# Call Graph Based Metrics To Evaluate Software Design Quality

Hesham Abandah[1] and Izzat Alsmadi[2]

*[1]JUST University; [2]Yarmouk University*
*heshama@just.edu.jo, ialsmadi@yu.edu.jo*

## *Abstract*

*Software defects prediction was introduced to support development and maintenance activities such as improving the software quality through finding errors or patterns of errors early in the software development process. Software defects prediction is playing the role of maintenance facilitation in terms of effort, time and more importantly the cost prediction for software maintenance and evolution activities.*

*In this research, software call graph model is used to evaluate its ability to predict quality related attributes in developed software products. As a case study, the call graph model is generated for several applications in order to represent and reflect the degree of their complexity, especially in terms of understandability, testability and maintenance efforts. This call graph model is then used to collect some software product attributes, and formulate several call graph based metrics. The extracted metrics are investigated in relation or correlation with bugs collected from customers-bug reports for the evaluated applications. Those software related bugs are compiled into dataset files to be used as an input to a data miner for classification, prediction and association analysis.*

*Finally, the results of the analysis are evaluated in terms of finding the correlation between call graph based metrics and software products' bugs. In this research, we assert that call graph based metrics are appropriate to be used to detect and predict software defects so the activities of maintenance and testing stages after the delivery become easier to estimate or assess.*

**Keywords:** *Software testing, defects prediction, software metrics, coupling metrics, call graph, software maintainability*

## 1. Introduction

In software development, the human creativities and abilities play a significant role in producing and directing the software product with the help of the tools and methodologies. However, humans also form the main source of the errors and defects that occur in the software and discovered before or after the delivery to the users. Producing software and projects free of defects is impossible. However, software developers struggle to keep such possible defects at minimum. Finding and fixing the defects and errors after delivery usually cost a large amount of the project budget and resources specially if compared with detecting them earlier. As such, trying to predict early the defects is valuable specially if detected before the delivery of the software to the user where that can also help the project to success in terms of cost and quality.

The coupling metrics play an important role in many development and maintenance activities such as effort estimation, improving the quality of the software products, test planning, anticipating and reducing future maintenance needs. These metrics assess the quality of the software by supporting the quality related factors such as evaluating: error proneness, changeability, and reusability. In those aspects, most relevant tools are either available as independent tools or as part of a development environment to compute the coupling metrics statically by tracing possible problems in the source code.

Call graphs represent the relationship between the modules of the software, reflect the degree of complexity of the software, and help to find some of software metrics such as coupling metrics.

One way to reduce cost through defects prediction is in using software metrics in general and based on the call graph in particular to predict and improve possible problems in the software design and hence code.

In this research, we tried to evaluate the effectiveness and the power of call graph based metrics in the detection and prediction the defects in software products. A tool is developed to generate call graph attributes and metrics from the evaluated open source projects. We selected three applications: (JEdit 4.2, Velocity 1.4, Velocity 1.6). The selection was based on two factors: First: open source projects and second projects that include users bug reports for users actual evaluation of those software products. The call graph based metrics that are programmed and evaluated in this paper include: LOC, FanIn, FanOut, SGBR, and IFC. Some of those metrics are known and popular while others (i.e. SGBR and IFC) are implemented in our tool based on their definition in some literature or research papers. Details on those metrics will come later in this paper.

This paper is organized into the following sections; Section 2 introduces some studies and researches related to the topic; Section 3 describes the steps of the methodology; Section 4 presents the analysis and the evaluation measurements adopted in the research; Section 5 shows the results of the experiments that were conducted; finally at Section 6 the conclusions and inferences from the paper are presented.

## 2. Literature Review

Many empirical studies used the call graph model for developing ways or methods to derive dependency metrics, especially code and size metrics. Each paper then proposed ways to utilize call graph based dependency metrics to improve the software quality through providing information for defect prediction and estimation. In the following sections we will list some related work in each step that was taken in our project and developed tool.

### 2.1 Call Graph Model

Many researchers studied software modeling and found that modeling techniques were grouped into largely two categories: Graphical modeling techniques that use a diagram with named symbols that represent the components and arcs that connect the symbols and represent the relationships and other notations to represent the constrains. Textual modeling techniques that use standardized notations and keywords to define major aspects of the software products call graph.

Bohnet and Döllner (2006) [4] considered the process of extracting call dependencies as one of the most important step in the reengineering process. Therefore they built a tool for call graph extraction based on OINK framework. The tool provides in addition to the call graph, a set of hierarchal data and information about the call type, methods definitions , and output these information (data set) to imporTable formatted file.

(Telea, *et al.*, 2009 ) [10] made an enhancement for hierarchical edge bundling (HEB) technique to be used as candidate visualization technique in their framework. So they build an experiment to compare their enhancement (HEB) and Tulip graph visualization framework , several large systems (Bison , Mozilla Firefox , and OINK) are analyzed to conclude with the differences between the two visualization techniques , the result viewed that (HEB) scheme perform better in typical comprehension tasks involving.

Honar and Jahromi (2010) [5] introduced a new framework for call graph construction to be used for program analyses , they choose (ASM and Soot) as a byte code reader for their environment to store information about the structure of the codes such as classes, methods, files, and statements .

As a next step in the proposed framework three algorithms have been implemented for call graph construction (CHA, RTA and CTA), finally the authors conclude with an experimental study on a verity of source code programs in order to compare the two byte code reader and the three supposed algorithms for call graph extraction.

### 2.2 Code Metrics Extraction

Analyzing the source code for any software and extracting code metrics is considered as the main preprocess for the reengineering operation. This information provides the maintainer a clear view about the complexity and difficulties of the software, as well as it provides full insight about the way to dived the tasks of the maintainer to milestones and phases in order to start the reengineering process easily. On the other hand many researchers considered the code metrics and the information about system complexity as a good defect tracker. They setup a number of hypothesis related with defect probability and code metrics and try to prove the correlation between them, but the hottest topics in this research field is how to define the set of metrics that we can considered them as the optimal defect predictor. The researchers run many studies to define this set of metrics each of them try to view it's set as the perfect one and give justifications for their results.

As we can see, code metrics play a major role in many research fields and extracting them accurately is important, many tools deployed to handle this extraction using different approaches.

Baroni and Abreu, (2003) [2] presented a new framework for metrics extraction and modeling the extraction data using UML Meta model called FLAME, they briefly mentioned the main characteristics of FLAME in fact extraction and recommended using them when firing a new tool for metrics extraction.

The authors introduce an approach to formalize the metrics design in optimal way, FLAME functions are used to extract a well known sets of metrics which are: MOOD, MOOD2, MOOSE, EMOOSE and QMOOD metrics (Baroni & Abreu, 2003) [2].

### 2.3 Defect Prediction from Source Code Metrics and System History

A number of approaches have been deployed for defect prediction, which they depend on different criteria and information. Some researchers turn to finding bugs in software code by analyzing the source code for that software and compute its complexity. This way depends on extracting call graph based metrics from source code, then use these metrics to decide which part (module) of the software code likely to be defected. While other researchers prefer to use the system history to decide which part of them has a big probability to be defected (useful when the application has many release). They saw that the system history is more accurate for predicting defected parts of the system more than the code complexity extraction predict. For these reasons another studies drawn that support the two approaches together and use them both in finding systems bugs.

Knab, *et al.*, (2006) [7] used decision tree learners to compare the influence of different metrics used in defect prediction and predicting defect densities. They collected the data needed (source code metrics and bugs report) in the experiment from seven version of open source code for Mozilla application. They applied J48 algorithm in WEKA data miner on the data set, then they setup a number of experiments to test their supposed hypothesis on defect predicting in software parts, and conclude with that a simple tree learner can produce good results with various sets of input data, and they find underlying rule for defect prediction.

Nagappan and Ball (2005) [8] used code churn measures to introduce a new technique for prediction defect density. The idea was drawn with a hypothesis that supposed that the code changes many times in the prerelease version then it will has a big chance to be defected in the post release. The authors build an experiment on W2K3 release and the release of W2K3 service back and showed with its result that code churn is a good defect predictor. Because they noticed that the increase of the code churn measures leads to an increase on the defect density in any software. They conclude that their metrics suit which are LOC churned , Deleted LOC , Files churned , Churn count , Weeks of churn , churn count , and file count can reported if the part of the system are defected or not with accuracy of 89 % .

Software developers aim to evaluate the software in terms of cost of and quality before delivering it to customers to predict and finding bugs and defects especially for critical systems that need low percentage of defects along using the software, since the consequence will be catastrophic in terms of cost or quality.

## 3. Methodology

Our methodology consists of six main phases, as shown in Figure 1, beginning by "Metric computing tool implementation" in order to built tool that can read source code of an application to compute some metrics related to coupling measurements. After that, the second phase comes; which is "Call graph model generation" so that application model is utilized. Proceeding to the third step, which includes "Call graph based metrics calculation" and we compute some call graph based metrics for our application model. Then we go to the forth step, where we "Data set generation", in this phase we also prepare data set consisted from metrics values for each class in application. In the next phase; the fifth one, we "Data set refinement with bug report" by assigning each record (class) in data set with its number of bugs if it is exist. Finally, we finish by the phase of "Data set analysis and evaluation using WEKA" for the purpose of evaluating its quality and find the correlation between its bug and its call graph based metrics.

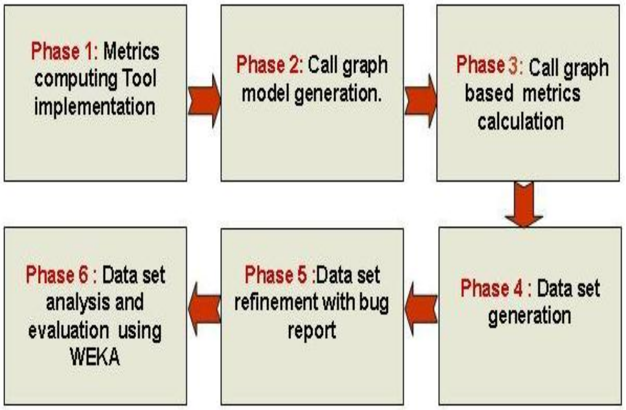

Phase 1: Metrics computing Tool implementation
Phase 2: Call graph model generation.
Phase 3: Call graph based metrics calculation
Phase 4 : Data set generation
Phase 5 :Data set refinement with bug report
Phase 6 : Data set analysis and evaluation using WEKA

**Figure 1. Overall Methodology Diagram**

The generated data set from the developed tool does not contain bug attribute for each class since the tool focus to extract the call graph based metrics. This phase is responsible to make some refinement on the CSV file. Firstly, the tool automatically fill the bug attribute filed for each class by providing it previous CSV file for the same application under investigation and contains the bug report for each class. The tool fill automatically the bug attribute of each class by mapping the name of classes between our CSV file and the pre worked CSV file. The source of previous CSV file gained from Promise Data Repository which contains several data sets in CSV (Comma-Separated values) or ARRF (Attribute-Relation File) format. These files created and prepared by researchers worked at the topic of software defects prediction and we use in our research the bug attribute for these files which related to the applications we use at our experiments. (Boetticher et al, 2007)[3].

## 4. Analysis and Evaluation

Before you begin to format your paper, first write and save the content as a separate text file. Keep your text and graphic files separate until after the text has been formatted and styled. Do not use hard tabs, and limit use of hard returns to only one return at the end of a paragraph. Do not add any kind of pagination anywhere in the paper. Do not number text heads—the template will do that for you.

After refine the generated CSV file that represent the data set of our research with bug attribute then it will be ready to analyze and evaluate using tool WEKA 3.7.5 as data miner tool. At this study we apply the following classifier algorithms such as J48 algorithm, Logistic Model Trees (LMT) Algorithm and Support Vector Machine Algorithm (SMO) classifier. The decision tree algorithms were chosen since we want to look at classifiers that were easy to understand, so we could see how valid the correlation between call graph based metrics and bug.

### 4.1 Evaluation Measures

The evaluation process of our C# testing tool depends on five matrices in term of call graph based metrics measurement. The five matrices shown in Table 1 are line of code, FanIn, FanOut, call graph based ranking (CGBR), information flow complexity (IFC). The metrics value for each type (LOC, FanIn, FanOut, CGBR, and IRC) depends on the functions that extracted from the application under investigation by which the higher metric value type achieves a higher complexity value. The values of metrics related to class level are computed by find the summation of all corresponded metrics to function level. For example: if we have 10 functions are included at such class and each function has FanIn metrics value equal 1 then the class has FanIn metrics value equal the summation of all FanIn metrics values related to functions of the class which equal to 10.

**Table 1. Call Graph Based Matrices Measurements**

| Metric Type | Metric Matrix |
|---|---|
| LOC | # of execuTable and non-commented line code for each function |
| Fan In | # of callee function list |
| Fan Out | # of called function list |
| CGBR | $(1 – d) + d * \sum i \frac{CGBR(Ti)}{C(Ti)}$ |
| IFC | IFC(M) = LOC(M) + [fan-in(M) x fan-out(M)]2 |

The five metrics we use at this research are related to size of the software or related to coupling and dependency between the components and functions of the application under investigation. LOC metric value represents the number of execuTable and non-commented lines of code. FanIn metric value for such function represents the number of function calling a given function. FanOut metric value for such function represents the number of function being called by a given function. CGBR metric is an abbreviation to call graph based ranking and proposed by (Turhan, *et al.*, 2008), this metric depends on the page ranking algorithm that used by almost of the search engines, where the ranking methodology is adopted to functions of the software. This metric hypothesis that more frequently used functions and less used modules should have different defects and bugs characteristic. The equation of the used to compute CGBR value is listed at Table 1. the value of d at the equation represents damping factor and refer to probability of such function being called or used and can be computed as the ratio of actual function calls to all possible function calls. CGBR(Ti) is the call graph based rank of module Ti which call for given function. C(Ti) is the number of outbound calls of function Ti. IFC metric is abbreviated to information flow complexity (IFC) and represents the measurement of the total level of information flow of given function. The value of this type of metric depends on the values of metrics LOC, FanIn, and FanOut for the given function and the equation required to compute the value is listed at Table 1

### 4.2 Principle Component Analysis using SPSS

The purpose of this analysis is to show how metrics in developed tool correlate to each other. Table 2 described PCA analysis for call graph based metrics in developed tool results in 2 orthogonal dimension components were identified from 5 call graph based metrics that have Eigen value more than 1. According to this, medium redundancy presented among these measures. In the Table 2, Eigen values, the variance of the data set explained by the PC (in percent), and the cumulative variance are provided for each PC. Values above 0.6 are set in boldface. The 2 PCs capture 89.963% of the variance in the data set.

**Table 2. Rotated Component Matrix for Developed Tool**

| | Component | |
|---|---|---|
| | 1 | 2 |
| Eigenvalue | 3.475 | 1.023 |
| % of Variance | 69.498 | 20.465 |
| Cumulative % | 69.498 | 89.963 |
| CGBR | .961 | -.115 |
| LOC | .930 | -.115 |
| IFC | - .025 | .966 |
| FanIn | .868 | .112 |
| FanOut | .961 | .011 |

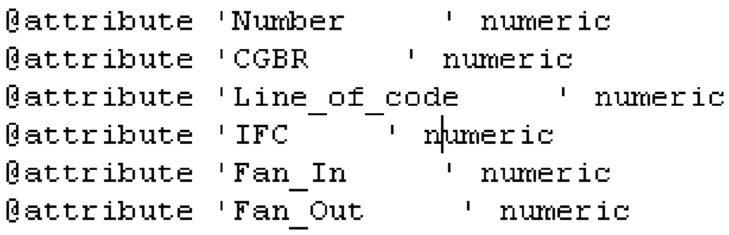
The PCs are interpreted as follows:

- PC1: CBGR, LOC, FanIn, and FanOut are coupling and size metrics. We have size and coupling metrics in this dimension. This shows that there are classes with high internal methods (methods defined in the class) and external methods (methods called by the class). This means coupling is related to number of methods and attributes in the class.
- PC2: IFC measure the total level of information flow of a module and reflect the degree of flow complexity among classes.

### 4.3 Experiments

At the first step, we collect the source code for the applications of the study, JEdit 4.2 application, Velocity 1.4 application, and Velocity 1.6 application. We enter the source code for each application to a developed C# tool in order to generate call graph model for each application. After that, the developed tool computes the call graph based metrics for each function extracted. Then compute the same metrics to classes and output the results into CSV file that represent the data set to be tested. The next step is refining the data set with bug report related to each application under investigation. Finally, evaluate the value of the metrics in terms bug and defect detection

The format of the data set should be ARRF file, since the classifier algorithms such as J48 algorithm and M5P algorithm accepts only the files with that format. The accuracy is calculated with ten-fold cross validation. The attributes of the file listed in the Figure 2.



```
@attribute 'Number      ' numeric
@attribute 'CGBR      ' numeric
@attribute 'Line_of_code      ' numeric
@attribute 'IFC      ' numeric
@attribute 'Fan_In      ' numeric
@attribute 'Fan_Out      ' numeric
```

**Figure 2. Attributes of the Data Set**

The attribute bug is classified into three categories based on the number of bugs for each class. Table 3 illustrates all types of bugs.

**Table 3. Categories of Bugs**

| TABLE I. Bug Categories | TABLE II. Metric Matrix |
|---|---|
| One | VL == 0 errors / L == 1 error / M == 2 errors / H == 3 errors / VH == > 3 error |
| Two | L == 0 errors / M == 1 - 2 errors / H == > 2 errors |
| Three | False == no errors / True == exist errors |

The results of experiments show that there is an obvious correlation between the call graph based metrics and the bug and defects of the application. Table 4 will summarize all the result of nine experiments.

**Table 4. Summary of the Experiments Results in Terms of Bug Categories**

| Bug Category<br>Application name | Category One | Category Two | Category Three |
|---|---|---|---|
| JEdit 4.2 | 80.4082 % | 81.6327 % | 86.1124 % |
| Velocity 1.4 | 60.1227 % | 72.3926 % | 80.8916 % |
| Velocity 1.6 | 67.052 % | 66.474 % | 72.8324 % |

As we show in the Figure 3 that correlation between bug and the call graph based metrics will be high when we split the bug class into small number of categories, like category three that split the bug class into two categories. So we take category three as criteria to compare the J48 classifier on the applications under investigation output to other classifier output such as Logistic Model Trees (LMT) classifier and Support Vector Machine Algorithm (SMO) classifier.

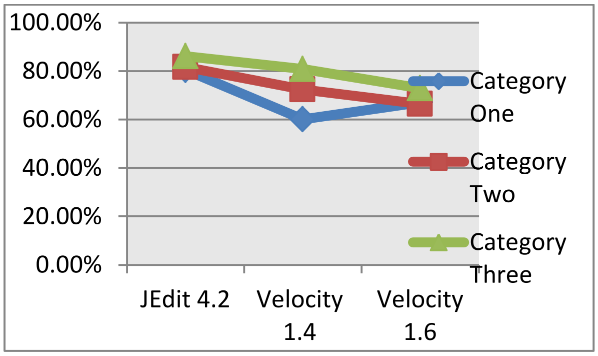


**Figure 3. Summary of the Experiments Results in Terms of Bug Categories**

As shown in Table 5 and Figure 4 the results of three classifier algorithm are approximately have similar values, where that leads us to conclude that correlation is very high between the call graph metrics and bugs of the application under investigation.

**Table 5. Summary of the Experiments Result in Terms of Algorithm Types**

| Classifier algorithm<br>Application name | J84 | LMT | SMO |
|---|---|---|---|
| JEdit 4.2 | 86.1124 % | 84.4898 % | 82.8571 % |
| Velocity 1.4 | 80.8916 % | 80.9816 % | 75.6401 % |
| Velocity 1.6 | 72.8324 % | 71.0983 % | 66.474 % |

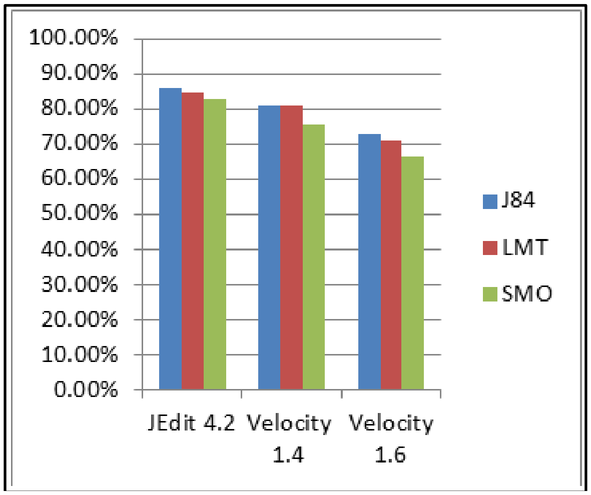


**Figure 4. Summary of the Experiments Result in Terms of Algorithm Types**

Finally, we make some normalization to our data set by excluding the non public functions such as private and protected functions from the computation of the call graph metrics for the applications under investigation, and we list the results of analysis at Table 6.

**Table 6. Summary of the Experiments Results of Data Set Excluding Non-public Functions**

| Classifier algorithm<br>Application name | J84 | LMT | SMO |
|---|---|---|---|
| JEdit 4.2 | 86.9338 % | 85.7143 % | 83.2653 % |
| Velocity 1.4 | 85.8896 % | 88.9571 % | 75.6401 % |
| Velocity 1.6 | 72.8324 % | 70.5202 % | 67.6301 % |

As we show in the Table 6 and Figure 5 the results of three classifier algorithm are approximately have similar values, where that leads us to conclude that correlation is very high between the call graph metrics that computed without non public functions and bugs of the application under investigation.

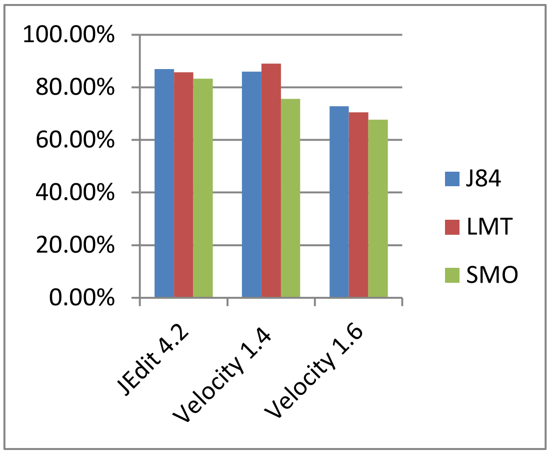


**Figure 5. Summary of the Experiments Results of Data Set Excluding Non-public Functions**

After comparing between the results in Table (5) and results in Table (6), we show that excluding the non-public functions such as private and protected functions in order to compute the call graph based metrics for the classes of the application under investigation will raise the percentage of the supposed correlation between call graph based metrics and bugs.

## 5. Conclusion

In this paper, we present the effectiveness and the power of call graph based metrics in detection and prediction the defects in software through our developed tool. We choose three application ( JEdit 4.2, Velocity 1.4, Velocity 1.6). We extract the call graph based metrics such LOC, FanIn, FanOut, SGBR, and IFC from the selected applications and evaluated their correlation according to many categories of bugs of the applications. All these experiments go together on the same direction, which is discovering how much the extracted call graph metrics are necessary and important in lightening the obstacles and problems of software that may arise after delivery phase, which is an expensive and time consumer phase. Therefore, it will be more effective to predict them and find their solutions earlier, if they occur at any time.

The results of our research improve the hypothesis of correlation between call graph based metrics and bugs in software design. The highest percentage of correlation was shown in results of the analysis JEdit 4.2 application using J48 algorithm classifier with metric correlation 86%, while the metric correlation resulted in analysis Velocity application with its versions 1.4 and 1.6 was 81% and 73% respectively. In addition, the results show that correlation between bugs and the call graph based metrics will be high when we split the bug class into small number of types, like category three that split the bug class into two types. In addition, the results show that excluding non- public functions such as private and protected functions in order to compute the call graph based metrics for the classes of the application under investigation will raise the percentage of the supposed correlation.

By this approach, we proved that call based metrics are appropriate criteria for helping the maintenance and developing stages to be more effective stages and less costly at the same time, especially for those systems that are very complex and hardly to understand.

## References

[1] N. Azeem and S. Usmani, "Defect Prediction Leads to High Quality Product", Journal of Software Engineering and Applications, vol. 4, no. 11, **(2011)**, pp. 639-645.

[2] A. L. Baroni and F. B. Abreu, "A Formal Library for Aiding Metrics Extraction", 4th International Workshop on Object Oriented Reengineering WOOR2003 at ECOOP2003, **(2003)** Dramstandt, Germany.

[3] G. Boetticher, T. Menzies and T. Ostrand, "PROMISE Repository of empirical software engineering data", West Virginia University, Department of Computer Science, **(2007)**, http://promisedata.org/?cat=11 repository.

[4] J. Bohnet and J. Döllner, "Visual exploration of function call graphs for feature location in complex software systems", Proceedings of the 2006 ACM symposium on Software visualization SoftVis 06, vol. 1, **(2006)**, pp. 95 – 104, ACM Press, http://portal.acm.org/citation.cfm?doid=1148493.1148508.

[5] E. Honar and M. Jahromi, "A Framework for Call Graph Construction", Student thesis At School of Computer Science, Physics and Mathematics, **(2010)**.

[6] M. Kaur, P. Batra and A. Khare, "Static Analysis and Run-Time Coupling Metrics", International Journal of Information Technology and Knowledge Management, vol. 3, no. 2, **(2010)**, pp. 707-710.

[7] P. Knab, M. Pinzger and A. Bernstein, "Predicting defect densities in source code files with decision tree learners", Proceedings of the 2006 international workshop on Mining software repositories, **(2006)** May 22-23, Shanghai, China [doi>10.1145/1137983.1138012].

[8] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density", Proceedings of the 27th international conference on Software engineering, **(2005)** May 15-21, pp. 284-292, St. Louis, MO, USA [doi>10.1145/1062455.1062514].

[9] W. Prins and P. Darbyshire, "Call-Graph Based Program Analysis with .Net", In Procs of the IRMA International Conference, **(2007),** pp. 794-798.

[10] A. Telea, H. Hoogendorp, O. Ersoy and D. Reniers, "Extraction and visualization of call dependencies for large C/C++ code bases: A comparative study", 2009 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, **(2009)**, pp. 81-88, IEEE, http://ieeexplore.ieee.org/lpdocs/epic03-/wrapper.htm?arnumber=5336419.

## Authors

### Hesham M Abandah

Hesham Abandah is a master student graduate in the computer information systems department at Yarmouk University. He works in Jordan University of Science and technology (JUST). His research interests are focused on software engineering in general, software testing and metrics in particular.

### Izzat M Alsmadi

Izzat Alsmadi is an associate professor in the department of computer information systems at Yarmouk University in Jordan. He obtained his Ph.D degree in software engineering from NDSU (USA). His second master in software engineering from NDSU (USA) and his first master in CIS from University of Phoenix (USA). He had a B.sc degree in telecommunication engineering from Mutah university in Jordan. He has several published books, journals and conference articles largely in software engineering different fields.