Proceedings of the IASTED International Conference on Software Engineering and Applications, November 19-21, 2007, Cambridge, Massachusetts, USA.

A Metrics Tool for Multi-language Software

Panos Linos, Whitney Lucas, Sig Myers
Butler University
Computer Science and Software Engineering Dept.
Indianapolis, Indiana 46208, USA
Email: {linos, wlucas, smmyers}@butler.edu

Ezekiel Maier
Walker Information, Inc.
3939 Priority Way South Dr
Indianapolis, Indiana 46240, USA
Email: zmaier@walkerinfo.com

ABSTRACT: In this paper, we present a prototype tool that automates the process of detecting, gathering and visualizing multi-language software metrics at an intermediate-language level. More specifically, the current version of our tool focuses on code written using the Microsoft Visual Studio .NET software development environment. It facilitates the process of locating and extracting software metrics found at the MSIL (Microsoft Intermediate Language) level. We illustrate the basic functionality of our tool and we discuss a preliminary case study performed in order to verify its functionality and validate its usefulness. Based on the results of this study, we continue improving the tool.

Our broader research goal is to show that complexity analysis of multi-language software, when it is done at an intermediate language level, it can be as effective as when conducted at the level of each individual language. This will eventually eliminate the need for developing different syntax parsers for each programming language used to develop multi-language software. The prototype tool described in this paper is the first step towards accomplishing such an objective.

Keywords: Software metrics, program analysis, multi-language software, intermediate language.

1. INTRODUCTION

Over the last two decades, we have seen many efforts towards developing multi-language software applications (i.e. software written using a combination of programming languages). In fact, Capers Jones has documented that at least one third of the software applications in the USA today are written using two different programming languages. In addition, he estimates that ten percent of all applications incorporate three or more languages [Jones, 1998, page 321]. In order to accommodate that trend, various commercial software development environments have been introduced which facilitate the development of multi-language software. As an example, Microsoft Corporation has produced such an environment known as Visual Studio .NET designed to support a wide range of programming languages [Microsoft Press, 2001].

Such multi-language software systems evolve continuously to meet the changing needs of their end-users and application domains. In addition, it has been acknowledged that continuous maintenance activities of such systems increase their overall complexity and makes them harder to understand and modify [Linos and Schach, 1999].

Today there are many metrics tools and techniques for assessing the complexity of software applications [Fenton, Pfleeger, 1996]. However, these tools mostly concentrate on analyzing the source code written using a single programming language. On the other hand, the software engineering community has made promising efforts towards analyzing multi-language distributed software systems [Deruelle, et al, 2005]. Moreover, researchers have made progress on building cross-language analysis and re-engineering prototype tools such as PolyCARE and X-DEVELOP respectively [Linos, 1995] [Strein et al, 2006]. Also, Linos et al have created a tool that detects multi-language program dependencies (i.e. program entities and their relationships) [Linos et al, 2003].

As the size and complexity of multi-language software applications increase, the need for building more effective analysis tools for such software becomes compelling [Kontogiannis, Linos and Wong, 2006]. Therefore, this research project focuses on facilitating the process of gathering metrics that can be effectively used to measure the complexity and quality of multi-language software. More specifically, we present a proof-of-concept prototype tool that automates the detection of such selected metrics. A key aspect of our approach is the emphasis on analyzing MSIL (Microsoft Intermediate Language) code [Hanson, 2002]. Our ultimate goal is to demonstrate that this can be a promising approach which leads to an effective way for detecting software complexity metrics found in multi-language software

For the purpose of our on-going research, *we hypothesize that the complexity analysis of multi-language software, when it is done at the intermediate-language level (such as MSIL), it can be as effective as when conducted at the level of each individual language (e.g. VB, J#, C# etc.).* As a result, we can avoid the need for developing different syntax parsers for each individual programming language used to develop the software.

The remainder of this paper is organized as follows. The second section briefly describes the Visual Studio .NET software development environment. After

that, in the third section, we introduce our Multi-language Metrics Tool (MMT). In the fourth section, we present some examples of analyzing multi-language source code using MMT. The fifth section discusses an empirical study using the tool. Finally, we close by mentioning our conclusions and some of our future plans.

2. The .NET ENVIRONMENT

Microsoft Corporation has produced a software development environment known as Visual Studio (VS) .NET (dotNet) designed to support a wide range of programming languages [Microsoft Press, 2001]. This environment entails a published virtual machine standard known as *Microsoft Intermediate Language* (MSIL) [Hanson, 2002]. Figure 2.1 shows the basic process of translation that takes place in the VS .NET environment. During the first phase of this process, the source code of any language (e.g. VB, C#) used to develop the application is translated by its corresponding compiler to the MSIL format. During the following phase the MSIL format is optimized by the JIT (Just in Time) compiler and then the object code is generated in the runtime environment for execution.

In the next section, we describe our tool which parses and analyzes MSIL code produced by the VS.NET environment.

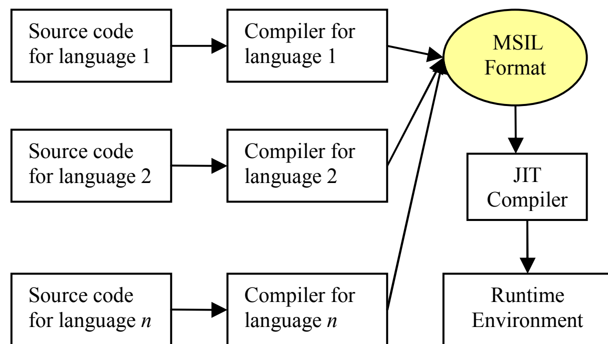

Figure 2.1: *The compilation process in the Visual Studio .NET software development environment.*

3. A MULTI-LANGUAGE METRICS TOOL

In this section, we describe our Multi-language Metrics Tool (MMT) developed at the CeASER (Center for Applied Software Engineering Research, http://ceaser.butler.edu). The purpose of MMT is to facilitate the process of gathering and presenting software metrics found in programs written using the Visual Studio .NET environment. Figure 3.1 shows a use-case diagram for MMT.

The use-case diagram in Figure 3.1 shows the various use cases including the user being able to analyze (i.e. detect and gather various metrics found in a .NET project), display file and class related metrics, view source code and export all the collected metrics to MS Excel so that bar charts and pie charts can be generated.

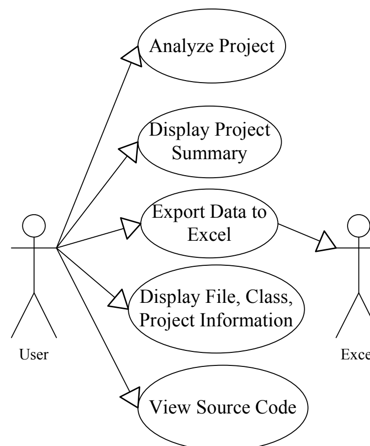

Figure 3.1: *A use-case diagram for MMT.*

3.1 Our Process

As shown in Figure 3.2, our overall process of analyzing Visual Studio .NET applications entails four stages. During the first stage, we obtain the .NET multi-language source code along with all related files that we wish to analyze. The second stage conducts the merging and disassembling of the corresponding executable files (e.g. .EXE and .DLL) by using the *Intermediate Language Merger (ILMerge)* and the *Intermediate Language Disassembler (ILDASM)* available packages respectively [Barnett, 2007] [Robbins, 2001]. As a result, the corresponding MSIL file is generated. After that our *Analysis Engine* parses the MSIL multi-language code in order to detect and extract typical metrics such as the number of projects, files, classes, variables, functions etc. So far, the MSIL metrics that can be gathered using MMT are summarized in Table 1 (note that in VS.NET a *solution* may have one or more *projects*, each project entails several *directories* or *files*, and each file may contain *classes)*.

During the third stage of our process, all the collected metrics are populated into a database (currently implemented as a simple text file). Lastly, during the fourth stage, our *Presentation Engine* displays the metrics stored in our database in a data grid window form. It also exports them into MS Excel for customized visualization (e.g. pie charts, bar graphs, etc.) In the following sections, we describe our tool which fully automates the process described in Figure 3.2.

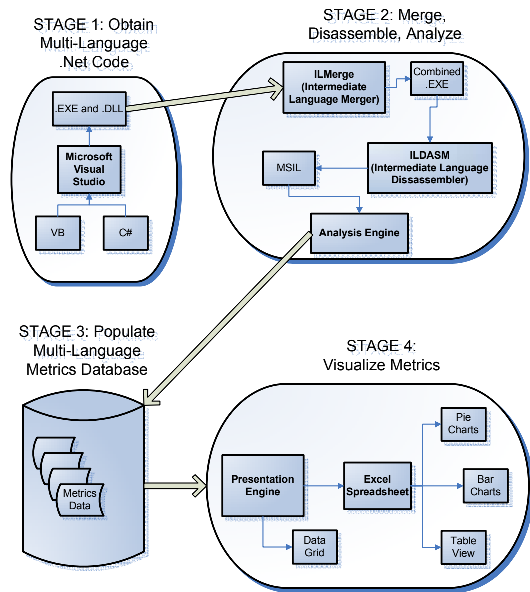

Figure 3.2: *Our process of merging, disassembling and analyzing multi-language VS.NET applications.*

3.2 Design

We have developed MMT using UML (Unified Modeling Language). Figure 3.3 shows a high-level UML class diagram used to design it. As you can see, the MMT driver class contains two classes namely the *read* and the *UIParser* classes. The *read* class is responsible to detect and extract all related metrics from the MSIL file. The *write* class populates those metrics into a database. The *UIParser* class manipulates the database entries, using the *data* and *reader* helper classes. These classes populate the *UIParser*'s display boxes with the number of projects, files and classes found in an analyzed program. After the *data* and *reader* classes are instantiated, the user can open the *UIProject, UIFile,* or *UIClass* forms. These forms inherit data sent from the *UIParser* class which was analyzed by the *data* and *reader* classes. The *UIProject, UIFile,* and *UIClass* forms each contain different metrics such as the number of projects, files or classes in a program, as well as the type of programming language, number of variables, and number of functions. In addition the *UIFile* and *UIClass* forms contain a list of metrics, ordered by the line number in which they appear in the corresponding source file.

In addition, Figure 3.4 depicts a sequence diagram used to document the "Analyze Project" use case. Similarly, Figure 3.5 presents the "Export Data to Excel" use case.

TABLE 1: *Summary of metrics detected by MMT*

Level / Metric	**Solution**	**Project**	**File**	**Class**
Number of projects	*yes*	*n/a*	*n/a*	*n/a*
Number of directories	*yes*	*yes*	*n/a*	*n/a*
Number of files	*yes*	*yes*	*n/a*	*n/a*
Number of lines of source code	*yes*	*yes*	*yes*	*yes*
Lines of MSIL code generated	*yes*	*yes*	*yes*	*yes*
Type of languages used	*yes*	*yes*	*yes*	*yes*
Multi-language code distribution	*yes*	*yes*	*n/a*	*n/a*
Number of identifiers	*yes*	*yes*	*yes*	*yes*
Identifier name, type & location	*yes*	*yes*	*yes*	*yes*
Number of read/write properties	*n/a*	*n/a*	*n/a*	*yes*
Ratio of variables over functions	*yes*	*yes*	*yes*	*yes*

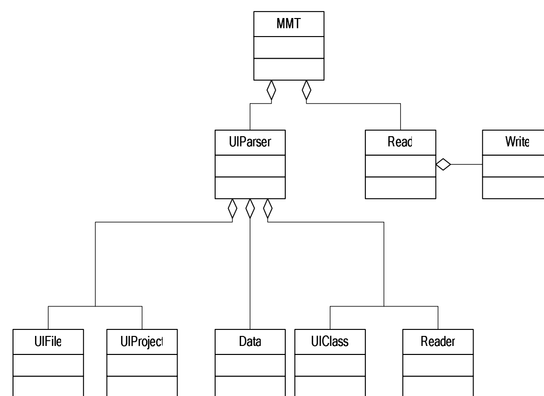

Figure 3.3: *A high-level UML class diagram for MMT.*

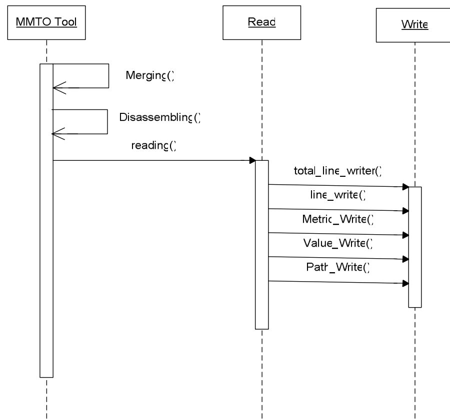

Figure 3.4: *Sequence diagram for the "Analyze Project" use case.*

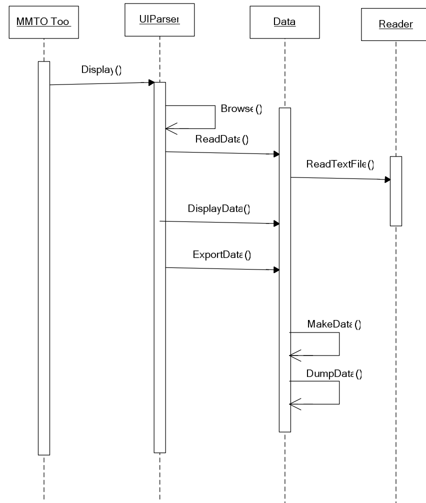

Figure 3.5: *Sequence diagram for the "Export Data to Excel" use case.*

3.3 A Tour of MMT

In this section, we describe our tool's simple user interface with a focus on its main features. The top display form is shown in Figure 3.6. Its two main features, *Analyze* and *Display,* are activated by clicking each of their respective buttons. This form also contains a *Help, Exit* and *Clear* button.

When the *Analyze* button is clicked, MMT brings up a browser window (see Fig 3.7) for locating and selecting the folder that contains the source files to be analyzed.

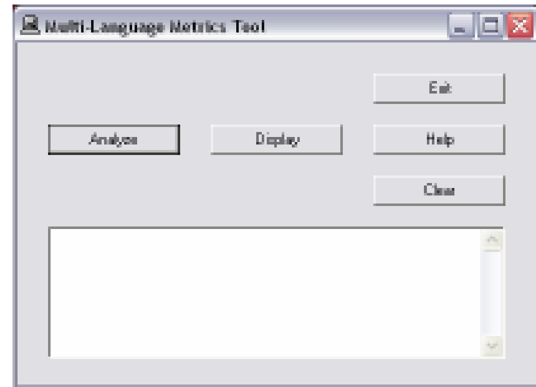

Figure 3.6: *Main display form of MMT.*

After highlighting a folder, all the .EXE and .DLL files under that folder are selected by the tool. MMT then begins the process of analyzing those files (see Fig. 3.8). The *Display* button on this form is used to display the window shown in Fig. 3.9 which is used to display all the extracted metrics that can be opened by clicking on the Browse button (shown in Figure 3.9).

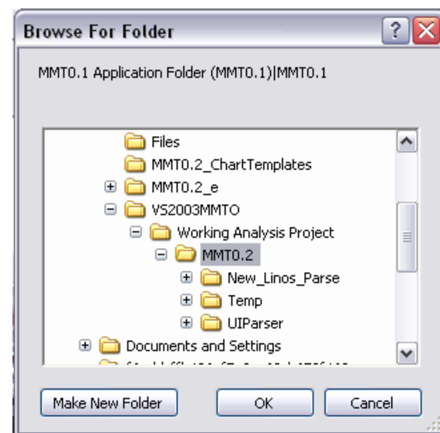

Figure 3.7: *Folder browser.*

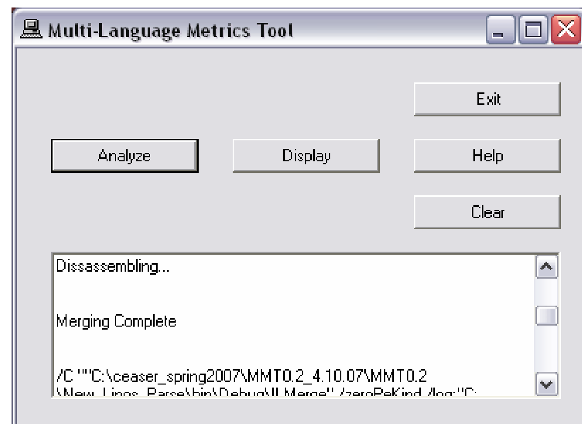

Figure 3.8: *Analysis in progress shown in text box.*

Figure 3.9: *Analysis display form.*

Typically, MMT displays a list of projects, files and classes in separate columns as shown in Figure 3.10. The user can then select a specific file, project, or class and press the corresponding *Display* button at the bottom of each column. After that, a new form is displayed as shown in Figure 3.11 which entails various related metrics.

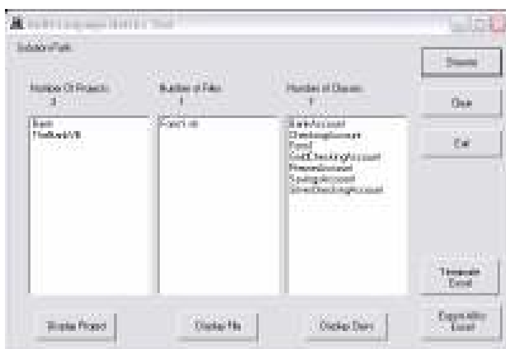

Figure 3.10: *Metrics display example.*

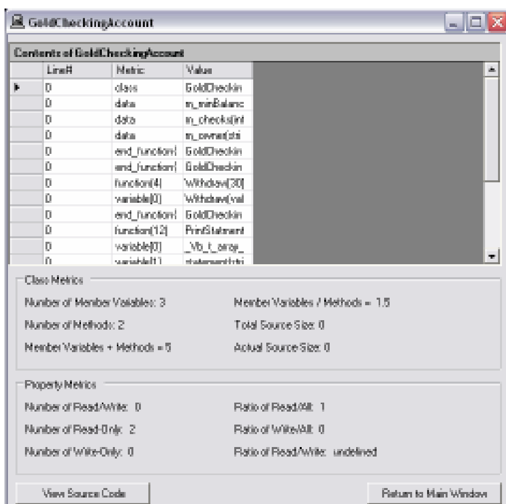

Figure 3.11: *Example displaying metrics for a class called GoldCheckingAcount.*

As shown in Figure 3.11, the form displays a list of all entities found in the program with their type, (e.g. class, variable, function etc.), the relative line number, and its corresponding identifier (e.g. name of class, variable, etc.). At the bottom of the form a summary of related metrics is displayed. This includes the total number of attributes (i.e. variables) and methods (i.e. functions) found inside a selected class as well as the ratio of attributes over methods. This form also has a *View Source Code* button which drills down to the source code of the file being analyzed when desirable.

Finally, the analysis window shown in Figure 3.9 has an *Export-All-to-Excel* button which exports all metrics data to Microsoft Excel. In that case, a spreadsheet is created and the metrics are displayed in a bar-chart form (see Fig 3.12).

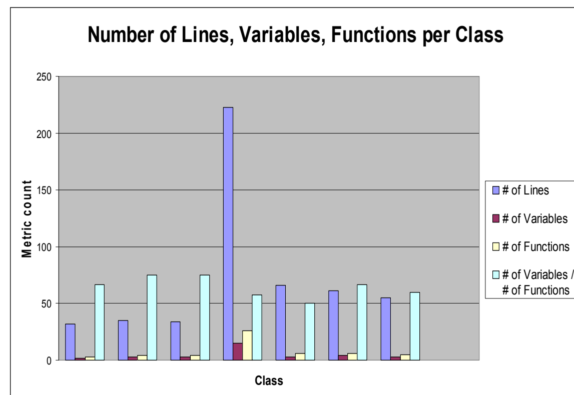

Figure 3.12: *Class metrics shown in a bar chart.*

We have created an Excel spreadsheet template designed to contain three kinds of charts that are used to display different types of metrics. More specifically, a pie chart is used to show the number of lines for each language used. The other two charts are self-expanding bar charts. We have used Excel's programmable features for creating the spreadsheet in such a way so that when data is added or removed from the spreadsheet the charts expand or collapse accordingly [Walkenbach, 2004, pp. 594-600]. These charts contain information about the number of lines in each file and the number of lines, variables, and functions found in each class. Also, *filters* available in Excel were used on the data to adjust the graphs and allow for more focused queries [Walkenbach, 2004]. For instance, a filter could be used to only show classes whose line count is less than one hundred (*filters* are used to create the bar charts shown in Figures 4.2 and 4.3 found in the next section).

4. EXAMPLES

This section describes how MMT is used to analyze the source code of three realistic examples. The first example is an open-source program called *NHibernate* [NHibernate, 2006]. The second example demonstrates the analysis of the source code of our own tool (MMT). Finally, for the third example we use the source code of

another open-source application called *Timecard CS Client* [Timecard CS Client, 2006]. Each example is described in a separate section below.

4.1 NHibernate

NHibernate is a tool which streamlines data persistence related programming tasks [NHibernate, 2006]. This is an open source application still under development. We have successfully analyzed its source code using MMT and we found that NHibernate consists of 74 projects, 486 files and 685 classes as shown in Figure 4.1.

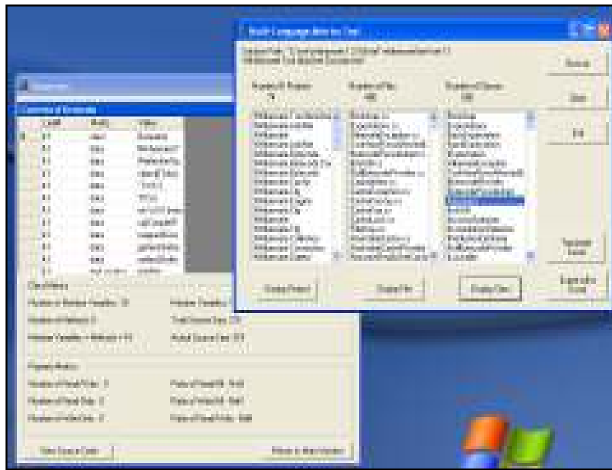

Figure 4.1: *MMT displays metrics for NHibernate.*

Next, we export the metrics data to Excel. The data is grouped into multiple sections in a spreadsheet entailing information about classes, files and directories. Figure 4.2 displays an example of the charts and data displayed after exporting the data to Excel. As we can see in Figure 4.2, the bar-chart shows the file size (i.e. lines of code) distribution for NHibernate. Excel allows us to organize the data into any desirable chart form effectively.

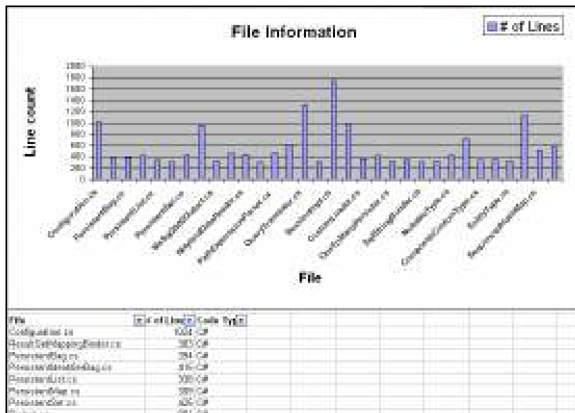

Figure 4.2: *NHibernate metrics shown in Excel.*

Similarly, Figure 4.3 depicts various metrics related to each class such as number of lines, variables and functions included in that class. Also the user can easily change the graph display criteria (ascending or descending order) by simply selecting a drop down box as shown in Figure 4.3. In this example, the MMT found a total of 134,601 lines of source code.

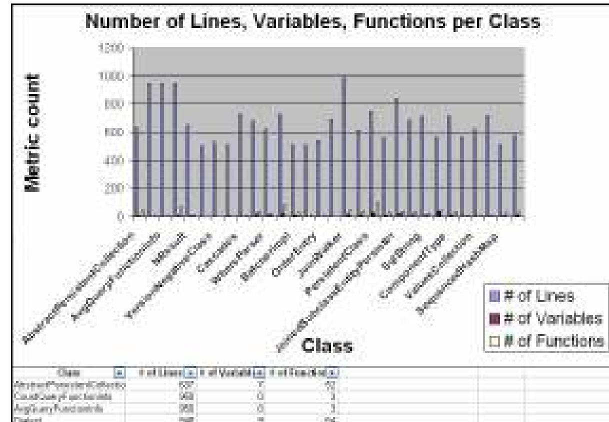

Figure 4.3: *Class metrics found in NHibernate.*

4.2 Multi-Language Metrics Tool (MMT)

In this section, we use MMT to analyze its own source code which is written using a combination of C# and VB programming languages available in the .NET environment. As we can see in Figure 4.4.1, a table with a list of file information is generated by MMT. It displays the size (i.e. lines of code) of each file along with the different type of language used (i.e. C# or VB). Also in Figure 4.4.2, there is a pie chart representing the percentage of each programming language used to write the source code. Similarly, Figure 4.5 shows class related metrics generated by MMT.

File Name	# of Lines	Language
Form1.cs	555	C#
read.cs	749	C#
Write.cs	11	C#
DClass.vb	26	VB
DFile.vb	26	VB
DProject.vb	25	VB
Reader.vb	190	VB
UIClass.vb	391	VB
UIFile.vb	220	VB
UIParser.vb	501	VB
UIProject.vb	272	VB

Figure 4.4.1: *MMT source file related metrics.*

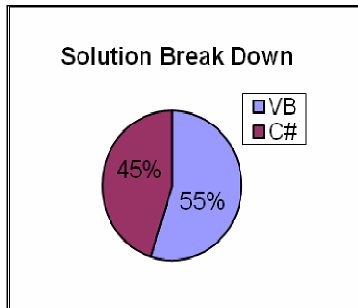

Figure 4.4.2: *MMT language distribution.*

Class Name	# of Lines	# of Variables	# of Functions
Form1	552	18	14
Read	748	17	17
Write	39	1	5
DClass	23	2	4
DFile	23	2	4
DProject	22	2	4
Reader	187	12	7
UIClass	388	17	38
UIFile	217	7	16
UIParser	498	32	49
UIProject	269	10	22

Figure 4.5: *Table view with class related metrics.*

4.3 Timecard CS Client

Our last example demonstrates the analysis of the *Timecard CS Client* application using MMT [Timecard CS Client, 2006]. This application is around 42 thousand lines of code. After analyzing its executable files (i.e. .EXE and .DLL) using MMT, we found that there are 238 classes in 34 different files (see Figure 4.6). Also, as shown in Figure 4.6, the lists displaying all project, file and class data are sorted alphabetically in order to help with searching through the different categories. It is worth mentioning that built-in VS.NET framework classes are also gathered and displayed by MMT.

We can also collect using MMT all the metrics within a project, file and/or class. In this example, we have selected the *Constant.cs* file and clicked on the *Display File* button. Then, MMT's metrics analysis window is displayed, showing the type of language(s), number of classes, source path and the actual metrics within the selected class along with their associated line numbers (see Figure 4.7).

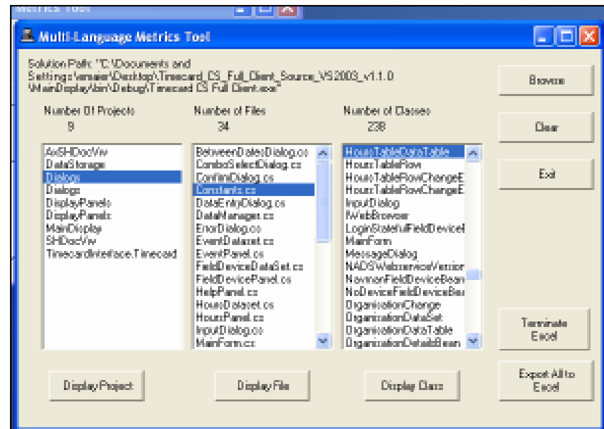

Figure 4.6: *MMT displays all related metrics found in the Timecard Client application.*

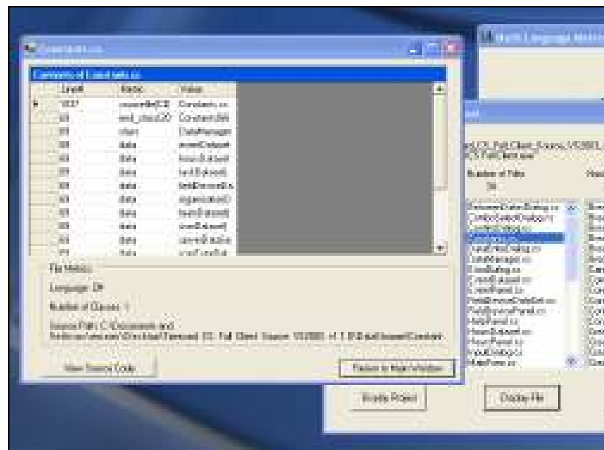

Figure 4.7: *MMT displays metrics found in the Constants.cs file.*

Next we click on the *Export All to Excel* button and take a look at some of the graphs generated from our Excel template. First we observe from the solution breakdown that the *Timecard CS Client* project is written in C# and has around 42,000 lines of code. Also, we can see from the spreadsheet generated by MMT (shown in Figure 4.8) that class sizes are of a wide range. Notice that some classes indicate that they have zero lines because they are inherited from the VS.NET framework's built-in classes. Finally, the distribution of variables and functions per class generated by MMT is shown in the bar chart in Figure 4.9.

Class Name	# of Lines	# of Variables	# of Functions
Constants	66	24	0
DataManager	733	12	26
EventsTableRowChangeEventHandler	0	0	3
EventsTableDataTable	618	14	35
EventsTableRow	590	1	38
EventsTableRowChangeEvent	614	2	2
EventDataset	128	1	10
FieldDeviceRowChangeEventHandler	0	0	3
FieldDeviceDataTable	654	15	36
FieldDeviceRow	626	1	42
FieldDeviceRowChangeEvent	650	2	2
FieldDeviceDataSet	128	1	10
HoursTableRowChangeEventHandler	0	0	3
HoursTableDataTable	731	17	38
HoursTableRow	703	1	52
HoursTableRowChangeEvent	727	2	2
HoursDataset	128	1	10

Figure 4.8: *Class metrics found in Timecard CS Client.*

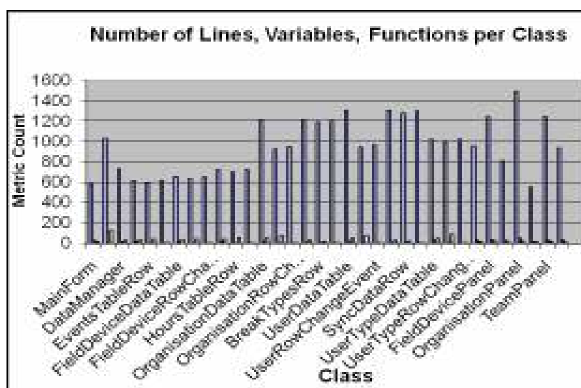

Figure 4.9: *Distribution of variables and functions per class found in Timecard CS Client application.*

5. AN EMPIRICAL STUDY

In this section, we describe an empirical study using MMT. The main objective of the study is to validate the usefulness of the tool and also to assess some of its functionality. During this exercise, we also wish to observe how the subjects detect and gather metrics found in multi-language software.

To this end, an exercise with four computer science students is conducted during this study. The students are divided into two groups (A and B) each having two members who are asked to gather related metrics from some given multi-language software. More specifically, Groups A and B were each given two programs to analyze, one using the MMT tool and one without the tool (i.e. only using Microsoft's Visual Studio). Group A analyzed a software system called *Simple Validation Tester* using the MMT tool. This system is an open-source .NET program which allows the user to implement a simple validation process. It is written using a combination of Visual Basic and C# programming languages.

On the other hand, Group B analyzed a multi-language application called *CPGui Observer* using the MMT tool. This software system is used as an example (see www.CodeProject.com) to help explain the process of handling data updates to complex controls in a Windows form. The *CPGui Observer* application is written using a combination of Visual Basic and C#. During the experiment, the students do not know anything about the internal structure of the given program to be modified other than the languages used to create it. It is also worth mentioning that both programs given to the students were around ten thousand lines of source code each.

During the experiment we gathered both quantitative (e.g. time needed to complete the exercise) as well as qualitative data by means of completing a questionnaire at the end of the exercise. Two of the students selected had some familiarity with the Microsoft Visual Studio .NET development environment, whereas the other two were not acquainted with the environment. Also, all four students were familiar with object-oriented programming practices and had an understanding of basic concepts such as classes, attributes and methods. Also, none of the students was familiar with the MMT. Therefore, before the exercise, we handed to each student a set of written instructions explaining how to use MMT and VS.NET. We also gave them a brief demonstration that helped them understand how to use the environments.

During the exercise, they were asked to find, collect and display many different metrics, such as the ratio of code written using different languages (e.g. VB vs. C#), the number of functions/methods in a class, how many projects, files and classes were found in the program, and how many lines of code for each structure.

After each student had completed the exercise using the MMT, we asked them to repeat the process of finding the given metrics without the tool (i.e. just using Visual Studio). Once each student analyzed the program using VS.NET it became evident from the data described in the next section that the metrics collected were inaccurate and took them more time to find. More detailed results of the study are described in the next section.

5.1 Results

The total time to complete the exercise (with and without the tool) can be seen in Figure 5.1. It entails all four test subjects and their time to completion. After analyzing the results it becomes clear that the MMT tool saved time with finding relevant information in a multi-language software system. In every test it became apparent that when solely using Visual Studio to gather metrics of a system, the process as a whole was more time consuming than if the MMT was used. For instance, in Figure 5.1 we can see that when the MMT was used, Project A took the subjects 4 and 13 minutes, respectively. When Visual Studio was used to analyze the test projects, it took them 10 minutes and 23 minutes,

respectively. This indicates that in analyzing a small-scale project, it took nearly double the time to gather metrics compared to using the MMT. As we mentioned earlier, both programs given to the students to analyze during the exercise were around ten thousand lines of code. We believe that if the programs given to the students were larger, the time to analyze them without the MMT would have been significantly larger.

In addition, one of the tasks given to the students during the exercise was to find the percentage of each language used (i.e. VB vs. C#) by the program to be analyzed. Only one of the four students managed to correctly answer when finding this metric manually (i.e. without the MMT). The other subjects assumed that the number of files of each programming language represented in the projects were the ratio rather than the lines of code in each of those files. Aside from that, simple errors such as miscounting the number of files and/or classes in a VS.NET solution were often occurring.

Figure 5.2 shows the correct number of files as compared to the subjects' answers to the question. It is clear from the results that our tool was accurate in counting the number of files in a project. All subjects found the correct number of files in each project when the MMT tool was used. In the absence of the MMT tool, the subjects only found accurate results in Project B. The complexity of Project A can account for the difficulty in trying to disseminate how many files actually existed in the project.

Analysis with the MMT	Subject A	Subject B
Project A	4	13
Project B	7	13

Analysis Without the MMT	Subject A	Subject B
Project A	10	23
Project B	11	16

Figure 5.1*: Time in minutes needed to complete the task with/without the tool.*

Analysis With the MMT	Subject A	Subject B
Project A	14	14
Project B	4	4

Analysis Without the MMT	Subject A	Subject B
Project A	16	9
Project B	4	4

Actual Project Values	
Project A	14 files
Project B	4 files

Figure 5.2*: Actual number of files vs. collected number of files by the subjects.*

5.2 Discussion

During this study, we also gathered some qualitative data by means of a questionnaire that the students were asked to complete at the end of the exercise. Although the results of this study are limited by the number of subjects asked to use our tool, some very useful commentary was gleaned from the user responses, particularly regarding the user interface, functionality and user-friendliness.

More specifically, the students were asked a series of questions regarding the functionality of the tool such as *how do you rate the usefulness of the MMT's Display Project button?* The subjects then gave a rating of 1 meaning *not used at all* to 5 representing *used very often.* The subjects were also asked to rank the usefulness of the generated Excel charts.

Moreover, at the end of the survey, the subjects were asked to write down their thoughts regarding the tool as a whole. Also, they were asked to propose any modifications they would wish to see made in MMT. In that context, the subjects' general response was the need for a more intuitive graphical user interface as well as additional metrics and charts to be displayed in Excel.

The results collected from the questionnaires are summarized in Figure 5.3. As you can see in that figure, the ratings of all four subjects were averaged. In the same figure one can notice that the majority of averages were above 4 (out of 5). Only two values were averaged to 3.75 which correspond to the *Display File* and *Display Class* buttons. Aside from that it is clear that the subjects found many aspects of the tool to be useful, especially that of the *Analyze* and *Display* button which are used to disassemble merge, analyze and display the metrics of multi-language software.

METRICS DATA & CHARTS	Average Rating
Class Metrics	4.25
File Metrics	4
Solution Metrics	4
Class Chart	4.25
File Chart	4.25
Solution Chart	4.25

MMT DISPLAY FORMS	Average Rating
Display project	4
Display file	3.75
Display Class	3.75
Layout of Information	4.67

Figure 5.3*: Average of ratings, based on a scale from 1 to 5, of the features supported by MMT.*

Overall it was found that the most useful features of the tool were the *Analyze* and *Display* functions. Also, it appears that exporting the data to Excel facilitated effectively the graphical and tabular presentation of the data and it was the second most liked feature of the MMT. The data organization in the display forms was well-liked because of its readability, whereas the display buttons for each project, class and file were not utilized as much as expected. Also, we noticed that the *Clear* and *Help* buttons were not used very much (or at all) during the exercise. Aside from that, certain metrics displayed were not deemed to be very useful by the subjects, such as the class and property metrics.

Finally, the majority of the subjects expressed the opinion that the MMT is a promising tool for analyzing, detecting and displaying metrics found in multi-language software applications. Some modification suggestions include an improved user interface, as well as more information provided about Excel. In summary, all subjects agreed, with a rating of 4 (and in some cases 5) out of 5, that the MMT has potential to be a useful tool in gathering and presenting metrics for multi-language software.

6. CONCLUSION AND FUTURE DIRECTIONS

Our long term research goal is to demonstrate that complexity analysis of multi-language software can be effectively conducted at an intermediate-language level. To this end, we have selected MSIL (Microsoft's Intermediate Language) as an example and we have developed a prototype tool that automates such a process successfully. We have also performed some small scale exercises using our tool in order to assess its usefulness and validate its functionality.

Based on some preliminary results gathered from this study, we are currently fine-tuning our tool's functionality by developing a more intuitive GUI. Finally, we are incorporating additional software metrics and also working on a larger scale empirical evaluation with a local software company.

7. REFERENCES

[Barnett, 2007] Michael Barnett, "ILMerge: Foundations of Software Engineering" *Microsoft Research posted 2-28-2007* *http://research.microsoft.com/~mbarnett/ILMerge.aspx*

[Deruelle, et al, 2005] L. Deruelle, N. Melab, M. Bouneffa, H. Basson, "Analysis and Manipulation of Distributed Multi-Language Software Code," *Fifth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), 2005.*

[Fenton and Pfleeger, 1996] Norman E. Fenton and Shari Lawrence Pfleeger, "Software Metrics. A Rigorous and Practical Approach," *Thomson Computer Press, 1996.*

[Hanson, 2002] D. Hanson, "Targeting the Microsoft .NET Common Intermediate Language", http://*microsoft.research.com* Nov. 2002.

[Jones, 1998] C. Jones, "Estimating Software Costs", *McGraw-Hill, New York, 1998*.

[Kontogiannis, Linos and Wong, 2006] K. Kontogiannis, P. Linos, and K. Wong, "Comprehension and Maintenance of Large Scale Multi-Language Software Applications", *Proc. 22nd International IEEE Conference on Software Maintenance, September 2006, Philadelphia, Pennsylvania, USA.*

[Linos and Schach 1999] P. Linos and S. Schach, "Comprehending Multi-language Multi-Paradigm Software," *Proc. International IEEE Conference on Software Maintenance, Oxford, England, August 30 - September 3, 1999.*

[Linos, 1995] P. Linos, "PolyCARE: A Tool for Re-engineering Multi-language Program Interactions," *Proc. 1st IEEE International Conf. on Engineering Complex Systems*, Ft. Lauderdale, FL, Nov. 6–11, 1995, pp. 338–341.

[Linos et al, 2003] P. Linos, Z, Chen, S. Berrier and B. O'Rourke, "A Tool for Understanding Multi-language Program Dependencies", *Proc. International Workshop on Program Comprehension, Portland, Oregon, May 5-11, 2003.*

[Microsoft Press, 2001] "Microsoft Unveils Visual Studio.NET Open Tools Platform", 2001 *http://www.microsoft.com/presspass/press/2001/Mar01/03-05PlatformPR.mspx*

[Nhibernate, 2006] Open Source, *http://www.hibernate.org*

[Robbins, 2001] John Robbins, "ILDASM is Your New Best Friend" *MSDN Magazine, May 2001* *http://msdn.microsoft.com/msdnmag/issues/01/05/bugslayer/*

[Strein et al, 2006] Dennis Strein, Hans Kratz, Welf Löwe, "Cross-Language Program Analysis and Refactoring", *Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM), pp. 207-216, Philadelphia, Pennsylvania, USA 2006.*

[Timecard CS Client, 2006] Open Source Software http://www.econz.org/wiki/index.php/Client_Software

[Walkenbach, 2004] John Walkenbach, "Microsoft Office Excel 2003 Power Programming with VBA*", pp. 594-600, Wiley Publishing, Inc. 2004.*