Parsing and Modeling C# Systems

ERMIRA DAKA
Master Thesis - submitted to the
University of Lugano
Faculty of Informatics

Under the supervision of
Prof. Michele Lanza

May 2009

The author declares that the text and work presented in this Master Thesis are original and that no sources other than those mentioned in the text and its references have been used in creating the Master thesis.

Ermira Daka
Lugano, May 2009

Acknowledgment

Coming from a small country that was celebrating its 8th year of freedom to continue my education with master studies in the University of Lugano, was not easy. Being away from the family and taking up on challenges and facing them on my own was hard but also a learning experience. I met great people during my two year studying and living in Switzerland and made very good friends, which I am grateful for. The whole experience taught me a lot of things and, while on one hand I am happy that this is my final step towards my Masters Degree and I will be heading home, on the other hand, I will miss everything about this place and all the people and friends I met here.

I would like to take this opportunity thank the people who made it possible for me to be here. First of all, I would like to thank Prof. Michele Lanza for all the help that he provided me throughout this time, for his advice and feedback. Special thanks go out to the dean of University, Prof. Dr. Mehdi Jazayeri, who has been a great support for me since the very beginning of my studies.

I would like to thanks my parents, sister and brother for their support and love, and having to put up with me during these two years.

Ermira Daka
Lugano, May 2009

Contents

List of Tables

List of Figures

Abstract

Software and its usage is growing everyday, and we can say that it is everywhere. During the maintenance phase, software needs updates and modifications, which we call changes in the software. Software may need updates because of hardware changes too, existing software system usually may not be compatible with new computers. These are the typical reasons when reverse engineering becomes an important field of the software development industry.

The main source of information for understanding existing software is its source code. Reverse engineering tools take the source code of a software system and produce the basic blocks of its general structure.

In this thesis we created the Parsing and Modeling C# Systems (PMCS) tool that parses the C# source code, models its entities based on the FAMIX meta-model structure, and exports these entities in the MSE file. MSE is a file format, which can be imported in the MOOSE reengineering platform and visualized by reverse engineering tools. PMSC is an reverse engineering tool which is done with C# .

Chapter 1

Introduction

Software is subjected to changes throughout its lifetime. New requirements, like functionality changes and new components usually drive software changes.

A legacy system is an old application program that continues to be used, typically because it still functions for the users' needs, even though newer technology is available. [1] A legacy system needs to be migrated with its current functionality to a new (hardware or software) environment or its current functionalities need to be changed. To modify these systems is difficult and costly. On the other side, there are many built-in business processes that have evolved over time. These processes have to be inherited during system migration.

The goal of reverse engineering is to understand existing software systems and analyze them. Through activities like reading the code, talking with developers or skimming the documentation, we gain knowledge about the system [2]. Often, the developers of the legacy systems are not available to verify or explain them. System documentation usually is old, unstructured or no longer available, so the main information or resource about a system is its source code.

To start the modification of a legacy system, the developer has to make some effort in understanding the structure and architecture of that system, and then change it.

Nowadays, there are automated tools which read and extract the structure of the system. The aim of a reverse engineering tool is to understand a software system and visualize its structure. These tools usually provide a set of functions for software developers to compose different views of the software depending on the comprehension task at hand.

1.1 Software Reverse Engineering

Within the overall software evolution cycle, changes in software are very important. After some changes a system can lose its original design, and its initial structure. Because there does not exist perfect software system developers try to build systems that keep up with the changing environment.

Useful systems tend to be very large and complex, and they may have gone through changes many times. Software change types were defined by Lientz and Swanson (1980)[14],

[1] *http : //en.wikipedia.org/wiki/Legacy − system*

which then were updated and normalized internationally in the ISO/IEC 14764 standard. Software changes are categorized as:

- *Adaptive changes* - these are changes done because of software integration in its new environment, and additional functions in the system,
- *Corrective changes* -these are changes done because of the need of fixing bugs in the delivered system,
- *Perfective changes* - these are changes done in the system because of the reason of making the system more maintainable, and
- *Preventive changes* - these are changes done in a system after its delivery to detect and correct latent faults in the software product before they become effective faults.

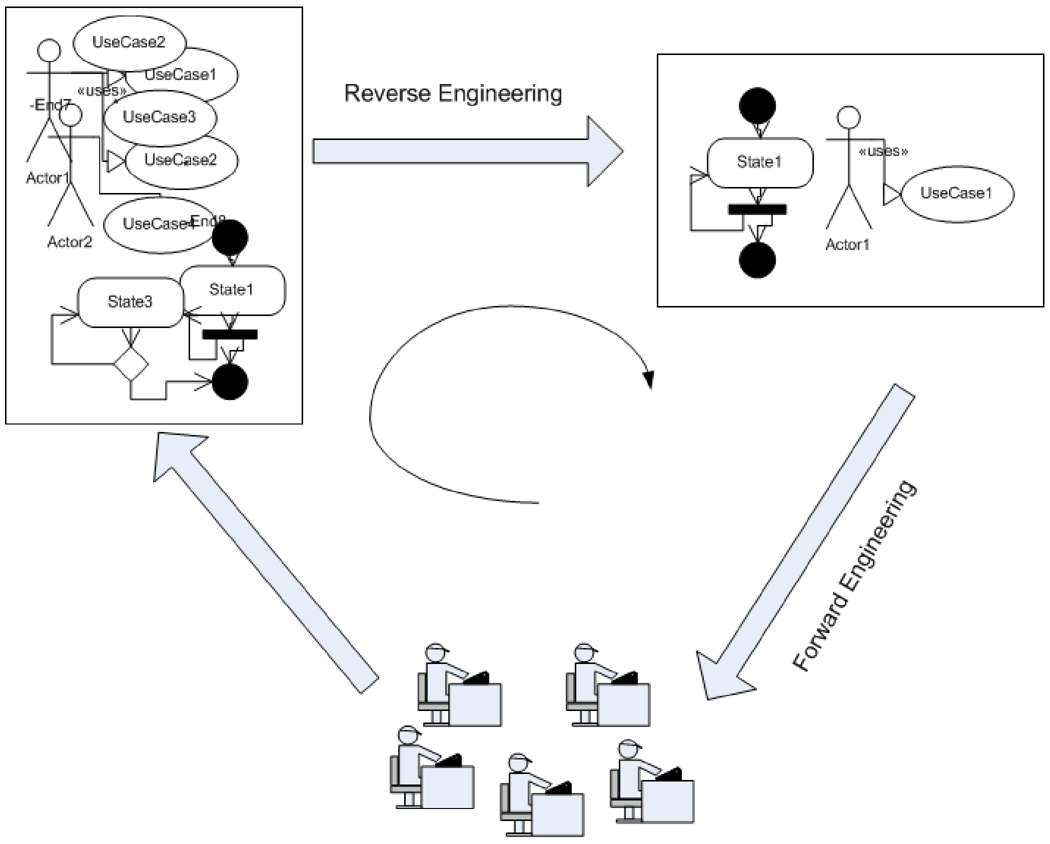


Figure 1.1: *The software evolution life-cycle, from the initial stage to the stage where a system becomes complex and unclear*

As you can see in the Figure 1.1, software evolution starts from some basic blocks, later people start to use it, and after a time it needs to be redesigned (reengineered) and then delivered back to its user. This defines a software systems' life cycle, and it continues like this until the system is not used anymore.

There are many reverse engineering tools, which help in the extraction of a systems' basic blocks, and understanding its structure.

1.2 Software Reverse Engineering Tools

Reverse engineering can be seen as part of Reengineering. Reengineering means redesigning and re-implementing an existing legacy system. [1] It is defined around a life cycle model that contains these steps:

- *Requirements Analysis* - identifying the concrete reengineering goals.
- *Model Capture* - documenting and understanding the software system.
- *Problem Detection* - identifying flexibility and quality problems.
- *Problem Resolution* - selecting new software architectures to correct the problems.
- *Reorganization* - transforming the existing software architecture for a new release.
- *Change Propagation* - ensuring that all client systems benefit from the new release.

Model Capture of large software systems can be very hard without appropriate tools, especially when a system is large and complex, and a developer does not have any other information other than the systems' source code. Good reverse engineering tools are those that support different tasks during this process, and enable analyzing and understanding the current system.

There are methods and automated (reverse engineering) tools that are applicable for different programming languages, from which we can mention: CodeCity [18], Softwarenaut [15], Mondrian [16].

Software systems can be developed in different programming languages. Each of these languages has it own structure. Object-oriented programming languages, in general, are built upon the same concepts, even if in detail they differ from each other.

Around all these programming languages and reverse engineering tool there exists a FAMIX [9] meta-model, which serves as an exchange information format.

FAMIX is extensible, which means that it does not depend on a specific programming language, and objects and properties of FAMIX may contain additional information or details.

Tools that use FAMIX meta-model saved in a specific format [5] are done on top of the MOOSE reengineering environment. Up to now, there is no parsing tool for C# systems. C# is one of the modern programming languages that evolved from C++.

1.3 Goals and Objectives

Goals and objectives of this project are:

- Developing a tool that will parse and model C# systems.
- Exporting models into *.mse* files complying with the FAMIX meta-model.

  FAMIX is based on the 4 levels of extraction listed below, and the objective of this thesis is to extract completely the first three levels.

  - Level 1 - classes, inheritance definitions, packages, methods.

  - Level 2 - level 1 entities, attributes, global variables.
  - Level 3 - level 2 entities, attributes accesses, and method invocations.
  - Level 4 - level 3 entities, local variables, implicit variables, and formal parameters.

1.4 Structure of the document

The rest of this document is organized as follows:

- *Chapter 2* describes the domain of the reverse engineering, and discusses related issues.
- *Chapter 3* describes our approach to parse and model C# systems, and also illustrates our tool implementation.
- *Chapter 4* summarizes the tests done with PMCS tool. The source code of several C# software systems are parsed and modeled. To give a validation to our tool, the parsing and modeling results are imported in MOOSE and visualized with CodeCity.
- *Chapter 5* concludes the thesis and discusses future work.

Chapter 2

Problem Definition

Reverse engineering is becoming more and more important as software system usage is growing. There are many reverse engineering tools that help on information extraction from a system in use.

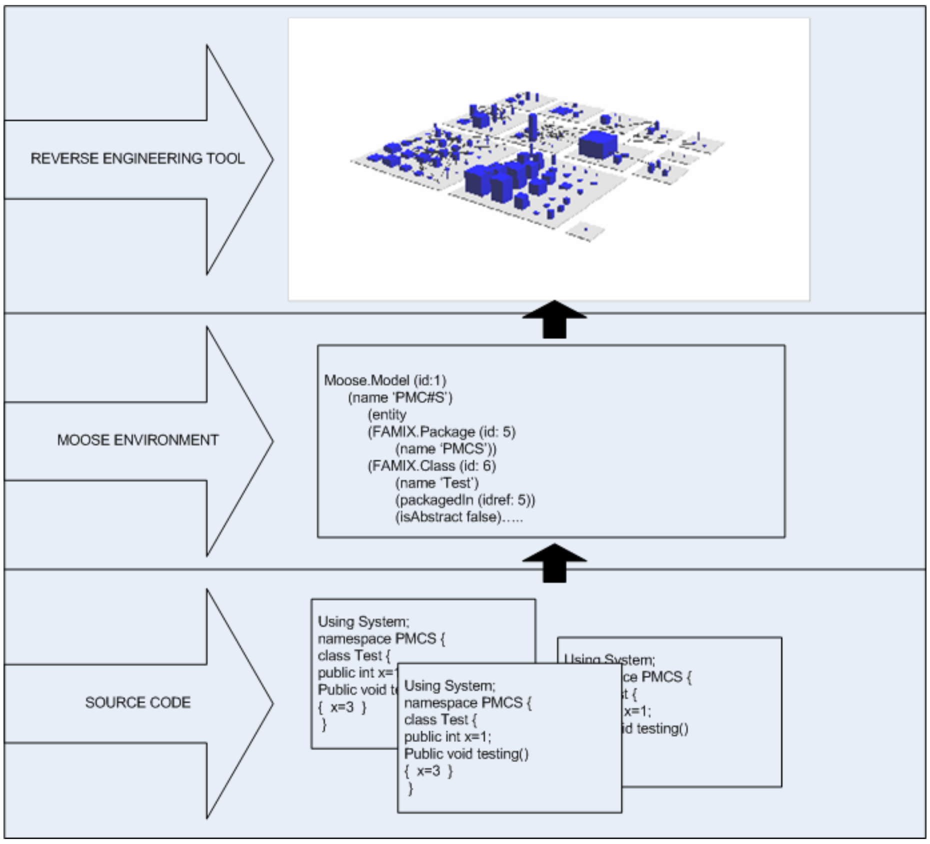
The current chapter describes the MOOSE analysis environment, the FAMIX meta-model, the MSE interchange file format and the C# programming language organization, in order to understand key points and problems that can be when building a tool for parsing and modeling C# systems based on the FAMIX meta-model structure. In the last part of this chapter there is a short section describing parsing techniques in general. Since parsing is a field in itself, the usage of parsing techniques and how we applied them in our tool is described in the next Chapter where we wrote in details about the solution that we applied to the problem.

2.1 System Modeling

Today software systems can be implemented in different programming languages. To avoid equipping reverse engineering tools with parsing technology for all existing programming languages, FAMIX is used as a common information exchange model.

Based on the content of the system there are created FAMIX entities (see Figure 2.3) and put in one FAMIX meta-model, which then is saved as an *.mse* file. FAMIX models are uniformly represented by MOOSE [1] as a collaborative platform for software analysis and information visualization.

[1]http://moose.unibe.ch



```
Moose.Model (id:1)
     (name 'PMC#S')
          (entity
          (FAMIX.Package (id: 5)
               (name 'PMCS'))
          (FAMIX.Class (id: 6)
               (name 'Test')
               (packagedIn (idref: 5))
               (isAbstract false).....
```

```
Using System;
namespace PMCS {
class Test {
public int x=1;
Public void testing()
{ x=3 }
}
```

Figure 2.1: *The MOOSE environment between a system source code and a reverse engineering tool*

2.1.1 What is MOOSE?

MOOSE is an Extensible Language-Independent Environment for Reengineering Object-Oriented Systems [7] developed at the University of Berne. As you can see from the Figure 2.1, MOOSE stands between reverse engineering tools and different programming languages such as Java, C++, Ada, Smalltalk, C# (in which we are interested in). In the other side, it provides infrastructure for tool integration, and since it is extensible and scalable reengineering environment, it makes different tools able to collaborate with each other.

To better understand the MOOSE reengineering environment, we can say that MOOSE is:

- An analysis platform, which covers:
  - complexity of the system, or class hierarchies (System Complexity [13])
  - classes and their internal complexity, (Class Blueprint [6])
  - package usage, (Package Blueprint [8])
  - how the system is distributed in an abstract space, (Software Map [11]).

- A modeling platform that works with:

    - FAMIX meta-model that shows system entities (namespace, classes, attributes, methods, inheritance definition, invocation, accesses)
    - MSE files that are readable by all MOOSE supportable environments. [12]

- A visualization platform, and some of tools running on the MOOSE are:
    - Mondrian that shows all the classes in the system, methods, and attributes in the class and every relationship that can be between these entities, [16]
    - CodeCity that shows every entity in the system, in the form of city with buildings on it, [18]
- A tool building platform,
    - there are many tools build on top of MOOSE (Chronia, CodeCity, DynaMoose, Softwarenaut,..) and many other that will be built,
- A collaboration
    - MOOSE is an extensible platform very flexible for research, for this reason it is used in many Universities around the world.

2.1.2 FAMIX *-language independent and extensible meta-model*

FAMIX stands for FAMOOS Information Exchange Model, which was created to support information exchange between reverse engineering tools and software systems. It works with different object-oriented programming languages like: C++, JAVA, SMALLTALK, and ADA.

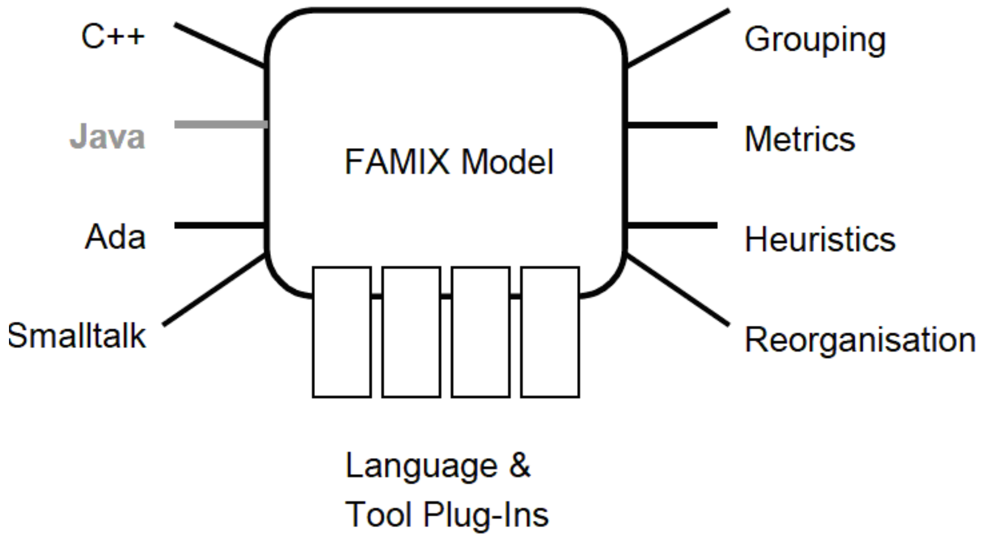


Figure 2.2: *The FAMIXs' basic concept [4]*

In the Figure 2.2 is given the basic concept of the FAMIX model. On the left side of the figure there are different programming languages, which are used to implement software applications, on the right side there are various results that user can get from different software analysis tools. Between the right and the left sides there is the FAMIX information exchange meta-model, which takes as input entities of an object-oriented system, and structures them in a format that is standard and understandable by all MOOSE based reverse engineering tools.

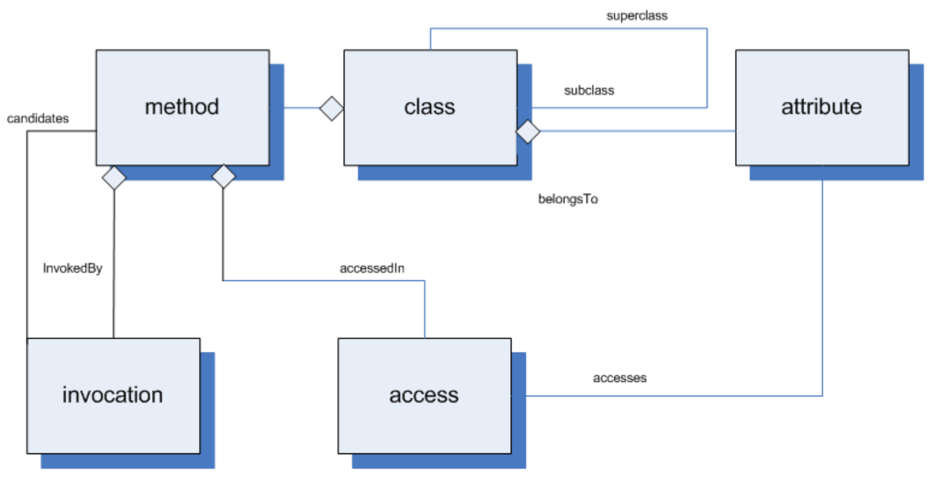


Figure 2.3: *The FAMIX core model [9]*

The FAMIX core model as in the Figure 2.3, describes software systems main entities and possible relations that can be between these entities. Classes, attributes, methods, inheritance definitions, invocations, and accesses, are the main entities of a software system. The last three entities define the fact of the relations between classes (inheritance definition), method invocation by another method (invocations), and method accesses to attributes (accesses), which as a group play an important role for reverse engineering tools, and provide to them special information in terms of better system description.

In the Figure 2.4, there is given the whole FAMIX class diagram, where are described all FAMIX entities and their attributes (details). Everything is organized around a root model (abstract object), which is very flexible to be extended with the new defined objects and properties.

LEVELS	**ENTITIES**
Level 1	Class, InheritanceDefinition, BehaviouralEntity, Packages
Level 2	Level 1, Attributes, GlobalVariables
Level 3	Level 2, Access, Invocation
Level 4	Level 3, FormalParameter, LocalVariable, ImplictVariable

Table 2.1: FAMIX Levels of Extraction

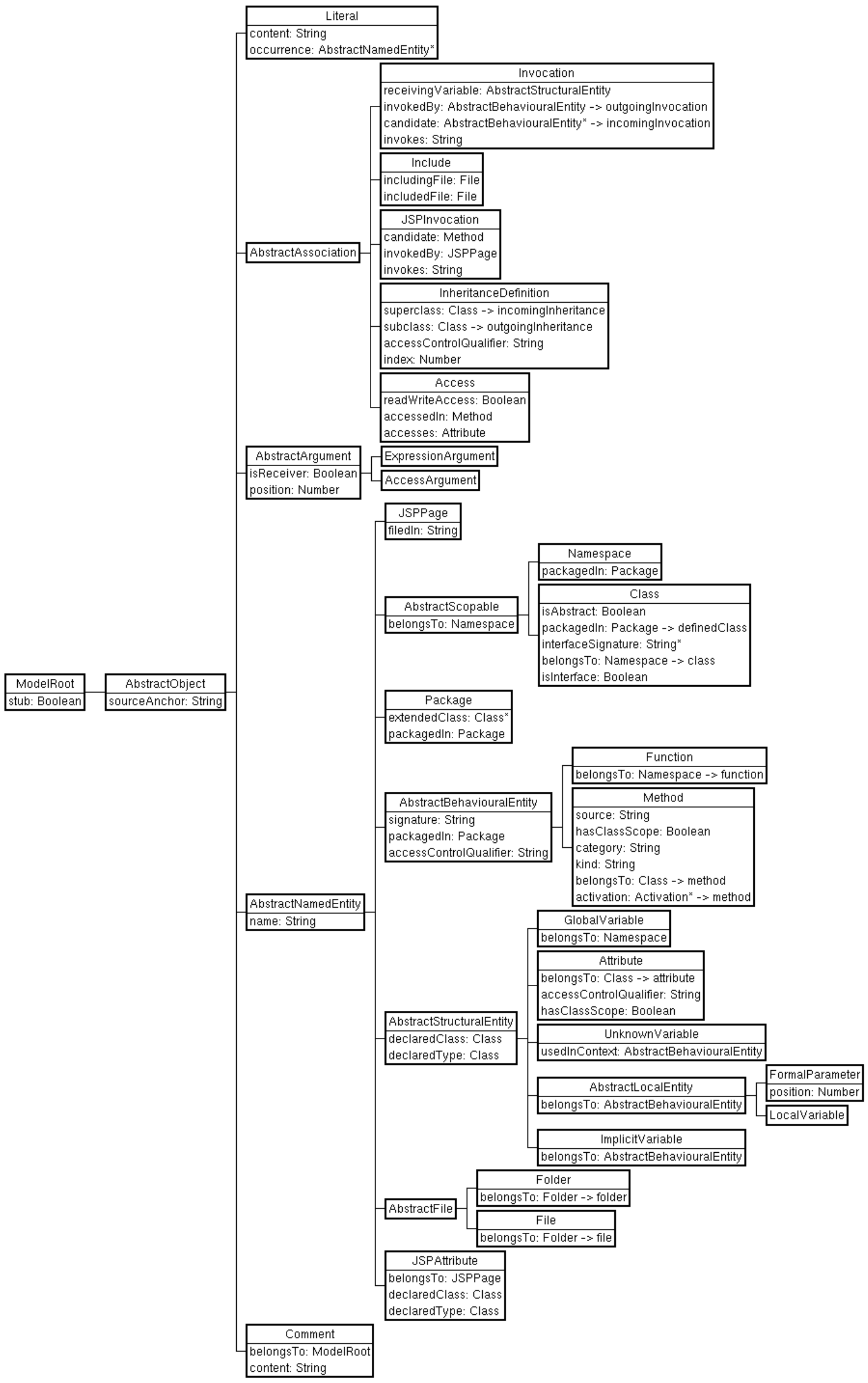


Figure 2.4: *Famix Class Diagram* [2]

Not all the existing parsers will provide all the entities of the FAMIX core model, neither all reverse engineering tools need all of them. For this reason in the FAMIX there are defined *four levels* with different model entities, which are written in the Table 2.1. The higher the level of extraction the more entities are extracted from a software system. FAMIX *levels of extraction* are defined with an order, where no entity of an lower level needs information from an entity of a higher level .

The FAMIX [3] meta-model is language independent because it needs to work with legacy systems in different implementation languages (C++, JAVA, SMALLTALK, ADA and C#), and it is extensible because not all information are known in advance that are needed in the future tools.

2.1.3 Why FAMIX?

UML is a well-known modeling language that is being used in almost every application domain. In the software industry UML seems to be very dominant, and it is applied as a standard modeling language.

In the other side, because of the reengineering needs in a software system, automated tools are becoming more and more important every day. One automated tool cannot cover all the development life cycle of one application, and using different tools need something that is familiar for all of them which they can use and share.

UML is not sufficient to serve as tool interoperability standard for integrating round-trip engineering tools, because one is forced to rely on UMLs built-in extension mechanisms to adequately model the reality in source-code. [3]

FAMIX is an alternative meta-model, which serves as interchange standard between different tools, by supporting the whole round-trip engineering (reengineering) tasks.

2.1.4 MSE

It does not matter in which programming language software is written, the FAMIX meta-model contains entities of the systems (namespaces, classes, methods, fields, invocations, accesses and class inheritances), and MOOSE environment allows system analysis in a uniform way.

MOOSE environment acts as a repository for many reverse engineering tools. Tools in this environment for analyzing a model need a specific information exchange format. There were used different exchange formats during the evolution of the MOOSE reengineering platform. First used information exchange format was CDIF, which is industry standard but since it is not developed any more MOOSE started to use XMI. XMI (XML Metadata Interchange) information exchange format is model driven standard that is based on Meta Object Facility (MOF).

Next interchange file format which MOOSE environment supports and is still using is MSE.

The MSE file in order to be used by reverse engineering tools has to: (1) contain entity level system model, and (2) all these models have to be written with a right format and a specified standard.

[3]http://scg.unibe.ch/archive/famoos/FAMIX/

The MSE format is organized in terms of nodes and commands. In the Table 2.2 [4] there are described all the details about nodes and commands that an MSE file must support. The MSE file *nodes* are: entities (element node) of a system and all entity properties (attribute and value nodes), and *commands* are: node properties that uniquely identify entities and are useful when we need to connect to them.

NODES and COMMANDS	**DESCRIPTION**
Element node	defines an element in the model by: 1) the name of its type, 2) an id, 3) and its attribute nodes.
Attribute node	defines an attribute of an element in the model by: 1) the name of attribute, 2) and its value nodes.
Value node	defines the value of an attribute of an element in the model. An value node is either a primitive value or any of the commands that returns an element or value.
ID command	assign an identifier to an element with 1) the command "id:", 2)and an identifier.
REF command	returns the element identified by an identifier with 1) the command "idref:", 2) and an identifier.
REF command (meta-models only)	returns the element identified by its unique name with 1) the command "ref:", 2) and a name.

Table 2.2: The MSE file - Nodes and Commands

In the Figure 2.5 we can see how a C# class is represented in an MSE file. Every entity and relation that can be between entities in the class are given as separated nodes, such as:

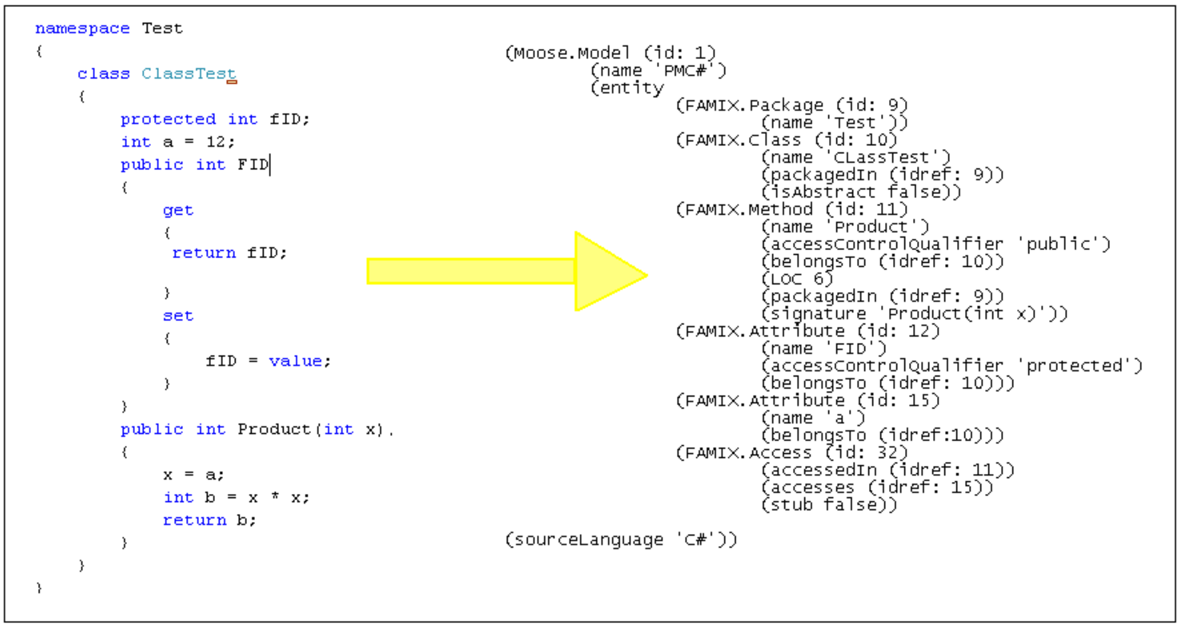
- Element nodes are:
  - *Package, Class, Method, Attribute and Access.*

- Attribute nodes are:
  - *id, names, packagedIn, isAbstract, accessControlQualifier, belongsTo, LOC, signature and stub.*

- Value nodes are values after all these attributes.

[4] http://scg.unibe.ch/wiki/projects/fame/

Since, MSE commands identify every *element node* with a unique identifier, you can see how in the Access-*element node* command *idref* is used to return a method by using its *id* - attribute.



```
namespace Test
{
    class ClassTest
    {
        protected int fID;
        int a = 12;
        public int FID
        {
            get
            {
             return fID;

            }
            set
            {
                fID = value;
            }
        }
        public int Product(int x)
        {
            x = a;
            int b = x * x;
            return b;
        }
    }
}
```

```
(Moose.Model (id: 1)
        (name 'PMC#')
        (entity
                (FAMIX.Package (id: 9)
                        (name 'Test'))
                (FAMIX.Class (id: 10)
                        (name 'ClassTest')
                        (packagedIn (idref: 9))
                        (isAbstract false))
                (FAMIX.Method (id: 11)
                        (name 'Product')
                        (accessControlQualifier 'public')
                        (belongsTo (idref: 10))
                        (LOC 6)
                        (packagedIn (idref: 9))
                        (signature 'Product(int x)'))
                (FAMIX.Attribute (id: 12)
                        (name 'FID')
                        (accessControlQualifier 'protected')
                        (belongsTo (idref: 10)))
                (FAMIX.Attribute (id: 15)
                        (name 'a')
                        (belongsTo (idref:10)))
                (FAMIX.Access (id: 32)
                        (accessedIn (idref: 11))
                        (accesses (idref: 15))
                        (stub false))
(sourceLanguage 'C#'))
```

Figure 2.5: *The MSE file format - C# class in an .mse file after PMCS tool parses and exports it*

2.2 C# Programming Language

C# (pronounced C sharp) is a new programming language designed for building a wide range of enterprise applications which runs on the .NET Framework. It evolved from Microsoft C [5]and Microsoft C++ and is a simple, modern, type safe, and object oriented programming language. C# code is compiled as managed code, which means that it benefits from the services of the common language runtime. These services include language interoperability, garbage collection, enhanced security, and improved versioning support.

C# sources are stored in the files with *.cs* extension. There are also compiled components that are stored in DLL (binary source code) files. The scope of this thesis is parsing C# source files stored with *.cs* extension.

C# programming language provides a unified type system. The main type is object and every other type is derived from the type object. Types in C# are divided in two groups:

- Value Types - all primitive types (int, float, string,...),
- Reference Types - classes, delegates, interfaces.

[5]http://msdn.microsoft.com/en-us/library/aa287558(VS.71).aspx

Value types differ from reference types, and this is because value types directly contain their data, whereas variables of the reference types store references to the object. With reference types, it is possible for two variables to reference to the same object, and is possible for operations of one variable to affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it is not possible for operations of one variable to affect the operation of the other.

Every programming language has its own grammatical structure, C# too, known as syntactic grammar and lexical grammar. Every source file in a C# program [6] shall conform to the input production of its lexical grammar.

Conform implementations it shall accept Unicode source files encoded with the UTF-8 encoding form (as defined by the Unicode standard), and transform them into a sequence of Unicode characters. Implementations can choose to accept and transform additional character encoding schemes (such as UTF-16, UTF-32, or non-Unicode character mappings).

2.2.1 C# Source File Organization

C# programming language is highly expressive, with less than 90 keywords. It is simple and easy to learn. Everyone familiar with C, C++ or Java will easily recognize the syntax of C#. There are no separate header files and no requirements where methods and types can be declared in a particular order. A C# [7] source file is a file with *.cs* extension and in this file may be defined any number of classes, structs, interfaces, and events. C# facilitates the development of software components through several innovative language constructs, including:

- *Delegates* - encapsulated method signatures called delegates, which enable type-safe event notifications.
- *Properties* - serve as access for private member variables.
- *Attributes* - provide declarative metadata about types at run time.
- Inline XML documentation comments.

Reference types in the software are organized into *Namespace* that is a logical organization of the software source code, and its nested types. This organization allows the developer to create globally unique types, and it is structured such as:

1. Namespace can contain inner types as: *Namespaces* , *Classes*, *Structures*, *Delegates*, *Interfaces*, *Enumerators.*

2. *Class* can contain declarations of: *constructors*, *destructors*, *constants*, *fields*, *methods*, *properties*, *indexers*, *operators*, *events*, *delegates*, *classes*, *interfaces*, *structs.*

   Classes in C# [8] are declared in this format :
   [attributes] [modifiers] class identifier [:base-list] class-body [;]

[6] http://www.ecma-international.org/publications/standards/Ecma-334.htm
[7] http://msdn.microsoft.com/en-us/library/z1zx9t92(VS.80).aspx
[8] http://msdn.microsoft.com/en-us/library/aa287558(VS.71).aspx

Where:

- *attributes* (Optional) - are additional declarative information.
- *modifiers* (Optional) - the allowed modifiers are new, abstract, sealed and four access modifier (public, protected, internal and private)
- *identifier* - is the class name.
- *base-list* (Optional) - is the list that contains one base class and any implemented interfaces, all separated by commas.
- *class-body* - contains declarations of the class members.

3. *Struct* type, is a valued type that can contain: *constructors, constants, fields, methods, properties, indexers, operators, and events.*

4. *Delegate* declaration defines a reference type that can be used to encapsulate a method with a specific signature.

5. *Interface* defines a contract. A class or struct that implements an interface must adhere to its contract. An interface can be a member of a namespace or a class and can contain: *Methods*, *Properties*, *Indexers*, *Events.*

6. The *enum* keyword is used to declare an enumeration, a distinct type consisting of a set of named constants called the enumerator list. This declaration takes the following form:
[attributes] [modifiers] enum identifier [:base-type] enumerator-list [;].
The enumerator identifier separated by commas, optionally includes a value assignment.

It is possible to split the definition of a class, a struct or an interface over two or more source files. Each source file contains a section of the class definition, and all parts are combined when the application is compiled. There are several situations when splitting a class definition is desirable:

- When working on a large project, and spreading a class over separated files allows multiple programmers to work on it simultaneously.

- When working with automatically generated source codes, and code can be added to the class without having to recreate the source file. Visual Studio uses this approach when creating Windows Forms, Web Service wrapper code, and so on. You can create code that uses these classes without having to edit the file created by Visual Studio.

2.2.2 C# Program System Organization

A C# program consists of one or more source files, known formally as compilation units. A source file is an ordered sequence of Unicode characters. Source files typically have a one-to-one correspondence with files in a file system, but this correspondence is not required. Conceptually speaking, a program is compiled using three steps:

3. Transformation - which converts a file from a particular character repertoire and encodes scheme into a sequence of Unicode characters.
4. Lexical analysis - which translates a stream of Unicode input characters into a stream of tokens.
5. Syntactic analysis - which translates the stream of tokens into executable code. [9]

C# files after compilation are combined in *.exe* and *.dll* files, where:

- Exe files are executable files that run under Windows Operating System.
- DLL are binary data that contain one or more *.cs* classes. The types in *.dll* can be invoked from *.exe* files or web servers (web pages in ASP).

The *.cs* files that do not contain namespaces are components.

2.3 Parsing Techniques

Parsing is the process of structuring a linear representation in accordance with a given grammar [10]. It is one of the components in the interpreter or compiler, which checks the correct syntax and build a data structure with its input tokens. The parser often uses a separated lexical analyzer to create a token from a sequence of input characters.

Parsing a sentence means computing its structural description in the form that the other part using the parsed sentence understands it.

There are two types of parsing:

- Top-down parsing - parsers using this method start from the largest element and break it down to the smaller parts. It can be viewed as an attempt to find left-most derivations of an input-stream.
- Button-up parsing - parsers using this method start from the input, and try to locate the most basic elements, and then the elements containing these. [11]

[9]http://www.ecma-international.org/publications/standards/Ecma-334.htm
[10]http://www.cs.vu.nl/ dick/PTAPG.html
[11]http://en.wikipedia.org/wiki/Parsing

Chapter 3

Solution

Solution Description

In previous chapters we identified requirements for a tool that will parse and model C# systems.

During this project we developed a Parsing and Modeling Application, which is a tool that parses C# source files from a specific C# system/application and models them based on the FAMIX structure. It is a third party tool for the MOOSE analysis platform.

As it is described in the Chapter 2, parser analyzes two levels of grammar: (1) lexical and (2) syntactic.

The first stage is token generation, by which the input character stream is split into meaningful symbols defined by grammar (e.g. an input such as 12*(3+4) / 2 is split into tokens '12', '*', '(', '3', '+', '4', ')', '/', '2'), and after token split, parser has to look about their meanings, whether token is context or arithmetic expression. The next stage is parsing or syntactic analyzing, which means checking the tokens. This is a process of defining components that can make expressions and the order in which they will appear.

To achieve meaningful parsing, successful modeling and exporting, we defined four components for PMCS tool:

- *C# Source Code Reader*,
- *C# File Parser*
- *C# Source Modeler*, and
- *MSE Exporter.*

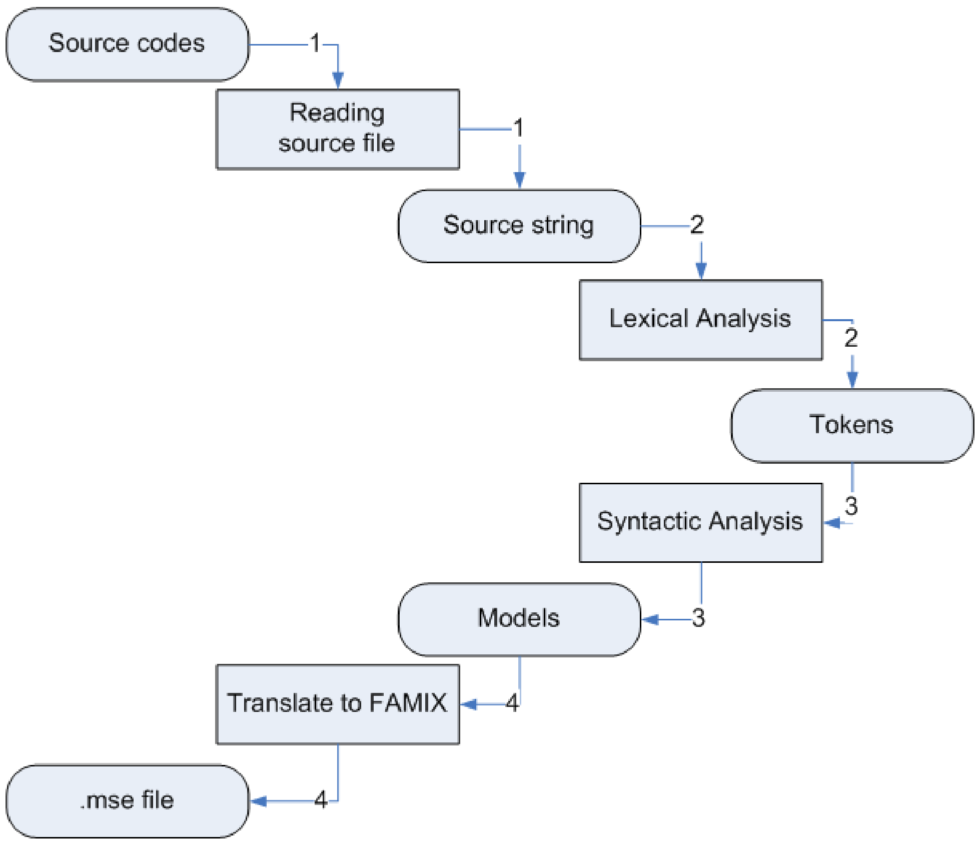


Figure 3.1: *PMCS logic flow*

PMCS tool work flow is shown in the Figure 3.1, which can be described with four steps:

- The input of our tool is existing software developed in C#. Source files are saved in a main folder that can contain subfolders. In one C# system not all the exiting files contain source code, and it is necessary for this tool to read the right files. From the file that is read it creates a string that has to be parsed.
- The string is complete source without comments and string/character values. From which the application creates tokens and words that it will parse.
- Application parses tokens in syntactic words and creates models.
- The last step is exporting parsed elements in *.mse* files.

This tool can parse systems developed in C# version 1.0 to version 2.0.

3.1 Source Reader

The first issue of PMCS is reading *.cs* files. To be read from PMCS application, a C# file has to be stored in one folder. User of the application will select that specific folder and parse it.

File reading has an order. Application starts to read files in the root folder. When all files in the root folder are read, application continues to read files in the first subfolder. If subfolder has another subfolder, application processes them in this order.

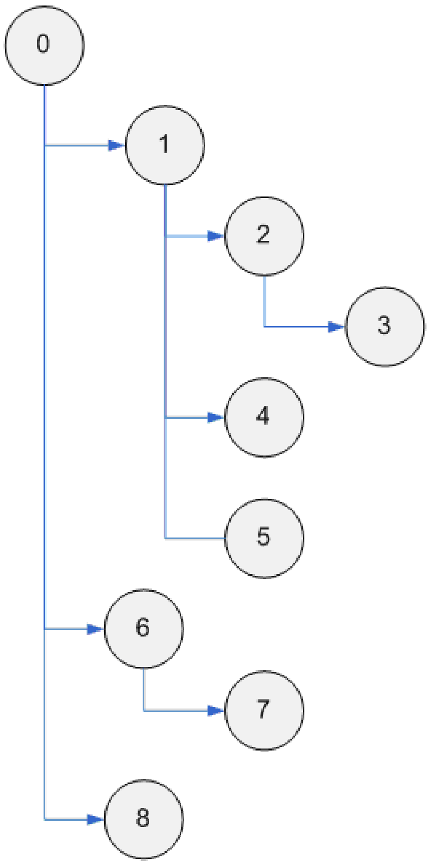


Figure 3.2: *Readers file processing order*

As it is shown in the Figure 3.2, each circle represents one folder. The number in the circle shows the order for reading files in this folder. If we have such a folder hierarchy and we choose to parse it, application will first read all *.cs* files that are in the folder 0, after completing them it will read files in the folder 1, and after finishing them tool will read files in the folder 2, continuing like this until it completes all *.cs files* in the folder 8.

Figure 3.3 shows how PMCS tool takes the command for reading *.cs* files.

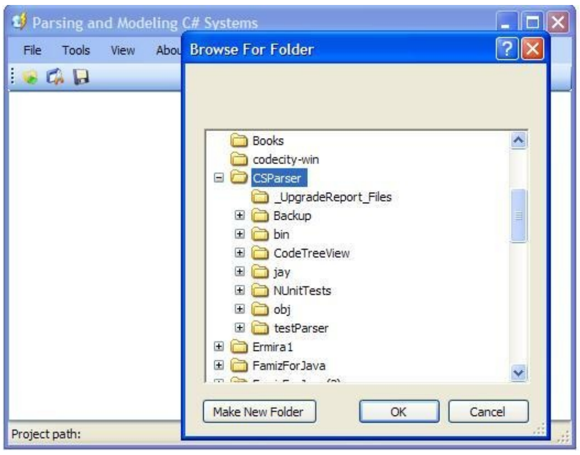


Figure 3.3: *PMCS - while user is choosing the folder to parse*

Technically, file reading is a process of opening the specific file and storing its content in one data holder that can be a string builder. The content of the C# file can be complex, it may contain many functions including comments, string values, character values, and so on. Comments, string and character values can be the same with keywords of the C# programming language. Since, our tool development logic is based on the language keywords, our approach to this problem was to remove all not used contents. Basically, application during the file reading process checks if the content is:

- Code comment - it starts with // or /* and ends with */ .
- C# preprocessor directive - it is only instruction on a line and starts with #.
- String value - values are under the signs " " .
- Character value - values are under the signs ' ' .

The code comments are stored in the string builder as white space characters. C# preprocessor directives are not read and in the string builder are saved as white space characters, too. This is because of making space between previous command and the next one. String values are not read and in the string builder all strings values are empty (string s = " "). In the same way are stored character values (char c =' ').

Files that do not contain namespace are internal components of the software. These files are not parsed by PMCS tool. Since the FAMIX meta-model contain entities, and each of these entities has to belong to its parent entity, like, *class* has to belong to a *namespace* otherwise it belongs to nobody, and parsing this element will not be meaningful for the FAMIX meta-model, we choose not to parse, model and export C# components.

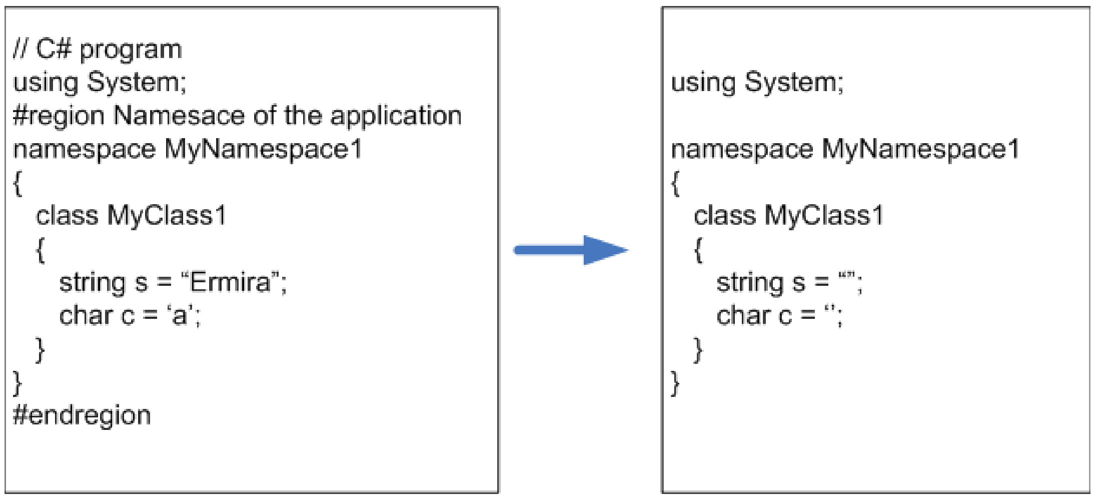
The file reading process and source code string creation has this logic:

1. PMCS checks the file name extension. If it is *.cs*, it follows:
   (a) PMCS opens the file content.
   (b) Software reads the content character per character.
   (c) If character is **'** :

        i. Character ’ is given to the code string.
        ii. Reader reads the next character until it finds the next ’ , if reader finds characters like \’ one after the other, means that it has to continue until it finds the next ’ character.
        iii. Last ’ character is given to the code string.
    (d) If character is ” :
        i. Character ” is given to the code string .
        ii. Reader reads the next character until it finds the next ” , if reader finds characters like \” one after the other, means that it has to continue until it finds the next ” character.
        iii. Last ” character is given to the code string.
    (e) If character is / and the next character is / - that means //:
        i. Reader reads till the next **\n** character .
        ii. White space character is given to the code string.
    (f) If character is / and next character is * - that means /*:
        i. Reader reads untill the next character is * , if after this it finds character / - that means */.
        ii. White space character is given to the code string.
    (g) If character is # :
        i. Reader reads till the next **\n** character.
        ii. White space character is given to the code string.
    (h) if the content has the keyword *namespace*
        i. PMCS starts parsing the module
    (i) If the content does not have the keyword *namespace*
        i. PMCS reads the next file

2. If file extension is not *.cs* PMCS checks the next file

The output of the reading process is a string builder which contains a C# source that has to be parsed.



```
// C# program
using System;
#region Namesace of the application
namespace MyNamespace1
{
  class MyClass1
  {
    string s = "Ermira";
    char c = 'a';
  }
}
#endregion
```

```

using System;

namespace MyNamespace1
{
  class MyClass1
  {
    string s = "";
    char c = '';
  }
}
```

Figure 3.4: *C# class transformation after the reading process*

The Figure 3.4 shows an example about how a source file can contain string or character values (as in the left part), and after reading process of PMCS tool, it is transformed in a string builder as in the right part of the figure.

3.2 C# Models

C# programming language contains its source code entities, which are organized in a vertical hierarchy. One entity can be the inner entity of the other or the opposite. The root of all entities is namespace that can contain other entities including inner namespace, but, does not belong to other types such as class, interface etc.

The flexibility of having inner entities makes the model of the C# more difficult. In the Figure 3.5 is presented this model, which contains all the entities that can be in one C# source code.

Namespace is on the top of the C# entity model. Every type in this hierarchy belongs to a namespace, and inner namespaces are modeled as individual namespaces. When PMCS tool reads a namespace, it creates new entity of that namespace. Other types of the second level of the hierarchy (looking from top to bottom) contain types that in PMCS tool are modeled as groups:

- As *class* entity are modeled: *class, interface, and struct.*

- There are not-modeled entities also, like: *delegates*, *enum*, and *formalParameters.*

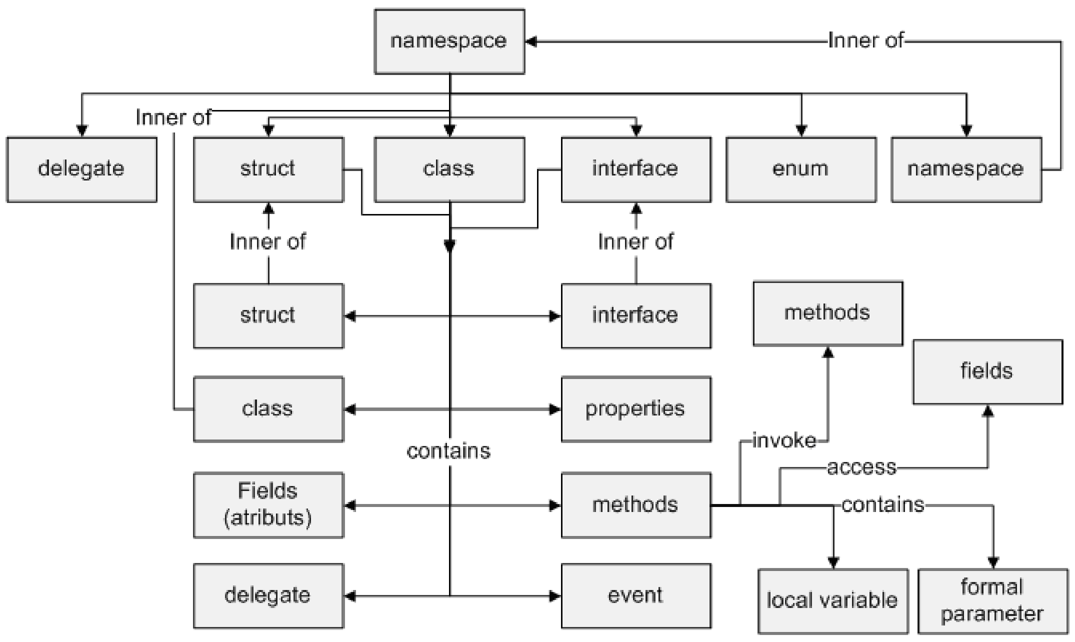


Figure 3.5: *C# language modeling*

Class, interface and struct in PMCS application are equivalent with classes for FAMIX. Class attributes in PMCS are:

- *Id* -unique id as entity in the system,
- *Name*- name of the class, interface, enum or struct,
- *Is Abstract* - an attribute that explains if the object is abstract,
- *Is Struct* - an attribute that explains if the object is struct,
- *Is Interface* - an attribute that explains if the object is interface,
- *Is Enum* - an attribute that explains if the object is enum,
- *Belongs To* - id of the namespace. Each modeled class (classes, interfaces, struct) and inner types of them belong to a namespace of the parent-modeled class. Inner classes automatically have parent classes.
- *Fields* - a list of attributes of the class,
- *Properties* - a list of properties in the modeled class,
- *Methods* - a list of methods and events in the modeled class.

Delegates are not modeled because do not exist in the FAMIX structure. Delegate in C# is an event with input parameter method that will be invoked. Delegates can be seen as methods of namespace. In the future work and with extendibility of FAMIX these can be added in PMSC as methods of a namespace. The hierarchy of PMCS allows adding an attribute without having to change other part of the source code.

PMCS application models class attributes same as the FAMIX meta-model structure is, as we explained in the Chapter 2.

Properties are specific feature in C# that can be seen as methods with get and set accesses. PMCS application models them as specific object of the C# system hierarchy.

In the PMCS modeling part as in the FAMIX structure the objects like local variable and formal parameter belong to a method. In the other side, methods can invoke other methods and access fields in the class, and they are modeled as a list of invocations and accesses.

Method entity attributes in PMCS application are:

- *Id* - method identifier,
- *Name* - method name,
- *Signature* - method signature (full name),
- *LOC* - lines of code for that method,
- *Accesses* - list of class attributes that the method access,
- *Invocations* - list of methods that are invoke in the method,
- *Formal Parameters* - list of parameters,
- *Local Variables* - list of variables declares in the method body.

The namespace that is on the top of the hierarchy, creates a tree as it is shown in the Figure 3.5. In the PMCS namespace entity represents a root entity that contains all other C# entities. Namespace is saved in the list of namespaces that in our case is a data holder. Namespace as an object has these attributes:

- *List of classes* - that holds a class entity (classes, struct, interfaces) with its attributes. The attributes of class and its sub attributes are explained above.
- *List of inheritances* - inheritance is the PMCS entity that specify which subclass inherits which super-class. It has these attributes:
  - *Id* - unique inheritance identifier,
  - *SupperClass* - class that is inherited, base class,
  - *SubClass* - class that inherits a (some) class(es).
- *List of accesses* - that holds the access entity updated from the method entity. Access is an attribute of a method, and it has:
  - *Id* - unique identifier,
  - *Name* - name of the field that is accessed,
  - *BelongsTo* - id of the class that method belongs to,
  - *AccessedBy* - id of the method that is making access,

  - *Accesses* - id of the field that is accessed, it is updated with access object creation process after all the files are parsed.

- *List of invocations* - invocation entity with its attributes:
  - *Id* - unique identifier,
  - *Name* - name of the method that is invoked,
  - *Parent* - the object that call the method,
  - *InvokedBy* - id of the method that makes invocation,
  - *Invokes* - method signature that is invoked,
  - *Candidate* - id of the method that is invoked.

3.3 Parsing Control Process

Parsing is a process of building data structures, which has two main sub processes:

- Lexical Analyzer, and
- Syntactic Analyzer.

This process depends on the programming language grammar. C# has its own grammar that is explained in the Chapter 2 and based on the C# keyword-list the parsing process can be done.

The body of a C# program can be:

```
using System;
namespace MyNamespace1
{
  class MyClass1
  { }
  struct MyStruct
  { }
  interface MyInterface
  { }
  delegate int MyDelegate ();
  enum MyEnum
  { }
   namespace MyNamespace2
  { }
  public class MyClass2
  {
    public static void Main (String[] args)
    { }
  }
}
```

As it is explained in the Figure 3.5, the entity model type in C# is very flexible, the only order that is strict is that namespace is always on the top. It means that the other types all belong to a namespace. Namespace itself can be inner type of a namespace but not an inner type of any other entity. Based on this logic PMCS looks to this hierarchy on two levels:

- *Namespace*- inner types of a namespace and inner types of class (classes are specified in this level) ,
- *Method* - body of the method.

The parsing process of the C# source file starts after file reading process. The file reading process output is the input of the *parsing process*. Input of the *parsing process* is string builder that we can call source string. As it is shown in the Figure 3.6, parser of PMCS locates the specific keywords of the C# programming language, and based on the keyword that it finds first it creates a specific token. Token string is a string that has to be parsed and gave meaning to that entity. The entity that can be created from a token string is namespace, inner entities of the namespace or inner entities of them.

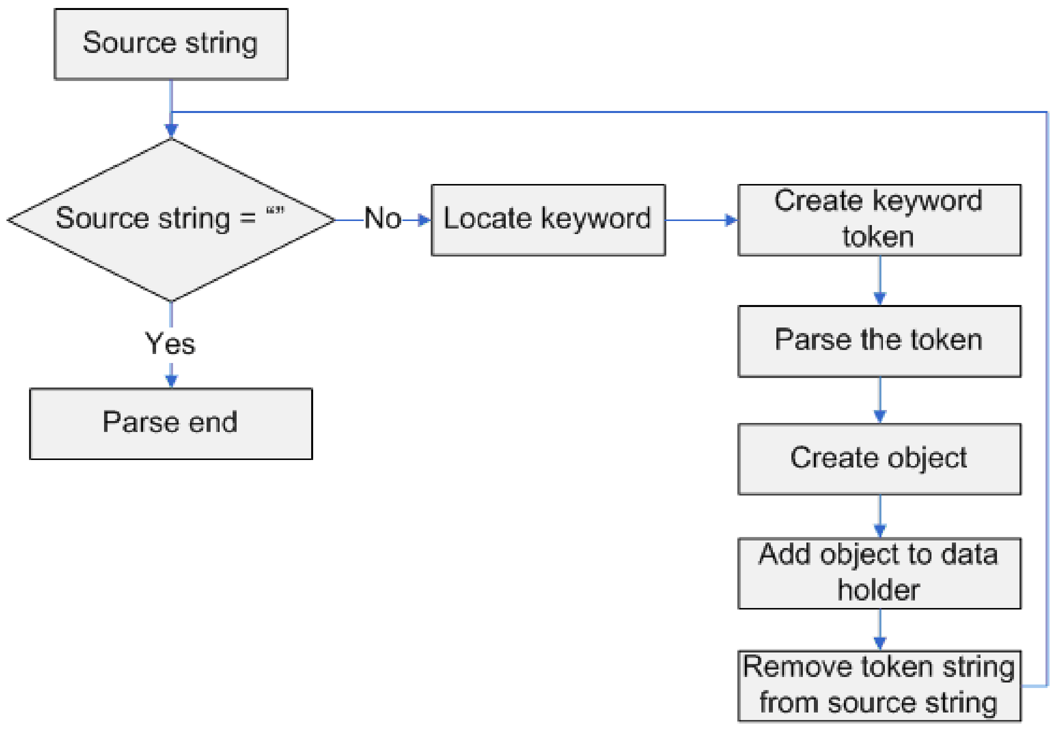


Figure 3.6: *PMCS logic flow*

Specific keywords are used in token creation process. We grouped C# keywords by where they can be used. We do not want to check if the specific keyword exists in the source string if it can not be in that part of the code where the parser is processing.

The parsing technique that application uses to find in which part of the code it is working, is LR Parser. Application searches for specific keywords in the source string and based on them it creates specific tokens.

In the first level, PMCS creates tokens like:

- *Library*,
- *Namespace*,
- *Class*,
- *Enum*,
- *Interface*,
- *Sruct*,
- *Delegate*,
- *Fields* object attributes,
- *Methods*,
- *Properties*,
- *C# Attributes*,
- *Class Inheritance.*

In the second level of parsing, PMCS works with:

- *Local variables*,
- *Invocations*,
- *Accesses.*

3.3.1 First Level Parsing

To create tokens for its first level, PMCS tool follows C# grammar rules. It creates tokens based on the keywords and the end of the specific programming command.

In C# programming language there are libraries that are used during the development. The grammar of C# language for library declaration is:

- using LibraryName;

Library declaration always starts with the keyword "using" and ends with ";" . PMCS after reading library declaration cuts the string from keyword "using" to the keyword ";" and extracts the name of the Library.

Namespace entity in C# can be declared in different ways, like:

1. One namespace in the file,

```
namespace N1
        {
          class A {}
          class B {}
        }
```

2. Inner namespace in the file,

```
namespace N1
        {
          namespace N2
                  {
                  class A {}
                  class B {}
                  }
          }
```

3. More namespaces in the file with the same name

```
namespace N1.N2
        {
        class A {}
        }
namespace N1.N2
        {
        class B {}
        }
```

And from the three examples above it can be seen that the keyword "namespace" is the starting position of the token. The token ends with "{" symbol that in the other side represents the start of the namespace body. PMCS application creates token for a namespace in this format:

namespace N1 or *namespace N1.N2*

Application splits the token based on the white space character, which means that it splits the token into individual words, and in order to extract the name of that namespace then it splits words based on the " . " character. There will be two kinds of results, like:

1. namespace name = N1,
   - the name of the entity is N1,
   - this entity is parent namespace of the C# system.
2. namespace name = N1.N2,
   - the name of the entity is N2,
   - this entity belong to namespace N1.

Before saving the namespace entity to the data holder, PMCS application checks if this namespace exists in the data holder. PMCS gets the index of the namespace in that data holder and puts all other entities under this namespace. Namespace can be the same in different source files, and PMCS looks at it as one namespace with many classes that can be in the same file or in different files.

Namespace parsing can be more understandable with this example as:

- PMCS has a list of parsed system namespaces,
- Parser has an array of namespaces - FN, populated with namespaces of one file,
- Token is copied from character in the position length + 1 of keyword "namespace" to the end of command that is character "{".
- The copied text is split based on "." character and instantiated in an array N.
- If N has more then 1 member:
  - Last member of N is new namespace N2,
  - Last member of Namespace belongs to previous member of array N, which is namespace N1.
- If FN Array has more then 0 member:
  - Last member of array N is new namespace N1,
  - N1 belongs to last member of FN array.
- If N1 does not exist in the PMCS namespace list:
  - N1 is new namespace in that list.

First two entities (library and namespace) have the same logic for token creation. The token string starts from a keyword and ends with a specific character. The problem is when the specific keyword is not stating position of the token. This is the case of *class, interface and struct* declaration.

In the Chapter 2 it is explained how the class can be declared in C#. In this case specific keyword is used to detect what parser is parsing. Token is created from the end of the previous token to the end of the current token. As it is explained in the Figure 3.6 after the model is created, parser removes the token string from the source string. In this case the token starting position will be starting position of the source string.

For: *class, interface, struct and enum* the end character of their token is "{". It presents class body start. When the parser locates keyword "class", the token for class will be created, when it locates keyword "struct" than the token for structure will be created, and when parser locates keyword "enum" than the token for enumerator will be created.

As mentioned, class, interface and structure in PMCS are modeled as classes that belong to a namespace. Token creation for these entities has the same logic. Next word to the specific keyword is the name of the entity. There are cases where they exist in different forms such as:

1. class - *public class MyClass : Class1, MyInterface2* {
   - Next to the *keyword class* is the name of the class. In this case it is *MyClass*
   - The list of classes from ":" to "{" are inheritance definitions.
   - from Chapter 2 we know that the class modifier *public* is optional

2. struct - *public struct MyStruct* {
    - following the same logic the name of the structure is *MyStruct*
3. interface - *public interfaceMyInterface* {
    - following the same logic the name of the interface is *MyInterface*

If the first case occurs we said that: from ":" to "{" character are defined superclasses of the current class. Inheritance definition means finding the sub and the superclass in that application. The mentioned sample is one of inheritance occurrences in an application , but there can be another form of inheritance definition, which is when a class contains an inner class.

Pseudo code for class, struct or interface parsing is:

1. Token T contains data from *keyword* to the first "{" character (do not contain "{" character),
2. Subtoken, temporary string temp is initialized with substring of T without *keyword* - from *keyword* to the end.
3. If temp contains ":":
    - string C is initialized with data of temp from the first character to the character ":",
    - string N is initialized with data of temp from ":" character to the end of the temp,
    - N is spilt by "," and put to the array P,
    - C is class CS,
    - CS inherits P.
4. If temp does not contain ":":
    - string temp is class CS.

C# has partial classes, which means that one class can be divide in two files. Parser controls if CS exists in the list of classes in the current namespace.

- If CS does not exist
    - CS is new class, and
    - CS belongs to the current namespace.

Delegates and enumerators are read by PMCS application. Tokens for both of them contain the full data but they are not processed. It means that in the future work, if FAMIX model requires this information, PMCS has to be modified in the methods for parsing delegate and enumerator.

The parser of PMCS application continues with field (attribute entity), property and method parsing in the class, structure or interface. There is no order for declaring a field, a property or a method. The first problem is to create a token, and then the parser has to locate and understand the differences in their declaration.

Fields in C# can be declared in class, interface or structure. The forms of *field* declaration are:

1. With one command line - command line ends with ";", and there are declared fields with same type:
    - *public static int x =1, y, z= 100;*
2. With more command lines:
    - *public static int x;*
    - *public static int y;*
    - *public static int z = 100;*

The field declaration has the mandatory part:

- the type of the field - *int, string, object,* etc, and
- the name of the field.

The PMCS parser first locates the ";" keyword and creates the token from the start of the source string to the keyword. There can be token formats like:

1. declaration and initialization of the field
    - *public static int x = 1;*
2. declaration of the field
    - *public static int c;*
3. multiple fields declaration and internalization in one command
    - *public static int x =1, y, z= 100;*

Parsing algorithm for filed that is used in PMCS is explained with pseudo code as:

1. Split the token string based on "," and give to an array A.
    - In the case of having multiple declaration, array A has more that one member.
    - In the case of one field declaration array A has one member.
2. For each member in the array A
    - Create a new array B by splitting the array A member based on the "=" character,
        - In the case of having the "=" character, in the string array B will be two members,

    - otherwise array B contains one member.

- For the first member of the array A,
    - Array B is separated with white space character and array C is created,
        * the last member of the array C is the name of the field,
        * last -1 member of the array C is the type of the field, or fields,
        * Other members of the array C are modifiers.
    - For other members of the array A,
        * The first member of array B is the name of the field,
        * Type and modifier are the same with the first member of the array A.

Field in the PMCS application model is inner type of a class. This means that every field is an entity added in the field list of the current class. These fields can be accessed from any method.

The property is one of the parsed entities in the first level in PMCS application. Properties in C# contain body, for accessing specific field in the class, in the body of the property there has to be at least one of the accessors: get or set. The body of the property starts with "{" and ends with "}". The PMCS parser creates a token for property from the beginning of the property declaration to the beginning of the property body "{". Reading from the right to left the words are:

1. the name of the property,
2. the type of property,
3. modifiers of the property.

Method in C# has grammatical rule of declaration. There are two main types of methods in C#:

1. methods with body,
2. methods without body - abstract methods.

Method declaration has this structure

attributes method-modifiers return-type member-name (formal-parameter-list).

PMCS application parses methods, and creates a token from the first character in the source string (remaining source string) to the first "(". After this it splits the token based on the white space character.

1. the first word is the name of the method,
2. the second word is the return type - if the word is not equal to void, and
3. other words are modifiers.

3.3.2 Second Level Parsing

Parsing a method from a source file starts by parsing the name of the method which is covered in the first level of the parser and then continues by parsing the body of the method. From the body of the method are extracted local variables, invocations and accesses. This is the second level of the parser.

Second level parsing is a process that uses:

- keywords, and
- operator appearances in the method.

The parser does not analyze loops, conditions or other blocks under the method. Based on the C# keywords it creates tokens from one keyword to the other. Parsed keywords are *operators, end commands, loops, and conditions* and other types of the keywords, which Parser analyzes and gives to token a specific meaning.

In the example below is represented a simple method.

```
public void MyMethod()
        {
        Student st = new Student();
        st.Name = "John";
        int count = 0;
        for (int i = 0; i < 10; i++)
                  {
                  count++;
                  }
        st.Print(field1 + " " + field2);
        }
```

Following this rule this simple method is parsed with the tokens from one operator to the other in one command line. Tokens that PMCS creates, are:
Student st, Student(), st.Name, "John", int count, 0, int i, 0, i, 10, i, count, st.Print(field1 + " " + field2), field1 , " " , field2.

From these tokens there are created models like:

- *Student st* - is a local variable because has a type in the declaration,
- *Student()* - is an invocation of the method *Student()*,
- *st.Name* - is an access of the attribute *Name*,
- *John* - is a value and PMCS do not model it,
- *int count* - where *count* is a local variable,
- *0* - is a value and as a numerical value PMCS do not model it,
- *int i* - where *i* is a local variable,

- *0* - is a value and as a numerical value PMCS do not model it,
- *i* - is a local variable that is already modeled,
- *10* -is a value and as a numerical value PMCS do not model it,
- *i* - is a local variable that is already modeled,
- *count* - is a local variable that is already modeled,
- *st.Print(field1 +   + field2)* - is an invocation of *Print* method of class *Student*,
- *field1* - is an access of *field1* attribute,
- *field2* -is an access of *field 2* attribute,

As mention in the example above, to parse body of a method the PMCS *parser* creates objects for Local Variables, Accesses and Invocations. Local variables have types in their declarations. As you can see in the example, *int count* is a local variable, which can be used in that method. During the parsing process of the token "count", parser checks if this token exists in the list of variables. If it exits, means that it is a local variable that is used.

Parser creates accesses for all variables that have no type in their declaration and are not local variables in the list. This is a rule because a variable that does not have a type in its declaration, means that it is declared before. The *id* of the field/attribute that is accessed by method is not initialized and it will be processed in the next step. It is because the fields in C# can be declared at the end of the file, and in that case parser still did not parse them. This object belongs to the method attribute of the access until the next process updates its data.

Invocations are parsed by following the rule of having character "(" and ")" in their token. The rule of invoking a method in C# programming language is to open and close the bracket, even if there is no parameter. The invoked object is given to the method object attribute.

3.3.3 Method Parsing Process

Invocations and Accesses during their creation are not initialized completely. It is because the *id* of the method or the attribute has to be found after reading and parsing all files. The PMCS Parsing process ends by checking invocations, accesses, and local variables.

There are three processes that run after the parsing:

1. inheritance definition,
2. invocation definition,
3. access definition.

Inheritance definition process starts by reading the list of inheritances in that class, and for each inheritance it finds the *id* of the class that is inherited. The inherited classes can be classes of the same namespace, or other. Parser checks them in all namespaces, and after it finds the class *id*, it creates an object inheritanceDefinition and holds it in the namespace attribute called inheritanceDefinition. It can be that class *id* is not in the list of classes, and this time Parser follows with its next process.

Invocation creating process starts with method body parsing. The parser has to find the *id* of that method that is invoked. This is done by checking the methods of the same namespace. If this method exists, parser puts the *id* of that method in the list of invocations in that namespace, otherwise it continues with its next step.

Access definition is similar to invocation definition. Parser checks for the fields in the class of that method and in its super-class. If it finds the field, parser puts its id to the list of accesses in that namespace entity.

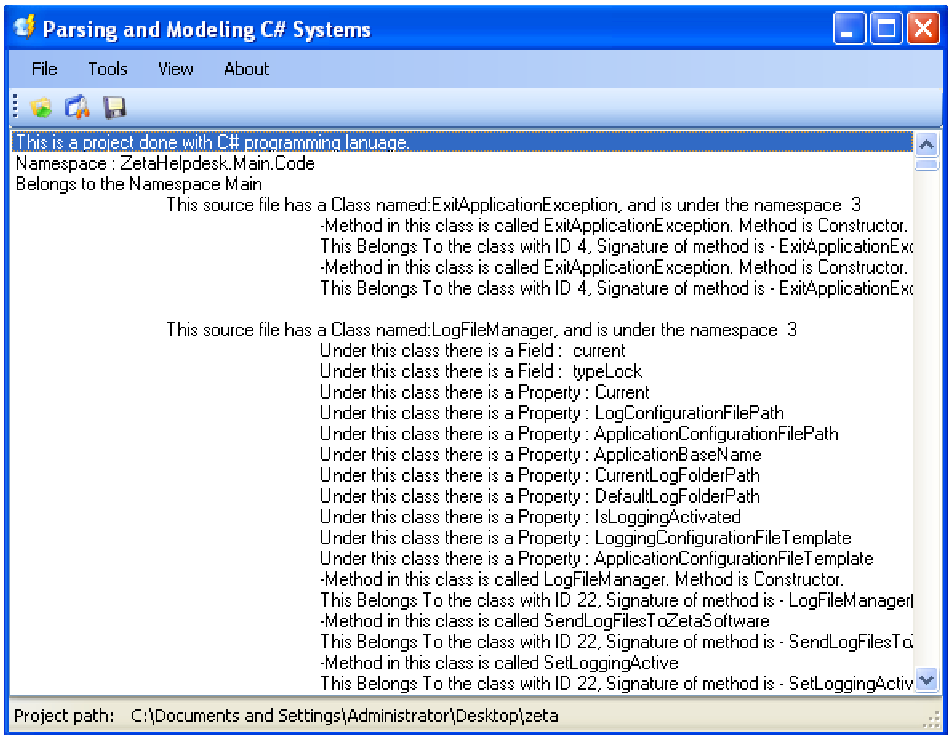
3.4 Modeling Process

During the parsing process, parsed entities are saved in a list of namespaces. Every namespace contains attributes as it is explained in the Figure 3.5. This hierarchy is modeled as follows:

- Namespace N1
  - Class 1
    - Fields
      - Field 1
      - Field 2
    - Properties
      - P1
      - P2
    - Methods
      - M1
      - M2
  - Class 2
    - Fields
      - Field 1
      - Field 2
    - Properties
      - P1
      - P2
    - Methods
      - M1
      - M2

- Namespace N2
  - Class 3
    - Fields
      - Field 1
      - Field 2
    - Properties
      - P1
      - P2
    - Methods
      - M1
      - M2
- Inheritance Definitions
  - C1 inherits C3
  - C2 inherits C3
- Invocation
  - M2 invokes M6
  - M6 invokes M1
- Access
  - M6 accesses Field1
  - M1 accesses Field1
  - M1 accesses Field2
- Local Variable
  - L1 belongs to M1
  - L1 belongs to M2

In PMCS it looks as in the Figure 3.7



```
This is a project done with C# programming lanuage.
Namespace : ZetaHelpdesk.Main.Code
Belongs to the Namespace Main
                This source file has a Class named:ExitApplicationException, and is under the namespace  3
                                -Method in this class is called ExitApplicationException. Method is Constructor.
                                This Belongs To the class with ID 4, Signature of method is - ExitApplicationExc
                                -Method in this class is called ExitApplicationException. Method is Constructor.
                                This Belongs To the class with ID 4, Signature of method is - ExitApplicationExc

                This source file has a Class named:LogFileManager, and is under the namespace  3
                                Under this class there is a Field :  current
                                Under this class there is a Field :  typeLock
                                Under this class there is a Property : Current
                                Under this class there is a Property : LogConfigurationFilePath
                                Under this class there is a Property : ApplicationConfigurationFilePath
                                Under this class there is a Property : ApplicationBaseName
                                Under this class there is a Property : CurrentLogFolderPath
                                Under this class there is a Property : DefaultLogFolderPath
                                Under this class there is a Property : IsLoggingActivated
                                Under this class there is a Property : LoggingConfigurationFileTemplate
                                Under this class there is a Property : ApplicationConfigurationFileTemplate
                                -Method in this class is called LogFileManager. Method is Constructor.
                                This Belongs To the class with ID 22, Signature of method is - LogFileManager(
                                -Method in this class is called SendLogFilesToZetaSoftware
                                This Belongs To the class with ID 22, Signature of method is - SendLogFilesTo
                                -Method in this class is called SetLoggingActive
                                This Belongs To the class with ID 22, Signature of method is - SetLoggingActiv
Project path:   C:\Documents and Settings\Administrator\Desktop\zeta
```

Figure 3.7: *C# system modeling with PMCS tool*

3.5 Exporter

The last function of PMCS tool is exporting parsed information from the system source code to the *.mse* file. We said that the process of parsing source code follows by saving these information, which later are used by **modeler** and **exporter**.

Exporter part in PMCS tool is based on the MSE grammar. It has a standard structure about how the system entities are written in the file. As we mentioned in the Chapter 2, the structure of the *.mse* file is organized with nodes and command.

Entities of the system are represented as nodes in the file, these nodes are filled with their attributes, and each element node attribute has a specific value. Below this paragraph you can see all the MSE element nodes, attributes and their values, and MSE commands that we extract with this tool from a C# source code.

```
(Moose.Model(id:1)
    (name 'PMCS')
    (entity
        (FAMIX.Namespace (id: namespace identifier)
            (name 'namespace name')
            (belongsTo (idref: namespace id)))
        (FAMIX.Class (id: class identifier)
            (name 'class name')
            (belongsTo (idref: namespace id))
            (isAbstract - true or false)
            (is Interface - true or false)
        (FAMIX.Method (id: method identifier)
            (name 'method name')
            (accessControlQualifier 'modifier type')
            (belongsTo (class id))
            (LOC - number of lines that method has)
            (signature 'method signature'))
        (FAMIX.Attribute (id: attribute identifier)
            (name 'attribute name')
            (accessControlQualifier 'modifier type')
            (belongsTo (class id)))
        (FAMIX.Invocation (id: invocation identifier)
            (invoked by (idref: method id))
            (invokes 'method signature'))
        (FAMIX.Access (id: access identifier)
            (accessedIn (idref: method id))
            (accesses (idref: field id)))
        (FAMIX.InheritanceDefinition (id: inheritance identifier)
            (subclass (idref: subclass id))
            (superclass (idref: superclass id)))
(sourceLanguage 'C#')
```

Exporter uses as input the results of the *parser* and in one way it can be seen as a parallel process with source code modeler. As in the Figure 3.8 the PMCS process starts from an *.cs* file which is parsed, modeled and at the end it is exported in the *.mse file.*

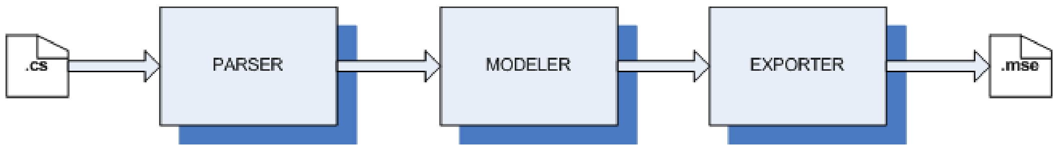


Figure 3.8: *Parsing, modeling and exporting with PMCS*

Based on the FAMIX Level of Extraction (Table 2.1) parser covers: Level 1, Level 2 and Level 3 completely, and entities of Level 4 that are FormalParameter and ImplicitVariables are not parsed, neither they are modeled. Exporter has all these entities as input from the parser part but, it does not exports them completely. It exports entities of Level 1, Level 2 and Level 3 completely, and entity of Level 4 that is LocalVariable is not exported.

Chapter 4

Experiments

In the previous Chapter we described the solution by explaining the logic of the software and its components. To validate usefulness of this software, tests are necessary. First objective of the test, is to validate the functionality of the software as it is required, which means to parse and model a C# system based on the FAMIX structure and to export that model in the *.mse file* format.

The PMCS experiment has three components:

- Source Code of a C# system,
- PMCS software itself, and
- CodeCity tool to verify the *.mse* file format.

Tests are done based on the real C# open source systems. Systems are small or large depending on the lines of code or source code files. PMCS is tested with systems starting from a small C# application with about 20 source files, continuing with a medium C# system with more than 450 source files, and a system with 800 source files. These systems can be differentiated by the complexity of the system components. With "complexity" of the system we understand systems that contain application and services that communicate between each other, and create one C# software system.

PMCS is tested with windows application, web application and complex C# systems that contain windows application in relation with windows and web services. Parsed C# system is exported in the *.mse file*, and validated using specific tools.

CodeCity [18] - MOOSE based reverse engineering tool is the tool that we used to import *.mse files* of a C# system meta-model. Further verifications are done using Mondrian, another reverse engineering tool.

One of the main objectives of software test is to find software bugs, but not limited to this. Software testing is a process of validating and verifying that software meets the business and technical requirement based on how the software is designed and developed. Testing can be implemented any time in the developing process, and the main effort of it, is after the coding process has been completed.

The PMCS testing process starts with:

- Unit testing - testing PMCS components during development and as a software.

- Integration testing - testing integration between Source Reading, Parsing, and Modeling components of PMCS.
- Systems integration testing - testing the integration of PMCS with MOOSE tools. Files exported from PMCS and imported in CodeCity.

PMCS as a software has to meet its functional and technical requirements. The functional requirements that are tested for each PMCS module are:

- *Source reading* module requirements:
  - Reading *.cs* files that are part of one namespace. Not reading components (C# sources that are not part of a namespace).
  - Creating the sources that will be parsed.
  - No memory overload.
- *Parsing* module requirements:
  - Parsing of namespaces, classes, fields, methods, inheritance, accesses, and invocations.
  - No overloading during parsing.
- *Modeling* module requirements:
  - Visualizing the parsed entities.
- *Exporting* module requirements:
  - Exporting all entities in the *.mse* file.

PMCS is developed to fulfill the requirements, and it has more parsing features that was expected to has. It parses and models Properties and Local Variables from a source code, and there is a method that can be easily modified to parse Formal Parameters, too. To conclude that PMCS fulfills the requirements, in this document are explained three examples with three different C# Open Source Code Systems.

4.1 First Test - Zeta Helpdesk Software

The Zeta Helpdesk (http://www.zeta-helpdesk.com/download.html) application is ticket system for Web and Windows. This Software consists of three main parts:

1. The main Windows Application - it is an application that is used to manage all parts of Zeta Helpdesk, like creating and editing data.
2. Scheduler - it is a console application without any interactive GUI. It is used to execute certain tasks.
3. End User Web - an ASP.Net 2.0 (C#) application that allow selected system users to manipulate specific data

The Zeta Helpdesk system has 420 source files with 90,435 LOC that PMCS reads. It is not a very large (medium) system based on the number of source files that it contains. In the other side it is a complex C# systems because it contains three different parts, windows, console and web Application. This test ensures that PMCS can parse a modular system with 420 source files (353 *.cs files*).

The parsing process of this system starts by selecting the root path of it, as it is shown in Figure 4.1, that Zeta Helpdesk is organized in 12 core folders that means 12 core *namespaces*. These namespaces contain *inner namespaces*.

Process for reading files in Zeta HelpDesk folder is done in the time interval with less than 30 seconds. Parsing process is done for 3 minute, and if we add the modeling process to it we can say that both are completed for 3 to 4 minutes.

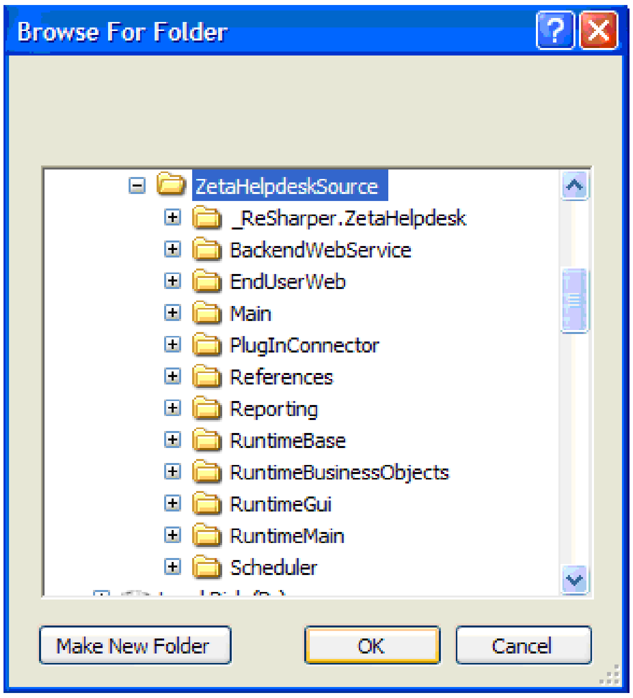
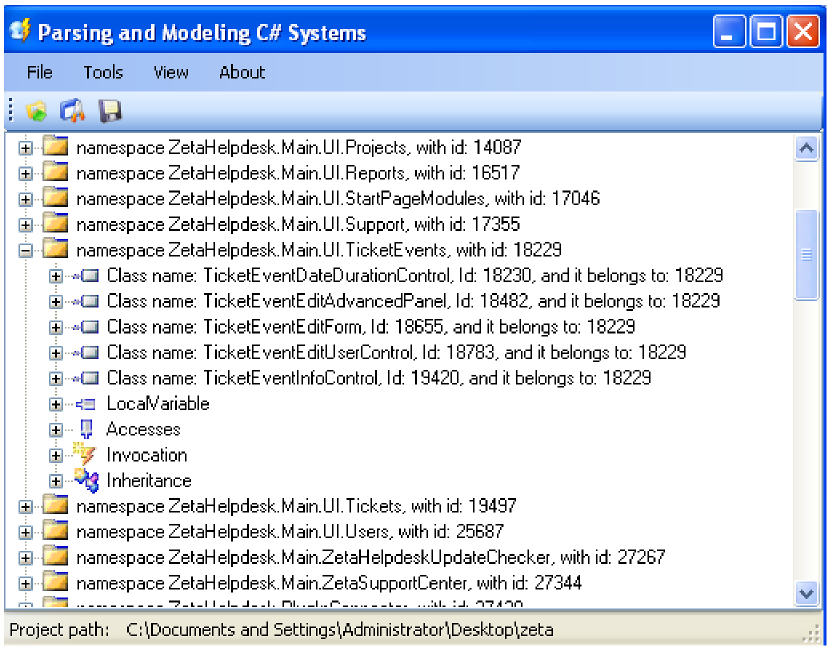


Figure 4.1: *Zeta Helpdesk folder organization*

Results for Zeta Helpdesk system are presented in the Figure 4.2. One of the namespaces in this system has five classes, local variables of methods, accesses, invocations, and inheritances. The classes listed under it have: fields, methods and properties.



```
namespace ZetaHelpdesk.Main.UI.Projects, with id: 14087
namespace ZetaHelpdesk.Main.UI.Reports, with id: 16517
namespace ZetaHelpdesk.Main.UI.StartPageModules, with id: 17046
namespace ZetaHelpdesk.Main.UI.Support, with id: 17355
namespace ZetaHelpdesk.Main.UI.TicketEvents, with id: 18229
    Class name: TicketEventDateDurationControl, Id: 18230, and it belongs to: 18229
    Class name: TicketEventEditAdvancedPanel, Id: 18482, and it belongs to: 18229
    Class name: TicketEventEditForm, Id: 18655, and it belongs to: 18229
    Class name: TicketEventEditUserControl, Id: 18783, and it belongs to: 18229
    Class name: TicketEventInfoControl, Id: 19420, and it belongs to: 18229
    LocalVariable
    Accesses
    Invocation
    Inheritance
namespace ZetaHelpdesk.Main.UI.Tickets, with id: 19497
namespace ZetaHelpdesk.Main.UI.Users, with id: 25687
namespace ZetaHelpdesk.Main.ZetaHelpdeskUpdateChecker, with id: 27267
namespace ZetaHelpdesk.Main.ZetaSupportCenter, with id: 27344
```

Project path: C:\Documents and Settings\Administrator\Desktop\zeta

Figure 4.2: *The PMCS parser results for Zeta Helpdesk - source code*

In the table bellow are presented the entitles that are parsed, modeled, and exported.

Level	Entities	Parsed	Modeled	Exported
1	Namespace	+	+	+
1	Class, inheritance, struct	+	+	+
1	InheritanceDefinition	+	+	+
1	Methods	+	+	+
	Property	+	+	-
2	Attributes, and Fields	+	+	+
2	GlobalVariables	+	+	+
3	Accesses	+	+	+
3	Invocations	+	+	+
4	LocalVariable	+	+	-
4	FormalParameters	+	-	-
4	ImplicitVariables	-	-	-
	Arguments	-	-	-

Table 4.1: Source code entities that are *parsed, modeled and exported* from PMCS tool

The *.mse file* that is exported from PMCS is imported in CodeCity, and statistics for Zeta Helpdesk are presented in the Figure 4.3 taken from the MOOSE Browser. In Figures 4.4 and 4.5 you can see the city view of this system from CodeCity and a part of blueprint complexity from Mondrian.

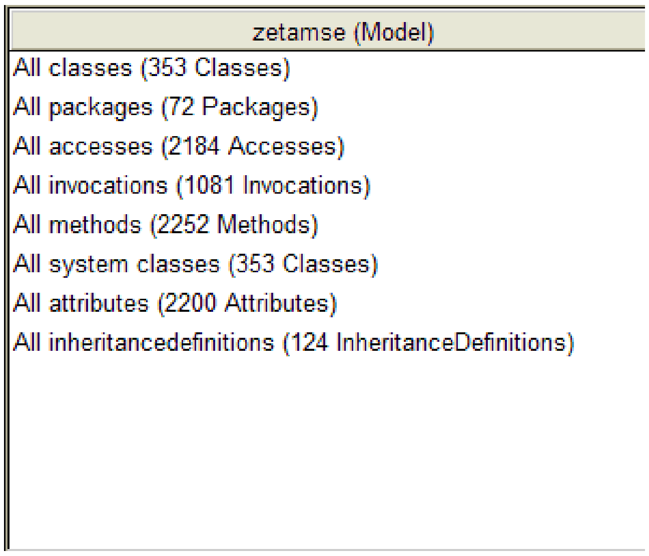

zetamse (Model)
All classes (353 Classes)
All packages (72 Packages)
All accesses (2184 Accesses)
All invocations (1081 Invocations)
All methods (2252 Methods)
All system classes (353 Classes)
All attributes (2200 Attributes)
All inheritancedefinitions (124 InheritanceDefinitions)

Figure 4.3: *The PMCS exporter results for Zeta Helpdesk - source code*

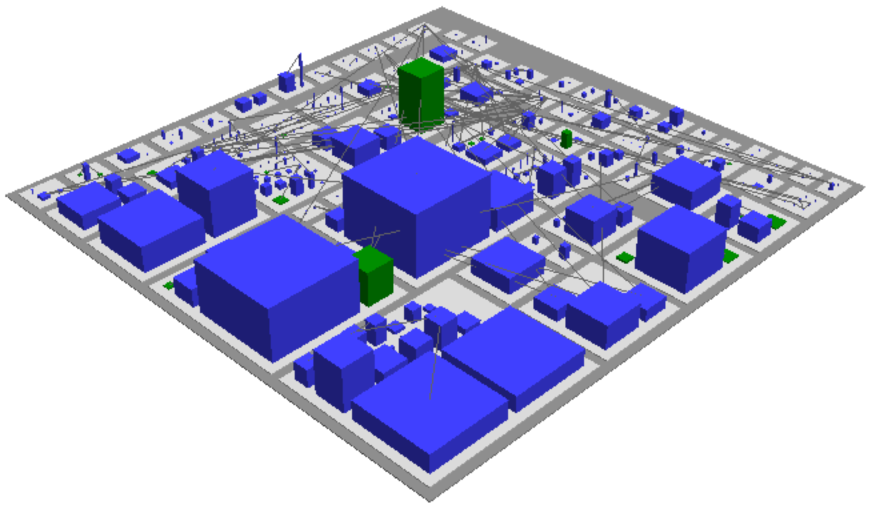


Figure 4.4: *Zeta Helpdesk in CodeCity*

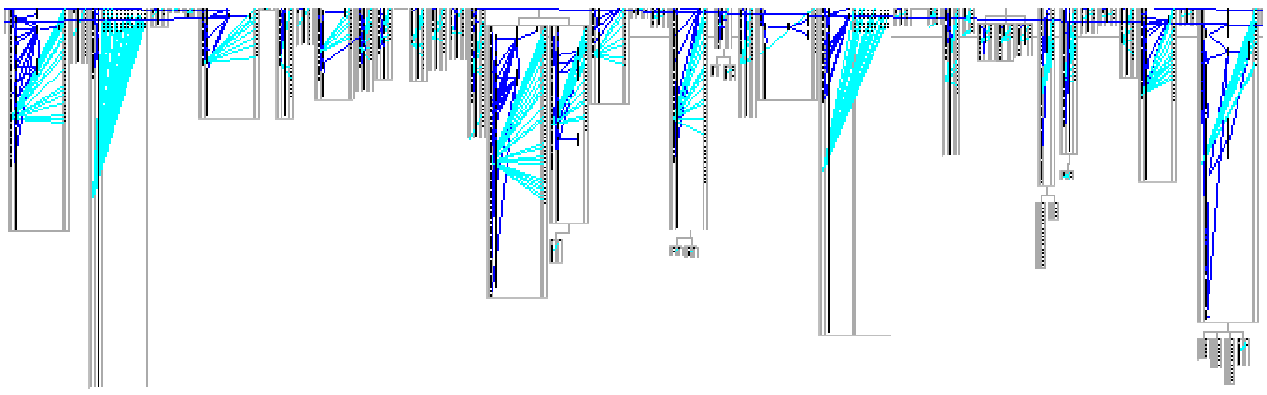


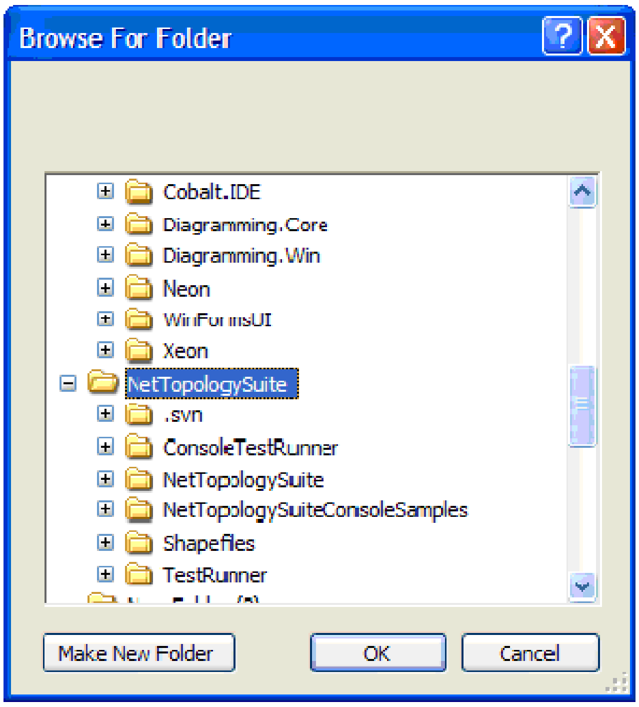
Figure 4.5: *Zeta Helpdesk in Mondrian*

4.2 Second Test - Net Topology Suite

Net Topology Suite software (http://nts.sourceforge.net/) is a GIS Solution that is fast and reliable for any king of C# systems as PocketPC and SQL Server 2005. It includes parts that integrate the capability to read/write data from file formats such as Shapefile, coordinate transformation and projection, and much more.

This software has 493 source files that PMCS reads, with 77,116 lines of code including code comments. The source code is organized into six folders with their subfolders (Figure 4.6). It contains two test runners, which are similar to software but used to test the software during its development.

PMCS reads and parses this software faster then software in the first example. In this software there is a file under the folder Net \NetTopologySuite \Utilities \RToolsUtil \StreamTokenizer.cs that has 1596 lines of code. In this file there is a method with signature NextToken(out Token token) that has 445 lines of code including comments. Parsing this method took more then 1 minute., and this happens because this method contains more accesses, invocations and local variables.

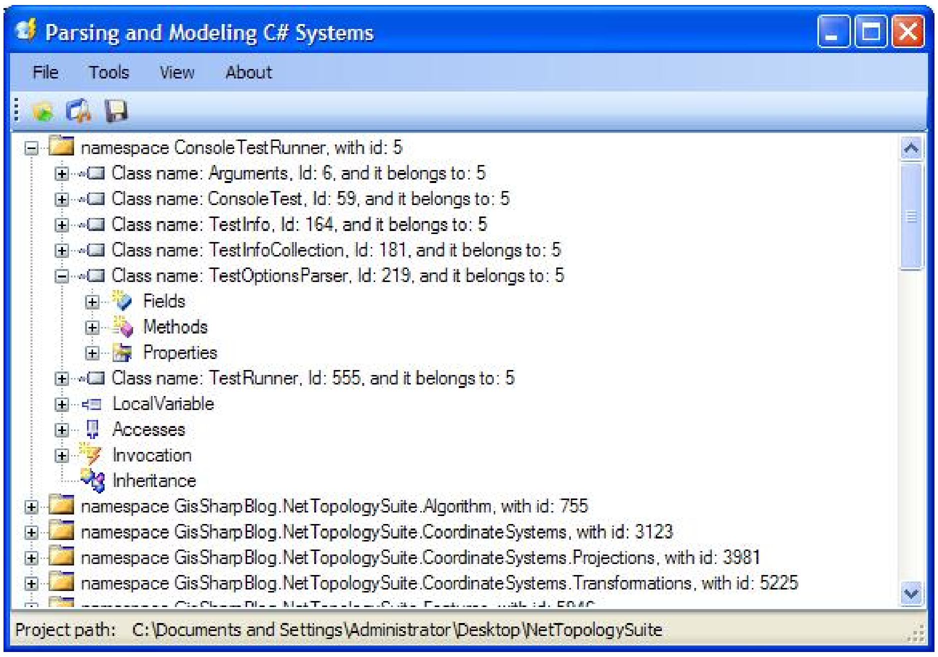

Browse For Folder

```
⊞ Cobalt.IDE
⊞ Diagramming.Core
⊞ Diagramming.Win
⊞ Neon
⊞ WinFormsUI
⊞ Xeon
⊟ NetTopologySuite
   ⊞ .svn
   ⊞ ConsoleTestRunner
   ⊞ NetTopologySuite
   ⊞ NetTopologySuiteConsoleSamples
   ⊞ Shapefiles
   ⊞ TestRunner
```

Figure 4.6: *Net Topology folder organization*

Net Topology Suite contains GUI forms and console source codes., and PMCS parsed it without problem.

As in the Table 4.1 the same entities are parsed, modeled and exported.


Parsing and Modeling C# Systems

```
⊟ namespace ConsoleTestRunner, with id: 5
   ⊞ Class name: Arguments, Id: 6, and it belongs to: 5
   ⊞ Class name: ConsoleTest, Id: 59, and it belongs to: 5
   ⊞ Class name: TestInfo, Id: 164, and it belongs to: 5
   ⊞ Class name: TestInfoCollection, Id: 181, and it belongs to: 5
   ⊟ Class name: TestOptionsParser, Id: 219, and it belongs to: 5
      ⊞ Fields
      ⊞ Methods
      ⊞ Properties
   ⊞ Class name: TestRunner, Id: 555, and it belongs to: 5
   ⊞ LocalVariable
   ⊞ Accesses
   ⊞ Invocation
     Inheritance
⊞ namespace GisSharpBlog.NetTopologySuite.Algorithm, with id: 755
⊞ namespace GisSharpBlog.NetTopologySuite.CoordinateSystems, with id: 3123
⊞ namespace GisSharpBlog.NetTopologySuite.CoordinateSystems.Projections, with id: 3981
⊞ namespace GisSharpBlog.NetTopologySuite.CoordinateSystems.Transformations, with id: 5225
```

Project path: C:\Documents and Settings\Administrator\Desktop\NetTopologySuite

Figure 4.7: *The PMCS parser results for Net Topology - source code*

The Figure 4.7 shows the entities that are parsed and modeled from Net Topology Suite system, and Figure 4.8 shows the entities imported in the MOOSE Browser. Imported *.mse files* are visualized with CodeCity and Mondrian like in Figures 4.9 and 4.10.

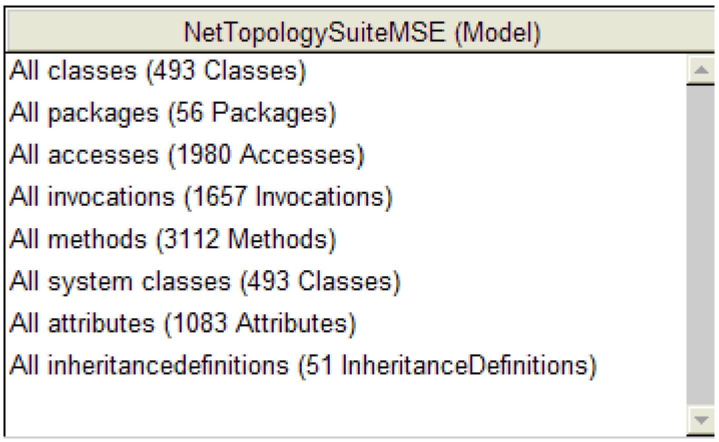

NetTopologySuiteMSE (Model)
All classes (493 Classes)
All packages (56 Packages)
All accesses (1980 Accesses)
All invocations (1657 Invocations)
All methods (3112 Methods)
All system classes (493 Classes)
All attributes (1083 Attributes)
All inheritancedefinitions (51 InheritanceDefinitions)

Figure 4.8: *The PMCS exporter results for Net Topology Suite - source code*

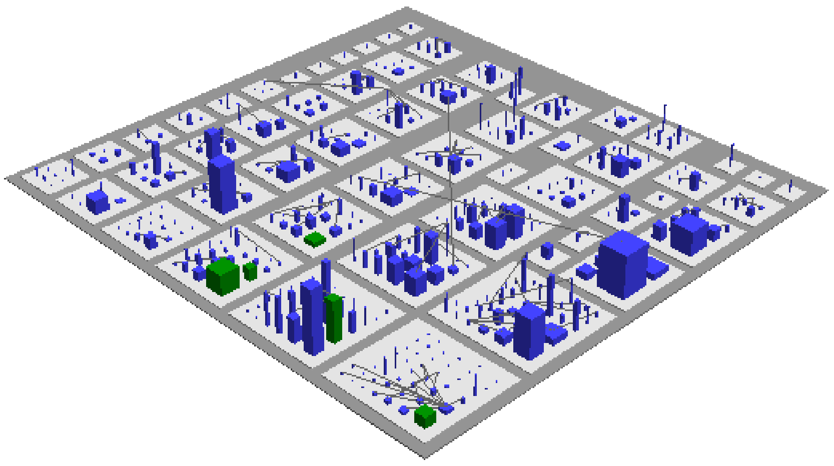


Figure 4.9: *Net Topology Suite in CodeCity*

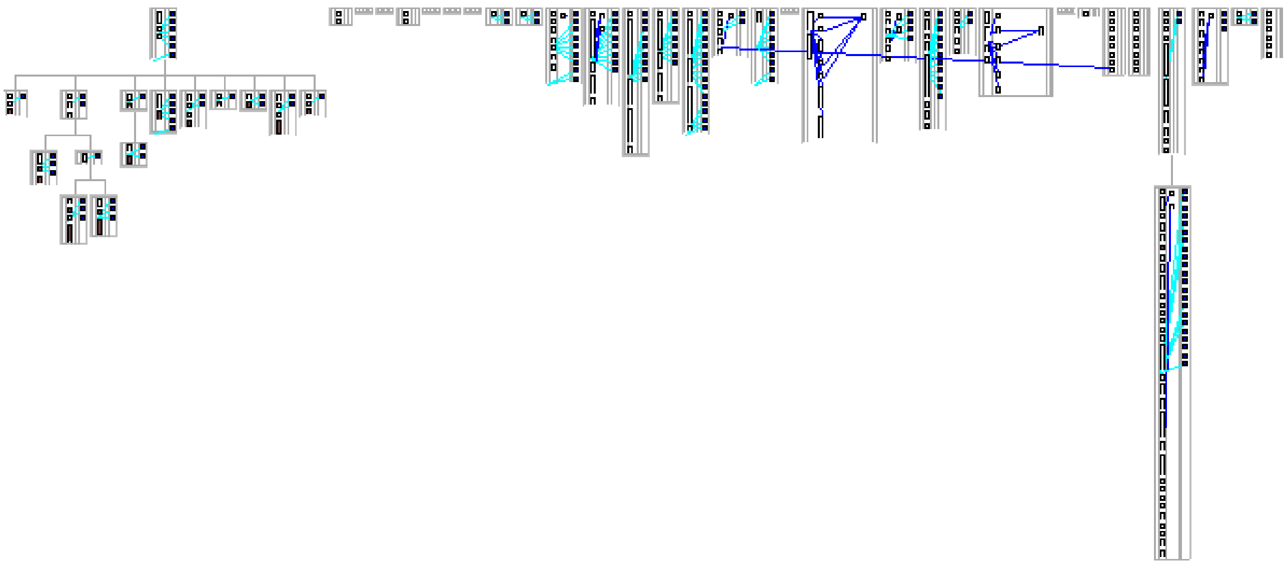


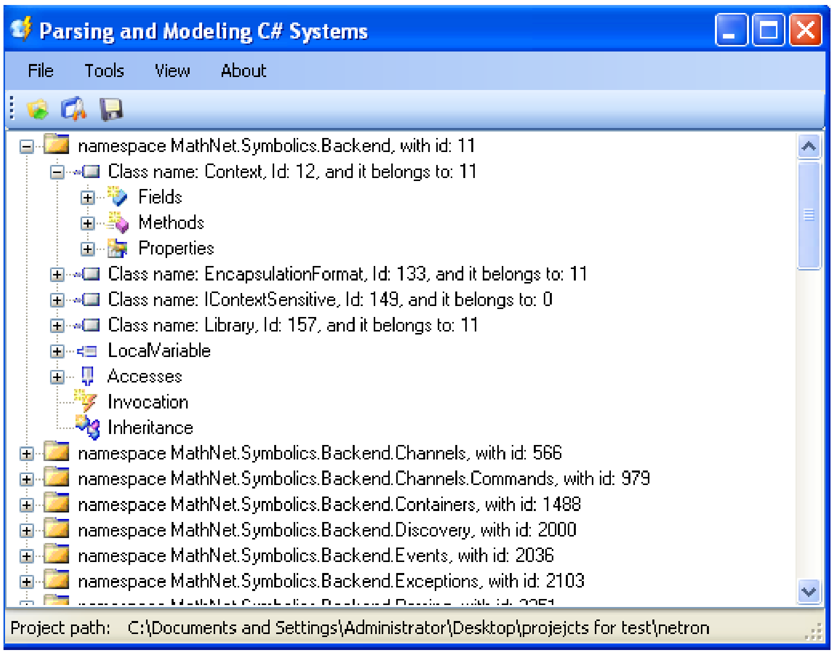
Figure 4.10: *Net Topology Suite in Mondrian*

4.3 Third Test - Netron

Netron ($http : //www.orbifold.net/default/page_id = 1322$) is diagramming or graph-drawing software. It allows creating interactive applications with minimum coding in C#. It is fully customizable via inheritances of the base classes. The shapes and functions you need for nodes or links can be compiled in separated libraries.

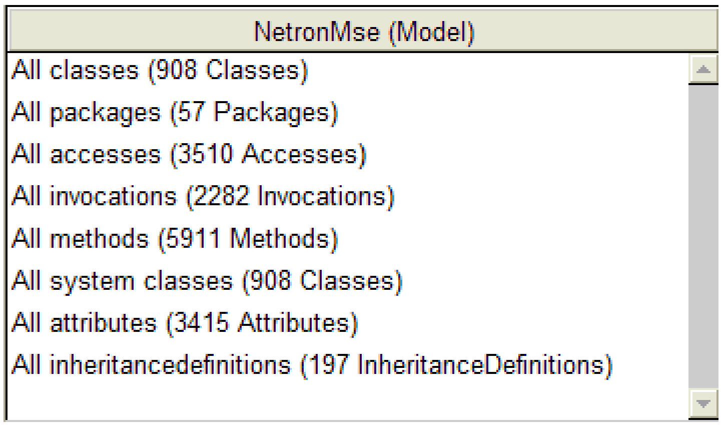
This software contains 908 source files with 128,677 lines of code including code comments. These files do not contain many lines and the largest file has 1,286 lines of code. The files of Netron software are organized with 8 main folders. The parsing and modeling process of this software is completed for 2 to 3 minutes in the optimal condition.

PMCS tool parses and models this system as in the Figure 4.11 , and exports the same entities as in the two previous examples (Table 4.1). In the Figure 4.12 these is given the screenshot of the MOOSE browser as a result of Netron *.mse file* content.



```
namespace MathNet.Symbolics.Backend, with id: 11
    Class name: Context, Id: 12, and it belongs to: 11
        Fields
        Methods
        Properties
    Class name: EncapsulationFormat, Id: 133, and it belongs to: 11
    Class name: IContextSensitive, Id: 149, and it belongs to: 0
    Class name: Library, Id: 157, and it belongs to: 11
    LocalVariable
    Accesses
    Invocation
    Inheritance
namespace MathNet.Symbolics.Backend.Channels, with id: 566
namespace MathNet.Symbolics.Backend.Channels.Commands, with id: 979
namespace MathNet.Symbolics.Backend.Containers, with id: 1488
namespace MathNet.Symbolics.Backend.Discovery, with id: 2000
namespace MathNet.Symbolics.Backend.Events, with id: 2036
namespace MathNet.Symbolics.Backend.Exceptions, with id: 2103
Project path:   C:\Documents and Settings\Administrator\Desktop\projejcts for test\netron
```

Figure 4.11: *The PMCS parser results for Netron - source code*



```
NetronMse (Model)
All classes (908 Classes)
All packages (57 Packages)
All accesses (3510 Accesses)
All invocations (2282 Invocations)
All methods (5911 Methods)
All system classes (908 Classes)
All attributes (3415 Attributes)
All inheritancedefinitions (197 InheritanceDefinitions)
```

Figure 4.12: *The PMCS exporter results for Netron - source code*

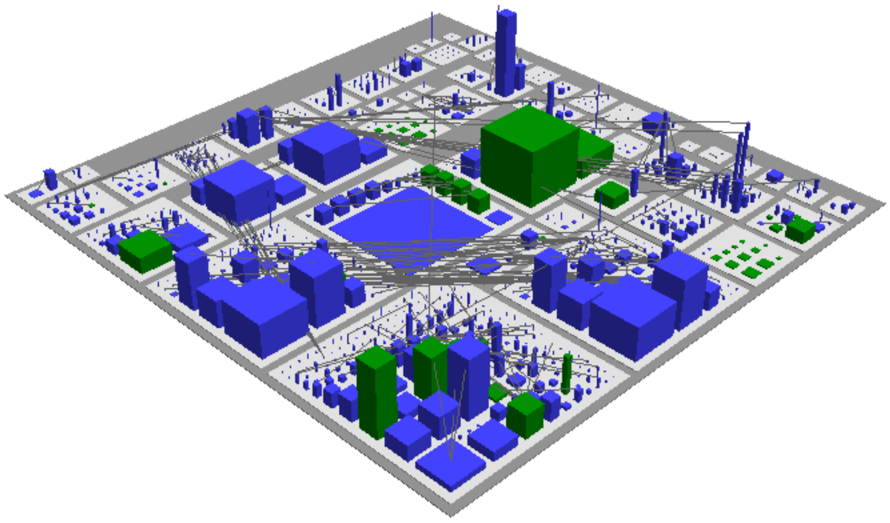


Figure 4.13: *Netron in CodeCity*

Imported *.mse files* for Netron system are visualized with CodeCity and Mondrian like in Figures 4.13 and 4.14.

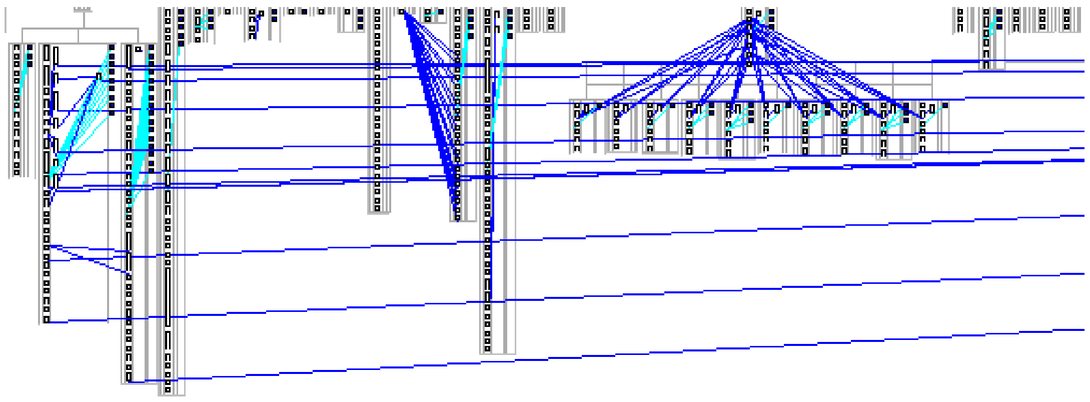


Figure 4.14: *Netron in Mondrian*

4.4 Test Results

There is no perfect software and it is not assumed that PMCS is software that is finished. It is easy to modify. During all the tests that are done and during the tests for the three examples explained above we can say that PMCS passes:

- *Functional test*:
  - Fulfills the requirements declared in objectives of the document.

  - No overload message occurs.
  - No infinite loop happens - it is a common problem when the reading file does not function correctly.

- *Stress test*:
  - Works under optimum hardware conditions (1 GB RAM and 1,7 GHz processor).
  - The large files are read and parsed.
  - Large methods are parsed by parsing invocations, accesses and local variables.

From the testing results we can conclude that PMCS is useful and can be implemented as a third party tool for MOOSE environment.

Chapter 5

In this Chapter we give a general conclusion about this document, what was the main purpose of it, possible problems and suggested solution, and future work that can be done.

The FAMIX meta-model contains entities that are:

- *Packages*,
- *Classes*,
- *Methods*,
- *Attributes*,
- *InheritanceDefinition*,
- *Invocation*, and
- *Accesses.*

These entities play the main role for analyzing and understanding a software system which is under reverse engineering.

FAMIX has four Levels of Extraction, which are based on the entities that meta-model contains. If we compare PMCS tool extracted entities with entities that are in the FAMIX Level of Extraction table (Table 2.1), we can see that PMCS tool does not cover all the entities that are in that table, but tends to create a general idea about the system.

In the Table 5.1 are shown all the entities and details that PMCS tool can extract. The rows that contain ”-” , describe that at that point PMCS does not show any result, for example PMCS tool does not export *Properties* of the C# source code.

PARSER	**MODELER**	**EXPORTER**
Namespace	Namespace	Namespace
Class	Class	Class
InheritanceDefinition	InheritanceDefinition	InheritanceDefinition
Method	Method	Method
Property	Property	-
Field	Field	Filed
Access	Access	Access
Invocation	Invocation	Invocation
LocalVariable	LocalVariables	-

Table 5.1: PMCS Entities

5.1 Conclusion

1. This document described a platform, technologies and methods, for building a parsing and modeling tool that will be useful for reverse engineering process. This project is based on a simple logic which at the same time is useful too. The whole idea was to create a parsing and modeling tool which will fit in the MOOSE analysis platform.

2. Even if for C# there are some reverse engineering tools, there does not exist any tool that is based on the source code of the system. Starting from this idea, project compares what exists and what not for C# systems, and runs a tool which parses, and models C# system source code.

3. State of the art of PMCS tool contains:

   - *Reading* part, which reads all *.cs* files in a specified folder.
   - *Parsing* part, which parses all *.cs* files, and from them extracts namespaces, classes, methods, fields, and properties. In order to be useful for reverse engineering process, PMCS tool extracts: inheritances between classes in that system, invocation between methods, and field accesses.
   - *Modeling* part, which has as input parsed entities from *.cs* file and saves them based on the *FAMIX language independent meta-model* structure.
   - *Exporting* part, which has as input parsed entities from *.cs* file and exports them in a *.mse* file.

4. And, the final contribution of this project is the part of using MOOSE environment with PMCS tool. C# was not in a list of MOOSE object oriented programming languages. With the fact of existence of a tool which parses C# systems and models them based on the FAMIX meta-model, the MOOSE analysis platform can be seen as extended in one way.

5.2 Future Work

In the previous paragraph we mentioned *Properties* which are C# special entities, and PMCS tool can parse and also model them. In the FAMIX core model there is no *property*-like entity. Based on the fact that FAMIX is extendable, this can be seen as an idea for the future work for any project that extends the FAMIX meta-model.

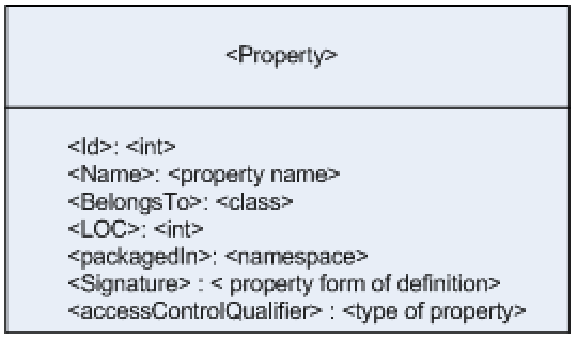

<Property>
<Id>: <int>
<Name>: <property name>
<BelongsTo>: <class>
<LOC>: <int>
<packagedIn>: <namespace>
<Signature> : < property form of definition>
<accessControlQualifier> : <type of property>

Figure 5.1: *Extending FAMIX*

Source code *Delegates*, and *Enumerators* are also C# entities which in FAMIX are not modeled as separated nodes. Like *Properties* which can be added in the FAMIX meta-model, these mentioned entities can also be part of the FAMX class diagram (Figure 5.1).

Summary

Software systems are developed to be used as long as possible. New technical and logical requirements drive the software system to modifications. Continuous modifications make software systems complex, very hard to maintain, and inefficient in some of its functions. This is basically called legacy system, which is characterized with lack of the software documentation. Information about the system has to be collected from business and development teams. Often, developers of a legacy system are not available, and business requirements change with time. The main and most structural information about a legacy system is its source code.

Reverse engineering goal is to understand the existing software system. During reengineering process there are functions that have to be reused or inherited for changes that have to be done.
Software changes are categorized as [10]:

- *Adaptive changes*,
- *Corrective changes*,
- *Perfective changes*,
- *Preventive changes.*

To achieve these changes during the reengineering process, it is necessary to capture the models of the legacy system. This is a task that directs to reverse engineering from the source code that is available. The object of this document is reverse engineering from precompiled source code that can be compiled and used as a legacy system. Sources as binary files or compiled files are not part of the document.

To capture the model without an appropriate tool is hard and time consuming, for this reason are developed Reverse Engineering tools, which help to understand the inner structure of a system. Good reverse engineering tools goes through parsing and modeling source code and visualizing its models.

MOOSE is an analysis platform that is used by different reverse engineering tools. Tools that work with MOOSE environment as: CodeCity [Wettel 2008], Softwarenaut [Lungu 2008], Mondrian [Meyer, Girba, Bergel], actually deal with systems developed in C++, Ada, Java and Smalltalk programming languages. MOOSE as an environment stands between source code of the legacy system and tools that model, visualize and make able developers to understand that system.

The FAMIX meta-model is used by MOOSE and enables communication between source code and reverse engineering tool, it is a language independent meta-model that is design to deal with object oriented programming languages. These FAMIX based models are exported from parsing and modeling tools in standard *.mse files* and imported in the MOOSE platform.

The FAMIX model has entities that are common for object oriented programming languages. In the MOOSE environment is missing C# programming language. It is missed because there does not exist a standard parsing and modeling tool that will work based on the FAMIX structure. The goal of this master thesis is to develop a tool that will parse software systems developed with C# , model the entities of a source code based

on the FAMIX meta-model, and visualize them under the MOOSE environment. Parsing and Modeling C# Systems tool or PMCS software is developed with C# and using Visual Studio 2005 with .Net Framework.

PMCS has four main modules:

- Source Files Reader module,
- Parser module,
- Modeler module,
- Exporter module.

PMCS reads C# source files with *.cs* extension. C# programming language has its own grammatical and file organization structure. On the top of the C# entities is namespace, but it can be that system contains source files that do not belong to any namespace, and they are called components. PMCS does not deal with components and during reading C# files it checks if sources are part of a namespace.

During this development we had to analyze parsing techniques. Parsing as a process is: analyzing of sequence of tokens for example words (command in C#). It is one of the components in an interpreter or compiler, which checks for correct syntax and builds data structures. There are two analyzing processes during the parsing:

- lexical analyzing process, and
- syntactic analyzing process.

In PMCS are implemented two types of parsing methods:

- Top-Down method, and
- Button-Up method.

Using these methods PMCS parses the entities that are:

- *Namespaces*,
- *Classes* - that will be simple classes, interfaces, structures,
- *Inheritance* - inheritance of classes or implementation of interfaces,
- *Behavioral Entities* - functions and procedures,
- *Properties* - entity in the C# that in the FAMIX structure do not exist,
- *Attributes* - fields,
- *Accesses*,
- *Invocations*,
- *Local Variables* - not part of the document goal.

Entities such as formal parameters, delegates, and enumerators are not parsed.

PMCS has its own part for entity modeling, and it is adapted to the FAMIX structure. On the top of the model is a namespace. Namespace has inner entities that can be namespace as entity itself, classes, inheritance definitions, accesses, and invocations. Class as an entity can be class, interface, and structure. Class entity contains inner entities such as attributes, properties and methods. Method entity contains local variable entity. Each entity has its own attributes that PMCS parses from the source code.

Access and invocation entities are parsed from methods and modeled as inner entities of a namespace. This is because of the flexibility of C# programming language to declare variables, attributes or methods without proper order. That is why PMCS when finishes parsing of all the files starts to define inheritances, accesses, and invocations.

These models are displayed in PMCS in two formats: as a tree view and as a text with better explanation.

The last module of PMCS is exporting models in *.mse files*. The models that PMCS exports are all entities of three first levels of the FAMIX Levels of Extraction. Local variable that is part of the fourth level is not exported even that it is parsed and modeled. Properties are parsed and modeled but are not exported because of no *property entity* in the FAMIX.

In this document there are tests that explain how PMCS did on them. Tests are done with C# systems source codes and the exported models are tested with MOOSE environment tools.

At the end we can conclude that:

1. PMCS is a tool that parses and models C# systems.
2. This tool fulfills the goal of the document, but, has to be upgraded to fulfill the FAMIX structure.
3. PMCS gives an extra contribution in the MOOSE environment by parsing and modeling C# properties.

The future works in PMCS and MOOSE can be:

1. Property modeling in FAMIX.
2. Parsing formal parameter,
3. Exporting formal parameter and local variables in the *.mse file*, and
4. The performance of the PMCS tool has to be modified in order to parse huge *.cs files*.

APPENDIX

Appendix A

User Manual

PMCS Parsing and Modeling Tool is designed to help you understand C# systems, by simplifying their internal content.

You will use PMCS to read C# *.cs files*, parse them, save their content in the different format, and model them. After you do modeling you can choose extracting these models in a specific file format (*.mse*). The exercises in the next section of this manual give you certain views that show the reasons of using PMCS in the software area.

This documentation will tell you what you need to run PMCS tool, and features of PMCS. Always remember that this is not a stable version of PMCS, and will not function perfect for every C# source code that you want to parse and model.

A.1 PMCS - Parsing and Modeling Tool Overview

A.1.1 System Requirements

PMCS tool can run as stand-alone application, and you need just to download it from the Web as a zip- folder ($http : //www.fpcg - ks.com/ermira/Project.htm$). At minimum you will need:

Operating System: Windows 95/98/NT/2000/XP/Vista

All users have the same responsibility while using PMCS, which means that they can use all its functions.

A.1.2 Main Menu

File

Use the *File* menu features to open a project from your computer, and to close PMCS.

Tool

Use the *Tool* menu features to choose what do you want to do with PMCS. You can choose between running the Parser and running the Exporter.

View

Use the *View* menu features to choose the form of modeling that you want to see. You can choose between TreeView and Text modeling forms.

About

Use the *About* menu features to read the PMCS main information.

A.1.3 Toolbar

The toolbar gives you quick access to many of the PMCS features that you will use most fre- quently. We can say that in the toolbar there are features of the tool that you have to choose during a whole parsing modeling and exporting C# system. As you can see in the figure 1.1 there are three features that completes PMCS tool goal.

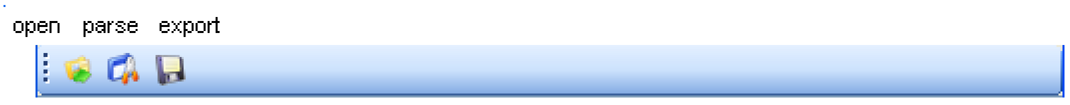


Figure A.1: *PMCS Parsing and Modeling Tool - ToolBar*

Workflow Overview

To create an *.mse file* compatible with the file format readable by CodeCity you need to:

1. Start PMCS

   You can use PMCS exe file by just running it in your Windows PC.

2. Read a Source File

   Choose File -> Open and it will appear a File Browser window. To read a C# system from your PC you have to select a folder that you are interested on.

3. Choose Parser

   Choose Tool -> Parser to process further with parsing the folder that you have chosen from the File Browser window.

4. Choose Text Modeling View

   Choose View -> Text to see the results that you have performed from the parser in the textual form, where you can read and easily understand the content of the project that you parsed.

5. Choose Exporter

   Choose Tool -> Exporter to export your parsed project in an *.mse file.*

6. Read about PMCS

   Choose About and see the main information of the tool.

7. Exit PMCS

   Coose File -> Exit and close PMCS parsing and modeling tool.

A.2 Getting Started

A.2.1 Running the Reader

Choose File -> Open. From the File Browser choose the folder to process.

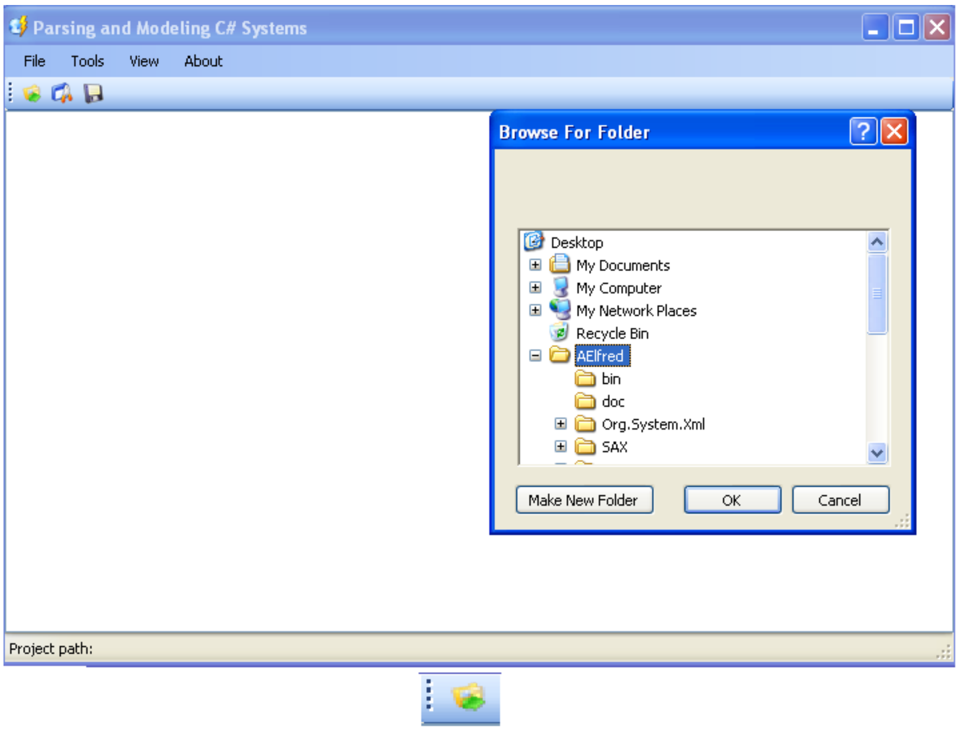


Figure A.2: *PMCS Reader*

Now when you see the path written in the bottom (as in the Figure A.3) of the tool window you are ready to run the parser.

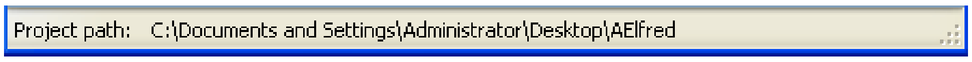

Project path: C:\Documents and Settings\Administrator\Desktop\AElfred

Figure A.3: *FIle path shower*

A.2.2 Running the Parser

Choose Tool -> Parser. This selection will start the parsing process for the folder that you have selected in the File Browser. You cannot run the parser before you choose the file from your directory list.

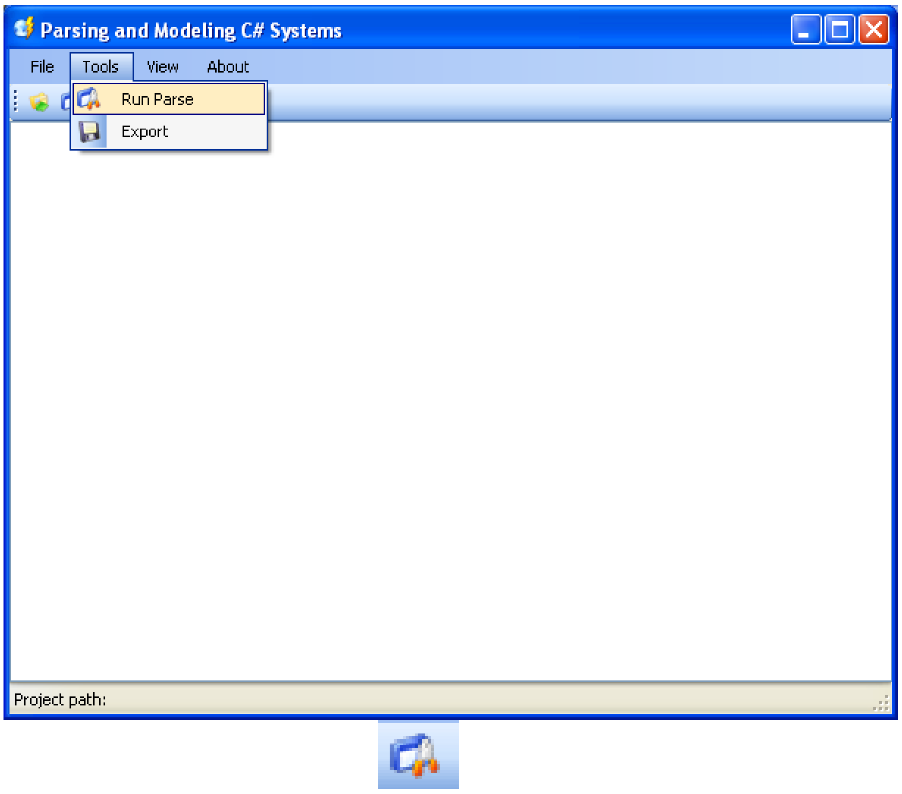


Figure A.4: *PMCS Parser*

Now when you see the progress bar running (Figure A.5), you are sure that progress of parser in that C# application have started.

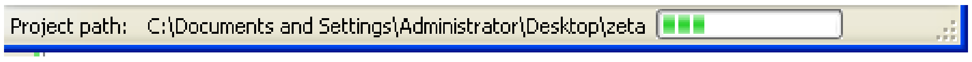
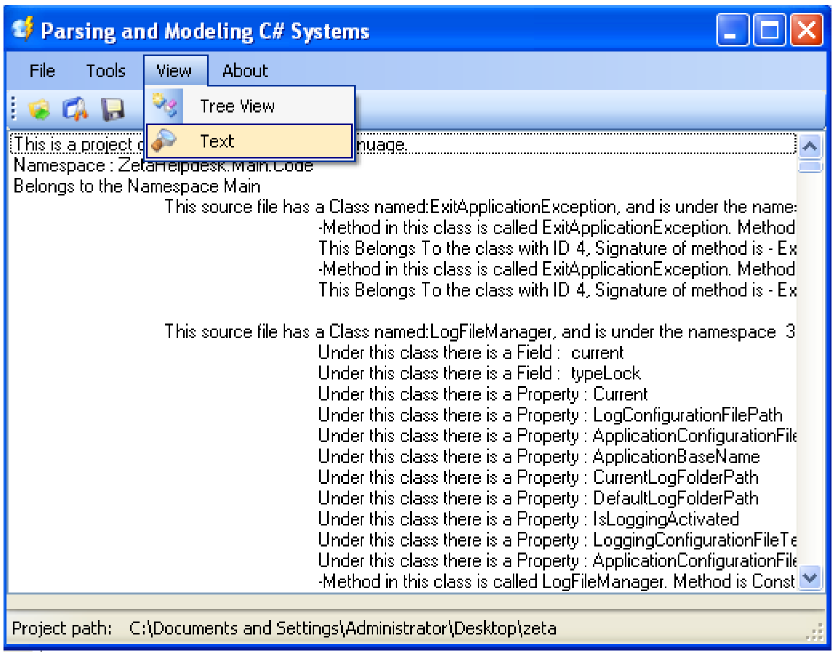

Project path: C:\Documents and Settings\Administrator\Desktop\zeta

Figure A.5: *Progress bar*

A.2.3 Change Modeling Form

Choose View -> Text to see the results that you got from the parser in the textual form. Modeling the parsed information in a textual form for users that want to read the content of the project like a specification, is the best way. With this modeler you see everything that is in the TreeView that appears after the parser runs. It does not take much to process, since it already have all information from the parser.

In the View menu item you will see the Tree View choice too. This is a process that appears every time after a successful parsing process. But if you want to go back from the textual form of modeling to the treeview form, you just have to choose View -> Tree View.



This is a project o... nuage.
Namespace : ZetaHelpdesk.Main.Code
Belongs to the Namespace Main
This source file has a Class named:ExitApplicationException, and is under the name:
-Method in this class is called ExitApplicationException. Method
This Belongs To the class with ID 4, Signature of method is - Ex
-Method in this class is called ExitApplicationException. Method
This Belongs To the class with ID 4, Signature of method is - Ex

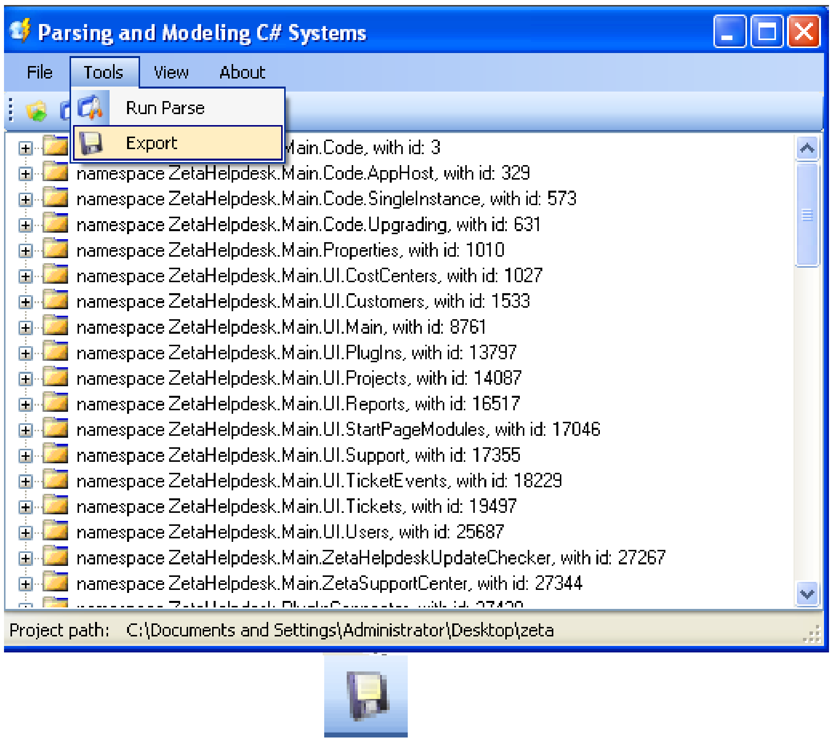
This source file has a Class named:LogFileManager, and is under the namespace 3
Under this class there is a Field : current
Under this class there is a Field : typeLock
Under this class there is a Property : Current
Under this class there is a Property : LogConfigurationFilePath
Under this class there is a Property : ApplicationConfigurationFile
Under this class there is a Property : ApplicationBaseName
Under this class there is a Property : CurrentLogFolderPath
Under this class there is a Property : DefaultLogFolderPath
Under this class there is a Property : IsLoggingActivated
Under this class there is a Property : LoggingConfigurationFileTe
Under this class there is a Property : ApplicationConfigurationFile
-Method in this class is called LogFileManager. Method is Const

Project path: C:\Documents and Settings\Administrator\Desktop\zeta

Figure A.6: *Modeler in the textual form*

A.2.4 Running the Exporter

Choose Tool -> Exporter to export your parsed project in the *.mse file*.



...ain.Code, with id: 3
namespace ZetaHelpdesk.Main.Code.AppHost, with id: 329
namespace ZetaHelpdesk.Main.Code.SingleInstance, with id: 573
namespace ZetaHelpdesk.Main.Code.Upgrading, with id: 631
namespace ZetaHelpdesk.Main.Properties, with id: 1010
namespace ZetaHelpdesk.Main.UI.CostCenters, with id: 1027
namespace ZetaHelpdesk.Main.UI.Customers, with id: 1533
namespace ZetaHelpdesk.Main.UI.Main, with id: 8761
namespace ZetaHelpdesk.Main.UI.PlugIns, with id: 13797
namespace ZetaHelpdesk.Main.UI.Projects, with id: 14087
namespace ZetaHelpdesk.Main.UI.Reports, with id: 16517
namespace ZetaHelpdesk.Main.UI.StartPageModules, with id: 17046
namespace ZetaHelpdesk.Main.UI.Support, with id: 17355
namespace ZetaHelpdesk.Main.UI.TicketEvents, with id: 18229
namespace ZetaHelpdesk.Main.UI.Tickets, with id: 19497
namespace ZetaHelpdesk.Main.UI.Users, with id: 25687
namespace ZetaHelpdesk.Main.ZetaHelpdeskUpdateChecker, with id: 27267
namespace ZetaHelpdesk.Main.ZetaSupportCenter, with id: 27344

Project path: C:\Documents and Settings\Administrator\Desktop\zeta

Figure A.7: *PMCS Exporter*

First system will ask you the path that you want to save this file, and after you choose the path and give a name to the file, you will be ready to import it in the CodeCity.

If you try to do Exporting as a process before process of *Parsing*, you will see a message that tells you that first you have to parse chosen file and then to export it.

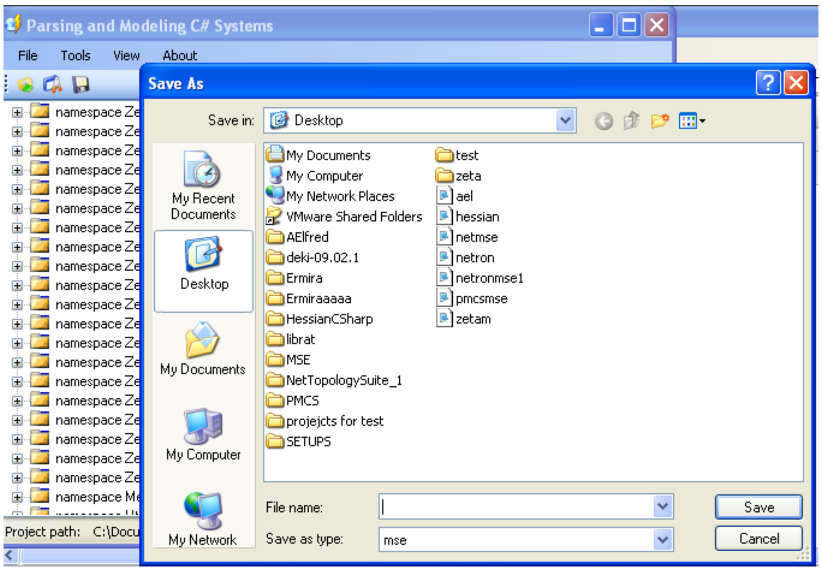


Figure A.8: *Process of saving the .mse file in a specific path*

A.3 Exit PMCS

Choose File -> Exit and you will close the PMCS tool.

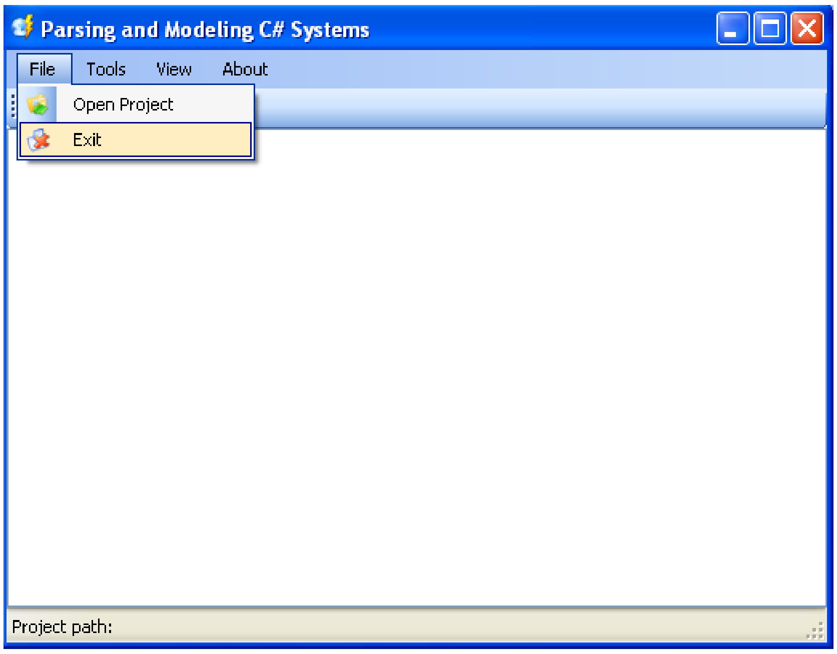


Figure A.9: *Exit PMCS parsing and modeling tool*

Bibliography

[1] E. Chikofsky and J. Cross II. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7(1):13–17, Jan. 1990.

[2] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan-Kaufmann, 2003.

[3] S. Demeyer, S. Ducasse, and S. Tichelaar. Why unified is not universal. UML shortcomings for coping with round-trip engineering. In B. Rumpe, editor, *Proceedings UML '99 (The Second International Conference on The Unified Modeling Language)*, volume 1723 of *LNCS*, pages 630–644, Kaiserslautern, Germany, Oct. 1999. Springer-Verlag.

[4] S. Ducasse and S. Demeyer, editors. *The FAMOOS Object-Oriented Reengineering Handbook*. University of Bern, Oct. 1999.

[5] S. Ducasse, T. Gîrba, M. Lanza, and S. Demeyer. Moose: a collaborative and extensible reengineering environment. In *Tools for Software Maintenance and Reengineering*, RCOST / Software Technology Series, pages 55–71. Franco Angeli, Milano, 2005.

[6] S. Ducasse and M. Lanza. The class blueprint: Visually supporting the understanding of classes. *Transactions on Software Engineering (TSE)*, 31(1):75–90, Jan. 2005.

[7] S. Ducasse, M. Lanza, and S. Tichelaar. The moose reengineering environment. *Smalltalk Chronicles*, Aug. 2001.

[8] S. Ducasse, D. Pollet, M. Suen, H. Abdeen, and I. Alloui. Package surface blueprints: Visually supporting the understanding of package relationships. In *ICSM '07: Proceedings of the IEEE International Conference on Software Maintenance*, pages 94–103, 2007.

[9] S. Ducasse and S. Tichelaar. FAMIX Smalltalk language plug-in. Technical report, University of Bern, 2001. To appear.

[10] S. Eick, T. Graves, A. Karr, J. Marron, and A. Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001.

[11] A. Kuhn, P. Loretan, and O. Nierstrasz. Consistent layout for thematic software maps. In *Proceedings of 15th Working Conference on Reverse Engineering*

(WCRE'08), pages 209–218, Los Alamitos CA, Oct. 2008. IEEE Computer Society Press.

[12] A. Kuhn and T. Verwaest. FAME, a polyglot library for metamodeling at runtime. In *Workshop on Models at Runtime*, page n10, 2008.

[13] M. Lanza and S. Ducasse. Polymetric views—a lightweight visual approach to reverse engineering. *Transactions on Software Engineering (TSE)*, 29(9):782–795, Sept. 2003.

[14] B. Lientz and B. Swanson. *Software Maintenance Management.* Addison Wesley, Boston, MA, 1980.

[15] M. Lungu and M. Lanza. Softwarenaut: Cutting edge visualization. In *Proceedings of Softvis 2006 (3rd International ACM Symposium on Software Visualization)*, pages 179–180. ACM Press, 2006.

[16] M. Meyer. Scripting interactive visualizations. Master's thesis, University of Bern, Nov. 2006.

[17] M. T. Paolo Tonella. Empirical studies in reverse engineering- state of the art and future trends. *Empir Software Eng*, 21(4):33–43, 2007.

[18] R. Wettel and M. Lanza. Visualizing software systems as cities. In *Proceedings of VISSOFT 2007 (4th IEEE International Workshop on Visualizing Software For Understanding and Analysis)*, pages 92–99, 2007.