, lots of different techniques and models for software development and testing has been proposed to get better quality products faster to the market. In old waterfall model [1] software testing is being done quite late in development cycle, right after software has been developed. Usually in waterfall model software testing is done by people specialized in software testing. Good in this old method is that extra perspective will be given to product and new type of problems can be found. Normally it is hard for traditional programmer to see his own mistakes and they might not have a good experience in testing software. When testing is done by software testing specialists, they have good knowledge about testing process and more problems can be found. Bad in this old method is that fixing problems found might require big changes to software. With possibly large scale of needed changes, temptation to ignore testing and discovered problems grows larger, especially when deadline is near. This affects negatively on product quality. Some of the proposed models for software development integrate parts of software testing to actual flow of the normal software development. Reason for this is to fix problems earlier and to detect new defects when they first surface. This shifts some work load from software testing professionals to regular coders and causes regular coders to think more about testability of the software. As a side effect, developers write also less error-prone software. When software is more tested its overall quality can also improve.

Different software testing methods can be divided to three major categories; unit testing, integration testing and system testing. In **unit testing** a small piece of software is verified that it works as it was specified to work. In **integration testing** different units are connected to each other and they are given stimulus to cause interaction between each other. Result of the interaction testing is being verified against expected functionality. In **system testing** complete system is being tested to verify its functionality against product requirements to verify that product itself complies with what it was built for.

Actual testing of the software is usually organized as a collection of multiple smaller tests. Collection of tests is called a test suite and a single test in this collection is called a test case. Test case prepares environment for running the actual test, setups input values for method and verifies that proper results are generated. If results match to predefined expectations set in the test case then test passes, if there is even one mismatch then test is seen as a failure. If failure is detected, problem must be analyzed and corrected. If expected functionality of the program has changed, test cases must then be refined to match those changes. One test case is usually limited to only test small set of features implemented in the software. Complete test suite is used to verify that all software components works as expected.

Software testing handled in this thesis is limited to automatic test case generation for unit testing and to support test-driven development process. In strict test-driven development process software testing is integrated to normal developer's software development process. It requires developers to first before writing expected functionality to write complete and runnable test case. If functionality needs new interfaces or methods, a dummy implementation must be written to allow compilation of the project. Written test case tests required functionality but when test case is executed it is failing as implementation is still missing. This causes developer first to think how functionality will be used and to think how it could be tested. As developer has already thought about functionality it will be easier to make actual implementation. After test case is written, functionality will be implemented and test case will be used to verify that it works as specified. This test-driven development process is illustrated in Figure 1. While guidelines presented in the test-

driven development process are good, in reality it usually differs a bit and test cases will be written after the functionality is implemented. Automatic test case generator can be used in this case to help user in implementing test cases by capturing restrictions and assumptions written in source code. As a result, user is presented with a runnable test case that verifies functionality.

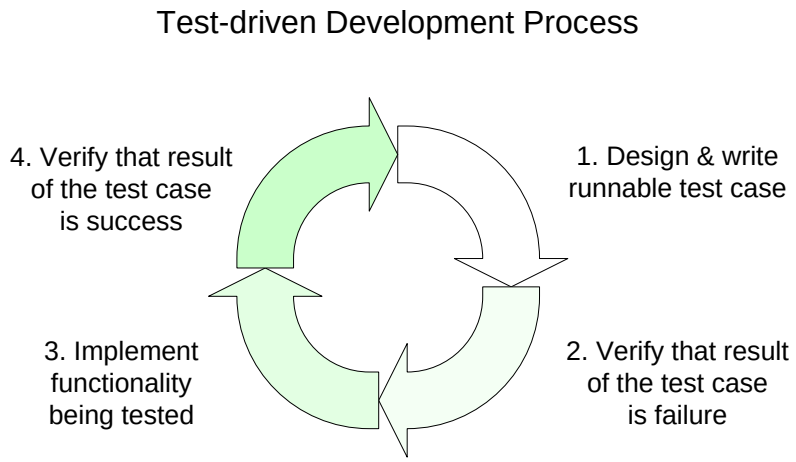

Figure 1 - Test-driven development process.

3 Test case generation

Test case generation is a technique to generate runnable test cases for given object. Its tasks are to analyze given object, determine how it could be tested, determine test case input values what will be used on testing, and lastly to generate runnable test case for target environment. In this thesis target object for test case generation is a C++ method (or C++ function) and its surrounding class, a structural analysis is done for them.

In order to generate test case for a C++ method, first user is asked to specify method for which to generate the test case. One way to do this is to analyze given project files and display object model to user and let user select from class hierarchy what method to analyze. When method has been selected, method's signature, method's body and method's surrounding class will be analyzed (see 3.1, Source code analysis and 3.2, Flow analysis). After all necessary sections of the source code have been analyzed; test case's input values can be determined (see 3.3, Test case data generation). There is couple of strategies here, generator can try to automatically generate all possible or relevant test cases, or alternatively let user to specify what paths to test from source code. Last step in test case generation is to generate code for test cases and write them to files for later compilation and running on target environment to verify functionality. Sequence of these steps is illustrated in Figure 2.

Automatic test case generation tries to fulfill specified test case adequacy criteria as well as it can. Adequacy criteria can be statement coverage, branch coverage, path coverage, or mutation adequacy [2]. In **statement coverage** every statement in tested code is executed at least one time during the test case. In **branch coverage** every branch in tested code is executed as true and false. In **path coverage** every path in tested code is executed from the beginning to the end. In **mutation adequacy** test case is tested against mutated tested code and if mutation is detected test case is considered to be good.

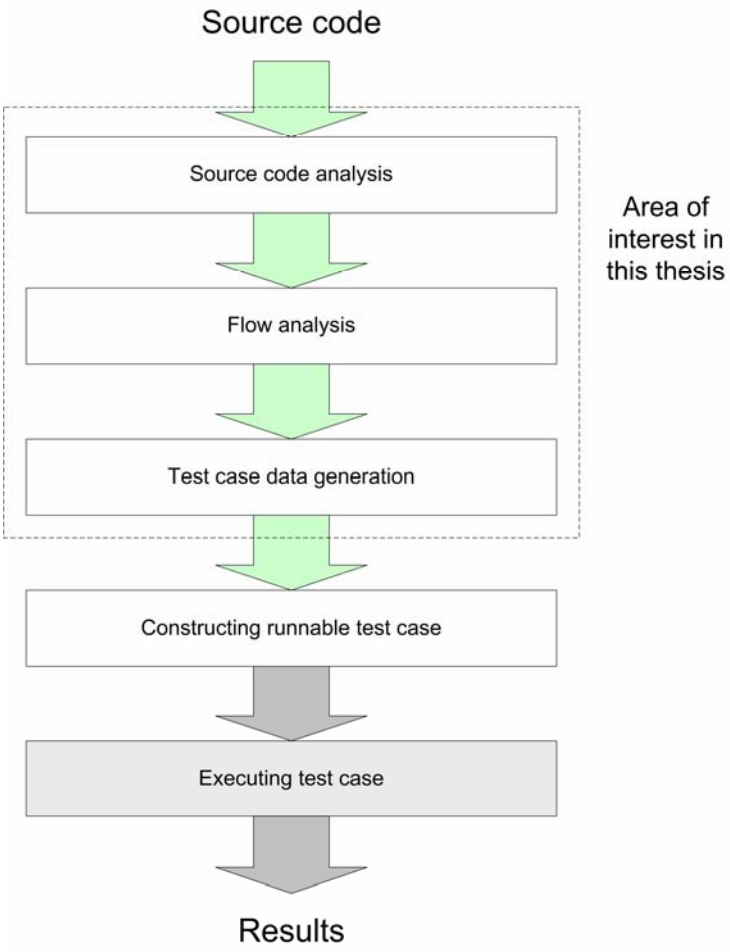

Figure 2 - Steps of test case data generation.

3.1 Source code analysis

In source code analysis given source code is analyzed to gather needed semantical information. Gathered information include abstract syntax tree (AST) of interested methods and symbolic information about used symbols. Semantical data model is being built from gathered information to be used on later phases of test data generation. Source code analysis is done exactly in same way as it would be done in normal compiler [3] as there is a need to have exactly same amount of information as normal compiler would have. If problem area only needs a subset of information then lesser amount of information provided by a simpler parser can be sufficient. Difference between normal compiler and test case generator is illustrated in Figure 3. The main difference is that in test case

generator there is no need to generate binary code and that phase can be omitted from the process and is being replaced by analysis of the flow, test case data generation and after that actual test case is being generated.

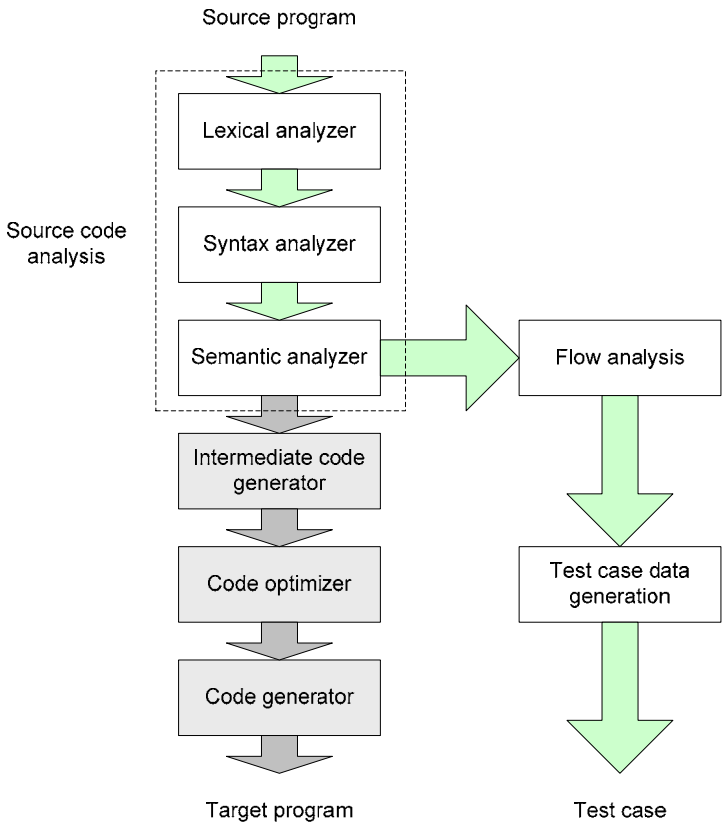

Figure 3 - Difference between normal compiler and test case generator.

As needed functionality is close to normal compiler it has been suggested by *G. Antoniol et al.* in their paper [4] that using C++ front-end from existing compiler would be a viable alternative for writing hand crafted parsers. Using well tested compiler front-end gives more confidential results, exactly same amount of information what compiler has and at same time fastens development time and most likely there will be lesser amount of problems related to source code parsing than on in-house made parser.

3.2 Flow analysis

Flow analysis uses semantical data model from source code analysis phase, builds up a data flow model and control flow graph of the source code being tested. **Data flow** model represents how variables interact with each other. **Control flow graph** is a model representing control structure of the code. Every branch makes fork in the graph and when branch ends it joins back to the previous level.

Tested method and its surrounding class will be analyzed for visibility and symbolical information. Analysis determines what variables are being used and what methods will be executed during the test case. When all dependencies have been determined, actual flow will be analyzed from start of the code to point of exit. This includes analyzing relations between variables and methods, and analyzing AST's of methods to extract branch structure and activation conditions.

Example for variable dependencies can be seen in Figure 4, it demonstrates how method argument *aArg* affects to class member variable *iMemberVar* and method's local variable *localVar*. In real applications variable relation information is more complex and it depends on position of the path being traversed. In example if *localVar* is not in range specified in *if* clause, relation to class member variable *iMemberVar* will not be made.

```
/* Example code demonstrating variable dependencies */

class CMyTestClass
        {
        public:
                void ExampleFunction(TInt aArg);

        private:
                TInt iMemberVar;
        }

void CMyTestClass::ExampleFunction(TInt aArg)
        {
        TInt localVar = aArg;

        if ((localVar >= 0) && (localVar <= 100))
                {
                iMemberVar = localVar;
                }
        }
```

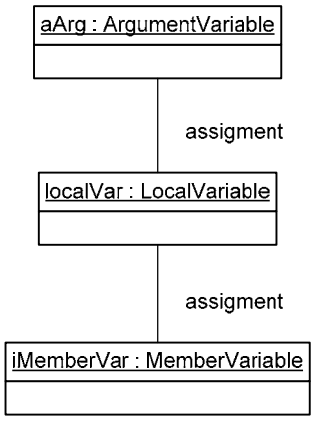

Figure 4 - Example of variable dependencies.

3.3 Test case data generation

Test case data generation uses models from previous phases to determine what values needs to be set as input values in order to execute specified path in test case. Results of the flow analyzer will be used to determine initial values for class members, global variables, and optionally to set return values for the stubbed methods. This includes analysis of branch activation conditions and relations between variables and return values of the called methods.

There are several different techniques to determine values for test case's input value. These techniques include symbolic execution [5], dynamic method [6], iterative relaxation method [7], and dynamic domain reduction [8]. In **symbolic execution** source code is analyzed and mathematical formula is constructed from expressions and solved to determine input values. In **dynamic method** initial set of input values is set, code is then executed and monitored that correct paths are taken, if there is a wrong path taken then input values are modified. In **iterative relaxation** method initial input values is set and modified iteratively to match branch conditions. It differs from dynamic method in way that it detects sooner paths that cannot be activated for given path in its iteration cycle and thus fastens processing of those cases. In **dynamic domain reduction** no initial values are

being set to input values, instead when control flow conditions and statements are executed, value ranges for input values is being refined, and at the end of code, value ranges are used to determine final input values for the test case.

4 Eclipse environment

Eclipse Software Development Kit (SDK) or more commonly referenced just as Eclipse is The Eclipse Foundation's multiplatform open source development environment for building Java products. One of the best known Eclipse based product is Eclipse Foundation's Java Integrated Development Environment (JDT). JDT features auto completing Java source code editor, integrated incremental Java compiler, integrated debugger, advanced code walking features like symbol reference and declaration locator, support for team development with integrated version control interfaces, and many more features to fasten software development. Eclipse has become quite popular development environment within Java developers [9] as its performance, stability and functionality has grown during the years of its development.

Eclipse allows integration of different development task directly to one single development environment. Example of this integration can be seen in Figure 5. It features Eclipse Foundation's Java IDE showing source code management, source code editing and features helping to locate problems in the source code. On top of the screen there is a toolbar featuring most commonly used features like compiling, running application in debugger and tools for walking in source code like search functionality. On center there is a source code editor which highlights programming language features and is showing one source code opened and displaying source of the detected problem. On left side there is a source code management in *Package Explorer* view showing opened projects and from which source files and libraries project builds from. On right side in *Outline* view there is a Java class viewer displaying contents of the active class being edited. On bottom there is a *Problems* view displaying automatically detected problems from the source code.

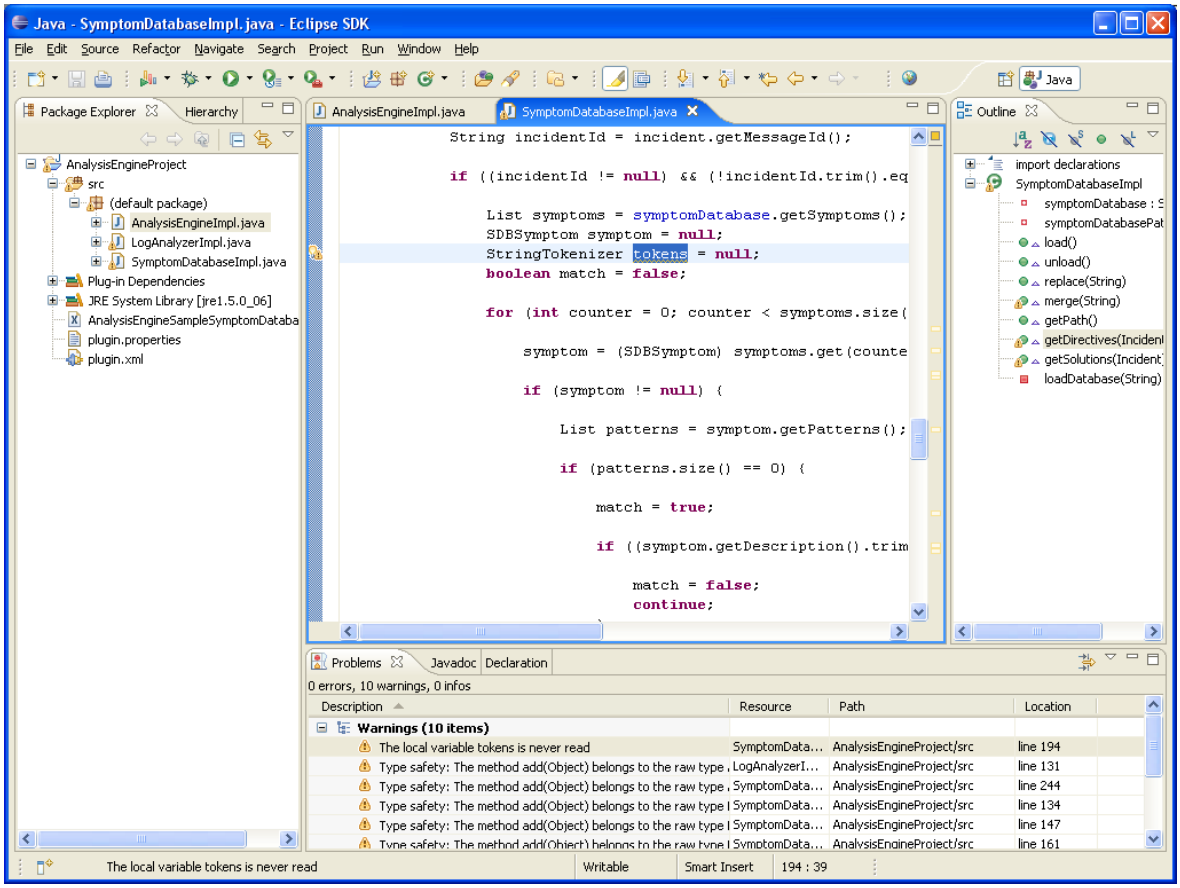

Figure 5 - Example project open in JDT.

Eclipse's software architecture has been designed to be easily extensible. It contains numerous different services that can be used and be further extended by using provided extension points. Extension points in Eclipse are attachment points where new software components can attach and add new features to existing services. Extensibility comes also with a down side; if subject of interest is not documented well enough its usage becomes harder. Eclipse has quite good technical documentation on their web site but lacks in some areas of in-depth information. To get in-depth information about components usually the source code and presentation notes from different Eclipse developer conferences are the best sources for information. Basic components of the Eclipse's software architecture and how test case generation is placed on it can be seen in Figure 6.

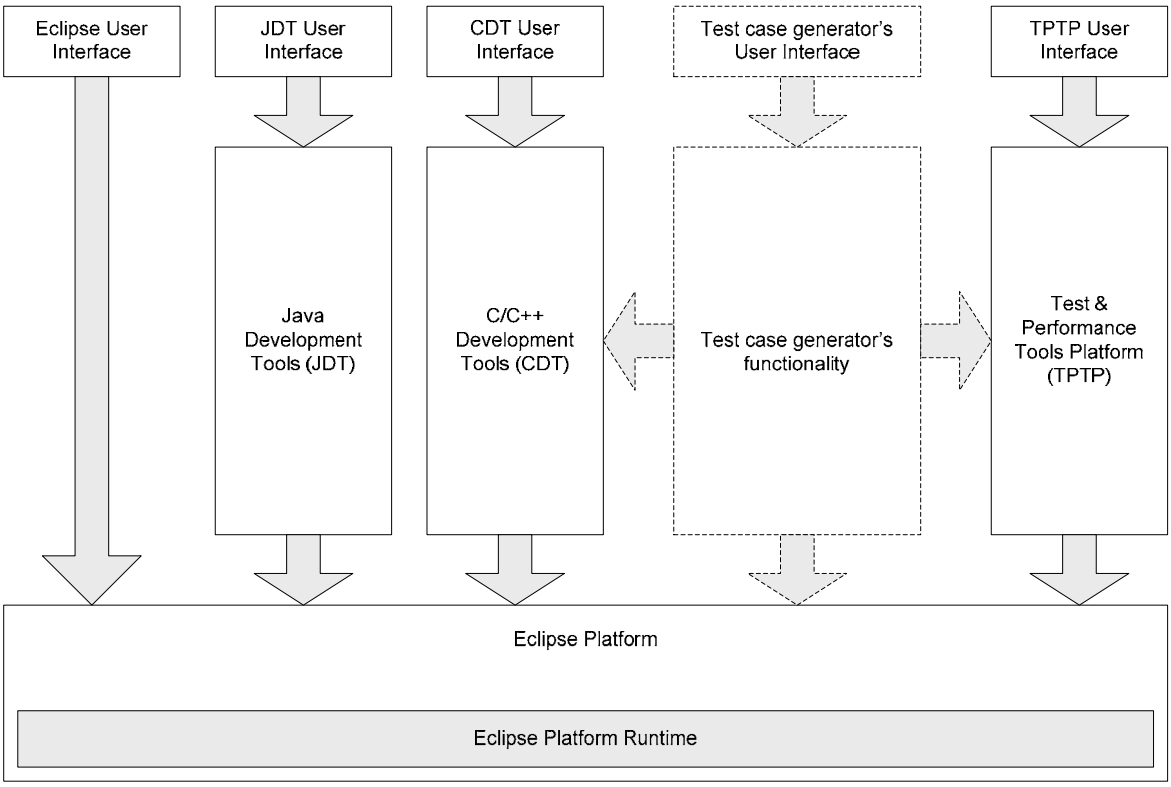

Figure 6 - Simplified software architecture of the Eclipse.

Eclipse Platform is the backbone of the Eclipse's software architecture and all Eclipse based products are built as plug-ins for Eclipse Platform. Platform itself also contains numerous services to aid in development and is elaborated in more detail in chapter 4.1.

As of success of Eclipse's Java IDE, there has been interest to develop also C/C++ IDE based on it and thus C/C++ Development Tools (CDT) project was born. At time of writing this document CDT is at version 3.1.1, its usability and extensibility has evolved dramatically from earlier versions. CDT is elaborated in more detail in chapter 4.2.

The Eclipse Foundation also offers tools to aid in testing of the software in form of The Eclipse Test & Performance Tools Platform (TPTP) project. Project includes source code and performance analysis tools, more details about TPTP can be found in chapter 4.3.

For generating test cases by analyzing C++ source code, new plug-ins needs to be developed. Developed plug-ins would interact with CDT to parse source code, and with TPTP to handle execution of test cases. During research part of this thesis, a prototype application was developed to demonstrate source code parsing using CDT. More details of prototype can be found in chapter 5.2.

Eclipse plug-ins provided by the Eclipse Foundation can also be used to build commercial applications as they are distributed under Eclipse Public License (EPL). EPL allows commercial usage of unmodified versions of the plug-ins. If modifications are to be done on plug-ins it is required by the license to share those modifications back to community. If new plug-in is to be developed its license can be different from EPL and therefore can contain confidential material. Optimal solution for commercial application developer would be to contribute changes needed to EPL components back to community and then provide closed source component that attach to existing components. Actual legal text of the license can be found from Eclipse Foundation's legal resources section on their website [10].

4.1 Eclipse Platform

Eclipse Platform is the basis of Eclipse based IDE's. Platform contains numerous services to aid in development of the features for IDE's and at same time it is platform independent allowing larger user base for Eclipse based products. Eclipse Platform has been so far ported to several different systems including Windows, Linux, Solaris, AIX, HP-UX and Mac OSX. This thesis contains only brief introduction to Eclipse Platform, more detailed information about the platform can be found from Eclipse Platform Technical Overview [11].

One of the advantages of the Eclipse platform is that it allows easy extensibility by plug-ins. This extensibility can be easily seen as almost all services in Eclipse are provided as plug-ins, there is only one component that is non-pluggable and that is Eclipse Platform Runtime. Plug-ins that forms some functionality together is called as features. To make

software components more easily manageable, features can be installed as one entity and Eclipse also provides an update service that allows user to download and install newer versions of the features from Internet. Some of the components found from Eclipse Platform can be seen in Figure 7. From platform most interesting components are *Workspace* for getting details about project being worked on and *Workbench* for interacting with end user and displaying results.

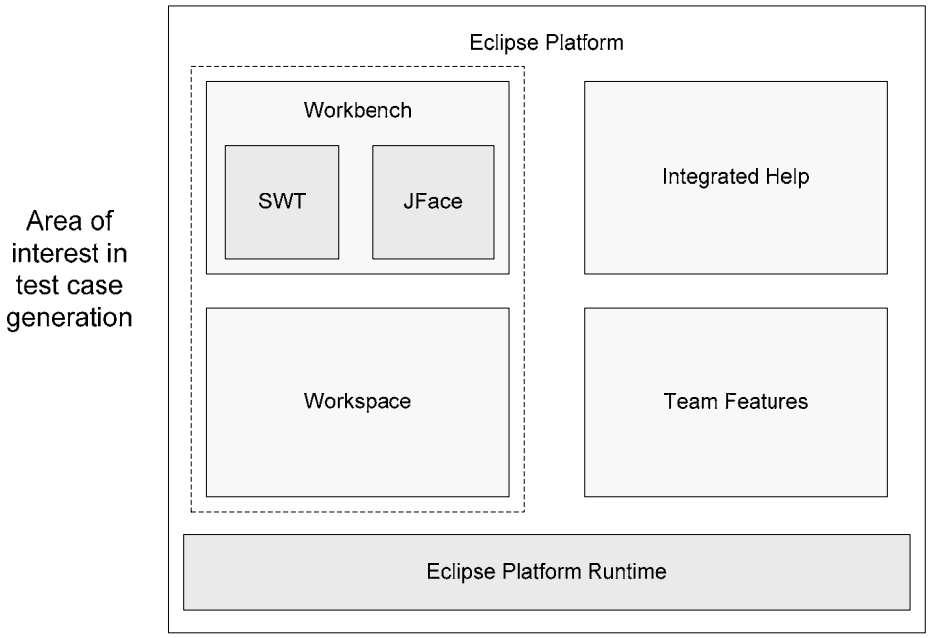

Figure 7 - Some of the components from Eclipse Platform.

Usage of the plug-ins to construct applications brings couple of interesting problems like how to protect crash of one plug-ins from crashing whole application and how to make sure that no other plug-in without permission to interfere with functionality of other plug-ins. As Eclipse is Java based Java Virtual Machine itself handles crashes gracefully and allows other parts of the software to continue so this crash protection comes almost for free. Eclipse maintains functionality of the whole platform by restricting unintended access between components by allowing components to specify what parts of the component can be accessed from other sources. Normally within Java program every class can use other classes public methods freely and this might not be intended as within module it would be required to have this public access, but if those interfaces are used wrongly it could cause

incorrect functionality. In normal Java application there is a class loader that loads new classes, in Eclipse there is new class loader that determines visibility information of classes and makes sure no protected parts of the classes gets accessed by non-approved components.

Eclipse Platform also allows building of standalone applications without overhead of the IDE components with subset of the platform called Rich Client Platform (RCP). RCP contains only components to enable multiplatform functionality including support for plug-ins, platform runtime and user interface.

Multiplatform products sometimes have quite poor performance as they try to emulate visual look and feel of the running system or bring completely new look and feel, Eclipse differs here in that it tries to maximize usage of native components of the system while providing same functionality across all platforms. If some feature is not available on ported platform it is only then being emulated. This guarantees that Eclipse based products give familiar look for users of specific platform allowing greater user friendliness.

Platform independent user interface is handled by Standard Widget Toolkit (SWT). One important difference of SWT as compared to other multiplatform user interface toolkits is that SWT uses user interface components from the running operating system to as far as possible, only non-supported components are being emulated. This allows users to more quickly learn usage of the new software as they are already using familiar user interface components, and at same time gives more robust user experience as often operating system components are faster and require lesser amount of resources.

4.2 C/C++ Development Tools

C/C++ Development Tools (CDT) is an Eclipse plug-in designed to aid in C/C++ software development. CDT has several features for software development, to extend it, and to build products based on it. Common usage scenario for CDT is development of C/C++ software in CDT's IDE and then using its services to compile, debug and execute developed

software. Similarity of the CDT and the JDT can be easily seen in Figure 8 as there are only some cosmetic changes and used programming language has changed. This similarity allows users familiar of the Eclipse to adapt quickly to different development environments based on it. CDT's popularity has been growing and it is backed by wide industry support so its future would seem to be quite bright. According to project organizers usage of the CDT has been growing as there are lots of direct downloads from Eclipse Foundation's web servers and additionally there are over 17 commercial products that are based on CDT [12].

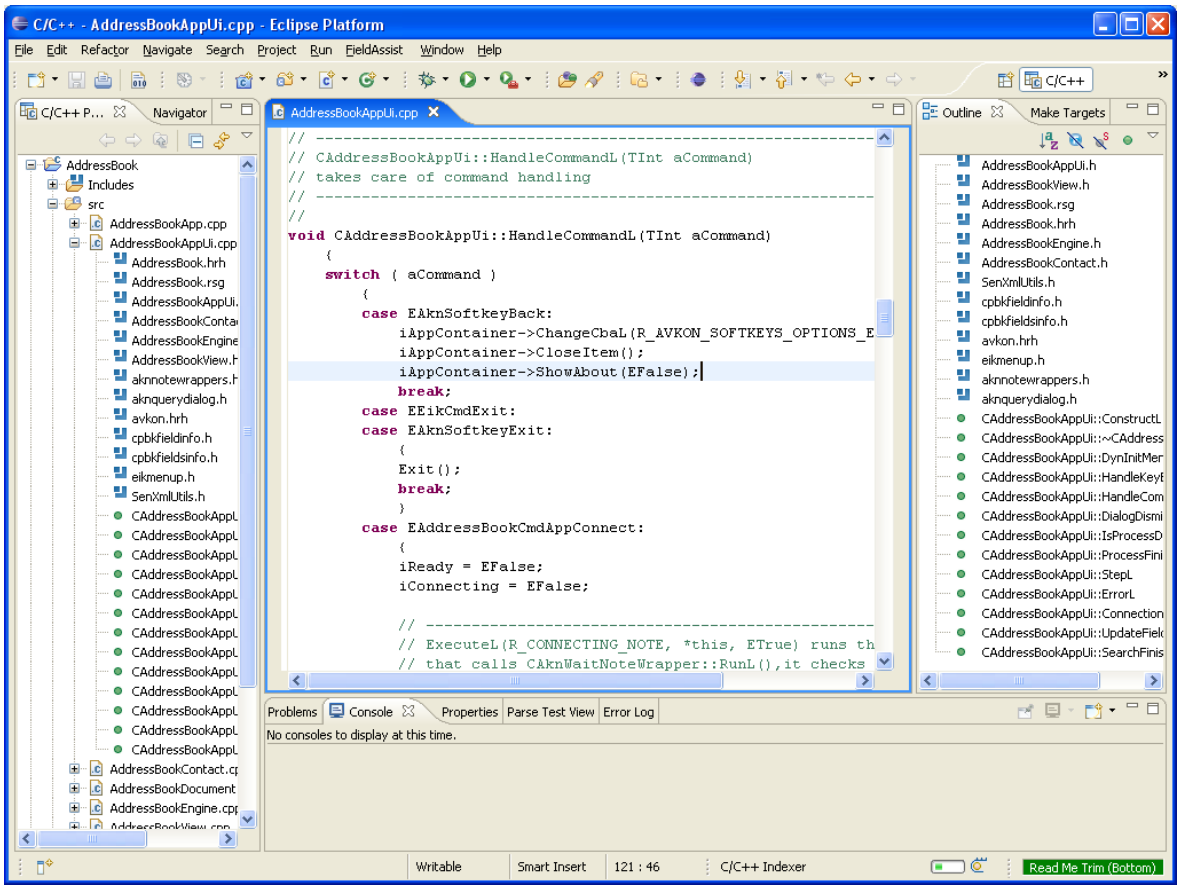

Figure 8 - Eclipse CDT IDE with Series 60 SDK example project open

As stated above CDT can also be used for building products using it. CDT provides application programming interface (API) for Eclipse plug-ins for accessing source code features like AST, object model and project settings. For test case generation ability to get method body in form of AST and object model for surrounding classes is major interest

from CDT. Construction of the AST tree from scratch can be time consuming process and if time spent on this step can be made smaller it affects greatly to end user's experience about the product. With CDT we can get AST for almost free as CDT's IDE uses it to support traversing within source code and it also stores this information to Persisted DOM (PDOM) database to fasten further queries of the symbols. As this work is now shared to some degree, time spent on querying AST's for interested methods gets shorter. One very handy feature for analyzing AST's is that CDT supports symbol binding. Symbol binding is technique used to resolve symbols found from the source code and in CDT's IDE this is used for locating references of methods and variables. When symbol is found from AST, it can be located from CDT's object model using binding resolver. CDT's object model is described in more detail in chapter 4.2.2. Binding resolver is context aware resolver that traverses object model to locate correct method overload or variable based on analysis of symbols found in expression AST tree. This saves some work and time on determining what method overload or variable is being used.

4.2.1 Persisted DOM

Persisted DOM (PDOM) in CDT is an indexing framework for finding symbols found from the source code and resolving them. PDOM consist of several components; PDOM database, PDOM indexer and PDOM clients. PDOM is also a research project to improve performance within Eclipse so it is designed to be neutral about programming language details stored on it. At time of writing of this thesis, technical documents about internals of the PDOM was not yet published, source for this information is from Eclipse Foundation's Wiki service [13] and source code of CDT.

PDOM database is a flat binary file that stores project specific symbols in meta-data directory. Data in disk is organized as different data type items and are currently limited as 4 KiB per data item. Data is also stored in memory cache to make queries fast enough to be usable on larger code bases.

PDOM indexer is a key functionality that extracts details from source code and then stores this information to PDOM database. PDOM indexer interface is designed in way that it allows different vendors indexers to be plugged in. Bundled PDOM indexer supports parsing of GNU C/C++ source code and its extensions. Features from C/C++ source code can be queried in form of AST or as CDT's Object Model nodes. When CDT's parser was studied, and used on building of the prototype application, no missing C/C++ features were found, so it would seem that CDT's bundled parser would be quite complete when compared against C++ standard [14].

PDOM clients are software components that use symbol information from PDOM database to do their task. Examples clients using PDOM are *C/C++ Search*, resolving of declarations and references of symbols and *C++ Outline* view.

4.2.2 CDT's Object Model

CDT's Object Model acts as a logical layer for modeling C++ source code. CDT object model contains two layers of information that are connected with each other. One way of accessing information is logical structure that is more abstract about language and second way is to access object details in AST level.

Object model in CDT builds from instances of source code element (objects) and their relations with other objects (scopes). When source code is analyzed, PDOM indexer builds a hierarchical representation of the definitions found. Different hierarchy levels are represented as different scopes in code. It represents quite closely to C++ language itself, in example, tree starts from global namespace scope and all objects that are defined in global namespace can be found from that scope. If new scope is introduced all of its objects are its children. Example of the scope hierarchy and how to get from AST level to object model level can be seen in Figure 9. In figure there is presented hierarchical scope levels for a class member variable *iMyVariable* and method *MyMethod()* and their AST level symbol names (*IASTName*). To get from AST level to object model level first a binding must be resolved by using *IASTName.resolveBinding()*. If symbol is resolved

correctly it returns matching object instance from object model. To move in object model hierarchy helper methods like *getScope()*, *getFunctionScope()* and *getParent()* can be used.

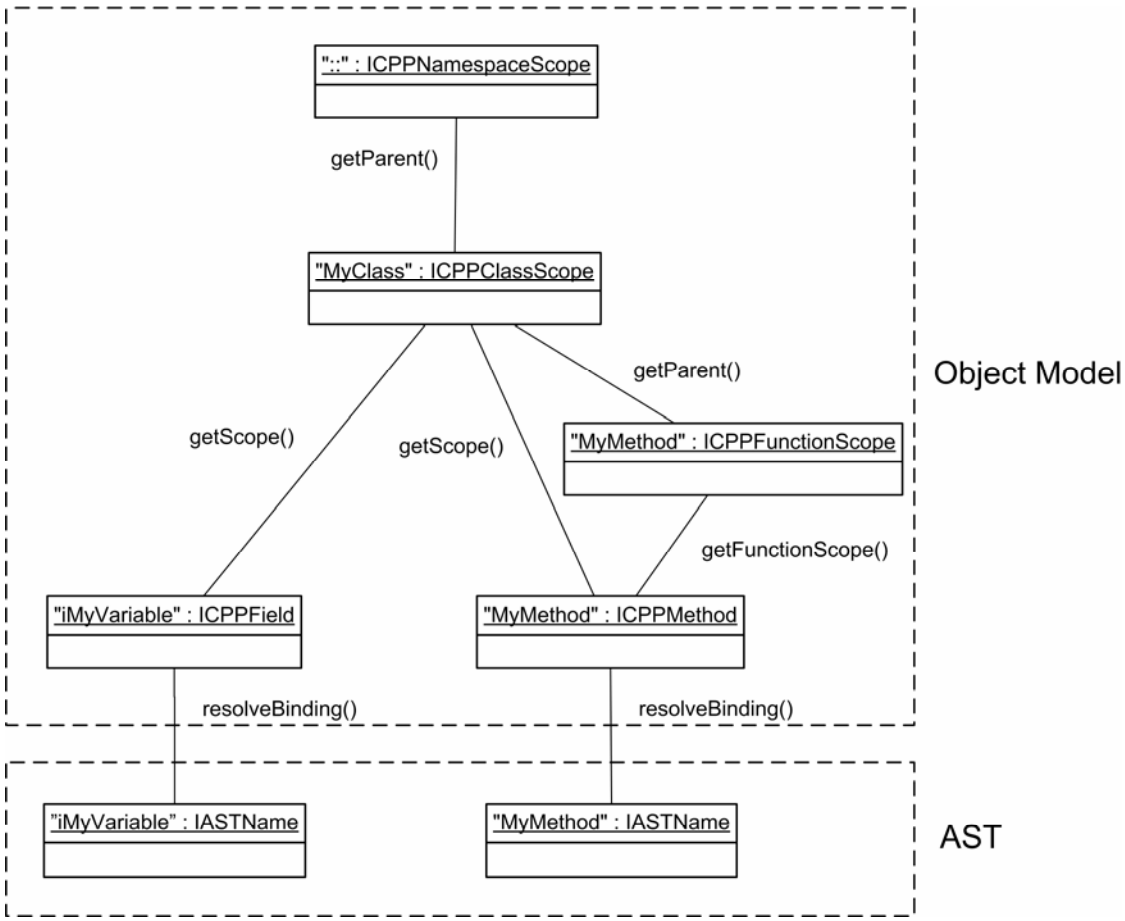

Figure 9 - Example of CDT's object model.

AST of the C++ source code is exactly what CDT's parser outputs and it is following quite closely schematics of the C++ standard [14]. AST level itself contains enough information to be used for extracting needed details from source code to build test cases. But in order to use CDT to its full extend; only interested parts of the AST should be evaluated and as there is already context aware symbol resolver in CDT and it should be used to ease and fasten the process. Only part of the AST that is interest for us is the target method body, rest of the needed information can be resolved from object model using PDOM.

Link between AST level and object model level is bidirectional. To get from AST level to object model level first a symbol having *IASTName* in must be located. It works like an indirect bridge for this direction. Symbol itself cannot be used to determine what object it is, instead its context must be analyzed to determine exact overload of the target method or variable. CDT has a helper functionality for doing this and is called as binding resolving and can be done by calling *IASTName.resolveBinding()*. It uses PDOM to locate objects and it returns object instance from object model that it implementing *IBinding* interface. Subtype of the object can then be determined by using Java's *instanceof* operator. To find context of the binded object from object model, its scope must be solved. This can be done by using *IBinding.getScope()* to get instance of *IScope* and then to traverse hierarchy to upper levels by calling method *IScope.getParent()*. It gives parent scope for an object, in example for methods it gives a class or namespace where it is being defined. To get method scope instead of the class scope or the namespace scope where method is being defined *IFunction.getFunctionScope()* can be used. It is also possible alternatively to do manual scanning of parent scope's children to locate correct definition. To get back to AST level from scope level *IScope.getPhysicalNode()* can be used to get corresponding *IASTNode*. Simplified class structure for C++ method (*ICPPMethod*) and C++ class member variable (*ICPPField*) can be seen in Figure 10.

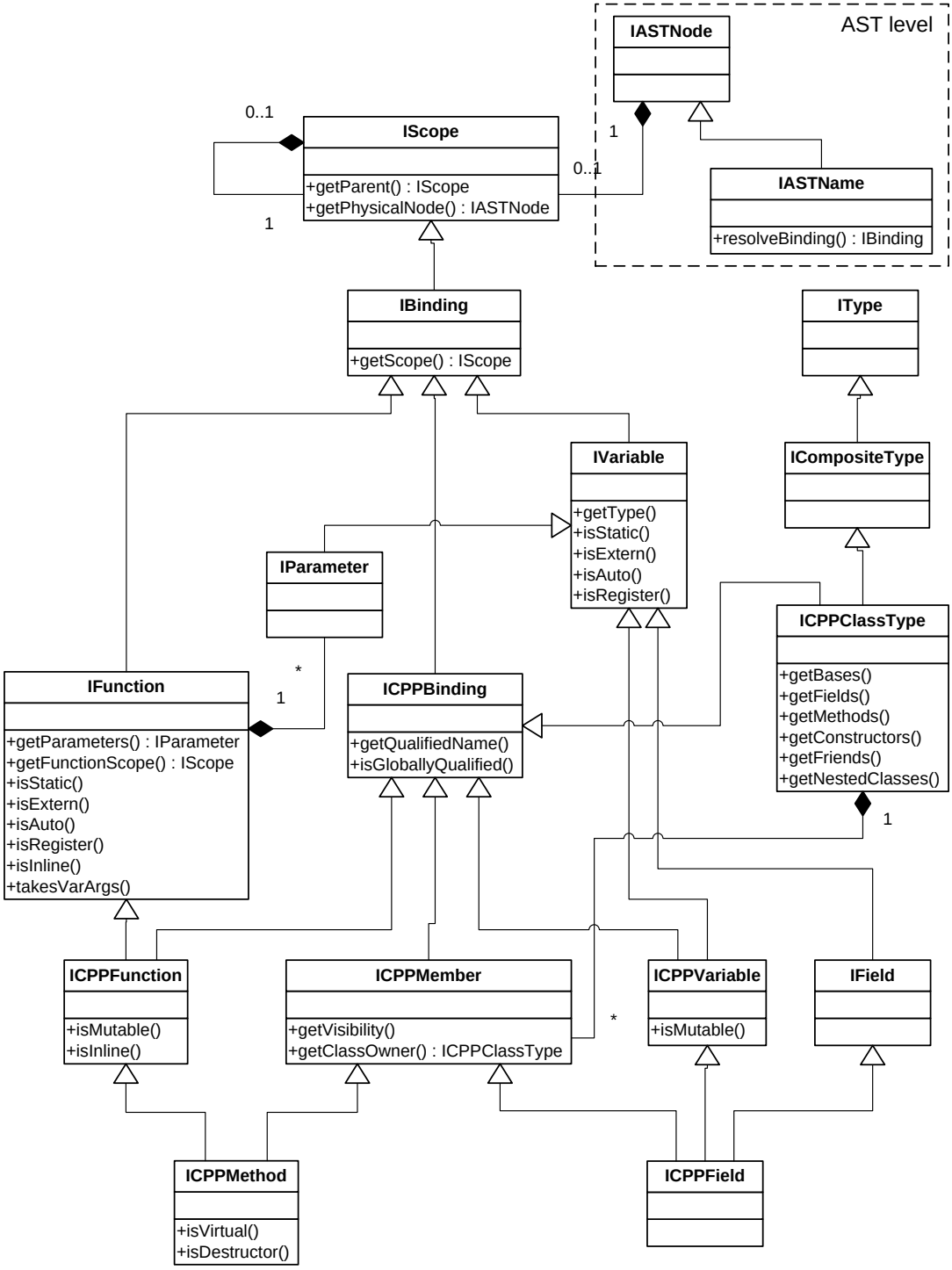

Figure 10 - Simplified class structure for C++ method and member variable.

4.3 Test & Performance Tools Platform

Test & Performance Tools Platform (TPTP) is a testing framework for Eclipse. It allows application developers to analyze their applications performance and functionality to guide process of making improvements to their software. TPTP is divided to four subprojects called TPTP Platform project, Testing Tools project, Tracing and Profiling Tools project, and Monitoring Tools project. These projects are all described briefly in below.

TPTP Platform project provides basis for other subprojects by providing common user interface, remote execution environment, data collection and other services. Platform provides several different level extension points to allow subprojects to share common look and feel, and at same time allowing different types of target platforms to be used for testing. Testing Tools project builds on top of the TPTP Platform and provides a framework consisting testing editors, test deployment, test execution, data collection, data history and reporting support. Tracing and Profiling Tools project builds on top of the TPTP Platform and provides framework to gather performance and tracing data and to do analysis on them. Monitoring Tools project builds on top of the TPTP Platform and provides a framework consisting capability for collecting and analyzing system and application resources.

TPTP also has some projects under technology preview to demonstrate future software components. At time of writing this thesis there is currently C++ Code Review Provider listed as one of the projects. C++ Code Review Provider gathers information about source code and at same time it checks out for some common programming errors and reports back about them to the user. C++ Code Review Provider works on AST level by using visitor pattern support provided by the CDT. Visitor walks on the AST and collects statistical details of the operators, definitions and declarations found from the source code. At time of writing this thesis there is no documentation available for C++ Code Review Provider, only source code is available on Eclipse Foundation’s CVS service.

As TPTP does not provide user interface to make method selection, primary usage for TPTP would be to use it as a platform for executing created test cases and collecting statistical information about the success. C++ Code Review might mature in future to also support doing more complex source code analysis, but as its name suggest it is only a tool meant to capture software problems. At this time there is a need to write custom source code analysis component that uses CDT directly and then either to create extension for TPTP to execute test cases, or write own component for doing also that task.

4.4 Case: Carbide.c++

Carbide is Nokia Corporation's newly introduced product family for developing applications and user interface themes for Symbian OS based mobile phones. Carbide products are based on Eclipse Platform and there are versions of Carbide for both Java and C++ development and for a theme development a custom user interface theme design tool is also available. For this thesis we are more interested on C++ development and this feature is provided by Carbide.c++ product series. Other Carbide products are not detailed on this thesis; refer to Nokia's mobile developer website [15] for further details about them.

Carbide.c++ is meant for developing Symbian OS C++ applications for mobile phones and comes in three different versions called Express Edition, Developer Edition and Professional Edition. From these **Express Edition** is free version for non-commercial usage and has limited set of features available, and rest of the editions is for commercial development. **Developer Edition** is targeted for regular software developers while **Professional Edition** is more targeted for device or firmware manufacturers. All these tools are based on CDT and use it as a basis for IDE functionality. To differentiate Carbide from normal CDT it provides Symbian OS specific add-ons like project creation wizard, project file management wizards, user interface designer, debugging support for emulator debugging, and in commercial versions also for in-device debugging. Additionally Carbide comes with bundled C/C++ compiler for both ARM and IA32 architectures in order to compile applications to be executed on mobile phone and in emulator.

Our interest to Carbide.c++ comes from fact that automatic test case generation case presented in this thesis is related to Symbian OS C++ application testing. Integrating application testing to development environment where normal development activities are performed could bring additional interests for the product. This would also allow more streamlined end-user's experience. However at time of writing this thesis Carbide.c++ is at version 1.1 and includes only modified version of the old CDT 3.0.1 and it has some limitations for doing source code analysis. This old version of CDT does not include PDOM and provided indexer is a limited one. It has been rumored that future version of Carbide.c++ would include newer version of the CDT and as both Symbian Ltd and Nokia Corporation can be seen in list of CDT's contributors it would suggest that they are planning to upgrade to newer versions on future releases. Again similarity between Carbide.c++ Developer Edition IDE and other Eclipse IDE's can be easily seen in Figure 11 as there are only small cosmetic changes and addition of new Symbian specific features. This similarity allows existing Eclipse users to be more conformable with it from start.

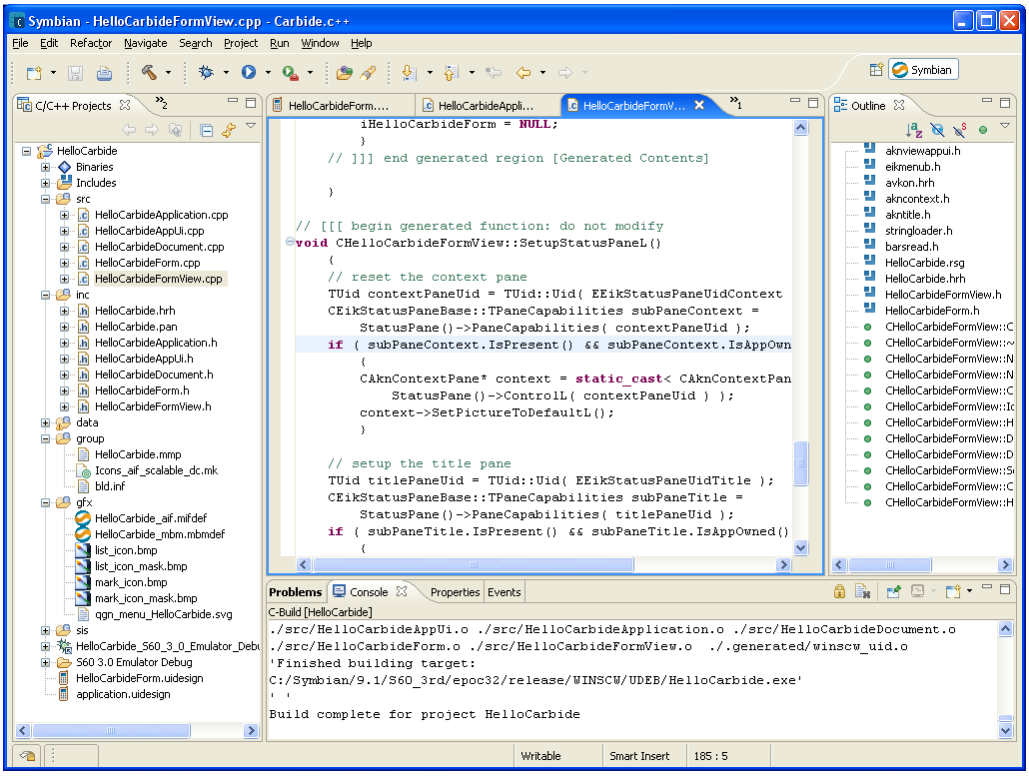

Figure 11 - Carbide.c++ Developer Edition IDE

Nokia has been extending CDT's features by improving its debugging facilities, and Symbian has been developing template engine and lately also offline indexes. **Template engine** can be used to generate project skeletons to make starting of the development of new applications easier. **Offline indexes** are targeted to solve problem of indexing complex and time consuming system headers beforehand to fasten usage of PDOM indexing service. For Carbide.c++ Symbian has developed support for Symbian OS SDK's and other Symbian OS specific features, and Nokia has developed support for compiling of source code, their own Symbian phone SDK's, debugging and running it in their emulation environment. Freescale Seminconductors has provided CodeWarrior compiler technology for compiling Symbian OS C++ code in Carbide.c++ products. [16, 17]

Nokia or Symbian has not publicly announced availability of SDK's for extending Carbide.c++ and as its modified CDT is a bit too old for real testing, its usability is still an unknown factor, but as their documentation states [16, 17] that other Eclipse plug-ins can be used to extend Carbide.c++ it could be assumed that in future version of Carbide.c++ using CDT from other plug-ins could be a possibility. To confirm this extensibility it would require some discussions between parties to verify it or to get proper SDK for extending it.

5 Case: EUnit Professional Edition

EUnit Professional Edition is testing software within SysOpen Digia's QualityKit 4.0 software suite. It is a standalone Java based tool for designing and running test cases for unit, module and integration testing on Symbian OS based mobile devices. Recently released EUnit Professional Edition version 4.0 features a functionality to automatically generate test cases for Symbian OS C++ source code. It lets user to choose method for which to generate test case and then possibility to generate all test cases or only one test case for user selected path. Software also contains other interesting features for software testing like memory leak detection, automated stub and adapter creation, possibility to automatically execute test cases and to generate reports on test runs. For this thesis our interests is automatic test case generation for Symbian OS C++ source code, more details about its other features can be found from SysOpen Digia's QualityKit 4.0 product brochure [18].

QualityKit is featured in Figure 12. Figure shows a class browser on middle of the screen presenting class structure for test project. On right side there is a source code viewer showing source code where selected method can be found. On left side there is a task selector for whether to create unit tests, module tests, user interface tests, functionality tests, configure tests suites and run them.

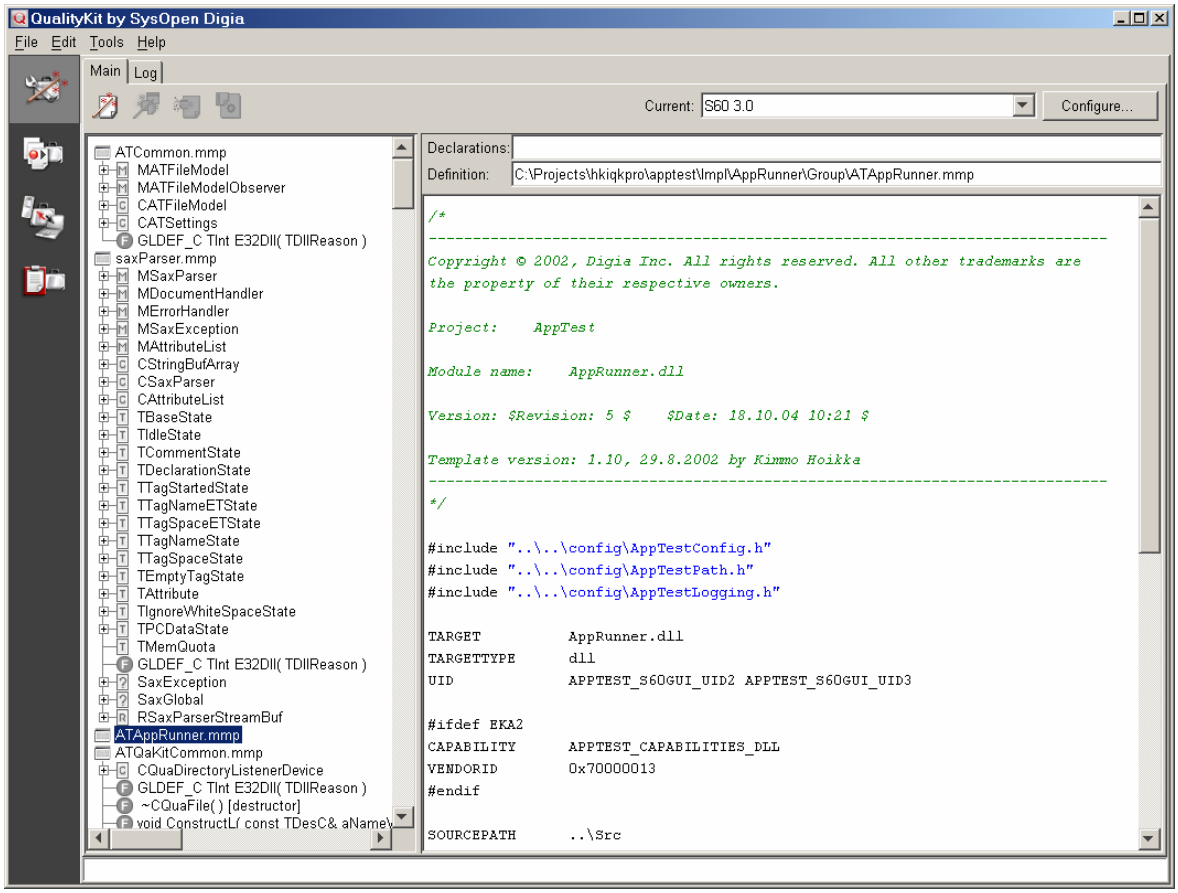

Figure 12 - QualityKit application from EUnit Professional 4.0 software suite.

As Nokia has recently introduced Carbide.c++ product family for Symbian OS C++ software development it would sound ideal to also integrate this testing environment to there. As Carbide.c++ is based on Eclipse technology it is also extensible and as it includes also CDT, source code analysis phase can be done using it. Symbol resolvation phase can be partly shared with other functionalities of the IDE as some of the needed source code features have already been processed and solved, so it fastens source code analysis phase. As both EUnit Professional and Eclipse are based on Java programming language some parts from the existing source code base can be reused while developing new plug-in for Eclipse. Proof of concept prototype application was developed during research part of writing this thesis to demonstrate feasibility of using Eclipse as an automatic test case generation platform.

Functionality of EUnit Professional Edition's automatic test case generation and prototype application is described more in depth in following chapters. First some details about current functionality of automatic test case generation in EUnit Professional Edition is described in chapter 5.1, and then some implementation details about prototype application for Eclipse environment is presented in chapter 5.2.

5.1 Automatic test case generation

Automatic test case generation in EUnit Professional works by first analyzing source code and then doing decisions based on it; lastly it generates a test case for later compilation and execution. These steps are briefly described below.

In early development phase of the product it was determined that doing a system with complete knowledge would require too much work with given resources and a limited set of source code knowledge could be sufficient to construct test cases for Symbian OS C++ source code if source code is written against Symbian OS C++ coding conventions [19, 20]. This decision led to development of the simpler handcrafted parser based on ANTLR parser generator [21] to extract source code features like class structure, branch model and expressions. Parsers are based on simplified version of the C++ standard [14] and it does not include any complex features like template support and has simpler variable type identification system. Different parsers in EUnit Professional and what they produce are illustrated in Figure 13.

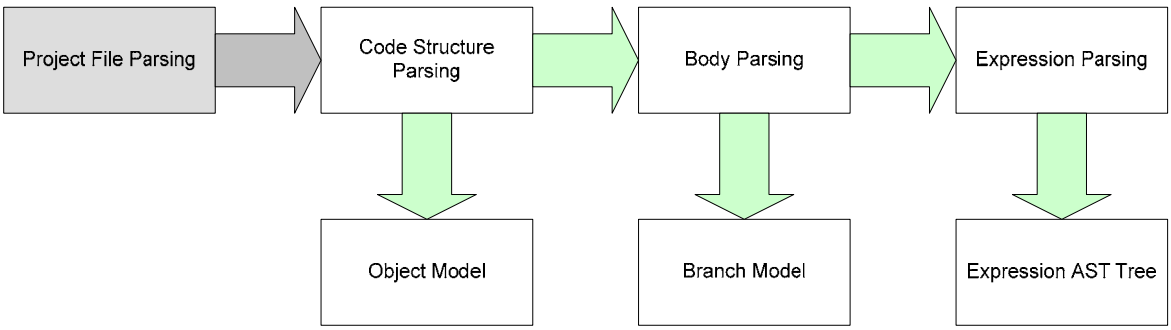

Figure 13 - EUnit Professional's different parsers.

Code structure parser is used to extract object hierarchies from the source code. It parses given source code and project's header files to extract methods, variables, classes, structures, enumerations and namespaces to builds up object model. Code structure parser does not analyze internals of method bodies or variable initializers, it only records contents of those as a strings and lets next parsers to handle those. Object model represents source code features as a hierarchical structure of elements. There are certainly similarities between CDT's object model and EUnit's object model but combining these two would require additional information to be stored on CDT's model and this requires more memory resources and slows down CDT. Better solution would be to build references from test case generator's object model to CDT's object model and store additional information only in generator. Example of EUnit's object model can be seen in Figure 14.

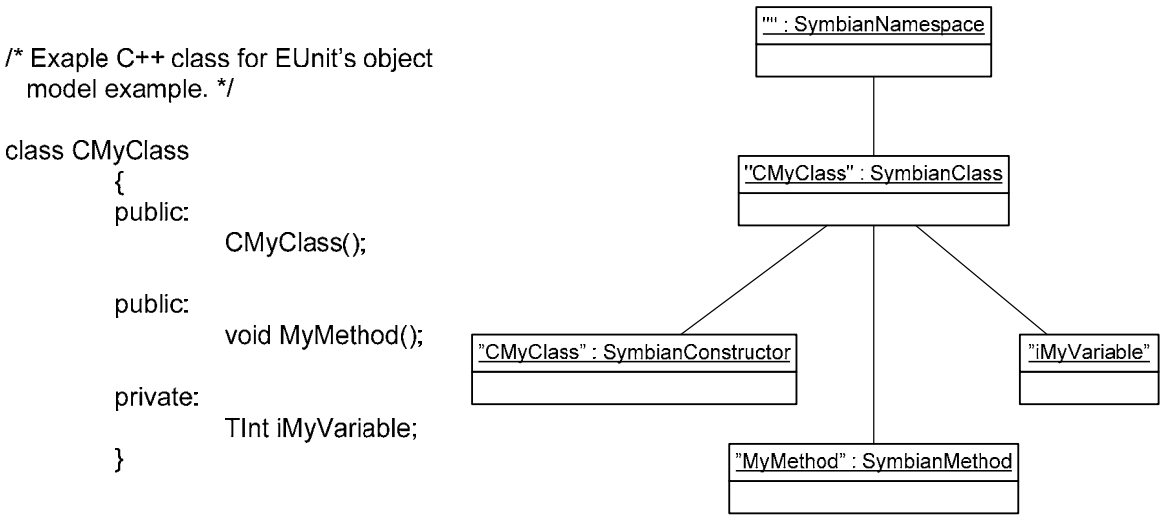

Figure 14 - Example of EUnit's object model.

At this point object model is built and user is presented a possibility to choose what method to analyze. After selection is done selected method body will be analyzed by body parser to build up a branch model for method. Body parser scans method body statements and expressions to locate branches and when branch is found it adds it as new children to active branch. Body parser only handles subset of the C++ standard [14] that is required on parsing of method bodies, it does not include parsing of expression and if expression is

found its condition is stored as a string for later processing using expression parser. Example of branch model can be seen in Figure 15.

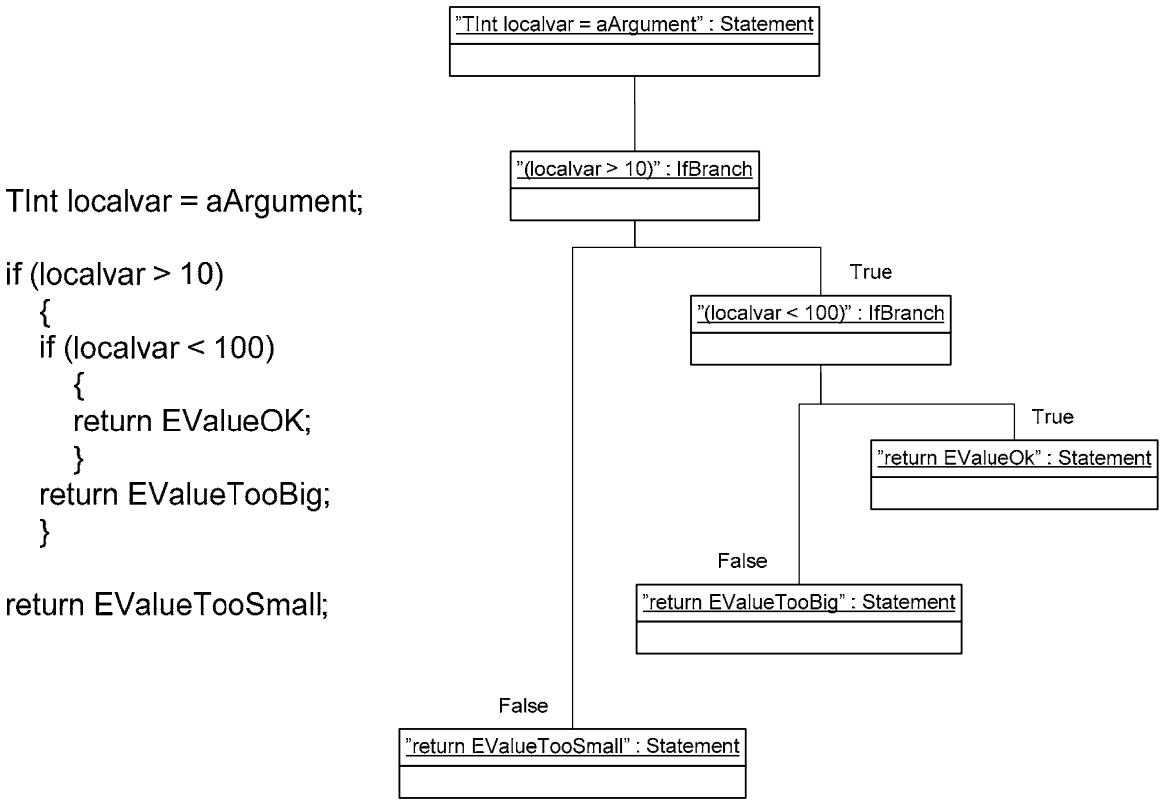

Figure 15 - Example of EUnit's branch model.

Now that branch model is complete analysis continues by analyzing of expressions found in statements and in branch conditions. Expression parser is built to quite closely follow the C++ standard [14] to get more details to make decisions based on what information can be found from expressions. It parses given expression strings and builds up an expression AST tree. Example of temperature conversion from Celsius to Fahrenheit for a return value of method call in expression AST tree form can be seen in Figure 16.

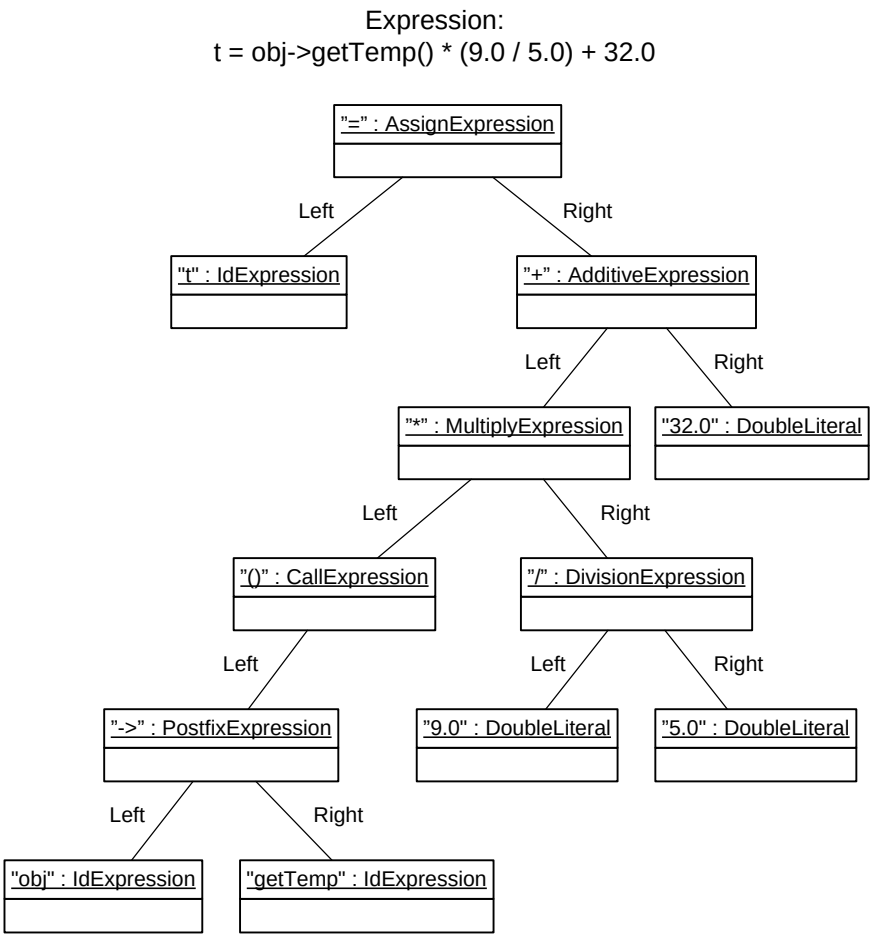

Figure 16 - Example of EUnit's expression AST tree.

After all syntaxical information has been extracted from source code, next step is to do semantical analysis and then perform test case data generation. This includes determining all relations between variables and methods are being used, and expressions are being analyzed to determine how they affect variables and methods being called. Class constructors and their initializer's are being analyzed to determine initial state for the class being tested. Last step is to generate a test case for analyzed path.

5.2 EUnit integration to Eclipse

A prototype application was developed to demonstrate feasibility of integrating EUnit Professional's automatic test case generation engine to Eclipse environment using CDT as

a parser. Current limitation of the EUnit Professional's parser is that it is not feature complete and does not process system headers at all. With CDT it could be possible to get more details of the source code and improve this situation. Main idea of the prototype application is that it is a proof of concept prototype. It does not try to be complete nor be used by end users so it lacks in features. Prototype application is detailed in more detail in following chapters. Overall details of software architecture are described in chapter 5.2.1. Integration issues and problems related to parsing and object model building is described in chapter 5.2.3. One solution to method selection is presented in chapter 5.2.2.

5.2.1 Software architecture

Prototype application builds on top of the Eclipse Platform. Prototype uses platform's services to handle user interface, uses its plug-in mechanism to make calls to other plug-ins and install itself on Eclipse. Functionality of the prototype is divided into automatic test case generation engine, object model builder and user interface that is sending control information to the engine. Figure 17 presents a simplified architecture of the prototype application featuring also not developed TPTP extension for executing test cases. As primary purpose of the prototype was to demonstrate feasibility of using Eclipse and CDT only parsing and generation aspects are demonstrated by prototype and rest of the needed functionality needed in final product has been omitted.

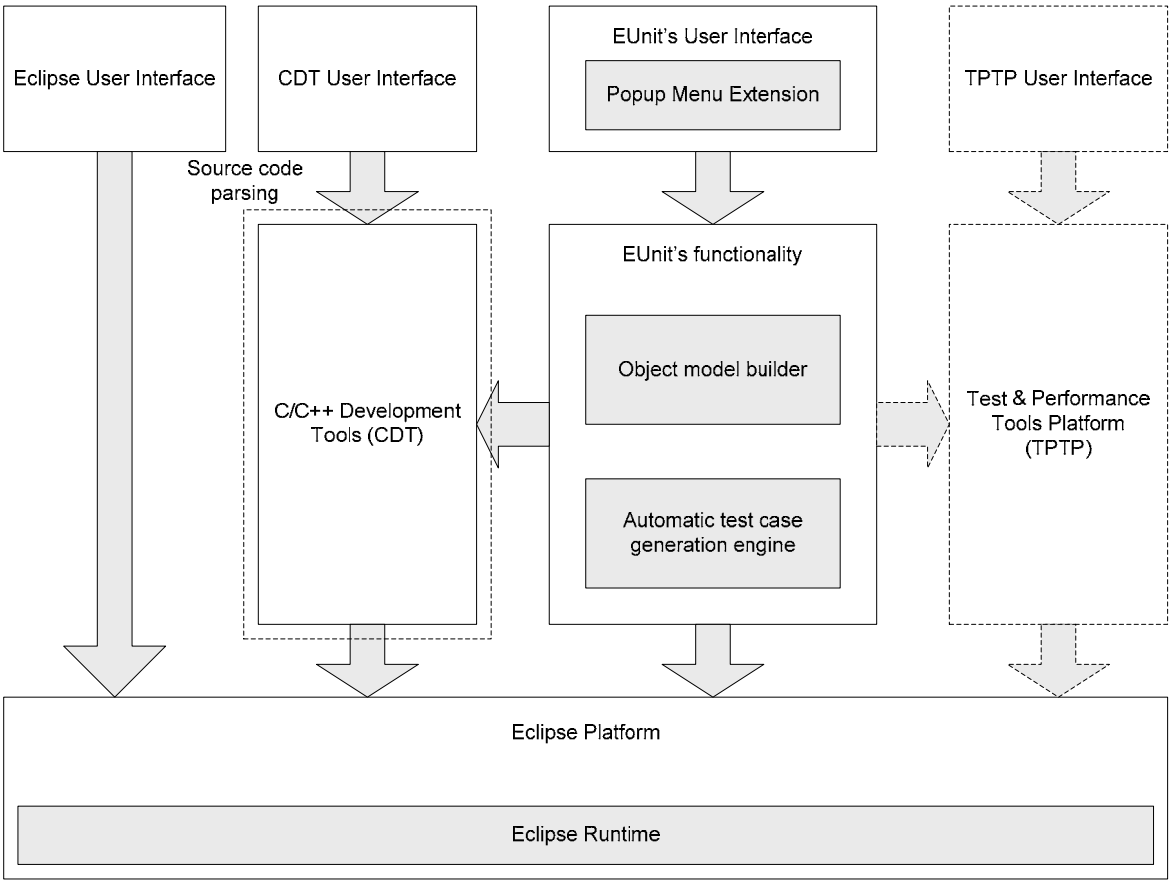

Figure 17 - Prototype application's simplified architecture.

5.2.2 Popup menu extension for method selection

There are basically two solutions to method selection problem, first one is to present new dialog to user with class browser, and the second one is to use existing one. It would be more logical and more integrated to use existing one and extend it to have feature to generate test case for given method. In Eclipse there is Outline view functionality that displays source code details like classes and their methods to user and there are also some other places where methods are presented. If user right clicks on selected method Eclipse will present a popup menu for user giving several actions what user can do with it. As CDT and other plug-ins add entries to these popup menu's it would be most logical place where to add new entry for automatic test case generation. One of the advantages in Eclipse's popup menu handling is that menu entries are registered for specific types of objects. In this case, everywhere where method name is presented same popup menu entry also

appears there. Example of popup menu extension showing additional entry *Analyze function* can be seen in Figure 18. Popup menu handling is called as an object action delegate in Eclipse. Object action delegate will receive events when user interacts with objects having specified registered type and will record selected method. When user makes a selection from popup menu entry, *run* event is sent to delegate and it can start to process command. Recorded selection (*ISelection*) can then be resolved to specific type with Java's *instanceof* operator. In example with method selection first an *IStructuredSelection* is returned and then from it *IFunction* can be queried to further refine selected method.

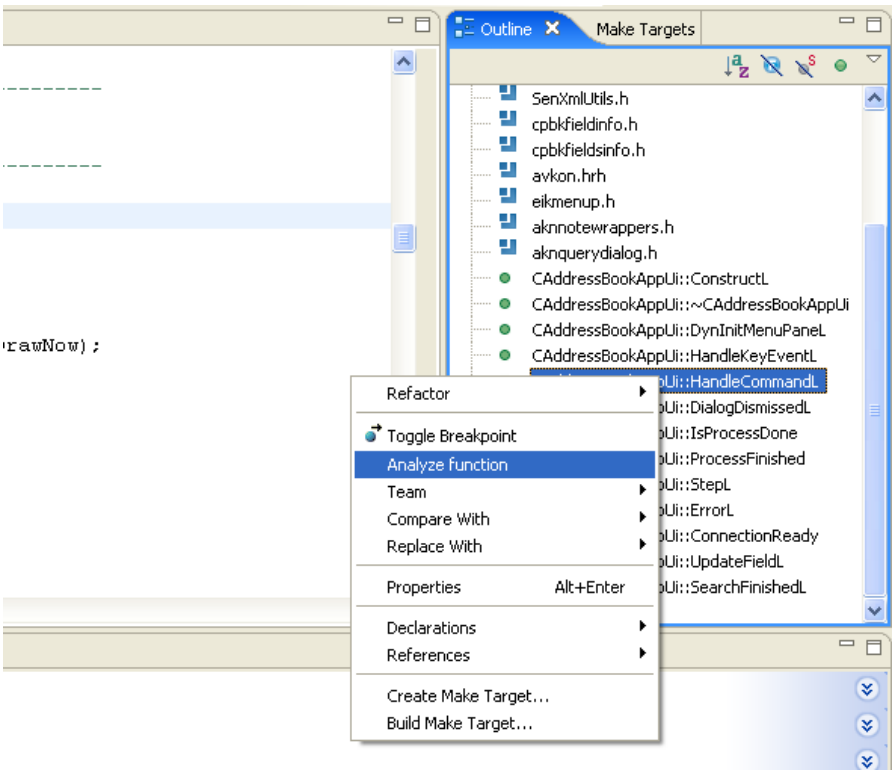

Figure 18 - Popup menu extension in prototype

5.2.3 Object model building

While CDT gives much more information about source code features than the old parser did, building of the object model is more challenging with a new parser. Problem comes from the fact that CDT gives far too much detail about the source code, it parses all Symbian OS system headers and if same strategy is used for object model building as in

old solution there would be many thousands of unneeded top level objects and then all those would have also sub-objects. Old parser was limited for parsing only parts of the application source code what were given for it and it did not parse Symbian OS system headers at all. This information loss was compensated by predefining most commonly used objects in configuration files. While this compensation mechanism can now be removed and more complete data is available, needed memory and processing power becomes problematic. One solution would be to filter objects or headers out of the model, but then situation would be similar as what it were before and this was unacceptable solution. Only viable solution left was that construction logic for the object model must be changed. If unknown symbol is found while parsing source code, CDT's binding mechanism is used to find symbol from CDT's object model, if object can be located it will be created on object model. This solution takes full advantage of CDT's PDOM indexer service as only the used symbols will be resolved and most likely they are the same what user has already queried indirectly by using IDE's features while developing his own application.

5.2.4 Test case data generation

As much as possible from the EUnit Professional's automatic test case generator source code base was intended to be left as unmodified and be reused without modifications. After analysis of the existing functionality, only parts related to source code parsing and construction of internal object model and expression trees needed to be modified. Figure 19 shows results of the test case data generation in raw format. In final product this information would not be presented to end user and EUnit Professional would use this raw form to construct final test case by replacing all tags to further customize outputted source code and builds a test case for later compilation and execution in simulator or in real mobile phone.

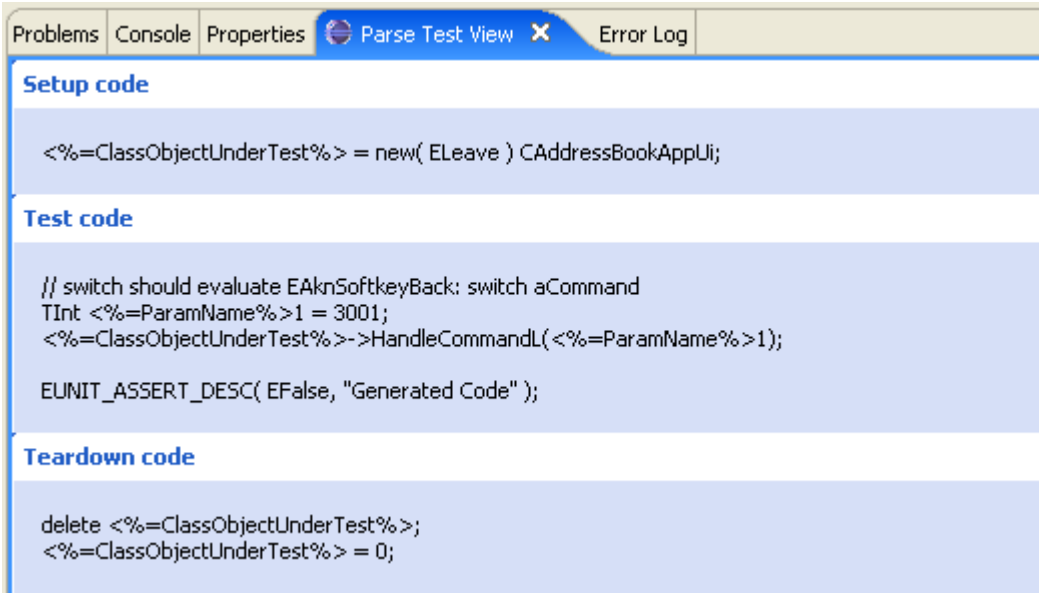

Figure 19 - Example of the raw output from automatic test case generation engine.

6 Results

In order to get picture how good CDT's parser is there needs to be comparable results between EUnit's parser and CDT's parser. To achieve this, a custom measurements application was built that can be integrated to both environments. Measurement application uses EUnit's object model as a source for its information and tries to locate symbols based on analysis of AST's of tested methods. Results of the measurements will provide details if there are any improvements on symbol knowledge as it were major interest for switching to more complete parser. Additionally some statistical information like object counts and execution time will be captured to give more information about how integration affects to other areas of the application and to the end user experience of using the application.

Following chapters will describe how measurements were done and what analysis gives as a result. In chapter 6.1 test setup used in measurements will be described. To give reader a better understanding about how measurement applications works, an execution flow of the both measurement applications will be presented in chapter 6.2. Analysis of the measurement data will be presented in chapter 6.3.

6.1 Test setup

As prototype application was developed before EUnit Professional Edition was released, prototype application's automatic test case generation engine was first synchronized with release version of EUnit to get comparable results. EUnit's automatic test case generation engine has some predefined constants and classes for most commonly used symbols. To get better comparison of different parsers these predefinitions were removed from prototype application's automatic test case generation engine; only basic C++ types were left. From EUnit only automatic test case generation engine were used and simple measurement application were created using it. Custom symbol resolver was written to traverse method's AST and locate symbols from expressions and try to resolve them. This same custom symbol resolver was used in both applications to give comparable results. For prototype application object model building was integrated to method analysis as it uses

CDT's symbol resolver to find symbols. Tested method selections were done manually by the author of this thesis from Nokia Corporation's Series 60 3rd Edition SDK. From SDK different types of projects were located and total of 16 methods were manually selected. List of selected methods can be found from Appendix A. Selection criteria for methods were to select different types of methods with different types of symbols. Results of both EUnit and prototype application were generated, compared and analyzed. All the tests were run on same computer system running Windows XP Professional SP2 with latest software updates. Hardware in system was equipped with AMD Athlon 64 3200+ processor and with 1 GiB of system memory. Latest versions of the components from Eclipse SDK and CDT were used on testing. Major component versions to mention were Eclipse Platform 3.2.1, CDT version 3.1.1 and Sun's Java VM 1.4.2-10.

6.2 Execution flow of the test applications

Before presenting measurements, first an introduction to execution flow of both applications is given. First an execution flow of the measurement application for EUnit will be presented and then execution flow of the prototype application.

EUnit's automatic test case generation starts by loading of predefined constants and types. This process uses Java's built-in XML loader. As there are lots of different objects and types, lot of the time is spent on allocating and initialization of those. This is only initial cost during software startup, but will be presented in measurements. Next step is to load Symbian OS C++ project file (bld.inf) to get knowledge about used source codes and to build up the object model. From created object model automatic selection of tested method will be done and symbol knowledge analysis will be done for it. As a last step results of the analysis will be written to text file. This execution flow of the EUnit measurement application can be seen in Figure 20.

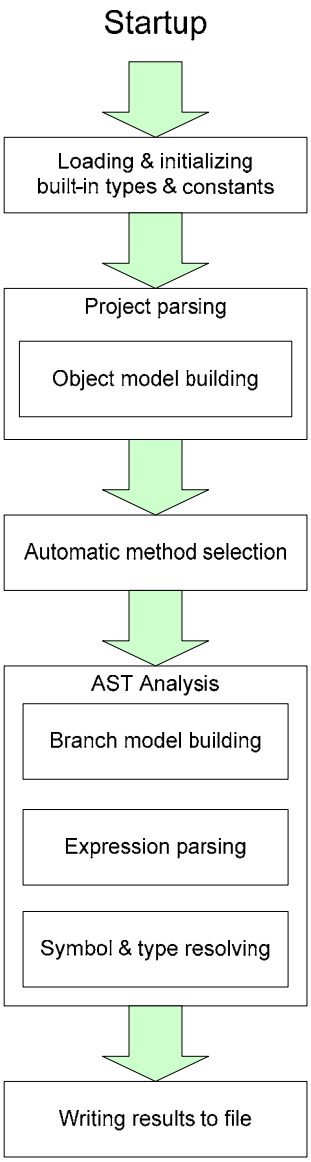

Figure 20 - Execution flow of the EUnit measurement application.

In **prototype application** first the Eclipse IDE will be started and it loads different plug-ins including CDT and developed plug-ins containing functionality of EUnit's automatic test case generation and measurement application. Loading time of the Eclipse IDE is omitted from results as it needs manual interaction before measurement can be started. As CDT comes with its own project manager, first a new project must be configured. This includes making a workspace where projects will be stored, adding a new C++ project, configuring source code folders, specifying include paths and predefined symbols, and

selecting used indexer for PDOM. For source code folders, project's *src* and *inc* folders were specified. Include path was configured to contain Symbian OS C++ system header directory and project's *inc* folder. As a predefined symbol *__CW32__* was defined to fake for system headers that we have a real compiler processing headers to resolve otherwise unknown symbols. Last step on configuration were to select used parser *Full C/C++ Indexer* or *Fast C/C++ Indexer* to cause CDT to parse source code files. After indexer selection is done, CDT automatically processes source codes to build up index. As this initial processing time is included in normal development process, time spent on it will not be measured. When IDE is back on idle state manual method selection will be done and measurements will be started for selected method. First a selection will be used to get translation unit from CDT and then selected method will be located. Once method is found, initial object model containing method and all its parent classes and namespaces will be built. During analysis of the AST of method body if yet unknown symbol is found CDT's binding resolver will be used to resolve symbol and then it will be added to object model. For namespaces, only definition will be made as it would otherwise include too much of unneeded objects. It should be noted that prototype application is not feature complete and is not optimized for speed so it sill lacks in functionality and this will negatively affect on measurements. Execution flow of the prototype's measurement application is presented in Figure 21.

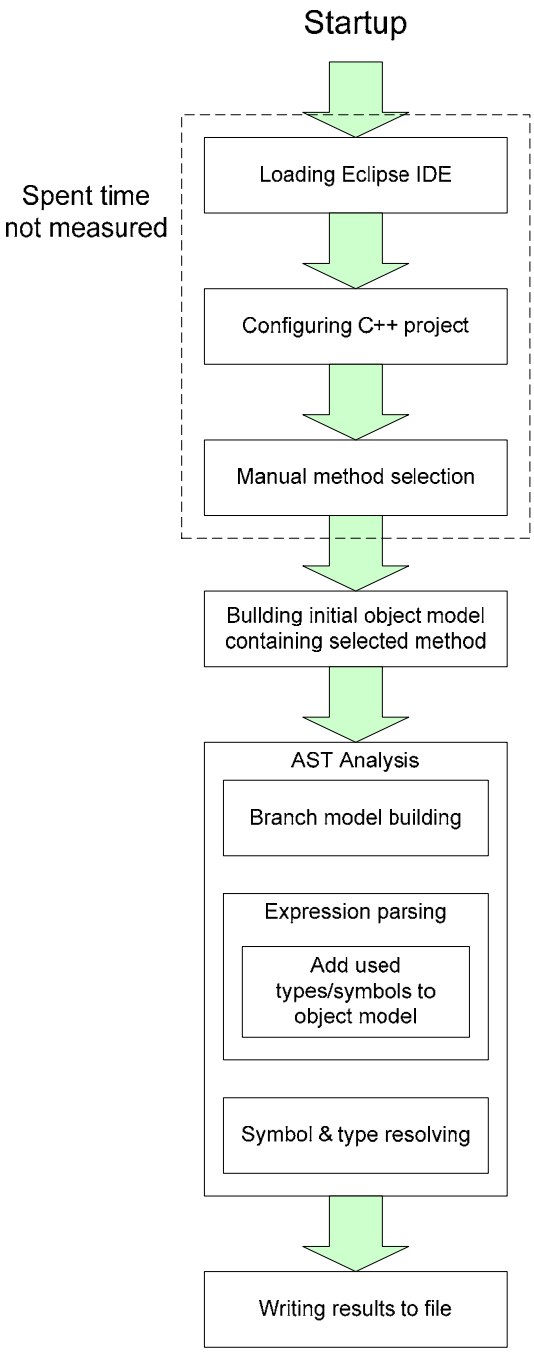

Figure 21 - Execution flow of the prototype's measurement application.

6.3 Measurements and analysis

Following details were measured from all tested methods: execution time, number of found methods and functions, number of found variables, number of found constants, number of

unknown symbols, and total number of objects in EUnit's object model. Raw results of these measurements can be found from Appendix A.

Analysis of CDT's symbol knowledge results reveals that there are still some unknown symbols and incomplete types. Some of these could come from fact that measurement application was developed quickly and its symbol and type locator is not following C++ standard too closely and some symbols and types might not be found. Some could come from fact that projects were not compiled beforehand to allow generation of additional project specific headers and projects settings might not match what project files would tell. Distribution of CDT's symbol resolving is illustrated in Figure 22. As figure shows in tested material there are still 15 % of symbols which could not be resolved correctly and this could affect results. It was also interesting to note that there weren't any differences in symbol knowledge whether *Full C/C++ Indexer* or *Fast C/C++ Indexer* were chosen.

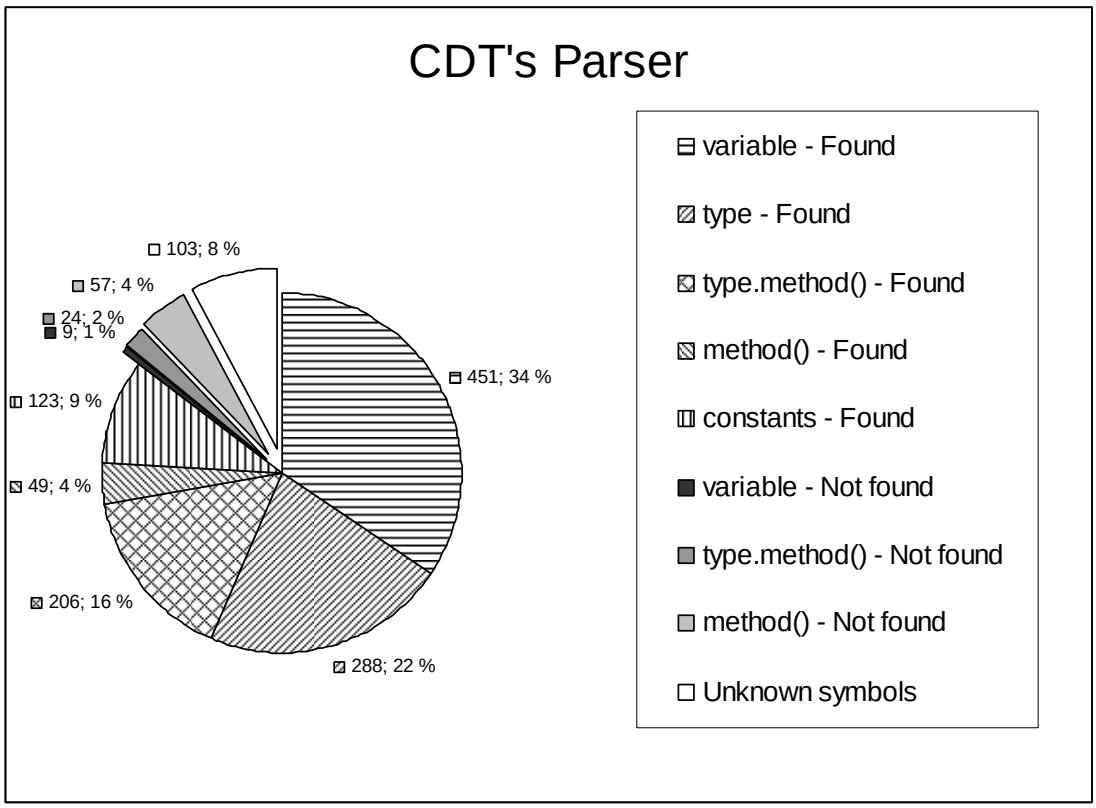

Figure 22 - CDT's parser analysis for symbol knowledge.

Analysis of EUnit's symbol knowledge results reveals that what was noticed in previously, there are some missing symbols and incomplete types. As Figure 23 shows in tested material there are about 39 % of unknown symbols or incomplete types. When these results are compared against results from CDT's analysis, there is definitely increase in symbol knowledge when CDT is used. As both measurement applications had exactly same symbol and type locator these results can be said to be comparable. Major difference can be seen on actual object models, there are lots of new classes in CDT's model that was resolved during processing of method's AST. As object model dumps are quite large, they are not included in here, numerical results can be found from Appendix A.

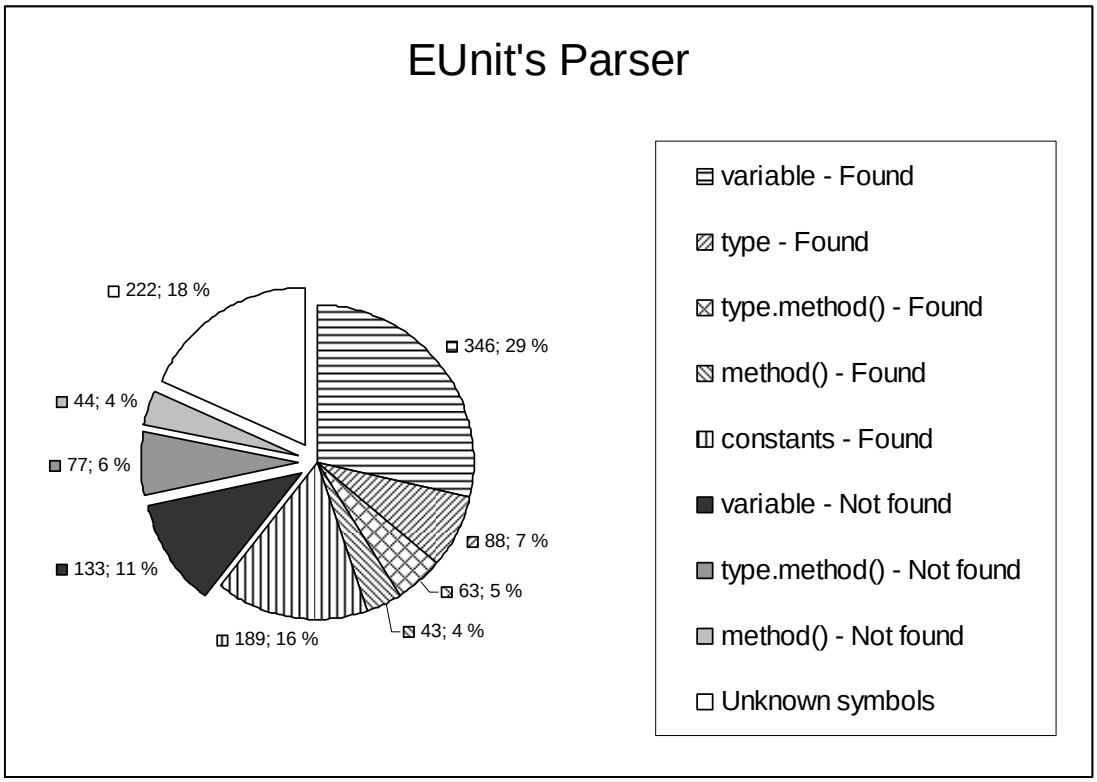

Figure 23 - EUnit's parser analysis for symbol knowledge.

Symbol knowledge increase in CDT comes also with a down side; execution time is increased. Even if we are not counting that CDT does some parsing of source code files before analysis, it still takes lot of time to process symbolically heavy methods. Total execution times with CDT ranged from about 6 seconds to worst case's 5 minutes where

EUnit spent on the average only about 6 seconds in every test. More complete measurements of execution times can be found from Appendix A. Time distributions between different steps in analysis are presented in Figure 24 and Figure 25. Figures show that time is still spent averagely on same steps but hide the fact that with CDT, processing takes more time.

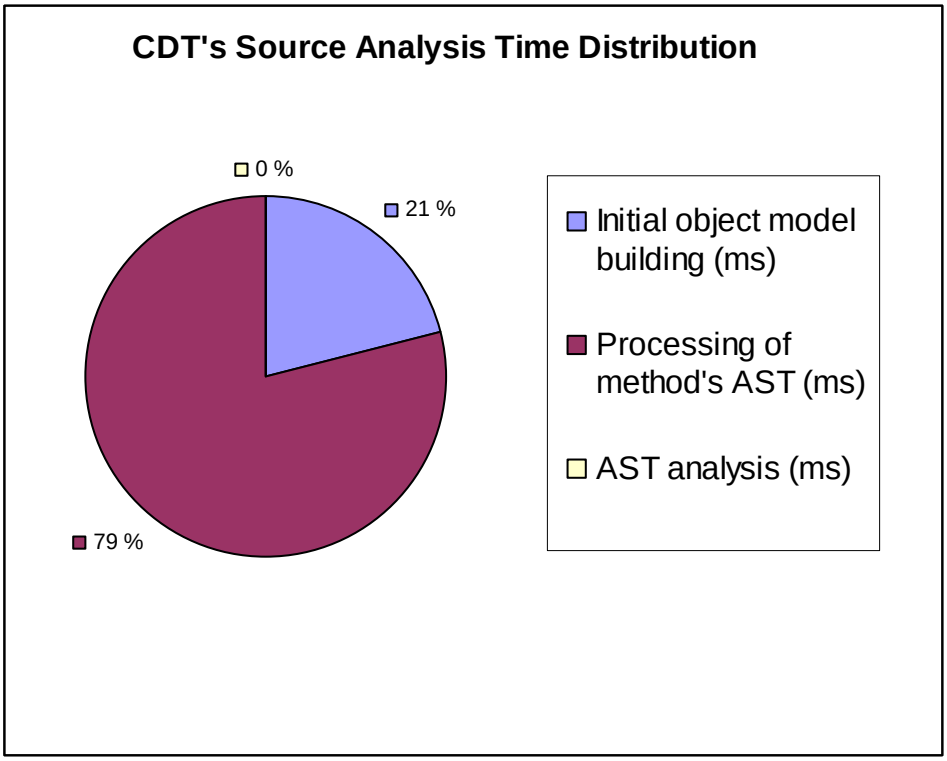

Figure 24 - CDT's source analysis time distribution

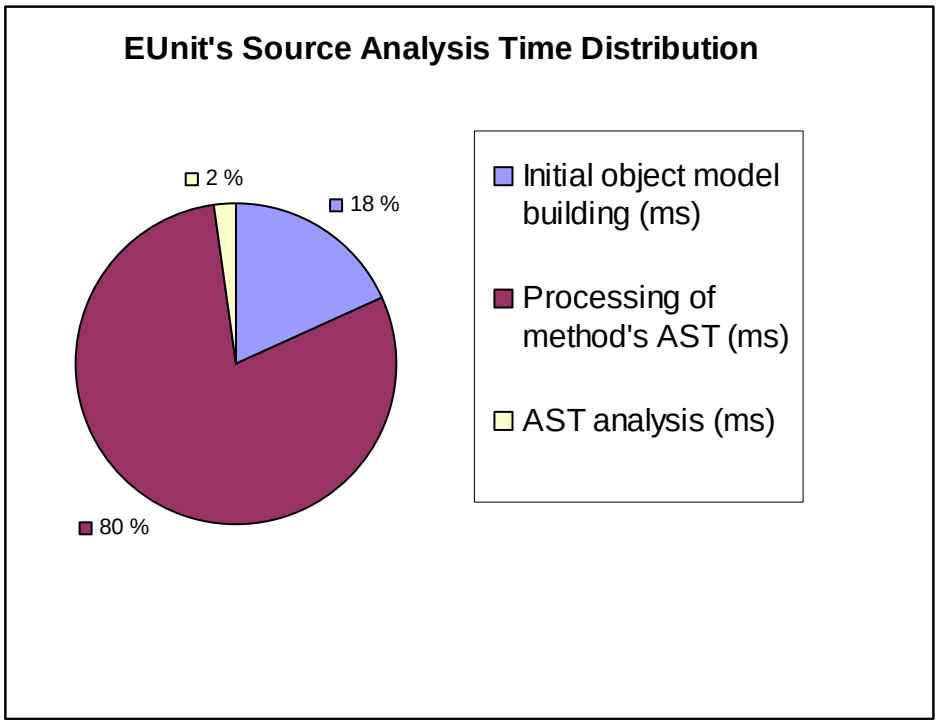

Figure 25 - EUnit's source analysis time distribution

7 Conclusions

In overall results of the work are pleasing. It showed that it is possible to use CDT for analyzing C++ source code. CDT also came with some nice features like PDOM to be used for symbol resolving. Even as there were some problems on getting working plug-ins built and understanding how Eclipse works, it seems to be quite nice platform to work with. During the study more amount of internal documentation of CDT would have been nice. Understanding of CDT's internals mainly came from its source code and unit tests. Other related documentation at time of writing this thesis was from really old version of the CDT and only some clues of its internals could be got from those. This might have caused that something was understood incorrectly and thus affected on results negatively.

7.1 Prototype application

As a working prototype application was completed during the study it would really seem that it is possible to use CDT for automatic test case generation. First version of the prototype application was completed within one month of one mans work, and during analysis part it needed additional week of work to get comparable results. Needed work in process of software adaptation to new platform was surprisingly low. Of course making a release quality application needs even more work and time, but overall impression that was left were that it is possible to do. It helped a lot as automatic test case generation engine in EUnit Professional Edition were quite isolated from other functionality so it was quite easy to get it running on Eclipse environment.

First version of the prototype application included parsing of the complete AST tree got from the CDT. Creating object model by walking the AST tree it was noticed that it was time consuming process. Analysis of the object model creation showed that it contained way too much of unneeded information. Most of the time was spent on allocating memory and setting up objects that were never used on the actual test case data generation. Process of creating object model took over one minute as it were not using benefits of the PDOM. After improving prototype application, object model creation was relaxed a bit and only

the needed objects were created. This improvement caused some drop in execution time. Processing time is still a lot longer when compared against EUnit's handcrafted parser but it also provides more symbolic information. With more information more complete test cases can be generated. Further improvements to execution time could come from ongoing project of adding support for offline indexes to PDOM, and from further improvements to measurement application. With offline indexes complex Symbian OS C++ system headers would be parsed and indexed beforehand. Resulting index can then be stored for later queries. At time of writing this thesis there has not been any end-user comments related to increase in execution time. It cannot be confirmed is this in some cases huge increase in execution time to be acceptable for them.

7.2 Carbide.c++

Carbide.c++ product contained old version of the CDT so it could not be tested how well the prototype application could be integrated to work on it. Integration of automatic test case generation to Carbide.c++ will have to be left for future studies. It would have been quite ideal to have all needed Symbian OS C++ software development components on the same IDE and probably creating of the test cases in this way would be more fitting for the end-users. However it would seem that in future there will be further improvements on Carbide.c++ as both Nokia and Symbian seem to be participating on development meetings and conferences related to CDT. Especially Symbian's work on offline indexes to PDOM could bring needed improvements on execution time when parsing complex Symbian OS system headers. Also Carbide.c++'s IDE could really take an advantage of the newer version of the CDT as some events take annoying amounts of time and during that time whole IDE is practically dead. Otherwise Carbide.c++ seems to be usable for development of Symbian OS C++ code.

7.3 Risks

Porting existing software to a new platform comes always with risks and with Eclipse this is no exception. Eclipse can be completely new style of architecture to learn and work

with. It is highly based on plug-in architecture and this brings some objected oriented designing challenges as code must really be developed to be co-operative with other features on the platform. Learning Eclipse internals and how to use it can be time consuming process and a book can be a good source when initially starting with Eclipse. For some internal components source code is the only source for information and this needs good skills on reading existing code developed by others.

Eclipse's plug-in architecture also comes with a nice additional feature for end users; effect of vendor lock-in issue becomes smaller but at same time it is also challenge for development team. Product has to be more users friendlier and to be better quality than a standalone application. If some competitor develops a better plug-in it is easier for end-users to take that into development process.

It is also possible that Eclipse component that is being used contains a software bug. Fixing these bugs can take some time if there is no in-house expertise related to that component. Distribution of these bug fixes for Eclipses components to end user is not fast process either, fixes must be first accepted to upstream and then new release has to be made.

It should also be noted that Eclipse is most likely based on different user interface concepts than what ported application were. This requires additional time spent on redesigning use cases and then using these new use cases to think how porting should be done. There are also some guidelines written for Eclipse on how to write code for different parts and these should be followed. If user interface works in completely different way than other Eclipse components or processing of something stalls whole Eclipse platform, it can be seen negatively by the end-users.

7.4 Future

As subject of one possible future study it would be interesting to know how work needed in automatic test case generation could be done as a background process while user is coding. In this way user could be displayed information in real time about impossible paths

and ranges for variables at certain point of the path. When new version of Carbide.c++ is published it would be good candidate for further studying how well automatic test case generation can be integrated to it, and does its Symbian OS specific additions give any improvements to results.

REFERENCES

1 Wiston. W. Royce. 1970. Managing the development of large software systems: concepts and techniques. Proceedings of IEEE WESCON, August 1970, pages 1-9. TRW.

2 Hong Zhu, Patrick A. V. Hall, John H. R. May. 1997. Software unit test coverage and adequacy. ACM Computing Surveys (CSUR), Volume 29 Issue 4, pages 366-427. ACM Press

3 Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. 1986. Compilers: Principles, Techniques and Tools. Addison-Wesley Longman Publishing Co., Inc., Boston, MA. ISBN 0-201-10194-7

4 G. Antoniol, M. Di Penta, G. Masone, U. Villano. 2004. Compiler Hacking for Source Code Analysis. Software Quality Control, Volume 12 Issue 4, pages 383-406. Kluwer Academic Publishers

5 M.E.M. Stewart. 2005. Towards a Tool for Rigorous, Automated Code Comprehension Using Symbolic Execution and Semantic Analysis. Proceedings of the 2005 29th Annual IEEE/NASA Software Engineering Workshop (SEW'05), pages 89-96.

6 Bogdan Korel. 1990. Automated software test data generation. IEEE Transactions on Software Engineering, Volume 16 Number 8, pages 870-879.

7 Neelam Gupta, Aditya P. Mathur, Mary Lou Soffa. 1998. Automated Test Data Generation Using An Iterative Relaxation Method. ACM SIGSOFT Software Engineering Notes, Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering SIGSOFT '98/FSE-6, Volume 23 Issue 6, pages 231-244. ACM Press.

8 A. Jefferson Offutt, Zhenyi Jin, Jie Pan. 1999. The Dynamic Domain Reduction Procedure for Test Data Generation. Software - Practice & Experience, Volume 29, Issue 2, pages 167-193. ISSN 0038-0644.

9 David Geer. 2005. Eclipse Becomes the Dominant Java IDE. Computer, July 2005 Volume 38 Number 7, pages 16-18. IEEE Computer Society.

10 Eclipse Foundation's Legal Resources. http://www.eclipse.org/legal/. 26.10.2006.

11 International Business Machines Corp. 2006. Eclipse Platform Technical Overview. http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf. 26.10.2006.

12 Richard Goering. 2006. EETimes - Objectives set for Eclipse C/C++ tools, EE Times, October 11, 2006, CMP United Business Media. http://www.eetimes.com/news/design/showArticle.jhtml?articleID=193200426. 26.10.2006.

13 PDOM Overview. Eclipsepedia. http://wiki.eclipse.org/index.php/CDT/designs/PDOM/Overview. 26.10.2006.

14 Programming languages - C++. ISO/IEC 14882:2003.

15 Forum Nokia - resources for mobile application developers, http://www.forum.nokia.com/. 26.10.2006.

16 Panos Asproulis. 2006. Introduction to Carbide.c++. Symbian Ltd. http://developer.symbian.com/main/downloads/papers/Carbide/Introduction_to_Carbide.pdf

17 Forum Nokia. 2006. Carbide.c++: Introductory White Paper. Nokia Corporation.

18 SysOpen Digia. QualityKit 4.0 Procuct Brochure. http://www.sysopendigia.com/C2256FEF0043E9C1/07D5035B5E00886CC22571250045682B/$file/SysOpen_Digia_QualityKit.pdf. 26.10.2006.

19 Forum Nokia. 2004. Symbian OS: Coding Conventions In C++ v1.0. Nokia Corporation.

20 Forum Nokia. 2005. S60 Platform: C++ Coding Conventions v2.0. Nokia Corporation.

21 ANTLR Parser Generator. http://www.antlr.org/. 26.10.2006.

CDT's Full C/C++ Indexer

	Total objects in model	Total symbols	variable - Found	type - Found	type.method() - Found	method() - Found	constant - Found	variable - Not found	type.method() - Not found	method() - Not found	Unknown symbol	Initial object model building (ms)	Processing of method's AST (ms)	AST analysis (ms)
CAddressBookAppUi::HandleCommandL	1460	134	50	32	26	9	11	0	0	1	5	37548	193239	78
CAddressBookContact::SelectedPhoneL	337	30	15	8	4	0	0	0	0	0	3	2500	3969	172
CAddressBookEngine::GenerateQueryRequestL	464	170	62	46	24	3	5	2	0	12	16	7219	30454	46
CAddressBookView::Draw	512	66	26	13	10	1	0	0	1	6	9	10751	11156	47
CChatAppUi::DynInitMenuPaneL	796	269	64	63	63	0	78	0	0	0	1	18766	92097	15
CChatBt::RunL	347	216	65	21	17	23	15	5	19	17	34	3000	6298	47
CChatContainer::ConstructL	437	38	14	9	8	3	2	0	1	0	1	11579	19500	16
CChatInet::StartL	510	73	22	17	15	3	3	0	3	1	9	2859	10438	47
CHWRMTestAppAppUi::LightsReserveL	992	72	32	22	12	1	0	0	0	0	5	35125	292289	16
CIsvTelInfoAppDlg::ExecuteLD	677	14	5	3	2	1	1	0	0	1	1	37720	7016	0
CMultiTexContainer::ConstructL	461	110	51	26	5	5	2	0	0	18	3	7875	7172	94
CMultiTexDocument::NewL	108	11	4	4	3	0	0	0	0	0	0	4093	2063	0
CRegistrationListbox::OfferKeyEventL	412	8	4	1	1	0	1	0	0	0	1	10938	391	141
CRegistrationView::ValidateInput	228	25	12	5	1	0	3	2	0	0	2	4875	5687	31
CWebClientAppUi::HandleUrlRequestL	1068	71	24	18	15	0	1	0	0	0	13	26578	121379	15
CWebClientContainer::ComponentControl	826	3	1	0	0	0	1	0	0	1	0	10235	65971	47
		1310	**451**	**288**	**206**	**49**	**123**	**9**	**24**	**57**	**103**	**231661**	**869119**	**812 ms**
			34 %	22 %	16 %	4 %	9 %	1 %	2 %	4 %	8 % % of total	231,66	869,12	0,81 s

CDT's Fast C/C++ Indexer

CAddressBookAppUi::HandleCommandL	1460	134	50	32	26	9	11	0	0	1	5	42704	216599	78
CAddressBookContact::SelectedPhoneL	337	30	15	8	4	0	0	0	0	0	3	2781	4016	125
CAddressBookEngine::GenerateQueryRequestL	464	170	62	46	24	3	5	2	0	12	16	8125	32907	47
CAddressBookView::Draw	512	66	26	13	10	1	0	0	1	6	9	12188	12422	16
CChatAppUi::DynInitMenuPaneL	796	269	64	63	63	0	78	0	0	0	1	20907	99721	47
CChatBt::RunL	347	216	65	21	17	23	15	5	19	17	34	3500	6641	125
CChatContainer::ConstructL	437	38	14	9	8	3	2	0	1	0	1	13047	21532	15
CChatInet::StartL	510	73	22	17	15	3	3	0	3	1	9	3468	11282	78
CHWRMTestAppAppUi::LightsReserveL	992	72	32	22	12	1	0	0	0	0	5	35001	295992	15
CIsvTelInfoAppDlg::ExecuteLD	677	14	5	3	2	1	1	0	0	1	1	40079	6484	0
CMultiTexContainer::ConstructL	461	110	51	26	5	5	2	0	0	18	3	8360	7281	110
CMultiTexDocument::NewL	108	11	4	4	3	0	0	0	0	0	0	4062	2047	16
CRegistrationListbox::OfferKeyEventL	412	8	4	1	1	0	1	0	0	0	1	11469	375	109
CRegistrationView::ValidateInput	228	25	12	5	1	0	3	2	0	0	2	5407	5766	15
CWebClientAppUi::HandleUrlRequestL	1068	71	24	18	15	0	1	0	0	0	13	27938	125191	15
CWebClientContainer::ComponentControl	826	3	1	0	0	0	1	0	0	1	0	10594	64861	47
		1310	**451**	**288**	**206**	**49**	**123**	**9**	**24**	**57**	**103**	**249630**	**913117**	**858 ms**
			34 %	22 %	16 %	4 %	9 %	1 %	2 %	4 %	8 % % of total	249,63	913,12	0,86 s

EUnit's Parser

CAddressBookAppUi::HandleCommandL	196	139	29	18	17	9	17	5	4	1	39	1109	5235	141
CAddressBookContact::SelectedPhoneL	196	26	9	1	0	0	1	4	0	2	9	1078	5188	110
CAddressBookEngine::GenerateQueryRequestL	196	111	18	4	0	1	3	31	3	8	43	1047	5204	172
CAddressBookView::Draw	196	46	20	2	0	0	0	10	1	6	7	1016	5156	110
CChatAppUi::DynInitMenuPaneL	295	226	64	20	20	0	78	0	43	0	1	1891	5234	156
CChatBt::RunL	295	240	66	20	17	26	41	34	3	5	28	1625	5406	265
CChatContainer::ConstructL	295	36	12	1	0	0	2	2	7	5	7	1594	5062	125
CChatInet::StartL	295	57	22	1	0	3	6	7	11	1	6	1578	5297	125
CHWRMTestAppAppUi::LightsReserveL	84	67	24	8	4	1	5	6	2	0	17	844	5110	125
CIsvTelInfoAppDlg::ExecuteLD	226	12	5	1	0	1	2	2	0	1	0	1562	5016	109
CMultiTexContainer::ConstructL	75	140	44	5	1	1	18	14	1	15	41	1172	5250	171
CMultiTexDocument::NewL	75	9	4	2	1	0	0	2	0	0	0	1063	5016	94
CRegistrationListbox::OfferKeyEventL	71	10	4	0	0	0	4	1	0	0	1	797	5079	125
CRegistrationView::ValidateInput	71	30	12	2	0	0	6	5	0	0	5	656	5156	125
CWebClientAppUi::HandleUrlRequestL	107	53	12	3	3	0	5	10	2	0	18	953	5141	140
CWebClientContainer::ComponentControl	107	3	1	0	0	1	1	0	0	0	0	875	4985	94
		1205	**346**	**88**	**63**	**43**	**189**	**133**	**77**	**44**	**222**	**18860**	**82535**	**2187 ms**
			29 %	7 %	5 %	4 %	16 %	11 %	6 %	4 %	18 % % of total	18,86	82,54	2,19 s

Following projects/methods were used from Nokia's Series 60 3rd Edition SDK:

AddressBook
- CAddressBookAppUi::HandleCommandL
- CAddressBookContact::SelectedPhoneL
- CAddressBookEngine::GenerateQueryRequestL
- CAddressBookView::Draw

Chat
- CChatAppUi::DynInitMenuPaneL
- CChatBt::RunL
- CChatContainer::ConstructL
- CChatInet::StartL

hwrmtestapp
- CHWRMTestAppAppUi::LightsReserveL

IsvTelInfoApp
- CIsvTelInfoAppDlg::ExecuteLD

OpenGLEx/MultiTex
- CMultiTexContainer::ConstructL
- CMultiTexDocument::NewL

Registration
- CRegistrationListbox::OfferKeyEventL
- CRegistrationView::ValidateInput

WebClient
- CWebClientAppUi::HandleUrlRequestL
- CWebClientContainer::ComponentControl