MAINTAINABILITY DYNAMIC METRICS DATA COLLECTION BASED ON ASPECT-ORIENTED TECHNOLOGY

Amjed Tahir , Rodina Ahmad and Zarinah Mohd Kasirun
Software Engineering Department
Faculty of Computer Science and Information Technology
University of Malaya
Email: amjedtahir@siswa.um.edu.my ; rodina@fsktm.um.edu.my; zarin@fsktm.um.edu.my

ABSTRACT

The increase dependence on software aspects has led the society to emphasize the importance of software quality and metrics. At present there are two categories of quality metrics; dynamic and static. Although dynamic metrics can provide a clearer insight into the software quality issue; it is observed that static metrics are often used for such a purpose. This is due mainly to the technical difficulties associated with the collection of dynamic metrics. One of the known issues when dealing with dynamic metrics is the need for instrument code by inserting points for data collection. This is a very tedious and counterproductive task. Aspect-Oriented Programming (AOP) is a promising technology that is currently used to add cross-cutting concerns to the software applications.AOP can be easily used to transparently instrument the code at compile-time. This work proposes AOP as a technique that can be used for collecting software maintainability dynamic metrics data. Therefore, an AOP-based framework for collecting dynamic coupling metric has been designed, implemented; and evaluated. Evaluation results showed that the framework is a reasonable approach for collecting a maintainability dynamic metrics. This approach allowed for the total separation of the metric's code (the metric aspect) from the application's source code. Thus, the AOP-based framework provides an effective way for the transparent collection of a maintainability dynamic metrics data. The designed framework can be extended to fit other kinds of quality dynamic metrics.

Keywords: Software Quality, Software Maintainability, Dynamic Metrics, Dynamic Coupling, Aspect-Oriented Programming

1.0 INTRODUCTION

Quality is currently considered the most important distinguishing factor between similar software products. The quality of the software system can be gauged from different aspects: project, process and product. Typically, the product quality is the main concern of the system end users; and its metrics are closely related to the software quality characteristics such as reliability and usability. In the present time, maintainability has becomes a key element in the development approaches. Literatures show that software maintenance takes around 70 percent of the total resources and 40-60 percent of the total life cycle efforts [1]. However, one of the factors that affect the software maintainability is coupling. Coupling describes the dependability between objects. Coupling is shown to have direct impact on software maintainability. High level of coupling means a high level of complexity; which in turn may lead to difficulties in the maintenance and reusability of the components.

Software quality deservedly acquires more attention nowadays. Many software organizations have been intensively looking for effective ways to improve the quality of their software products. One method was to give an indicator about quality is to quantify or measure some aspects of the software. This is exactly the role of software measurement and metrics. Without measuring the quality of the software, it would be difficult to determine the software strengths and weaknesses. Measurement is the fundamental phase of software quality. Software metric is any measurement that is used to quantify any characterise software projects, processes or products. Conversely, software quality metrics are subsets of software metrics that are concerned about the system quality aspects. Quality metrics therefore, provide quantitative measures of the software quality.

There are two types of metrics that could be used to measure the end-product's characteristics the static and the dynamic metrics. However, some metrics require the program to be executed (or or execution simulation). Other metrics in contrast do not need the software to run; they can be measured without executing the program. The first one is known as dynamic metrics and the second is the static metrics. Static and dynamic metrics provide important quality indicators for the end software product. Nevertheless, it is observed that many software projects focus entirely on static metrics although dynamic metrics can provide a deeper insight into the software quality issue [2]. In addition, dynamic metrics can capture and reflect dynamic behaviour of the software better. This mainly results

from the technical difficulties associated with the collection of dynamic compared to static metrics. In contrast to static metrics, dynamic metrics requires the instrumentation of the software code. . In addition dynamic metrics collection would not be possible without program execution. It is more difficult to measure the external attributes of the system using the dynamic metrics than the attributes that are measured by using static metrics [3].

As there are many difficulties associated with collecting dynamic metrics, we started looking for a solution to make the use of dynamic metrics easier and more practical. . Aspect-oriented programming (AOP) is a relatively new technology that was initially introduced in 1997. AOP is used currently to add cross-cutting concerns1 to the software applications. It is mainly used to add these cross-cutting concerns to the executed program at the run-time or executing time. AOP can be used to transparently instrument the code at compile-time. As noted in [4], the goal of AOP is to make it possible to deal with cross-cutting aspects of the system's behaviours. However, both dynamic metrics and AOP deal with a software system in run-time [5] [6]. This work investigates the potential of AOP in facilitating the collection of maintainability dynamic metrics.

2.0 BACKGROUND

Nowadays, quality plays a central role in the competitive market. Higher customer satisfaction may be achieved by high quality end products. The users always expect a high quality software product. As mentioned by [7], users are mainly interested in using the software product, and they normally evaluate a system's performance rather than the internal development process. This will lead us to focus on the end product.

In practice, software quality is a mix of factors that is different from applications, programs and customers (who request them). Furthermore, these factors are diverse depending on the type of application. However, the quality factors can be a feasible approach that quantifies some aspects of the software quality such as maintainability, reliability, usability and integrity. As this work focuses on software maintainability, the next section elaborates on software maintainability.

2.1 SOFTWARE MAINTAINABILITY

Software maintainability is quite complicated and a dependent factor. Maintainability, nowadays, becomes a key element in any software development methodology. Global business changes very fast. Therefore, software that controls these businesses also needs to frequently adapt to these changes. This means that the software needs to be adaptable and easy to maintain. Literatures show that software maintenance takes around 70 percent of the total resource and 40-60 percent of the total life cycle efforts [1] [8]. Software maintainability can be defined as: "The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment" [9].

Many software elements are involved in the maintainability process of a system. Different focus identifies the maintainability aspects of the system. There is also a difference between the maintainability of different programming paradigms. The maintainability of object-oriented-based program, for instance, is different from the procedural and functional programs. In object-oriented programs, different principles, structures and roles are followed compared to other programming paradigms. However, the main goal of software development is still the same for all different paradigms i.e. to produce a highly maintainable system which is one of the quality's attributes of any software system.

The ISO 9126 model shows software maintainability as a super category (head) which includes sub categories. It's divided into four sub-categories. Maintainability is one of six key factors that include: functionality, reliability, usability, efficiency, and portability. The four core sub-factors of software maintainability are: analyzability, changeability, stability, and testability.

One of the factors that affect the maintainability is the coupling. Coupling is the interconnection between objects. Coupling was shown to have a direct impact on quality attributes of the software. Coupling in object-oriented related directly to the maintainability, understandability and testability; which affect the overall quality of the system [10]. However, one of object-oriented goals is to produce a system with low level of coupling between components. This

---

[1] Areas of functionality (features) of the software to be implemented separately and which influence (crosscut) other concerns (e.g., security, logging…etc.)

will lead to a reusable component and reusable system. Also, it will improve the modularity of the system which leads to an improvement of the maintainability and the testability of the system. For all these reasons, coupling has been chosen to test our approach (coupling effect the maintainability of object-oriented system). The greater the coupling between objects, the harder the maintainability of the system [11].

However, it is important to be reminded that high maintainable software will demand less rework and maintenance activities and, therefore, decreasing overall cost of the product at the end of the life cycle [1].

2.2 SOFTWARE QUALITY METRICS

Software metrics become an integral part of the development process [2]. The quantification of software quality aspects of the product is known as software quality metrics. Without measuring the quality of the software, it is difficult to determine the strengths and weaknesses of the intended software.

There are two categories of quality metrics: in-process and end-product metrics [12]. In-process metrics are those that can be used for measuring and improving the software development process. End-product metrics are the metrics that evaluate the final product characteristics (such as the performance).

Furthermore, there are two main types of metrics that gauge different aspects of the software system: static and dynamic metrics. The metrics that are used to check the static attributes of the software such as the length of code and the complexity of the program are known as static metrics. Static metrics are invariant and they do not change if the program works or not. On the other hand, the dynamic metrics focus on verifying the dynamic attributes and behaviour of the system such as the maintainability, usability and reliability of the system. Dynamic metrics are the class of the software metrics that capture the dynamic behaviour of the software. They are usually obtained from the execution traces of the code, the executable models, or by simulating the execution.

There are a number of static metrics that have widely been used. The Lines of Code (LOC) is one of the famous static metrics that is used to measure the program size by counting the physical or logical lines of code. The complexity metric “Cyclomatic Complexity” is widely used for assessing the maintainability and testability of software system. Chidamber and Kemerer [13] introduced one of the well-known object-oriented static metrics (besides other metrics) called Response for Class (RFC). The RFC metric measures the number of methods that can potentially be executed in response to a message received by an object of the class. Yacoub et al [5] concluded in the end of their study on static and dynamic metrics of object-oriented based systems that dynamic metrics can be used to measure the actual run-time properties and attributes of a software programs as compared to the expected properties measured by static metrics. Currently, there are a number of dynamic metrics which are implemented for such a purpose. The Mean-Time-To-Failure (MTTF) and the Mean-Time-Between-Failures (MTBF) metrics, for instance, are used to measure the system’s reliability which is part of the software quality. However, all these metrics need to be collected at the execution time and they cannot be measured without running the program or the software.

Different types of metrics (static and dynamic) are related to different quality factors and attributes. For instance, static metrics help assessing the size and the complexity of the program while dynamic metrics help to evaluate the efficiency and the reliability of the software [14]. Based on their experience, Dufour et al. [15] note that developers focus more on static rather than dynamic metrics at least partly because static metrics are much easier to compute as compared to dynamic metrics. The collection of dynamic metrics usually requires the instrumentation of the software code by inserting additional lines of code for data collection purpose.

Dynamic metrics are computed based on the data collected during execution of the system, and therefore directly reflect the quality attributes of the system [2].These dynamic metrics are normally collected by inserting code into the source program. However, static metrics should be used alongside dynamic metrics to give better indicator of the quality. Using of static metrics alone may not be efficient to capture the dynamic or runtime behaviours of an application [16].

In the previous sections of this paper, areas of software maintainability and software metrics and measurement (focusing on maintainability metrics) are covered by providing examples of static and dynamic metrics. The following section explains and provides a brief clarification about the AOP technology itself.

2.3 ASPECT-ORIENTED PROGRAMMING (AOP)

AOP is a relatively new technique that was introduced for the first time in 1997 by Gregor Kiczales and his research group of Xerox Palo Alto Research Center (PARC). AOP aims mainly at capturing all of the important elements

(called aspects) of a system's behaviour (such as security and logging). AOP will lead to improvement in the overall system's design modularity [4]. The technology was successfully utilized in software security, error handling, reliability and so on.

One of the AOP goals is to make programming easier and faster for developers [17]. The main purpose of AOP is to capture the cross-cutting concerns in the system, then promote them as first class citizens [18], in order to enable the modelling and reusing of them in the system. AOP mainly handles cross-cutting concerns by providing a mechanism, called aspect, for expressing these concerns and automatically incorporating them into the system [19]. Well known examples of cross-cutting concerns are logging and security.

As noted in the previous sections, dynamic metrics can provide deeper insight into the software quality issue. Despite the case, it is observed that static metrics are more often used for such a purpose. This is due mainly to the technical difficulties associated with the collection of dynamic metrics. One of the known issues when dealing with dynamic metrics is the need to instrument the code by inserting points for data collection.

In this work, the utilization of AOP for facilitating the collection of dynamic metric is proposed. It is suggested that dynamic metrics be treated in a similar manner to other cross-cutting concerns. By using AOP, the code can be instrumented automatically by inserting data collection points to the source code. This may significantly reduce the effort needed for code instrumentation compared to manual instrumentation.

3.0 RELATED WORKS

3.1 MAINTAINABILITY DYNAMIC METRICS

There are a number of dynamic metrics and metrics tools which have been developed. These metrics, in addition to static ones, are designed to measure different characteristics of object-oriented design such as coupling, cohesion, inheritance, and so forth. However, it has been observed that the number of dynamic metrics that developed for object-oriented systems is less than the number of static ones. There is lack of researches on dynamic metrics in general, when compared with static metrics. However, the focus here is on software maintainability; in the following we will note down some of dynamic metrics that are related to coupling, and therefore to software maintainability will be covered.

Two dynamic coupling metrics were proposed in [5]. The two metrics are the Import Object Coupling (IOC) and the Export Object Coupling (EOC). In fact, these metrics were designed to measure the coupling in an early level of development (design stage). The two metrics are concerned with the coupling between objects (not classes), therefore they are dynamic metrics.

The Information Flow-based Coupling (IFC) metrics is also considered as a dynamic metric. It measures the number of invoked methods from other classes during execution[20].Continuously, the Message Passing Coupling (MPC) [21], counts the number of sent statements that is found in the methods of one class to other classes. However, most of Chidamber and Kemerer metrics can be also, beside the static form, considered as dynamic metric[2]. It depends on the way we collect them. If we collect the metrics by investigating the source code or diagrams, and then it will be considered static metrics. Otherwise, if we are collecting it by running the program (execution), then it will be considered a dynamic metric.

For example, the CBO measures the coupling level between classes in static mode. In the runtime, the CBO works by measuring object coupling by counting the number of coupled objects from other classes. Fig 1 shows the difference between class-level and object-level couplings. The coupling between classes can be called: "coupling on attributes", and the coupling between object called: "coupling on operation" [22]. This run time coupling metric is also known as "Dynamic CBO" [16]. The DCBO metric is the dynamic form of the CBO metric that proposed by [13]. It counts the number of method calls from other classes at the run-time. Other works on dynamic complexity measurement can be found in [22] , [23] and [24].

[2] Exclude Depth of Inheritance Tree (DIT) and Number of Children (NOC) metrics

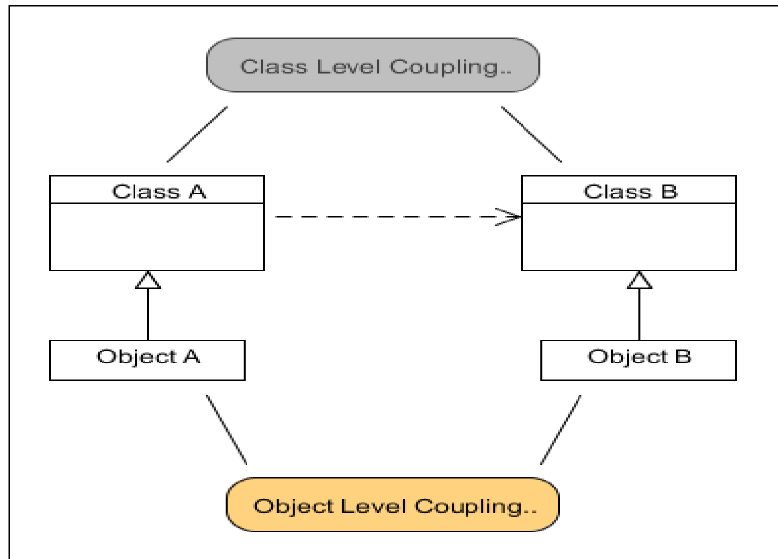

Fig 1: Difference between class –level and object-level couplings

A broad work on the relation between object-oriented characteristics and software quality has been done in [25]. The authors have done an empirical study on the use of object-oriented design as a quality indicator. As a result of the empirical study, it has been found that five out of the Chidamber and Kemerer's six object-oriented metrics appear to be useful to predict class fault-proneness; therefore they can be useful as quality indicators.

For the metrics tools, we observed that there are a limited number of runtime analysis tools available. Most of these tools deal with memory control function related to a software application at runtime such as [26], [27] and [28]. Additionally, a metric-based software performance measurement tool has been introduced in [29]. The work presented a measurement tool that includes a set of static and dynamic metrics for Java and C++ programs called DynaMetrics.

3.2 AOP AND SOFTWARE QUALITY

As aspect-oriented is relatively a new technology, only few works have been done on the relation between software quality and aspect-oriented quality. These researches mostly focus on the effect of AOP on program quality. Least researches focuses on the use AOP as a quality indictor for non-AOP systems.

Kulesza et al. [30] studied the effect of developing systems in an AOP manner on software maintainability. They found that aspect-oriented design has exhibit higher stability and reusability through the changes. Separation of concern will improve the maintainability of the software by reducing the complexity and increasing the reusability of the components. Moreover, Kaeli et al. [31] adopts the AOP technology to profile the software runtime behaviour and to increase the program comprehension of a given fragment of Java code.

Figueiredo et al. [32] present a set of metrics capturing several object and aspect oriented artefacts such as coupling and cohesion . However, the work introduces an aspect-oriented tool to collect metrics' data statically. Other works on the impact of the use of AOP on software quality can be found in the following sources [33], [34] and [35]. Tarta and Moldovan [36] used an AOP approach to evaluate system's usability. The study used a dynamic approach to measure usability in the run-time using AOP. Then, the study presented a case study in which AOP was used to collect a set of data needed to evaluate the usability. The experiment shows that usability evaluation module is easy to integrate with the other modules. Also, the usability evaluation module is easy to plug-in and out of the system to be measured.

4.0 RESEARCH METHODOLOGY

The experimental method is considered part of the empirical research methods. The empirical research is a research based on the real observation to test a specific hypothesis or discover the unknown. This type of research involves setting a hypothesis and a test to evaluate it. Experimental methodology is commonly used for evolution and problem solving projects [37] .

Kitchenham [38] stated that the goal of any empirical software engineering research is to positively influence the practice of software engineering. This implies the need of empirical methods that provide us with insights into how

software engineering works in real-life situations. The empirical method is highly recommended for software engineering projects [39].

However, the spirit of an experimental study is the attempt to practice and learn by comparing different theories (hypothesis and literatures) with reality (prototype). In this work, the hypothesis has been developed based on the problem structure. Then, a prototype has been developed in order to demonstrate and test the hypothesis. The prototype will be evaluated based on the results of the implementation stage.

The following is the structure of the experimental study:

4.1 RESEARCH HYPOTHESIS

The main hypothesis of this research work is “AOP is able to facilitate the transparent collection of maintainability dynamic metric”

The null hypothesis is therefore “AOP is not able to facilitate the transparent collection of maintainability dynamic metric”.

Consequently, the research question would be “Is AOP able to facilitate the collection of dynamic metrics for software maintainability?”

4.2 DESIGN OF THE STUDY

For the proposed approach, the dependent variables are the “manual instrumentation” and the “separation of the concerns”. Based on our hypothesis, the system should facilitate the collection of maintainability dynamic metrics without the need for manual instrumentation. In addition, the metrics logic should be separated from the application logic (source code). The independent variables are the “reusability of the metrics” (the metrics can be reused with other applications) and “size invariance” (The size of the metric code should be totally independent of the program’s size).

This work intends, in the first place to design a general framework for facilitating the collection of dynamic quality metrics using AOP. A representative dynamic quality metric for software maintainability will be then selected. Based on the designed framework, the selected metric will be developed as an AOP aspect. Thereafter, and in order to test the proposed approach, the implemented metric will be prototypically applied to some existing software applications (for example, open source software). Finally, the approach will be evaluated according to clearly defined criteria.

4.3 SYSTEMS INVESTIGATED

In this stage the dynamic metrics–AOP framework is going to be implemented prototypically (The AOP-based framework is at its initial stage). For this purpose, the designed framework is going to be implemented and applied to two open source software applications (AddressBook and Paint applications).

The AddressBook application is a Java example that uses the swt library to implement a simple electronic address book. This application has some basic and common functions of a simple address book such as save; load, sort, and search functions. On the other hand, the Paint application is also Java-based software that helps creating simple drawings and graphic shapes. The Graphical User Interface (GUI) a quite similar application to the famous Microsoft Paint application[3].

4.4 TEST AND EVALUATE THE PROTOTYPE

In order to assess the hypothesis, we need first to define clear evaluation criteria. The evaluation criteria are [11] :

a. Feasibility: Is it viable to use AOP for facilitating the collection of dynamic quality metrics?
b. Separation of concerns: It is expected that the metric code is totally separated from the software application’s code for which the metric to be measured.

[3] http://www.microsoft.com/windows/windows-vista/default.aspx

c. Transparency: Was it possible to collect dynamic metrics without any manual modification of the software application's code for which the metric to measured?

d. It is expected that no modification will be needed for the software code, and therefore the software code will preserve its consistency and readability.

e. Size invariance: The size of the metric code is totally independent of the software application's size. It is expected that the number of lines of AOP code used to define a specific dynamic metric is not going to change when the size of the software application changes.

f. Reusability: It is expected that the AOP code used to define a dynamic metric for a specific target software application can easily and with almost no modification be reused with other software applications

These evaluation criteria are used to evaluate the two prototypical implementations of the concept presented in this research for facilitating the collection of dynamic metrics using AOP.

4.5 PROPOSED FRAMEWORK

Before explaining the framework, the general mechanism of AOP will be explained. The concept of AOP is well-established and it is used in many different kinds of applications such as logging, tracing, and security. However, and in order to be able to prototypically implement and evaluate the framework, it is important to select a representative dynamic maintainability metric.

In this context, the Dynamic Coupling Between Objects (DCBO) metric has been chosen as a representative dynamic maintainability metrics. As noted earlier, this metrics is originally designed to measure dynamic coupling of object-oriented systems. This metric measures the coupling at objects level. This can only be done during execution time. Otherwise, at the class level, coupling can be measured only statically by means of source code inspection [22]. Figure1 shows the different between class and object level coupling.

Coupling describes the dependability between objects. High level of coupling means a high level of complexity; which leads to difficulties in maintenance and reusability of the components. However, there are two outputs of the DCBO metrics: the total number of couplings in the program and the percentage of coupling for individual classes within a given set of classes. This percentage is based on the system's total number of couplings.

The DCBO metric measures the number of methods that are called dynamically (at run-time) by a given method. Essentially, this metric counts the number of times that a given method accesses other methods in other classes. First it will count the number of accessing methods using the following formula (for each method):

$$M = \sum \text{Number of methods called by a method in the class}$$

Then, the DCBO can be founded using this formula

$$\sum \text{Number of coupled methods for all methods within class A}$$

$$OR$$

$$\sum Mn$$

n is the number of methods in class ***A***

Finally, the percentage of coupling for a selected class within a given set of classes can be defined as [16]:

$$\frac{\text{Number of coupled methods for all methods in class A}}{\text{Total number of coupled classes}} * 100$$

The use of the DCBO metric can be a good indicator for the software maintainability and complexity. The greater the coupling percentage, the greater the complexity and the harder maintainability of the system will be. Lower coupled components are easier to modify and correct faults within, improve performance and other attributes, or adapt to a changed environment.

After selecting and defining the metrics, the AOP-based framework for collecting software dynamic metric should be developed. The idea is to develop the DCBO metric as an AOP aspect. This aspect will be injected into the software program (for which the metric to be collected) by the aspects compiler. However, the metric collection will

start at the execution time. When the program runs, the injected metric code will start collecting the needed metric data at the executing time.

Fig 2 shows the general AOP framework. The Aspect Weaver adds the separate aspect of the program to the source code. In this study, compile-time instrumentation is used. However, the final framework is shown in Fig 3. The Aspect Weaver is the AOP's pre-compiling step. The weaver weaves the aspects into the source code at the compile-time. The aspect will contain the DCBO metric. In addition, it will be supported with list of join point s. The join point is the place where the aspect joins the source code. The aspect point cut is used to identify areas (exact methods) at which the metric will join the original source code.

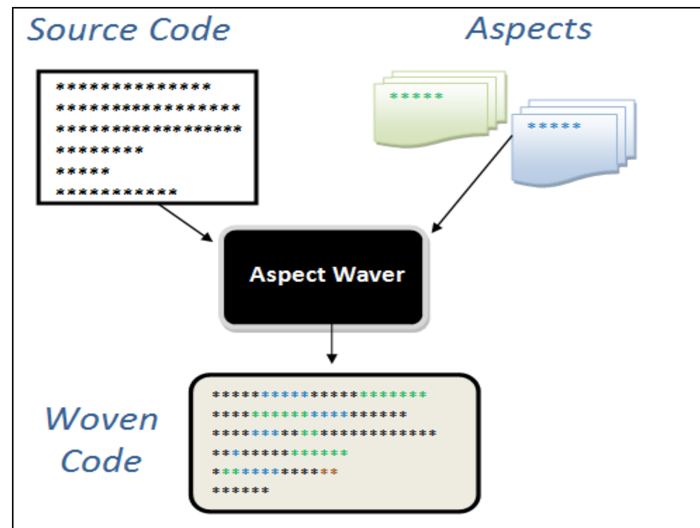

Fig 2: General AOP framework

As shown in Fig 3, after completing the weaving process, the original source code and the aspect (DCBO metrics) will become as one component. Therefore, combined elements will go to the compilation process. The program and the aspect becomes a one compiled piece. The metric data will be collected during the execution of the compiled code. The output is the metric's data that shows the number of coupled methods of each class. However, the DCBO metric will be written in an aspect advice. This advice will join the code at the execution time using the identified point cut. This point cut points on specific join points.

The advice is the action or decision of the cross-cutting parts. The advice provides a way to express a cross-cutting action at the join point s that are captured by a point cut [40]. The advice joins the code at the defined point cut. The Aspect Weaver will link the source code (the original application) with the advice (our DCBO metric). This is done by investigating the join point (through several point cuts) and adding the advice to the related point cuts.

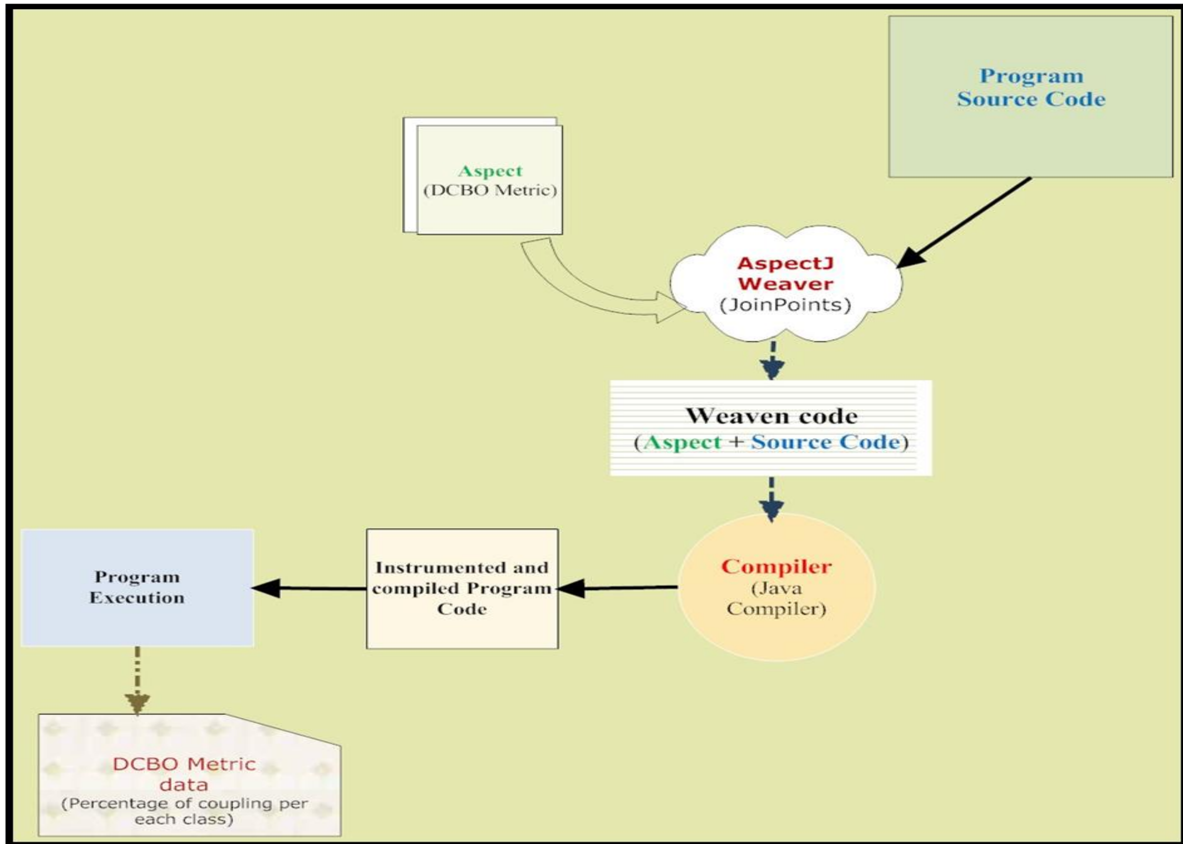

Fig 3: Proposed Framework of collecting DCBO metrics using AOP

5.0 IMPLEMENTATION

In order to implement and then test the proposed framework, two java programs have been chosen in which used to measure their coupling (AddressBook and Paint applications4). Both of these programs are sample open source Java projects provided by the Eclipse IDE.

We have showed and prepared our framework in section 5. Here, the DCBO aspect's code that used during the development is explained. The AOP-dynamic metrics to measure coupling is detailed as the following:

```
point cut   capture() : call(* *..*(..))
```

Here, the capture point cut (with no parameters) captures all method calls within the program. This includes all classes such as system classes and Java classes. Then, one additional point cut (called excluded) have been added which can help to exclude all unnecessary classes, as the metrics intend to measure the coupling within the program and without all attached and unnecessary classes (such as Java default classes). The following shows a combined join point (all in methods call):

```
point cut   excluded():
call(* org.aspectj..*(..)) || call(* java..*(..)) || call(*
        org.eclipse.swt..*(..));
```

[4] http://www.eclipse.org/swt/examples.php

The excluded point cut use to exclude all unwanted classes from any further calculation. It has a combined join point s with different signatures. Actually, if any of these methods matched, then the excluded point cut will capture them.

Now, there are two point cuts, one point cut that captures all calls and another one to exclude all unwanted calls that happen with three specific classes (AspectJ, Java, and swt classes). The coupling capturing process will start from here. The before advice is used to capture all calls of all methods. It will identify only methods that are captured by the capture point cut. Other method calls are excluded by the excluded point cut. The following code shows the before advice for the capture and excluded point cuts.

```
before() : capture() && !excluded()
```

The advice defines composed point cuts that should be captured by the "capture" point cut and not excluded by "excluded" point cut. If the composed point cut returns "true", then the advice will capture the class name and the caller class name (caller and the called class of the method). If both captured classes are different, then the metric will record this coupling information. This information will be stored in a hash table. Fig 4 shows an activity diagram that explains the process of capture and excluded point cuts, and how the method matches occur.

```
before() : capture() && !excluded()
{
        String callee = thisJoin point .getSignature().getDeclaringTypeName();
        String caller =                thisEnclosingJoin point
        StaticPart.getSignature().getDeclaringTypeNam();

if (!caller.equals(callee))
 {
update(callee, caller);
 }
}
```

The advice defines composed point cuts that should be captured by the "capture" point cut and not captured by the "excluded" point cut. If the composed point cut returns "true", then the first line will get the called class of the method and store it in string "callee". Then it will get the class name of the caller method and store it in the string "caller".

The next step is to compare the caller and callee. The aim is to differentiate between the caller and the callee classes. If the caller and callee methods are from different classes, then the advice will store all the information in a table. This table includes information of callees, number of calls, and names of all callers.

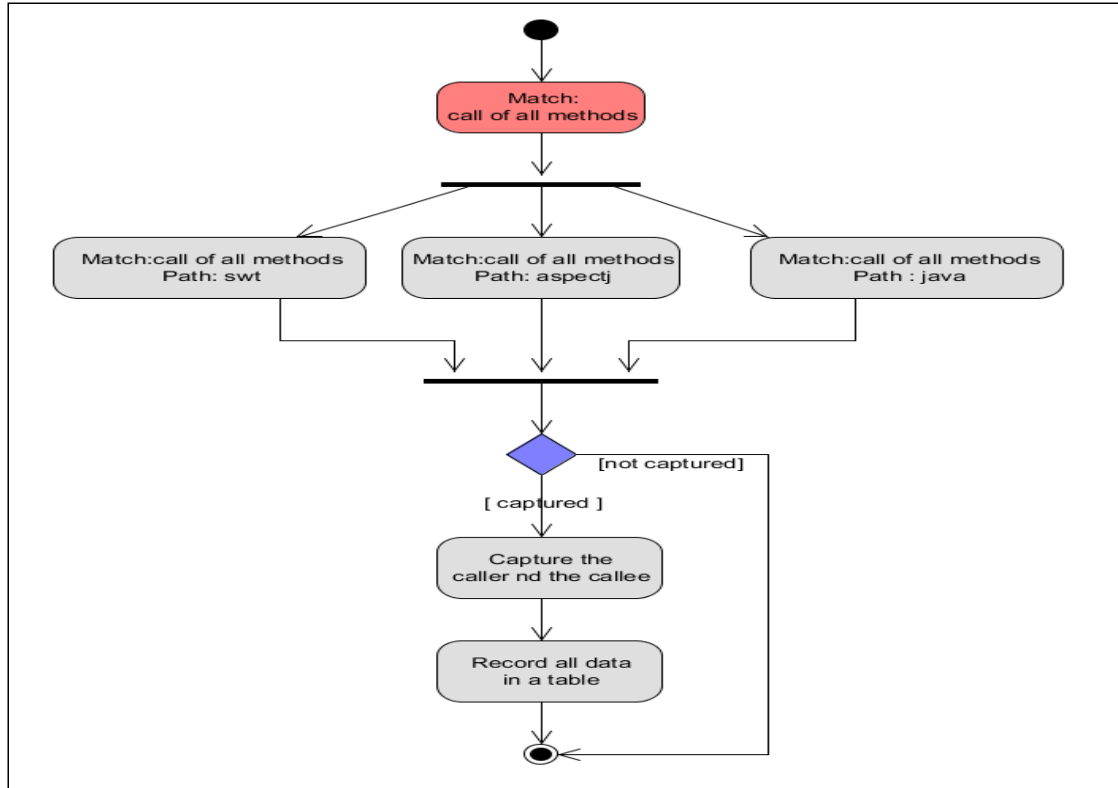

Fig 4: Activity diagram for the capturing process

6.0 RESULTS AND DISCUSSION

Here, the evolution of the proposed framework takes place. The evolution is done by implementing and applying the DCBO metric on two different software programs. As explained in section 4.3, AddressBook and Paint applications are used to measure their maintainability using the DCBO metrics. Fig 5 and Fig 6 show screenshots of the homepage of the AddressBook and Paint applications respectively.

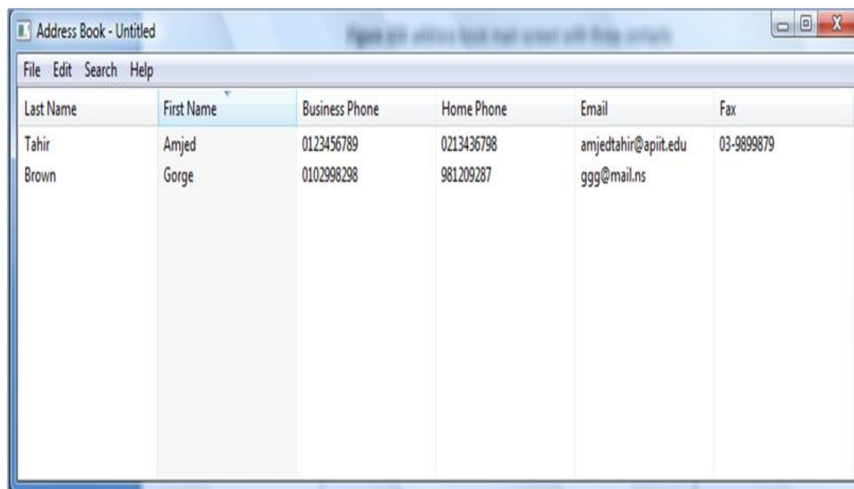

Fig 5: a screenshot of the AddressBook application

Table 1 shows the DCBO metrics data for the AddressBook application. Table 2 shows detailed results of the coupling measurement process for the Paint application. These two tables (1 and 2) show the number of the coupled classes for a particular class, name of the coupled classes and percentage of the coupling.

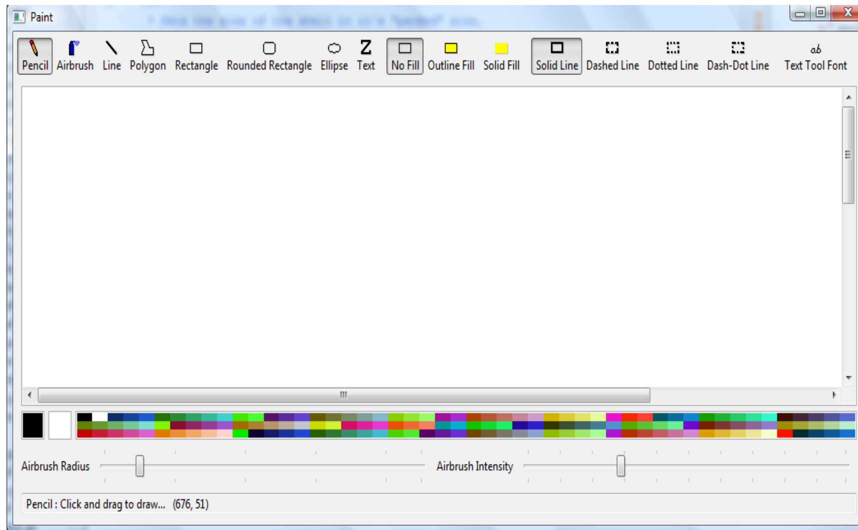

Fig 6: a screenshot of the AddressBook application

***Table 1**: DCBO metric data for the AddressBook application*

Class	*Number of Coupled classes*	*Callers*	*percentage*
addressbook.DataEntryDialog	1	addressbook.AddressBook	50%
addressbook.SearchDialog	1	addressbook.AddressBook	50%

Fig 7 shows the level of coupling for individual classes of the Paint application. It appears that the PaintExample class has the highest level of associated coupled classes. Thus, this class is the most complex and difficult to maintain than other classes. On the other hand, the PaintSession and PaintTool have the lowest number of associated classes and, thus, there are easier to maintain than other classes. Higher level of coupling produces classes that are more difficult to maintain, which results more complex software system.

Table 2: DCBO metric data for the Paint application

Class	***Number of Coupled classes***	***Callers***	***percentage***
Paint.ContainerFigure	3	Paint.PaintSurfac - Paint.AirbrushTool - Paint.RoundedRectangleTool	11%
Paint.Figure	2	Paint.PaintSurface - Paint.ContainerFigure	7%
Paint.FigureDrawContext	5	Paint.PointFigure - Paint.RectangleFigure - Paint.RoundedRectangleFigure - Paint.TextFigure - Paint.LineFigure	18%
Paint.PaintExample	9	Paint.PaintSurface - Paint.TextTool - Paint.PencilTool - Paint.ContinuousPaintSession, Paint.RectangleTool - Paint.DragPaintSession - Paint.AirbrushTool - Paint.RoundedRectangleTool - Paint.LineTool	33%
Paint.PaintSession	1	Paint.PaintSurface	3%
Paint.PaintSurface	6	Paint.PaintExample - Paint.ContinuousPaintSession - Paint.PencilTool - Paint.DragPaintSession - Paint.AirbrushTool - Paint.TextTool	22%
Paint.PaintTool	1	Paint.PaintExample	3%

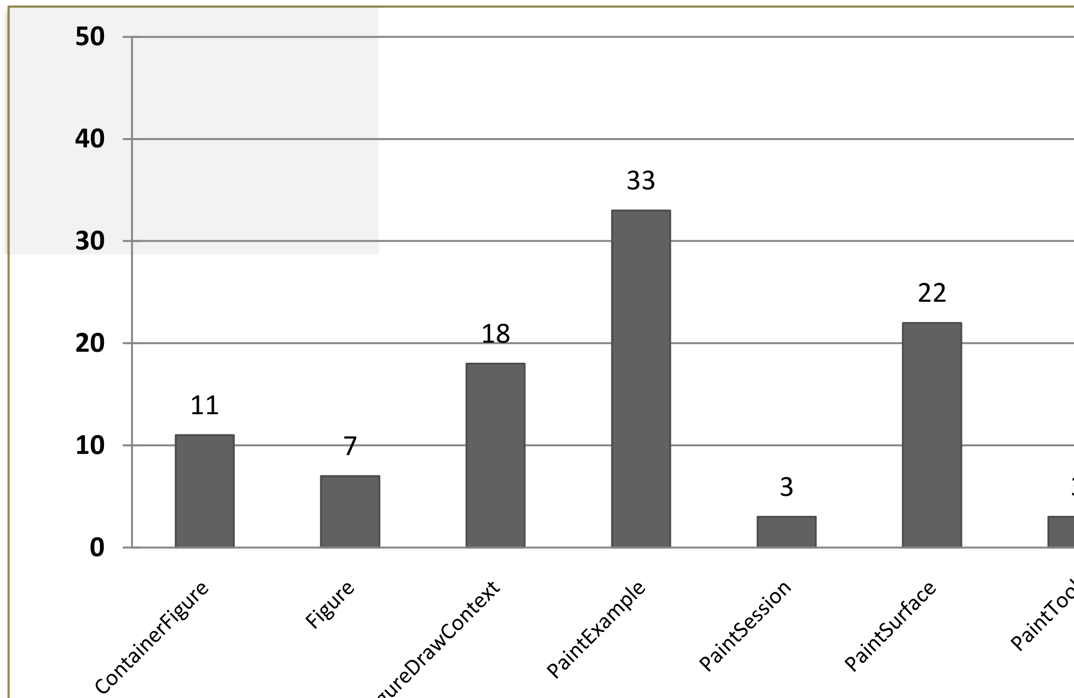

Fig 7: Level of dynamic coupling of all individual classes (Paint application)

The evaluation of the purposed framework is based on the following evaluation criteria (see 4.4): feasibility, separation of concerns, transparency, size invariance and reusability. The result of the evaluation detailed as the following:

a. ***Feasibility:*** It was feasible to use AOP to collect a dynamic maintainability metric. DCBO metric data was collected using AOP for two open source applications: AddressBook and Paint applications.

b. ***Separation of Concerns:*** the metrics logic and code was totally separated from application's code in both cases (i.e., for AddressBook and Paint applications).

c. ***Transparency:*** It was possible to collect a maintainability dynamic metrics without any manual instrumentation of the software application's source code. AddressBook or Paint applications source code have not been modified. Therefore the software code in both cases preserved its consistency and readability.

d. ***Size invariance:*** the size of the metrics was totally independent of application size. Table 3 shows the total number of LOC for the DCBO metric in the two different applications (AddressBook and Paint applications).

Table 3: LOC of the DCBO metrics for the two examples

	# LOC	*# BCL*	*DCBO LOC (without BCL)*
DCBO for the *AddressBook Application*	192	82	**110**
DCBO* for the *Paint Application	169	95	**110**

 # LOC: total number of Lines of Code with comments and blank lines
 # BCL: total number of blank and comment lines

e. ***Reusability:*** the DCBO metrics that is collected using AOP can be reused with other applications. The metrics is applied on the AddressBook application. Then, the same AOP code has reused and applied on the Paint application. It was possible to reuse the DCBO metrics with almost no changes or modification.

7.0 CONCLUSION AND FURTHER WORK

This work was motivated by the many difficulties associated with the collection of dynamic metrics. As the main contribution of this research, the authors could demonstrate that AOP is able to facilitate the collection of a dynamic maintainability metric. In addition, it is feasible to collect a maintainability dynamic metrics (the DCBO) using AOP. AOP could separate the DCBO metric logic from the application source code. It also allow for a transparent collection of a dynamic maintainability metric without any manual instrumentation or modification of the targeted software. Moreover, the metric's code could be reused with other applications with almost no changes. Finally, it has been found that the size of the metric code is totally independent of the software application's size.

In general the AOP-based framework for dynamic maintainability metrics data collection can also provide an effective and concise approach for collecting a dynamic metric. As noted before, this framework has been applied to two different applications (the AddressBook and Paint application). The authors believe that there will probably no significant inherent problems encountered with applying this method to applications of any size.

As this research project was limited, other aspects such as cohesion should be measured besides coupling in order to properly evaluate the maintainability of object-oriented based systems. Furthermore, in order to get clearer picture about the maintainability and quality, the dynamic metrics data collected should be used alongside static metrics.

Future work can be done in the following areas:

- Measure other object-oriented characteristics (such as cohesion) which are related to the system's complexity and maintainability. This could be done using the same approach but with one of the cohesion dynamic metrics instead such as the Dynamic LCOM [16] metrics.
- Evaluation of the current framework using additional applications. It is recommended to use applications of larger size.
- Use of dynamic metrics collected data alongside static metrics to give a better indicator of the quality.

REFERENCES

[1] S. S. Dahiya, J. K. Chhabra, and S. Kumar, "Use of Genetic Algorithm for Software Maintainability Metrics' Conditioning," presented at the Proceedings of the 15th International Conference on Advanced Computing and Communications, 2007.

[2] R. Gunnalan, M. Shereshevsky, and H. H. Ammar, "Pseudo dynamic metrics [software metrics]," presented at the Proceedings of the ACS/IEEE 2005 International Conference on Computer Systems and Applications, 2005.

[3] N. E. Fenton and S. L. Pfleeger, *Software Metrics: A Rigorous and Practical Approach*: PWS Publishing Co., 1998.

[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect Oriented Programming," presented at the European Conference on Object-Oriented Programming (ECOOP), Finland, 1997.

[5] S. M. Yacoub, H. H. Ammar, and T. Robinson, "Dynamic Metrics for Object Oriented Designs," presented at the Proceedings of the 6th International Symposium on Software Metrics, 1999.

[6] G. Murphy and C. Schwanninger, "Guest Editors' Introduction: Aspect-Oriented Programming," *IEEE Softw.,* vol. 23, pp. 20-23, 2006.

[7] J. McManus and T. Wood-Harper, "Software Engineering: a Quality Management Perspective," *The TQM Magazine,* vol. 19, pp. 315-327, 2007.

[8] D. A. Sunday, "Software maintainability-a new `ility'," in *Reliability and Maintainability Symposium, 1989. Proceedings., Annual*, 1989, pp. 50-51.

[9] IEEE, "IEEE Standard Glossary of Software Engineering Terminology," in *IEEE Std 610.12-1990*, ed, 1990, p. 1.

[10] M. Hitz and B. Montazeri, "Measuring coupling and cohesion in object-oriented systems," presented at the International Symposium on Applied Corporate Computing, Monterrey, Mexico, 1995.

[11] A. Tahir and R. Ahmad, "An AOP-Based Approach for Collecting Software Maintainability Dynamic Metrics," in *Second International Conference on Computer Research and Development (ICCRD)*, 2010, pp. 168-172.

[12] S. H. Kan, *Metrics and Models in Software Quality Engineering*: Addison-Wesley Longman Publishing Co., Inc., 2002.

[13] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object Oriented Design," *IEEE Trans. Softw. Eng.,* vol. 20, pp. 476-493, 1994.

[14] I. Sommerville, *Software Engineering: (Update) (8th Edition) (International Computer Science)*: Addison-Wesley Longman Publishing Co., Inc., 2006.

[15] B. Dufour, K. Driesen, L. Hendren, and C. Verbrugge, "Dynamic Metrics for Compiler Developers," Sable Research Group - McGill University, Technical Report 2002-11, Nov 2002 2002.

[16] A. Mitchell and J. F. Power, "Toward a definition of run-time object-oriented metrics," in *7th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering* Darmstadt, Germany, 2003.

[17] D. Shukla, S. Fell, and C. Sells. (2002) Aspect-Oriented Programming Enables Better Code Encapsulation and Reuse. *MSDN Magazine*. Available: http://msdn.microsoft.com/en-us/magazine

[18] O. Papapetrou and G. A. Papadopoulos, "Aspect Oriented Programming for a component-based real life application: a case study," presented at the Proceedings of the 2004 ACM Symposium on Applied Computing, Nicosia, Cyprus, 2004.

[19] G. Pollice. (2004, 20-01-2008). A look at aspect-oriented programming. *The Rational Edge*. Available: http://www.ibm.com/developerworks/rational/library/2782.html

[20] Y. S. Lee and B. S. Liang, "Measuring the Coupling and Cohesion of an Object-Oriented Program Based on Information Flow," presented at the Proceedings of International Conference on Software Quality, Maribor, Slovenia, 1995.

[21] W. Li and S. Henry, "Maintenance metrics for the object oriented paradigm," in *Proceedings of the First International Software Metrics Symposium*, 1993, pp. 52-60.

[22] E. Arisholm, L. C. Briand, and A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software," *IEEE Trans. Softw. Eng.,* vol. 30, pp. 491-506, 2004.

[23] Y. Liu and A. Milanova, "Static analysis for dynamic coupling measures," presented at the Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, Toronto, Ontario, Canada, 2006.

[24] L. C. Briand, J. W. Daly and J. Wüst, "A Unified Framework for Cohesion Measurement in Object-OrientedSystems," *Empirical Softw. Engg.,* vol. 3, pp. 65-117, 1998.

[25] V. R. Basili, L. C. Briand and W. L. Melo, "A Validation of Object-Oriented Design Metrics as Quality Indicators," *IEEE Trans. Softw. Eng.,* vol. 22, pp. 751-761, 1996.

[26] Y.-G. Guéhéneuc, R. Douence, and N. Jussien, "No Java without Caffeine: A Tool for Dynamic Analysis of Java Programs," presented at the Proceedings of the 17th IEEE international conference on Automated software engineering, 2002.

[27] B. Dufour, L. Hendren, and C. Verbrugge, "*J: a tool for dynamic analysis of Java programs," presented at the Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA, USA, 2003.

[28] A. Mitchell and J. F. Power, "An empirical investigation into the dimensions of run-time coupling in Java programs," presented at the Proceedings of the 3rd international symposium on Principles and practice of programming in Java, Las Vegas, Nevada, 2004.

[29] P. Singh and H. Singh, "DynaMetrics: a runtime metric-based analysis tool for object-oriented software systems," *SIGSOFT Softw. Eng. Notes,* vol. 33, pp. 1-6, 2008.

[30] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. v. Staa, and C. Lucena, "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study," presented at the Proceedings of the 22nd IEEE International Conference on Software Maintenance, 2006.

[31] D. Ng, D. R. Kaeli, S. Kojarski, and D. H. Lorenz, "Program comprehension using aspects," *IEE Seminar Digests,* vol. 2004, pp. 89-96, 2004.

[32] E. Figueiredo, R. Garcia, U. Kulesza, and C. Lucena, "Assessing Aspect-Oriented Artifacts: Towards a ToolSupported Quantitative Method," presented at the Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Glasgow- UK, 2005.

[33] J. Viega and J. Voas, "Can Aspect-Oriented Programming Lead to More Reliable Software?," *IEEE Softw.,* vol. 17, pp. 19-21, 2000.

[34] R. Laddad, "Aspect-oriented programming will improve quality," *Software, IEEE,* vol. 20, pp. 90-91, 2003.

[35] L. Madeyski and L. Szala, "Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study," *Software, IET,* vol. 1, pp. 180-187, 2007.

[36] A. M. Tarta and G. S. Moldovan, "Automatic Usability Evaluation Using AOP," in *IEEE International Conference on Automation, Quality and Testing, Robotics*, 2006, pp. 84-89.

[37] C. W. Dawson, *Projects in Computing and Information Systems: A Student's Guide*: Pearson Prentice Hall, 2009.

[38] B. Kitchenham, *Empirical Paradigm – The Role of Experiments* vol. 4336/2007: Springer Berlin / Heidelberg, 2007.

[39] B. Curtis, H. Krasner, and N. Iscoe, "A Field Study of The Software Design Process for Large Systems," *Communications of the ACM,* vol. 31, pp. 1268 - 1287 1988.

[40] R. Laddad, "Aspect-oriented programming will improve quality," *IEEE Software,* vol. 20, pp. 90-91, 2003.

BIOGRAPHY

Amjed Tahir received his M.Sc. in software engineering from the faculty of computing, engineering and technology, of Staffordshire University in 2009. Currently, he is a research assistant at the software engineering department, University of Malaya. His current research work focuses on the role of software metrics in software quality. His other research interests include software quality management, aspect-oriented programming, requirement engineering and agile software development.

Rodina Ahmad received her Bachelor and Master Degree in Computer Science from United States in 1988 and 1991 respectively. She obtained her PhD in the field of Management Information System from National University of Malaysia in 2006. She has been teaching and researching in Software Engineering department, University Malaya in the area of Information systems and Software Engineering since 1993. At present, she is supervising seven PhD students and more than twenty master students. She has published more than 14 journal articles and more than 20 conference papers at national and international level. She is a member of IEEE.

Zarinah Mohd Kasirun received her BSc (CS) and MSc (CS) from National University of Malaysia (UKM) in 1989 and 1993, respectively. She obtained her PhD in the field of Requirements Engineering from University of Malaya in 2009. Currently, she is a Senior Lecturer in Software Engineering Department at the Faculty of Computer Science and Information Technology, University of Malaya. Her current research interests include requirements engineering, requirements visualization, object-oriented software engineering and computer-supported collaborative learning.