A Class Coupling Analyzer for Java Programs

Boukari Souley and Baba Bata

Mathematical Sciences Programme
Abubakar Tafawa Balewa University(ATBU), Bauchi, Nigeria
bsouley2001@yahoo.com +2348069667696, +2347034568741

Abstract
Increasingly, object-oriented measurements are being used to evaluate and predict the quality of software. A growing body of empirical results supports the theoretical validity of these metrics. Strategies were presented on how analysis of byte code with metrics can be integrated in an ongoing software development project and how metrics can be used as a practical aid in code- and architecture investigations on already developed systems. An experimental study was conducted as an attempt to further validate each metric and increase knowledge about them. The tool was fully tested and can serve as a guide for software developers and maintainers to identify early enough what quality measures (coupling or cohesion) may affect the quality of the software.

Introduction

Software complexity measures are meant to indicate whether the software has desirable attributes such as understandability, testability, maintainability, and reliability. As such, they may be used to suggest parts of the program that are prone to errors. An important way to reduce complexity is to increase modularization [9] . Modularity of software design can be measured with two qualitative properties: cohesion and coupling [2].

Cohesion describes a module's functionality; the highest degree of cohesion is obtained when a module performs one function. On the other hand, coupling is the degree of interdependence between pairs of modules; the minimum degree of coupling is obtained by making modules as independent as possible. Ideally, a well designed software system maximizes cohesion and minimizes coupling.

An external attribute is concerned with how the product relates to its environment. Practitioners, whether they are developers, managers, or quality assurance personnel, are really concerned with the external attributes. However, they cannot measure many of the external attributes directly until quite late in a project's or even a product's life cycle. Therefore, they can use product metrics as leading indicators of the external attributes that are important to them. By having good leading indicators, it is possible to predict the external attributes and take early action if the predictions do not fit a project's objectives. For instance, if we know that a certain coupling metric is a good leading indicator of maintainability as measured in terms of the effort to make a corrective change, then we can minimize coupling during design because we know that in doing so we are also increasing maintainability [4] [11].

Internal structure attributes characterize software products used or produced in the early stages of software development. Moreover, these attributes can be measured directly. It is therefore common practice to use internal structure measures as early indicators for software quality.

Related Literature

Traditional Measures of Complexity

The earliest software measure, which was proposed in the late 1960s, is the Source Lines of Code (SLOC) metric, which is still used today. It is used to measure the amount of code in a software program. It is typically used to estimate the amount of effort that will be required to develop a program, as well as to estimate productivity or effort once the software is produced. Two major types of SLOC measures exist:

- **Physical SLOC and**
- **Logical SLOC**.

Physical SLOC: This is a count of non-blank, non-comment lines in the text of the program's source code.

Logical SLOC: This measures attempt to measure the number of statements; however their specific definitions are tied to specific computer languages. Therefore, it is much easier to create tools that measure physical SLOC, and physical SLOC definitions are easier to explain. However, physical SLOC measures are sensitive to logically irrelevant formatting and style conventions, while logical SLOC is less sensitive to formatting and style conventions.

In [8] a measure known as ***Cyclomatic Complexity*** was defined. It may be considered as a broad measure of soundness and confidence for a program. It measures the number of linearly-independent paths through a program module and it is intended to be independent of language and language format.

Function points, which were pioneered in [1], are a measure of the size of computer applications and the projects that build them. The size is measured from a functional, or user, point of view. It is independent of the computer language, development methodology, technology or capability of the project team used to develop the application. The original metric has been augmented and refined to cover more than the original emphasis on business-related data processing.

Object-Oriented Metrics

Object-oriented design and development is becoming very popular in today's software development environment. Object-oriented development requires not only a different approach to design and implementation but it requires a different approach to software metrics. Since object oriented technology uses objects and not algorithms as its fundamental building blocks, the approach to software metrics for object oriented programs must be different from the standard metrics set. Metrics, such as Lines of code and Cyclomatic complexity, have become accepted as standard for traditional functional or procedural programs and were used to evaluate object-oriented environments at the beginning of the object-oriented design revolution. However, traditional metrics for procedural approaches are not adequate for evaluating object oriented software, primarily because they are not designed to measure basic elements like classes, objects, polymorphism, and message-passing [7].

Definitions of Coupling

Myers defined six distinct levels of coupling to measure the interdependence among the modules; the coupling levels were ordered by **Page-Jones** according to their effects on the understandability, maintainability, modifiability and reusability of the coupled modules.

Coupling is increased between two classes ***A*** and ***B*** if:

- ***A*** has an attribute that refers to (is of type) ***B***.
- ***A*** calls on services of a ***B*** object.
- ***A*** has a method which references ***B*** (via return type or parameter).
- ***A*** is a subclass of (or implements) ***B***.

If two modules are coupled in more than one way, they are considered to be coupled at the highest level:

Coupling types are ranked on a 6 point ordinal scale from loosely coupled (i = 1) to tightly coupled (i = 5). If there is no coupling between x and y then i = 0. The types of coupling, in order of lowest to highest coupling, are as follows:

1. **DataCoupling (low and best)**

Data coupling is when modules share data through, for example, parameters. Each datum is an elementary piece, and these are the only data which are shared (e.g. passing an integer to a function which computes a square root).

2. **Stamp Coupling**
(Data-structured coupling)

Stamp coupling is when modules share a composite data structure, each module not knowing which part of the data structure will be used by the other (e.g. passing a student record to a function which calculates the student's GPA).

3. **Control Coupling**

Control coupling is one module controlling the logic of another, by passing it information on what to do (e.g. passing a what-to-do flag).

4. **External Coupling**

External coupling occurs when two modules share an externally imposed data format, communication protocol, or device interface.

5. **CommonCoupling**

Common coupling is when two modules share the same global data (e.g. a global variable).

6. **ContentCoupling(worst)**

Content coupling is when one module modifies or relies on the internal workings of another module (e.g. accessing local data of another module).

Disadvantages of high coupling include:

- A change in one class forces a ripple of changes in other classes.
- Difficult to understand a class in isolation.
- Difficult to reuse or test a class because dependent class must also be included.

Definitions of Cohesion

Cohesion was first introduced within the context of module design.. The cohesion of a module is measured by inspecting the association between all pairs of its processing elements. The term *processing element* was defined as an action performed by a module such as a statement, procedure call, or something which must be done in a module but which has not yet been reduced to code. A scale of cohesion that provides an ordinal scale of measurement that describes the degree to which the actions performed by a module contribute to a unified function was developed. There are seven categories of cohesion which range from the most desirable (functional) to least desirable (coincidental). They stated that it is possible for a module to exhibit more than one type of cohesion; in this case the module is categorized by its least desirable type of cohesion. In the principle of good software design it is desirable to have highly cohesive modules, preferably functional[9]. Cohesion is decreased if:

- The responsibilities (methods) of a class have little in common.
- Methods carry out many varied activities, often usingcoarsely-grainedor unrelated sets of data.

The types of cohesion, in order of the worst to the best type, are as follows:

CoincidentalCohesion (worst)

Coincidental cohesion is when parts of a module are grouped arbitrarily (at random); the parts have no significant relationship (e.g. a module of frequently used mathematical functions).

Logical Cohesion

Logical cohesion is when parts of a module are grouped because they logically are categorized to do the same thing, even if they are different by nature (e.g. grouping all I/O handling routines).

Temporal Cohesion

Temporal cohesion is when parts of a module are grouped by when they are processed - the parts are processed at a particular time in program execution (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

Procedural Cohesion

Procedural cohesion is when parts of a module are grouped because they always follow a certain sequence of execution (e.g. a function which checks file permissions and then opens the file).

Communicational Cohesion

Communicational cohesion is when parts of a module are grouped because they operate on the same data (e.g. a module which operates on the same record of information).

Sequential Cohesion

Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line (e.g. a function which reads data from a file and processes the data).

Functional Cohesion (best)

Functional cohesion is when parts of a module are grouped because they all contribute to a single well-defined task of the module

Disadvantages of low cohesion (or "weak cohesion") are:

- Increased difficulty in understanding modules.
- Increased difficulty in maintaining a system, because logical changes in the domain affect multiple modules, and because changes in one module require changes in related modules.
- Increased difficulty in reusing a module because most applications would not need the random set of operations provided by a module. Types of cohesion

Methodology

Analysis: Identification of Coupling Using Byte Code Analysis

The coupling types are determined by information that is usually not available at the design level and they must be computed from the program source code. Doing so, however, is difficult because some of the relationships appear only implicitly. The information must be extracted from the byte code of the software by an analysis tool. Programming languages have subtle complexities that make finding coupling information more difficult than might be expected. Some calls are implicit instead of explicit. There are several types of global variables and uses, and the effect of inheritance with regard to coupling is not obvious.

Occurrences of parameter coupling

In Java, *parameter coupling* occurs through only method and constructor calls. Parameter coupling to be viewed as the occurrence of an invocation of a call to a method or constructor through an object or class.

Java allows two explicit types of method calls, instance and static, and one implicit type, through a constructor. If a method in class `A` explicitly calls `method m()` in class `B` through an object instance (`b.m()`), this represents parameter coupling between **A** and **B**. An explicit static call occurs when class**A** calls a public static `method m()` in class B (that is, `B.m()`). An implicit constructor call is made when a variable of type **B** is defined and instantiated in class **A**, for example, **Bb** = new **B().** All three of these types are considered to be parameter coupling in this research, even if no actual

parameter value is passed in and no value is returned.

The three types of parameter coupling can be summarized as:

1. **Bb** = new **B();** // implicit, constructor
2. **b.m()**; // explicit, through an object reference
3. **B.m();** // explicit, static

It is very important to differentiate between different classes (inter-class) as opposed to couplings between methods in the same class (intra-class). The effects of intra-class couplings are very different from the effects of inter-class couplings. Intra-class coupling has no direct impact on the external system, although it can have an indirect impact. If the class is viewed as a black box, then intra-class coupling is invisible. Inter-class parameter coupling has the potential to propagate problems from within one class to other classes, especially during maintenance and reuse.

Global, Inheritance, and External Coupling

1. Global coupling is a kind of inter-class coupling that refers to the coupling that takes place through variables that are defined in one class and used in others. These variables will typically have public or protected package access specifiers. Public variables represent a traditional, or true global coupling, if the variable is static; otherwise it is a global coupling with an object reference. All of these variations must be detected.

2. Inheritance coupling refers to the coupling that is related to the inheritance between pairs of classes. The coupling takes place through attributes and methods that are inherited and used by a subclass but that are not re-defined. If a subclass does not actually use anything from its super class, or if it re-defines everything it uses, this is not considered to be inheritance coupling.

In Java, the inheritance relation is established through the keywords `extends` and `implements`. Therefore, an implementation can detect an inheritance coupling between two classes or interfaces if one class extends from another class or implements one or more interfaces, or if an interface extends from another interface.

3. External coupling is defined as access to an external device by two or more classes. In other words, external coupling happens when two classes share something that is outside the application that owns the classes. External resources can include files on a hard disk, printers, or other shared devices. The challenge in designing an algorithm to analyze coupling is to find out the unique interfaces between these resources and the application. Specifically, different classes or applications may use the same resource, but refer to them with different names. Binding to a physical device may be done at the OS level, not the program. This is necessary for symbolically linked files and devices with multiple names. If there is no unique interface, then all interfaces must be enumerated.

Actual types and dynamic binding

When analyzing the software for coupling, the analyzer must first discover the types of each reference. This is simple for direct references to names. However, when a reference is made through an object reference (`o.b` or `o.m()`), the type of `o` must first be found. Inheritance and dynamic binding means that the type of `o` cannot be determined statically, because it can change during execution.

Metric Descriptions

In the process of choosing what metrics are to be used as measurement, the first thing that has to be considered is from what viewpoint the measure is to be evaluated. What the main goal of the measurement is. As an example consider a metric for evaluating the quality of a text.

Some observers might emphasize layout, others might consider language or grammar as quality indicators. Since all of these characteristics give some quality information it is difficult to derive a single value (metric) that describes the quality of a text. The same problem occurs for computer software. This observation indicates that a metric must be as unambiguous and specific as possible in its measure. The metrics, the code analyzer will calculate and displays for each class are :

Coupling between object classes (CBO):

The coupling between object classes (CBO) metric represents the number of classes coupled to a given class (**efferent couplings, Ce**). This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions [12].

Lack of cohesion in methods (LCOM):

A class's lack of cohesion in methods (LCOM) metric counts the sets of methods in a class that are not related through the sharing of some of the class's fields. The metric considers all pairs of a class's methods. In some of these pairs both methods access at least one common field of the class, while in other pairs the two methods do not share any common field accesses. The lack of cohesion in methods is then calculated by subtracting from the number of method pairs that don't share a field access the number of method pairs that do [12].

Weighted methods per class (WMC):

A class's weighted methods per class (WMC) metric are simply the sum of the complexities of its methods. As a measure of complexity we can use the *cyclomatic complexity*, or we can arbitrarily assign a complexity value of 1 to each method. The program assigns a complexity value of 1 to each method, and therefore the value of the WMC is equal to the number of methods in the class.

Depth of Inheritance Tree (DIT):

The depth of inheritance tree (DIT) metric provides for each class a measure of the inheritance levels from the object hierarchy top. In Java where all classes inherit Object the minimum value of DIT is 1.

Number of Children (NOC):

A class's number of children (NOC) metric simply measures the number of immediate descendants of the class.

Response for a Class (RFC):

The metric called the response for a class (RFC) measures the number of different methods that can be executed when an object of that class receives a message (when a method is invoked for that object). Ideally, we would want to find for each method of the class, the methods that class will call, and repeat this for each called method, calculating what is called the transitive closure of the method's call graph. This process can however be both expensive and quite inaccurate. In this program, we calculate a rough approximation to the response set by simply inspecting method calls within the class's method bodies.

Afferent couplings (Ca):

A class's afferent couplings are a measure of how many other classes use the specific class. Ca is calculated using the same definition as that used for calculating CBO (Ce).

Number of Public Methods (NPM):

The NPM metric simply counts all the methods in a class that are declared as public. It can be used to measure the size of an API provided by a package.

Afferent Couplings (Ca):

The number of other packages that depend upon classes within the package is an indicator of the package's responsibility.

Efferent Couplings (Ce):

The number of other packages that the classes in the package depend upon is an indicator of the package's independence.

In addition the following metrics are also of great importance while trying to consider quality of large object oriented programs.

Instability (I):

The ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce / (Ce + Ca). This metric is an indicator of the package's resilience to change.

The range for this metric is 0 to 1, with I=0 indicating a completely stable package and I=1 indicating a completely instable package [10].

Abstractness (A):

The ratio of the number of abstract classes (and interfaces) in the analyzed package to the total number of classes in the analyzed package [10] The range for this metric is 0 to 1, with A=0 indicating a completely concrete package and A=1 indicating a completely abstract package.

Distance from the Main Sequence (D)

The perpendicular distance of a package from the idealized line A + I = 1. This metric is an indicator of the package's balance between abstractness and stability.

A package squarely on the main sequence is optimally balanced with respect to its abstractness and stability. Ideal packages are either completely abstract and stable (x=0, y=1) or completely concrete and instable (x=1, y=0).

The range for this metric is 0 to 1, with D=0 indicating a package that is coincident with the main sequence and D=1 indicating a package that is as far from the main sequence as possible

System model:

Figures 1 and figures 2 below illustrate the and the Class diagram of the coupling Analyzer.

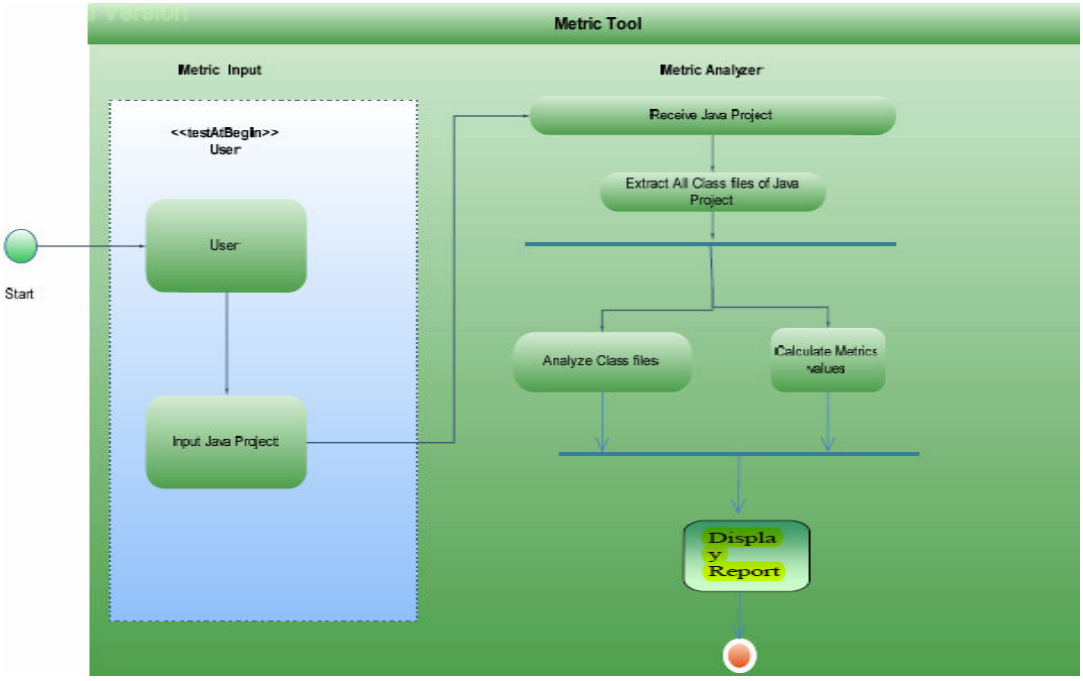

Figure 1: Activity diagram

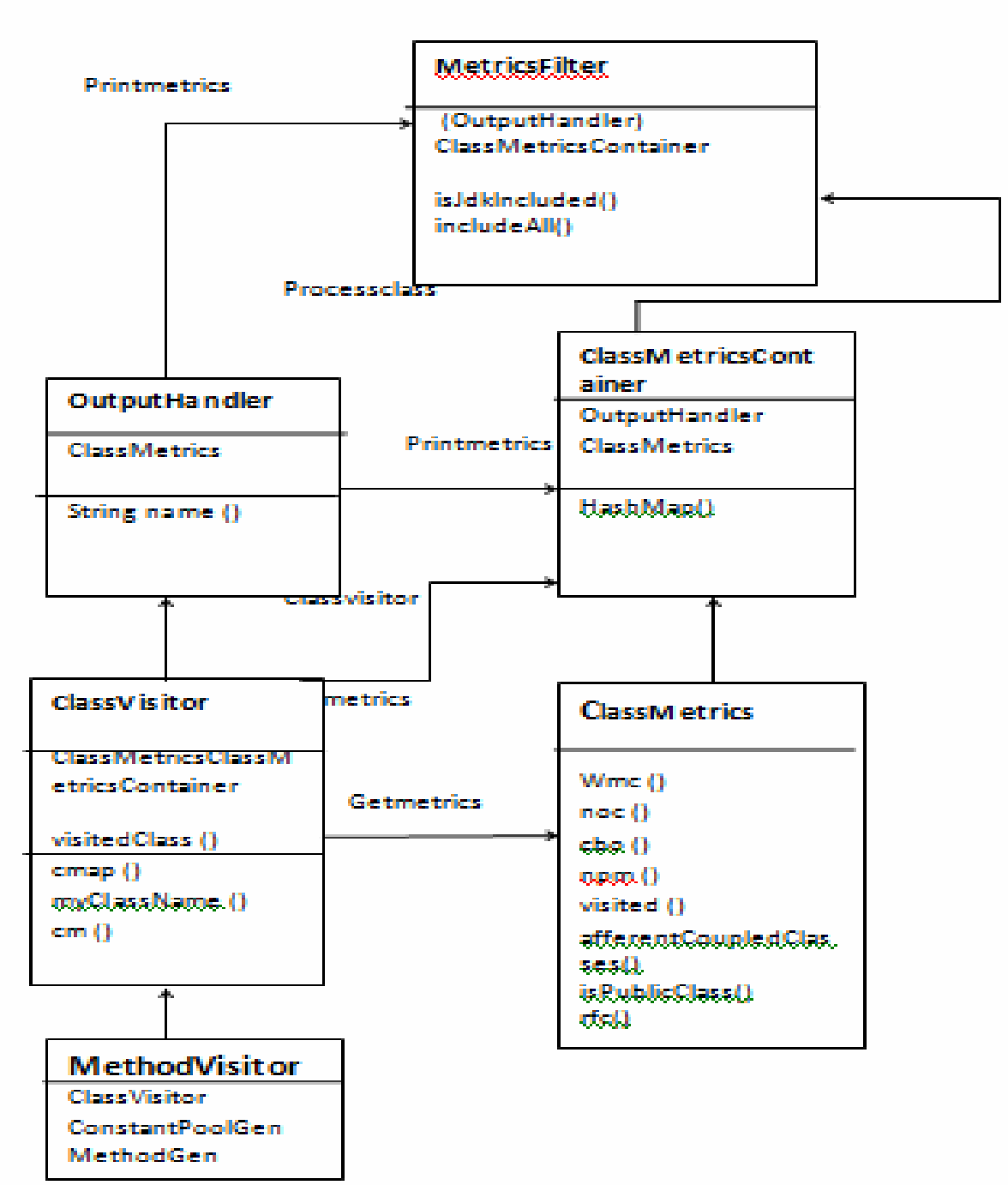

Figure 2: class diagram of class Analyzer

Testing And Results

The analyzer was developed using Java. T he various components were fully tested.

After running the analyzer on a certain numbers application on a number of test cases, the following results were obtained:

Package: epayment.adapters

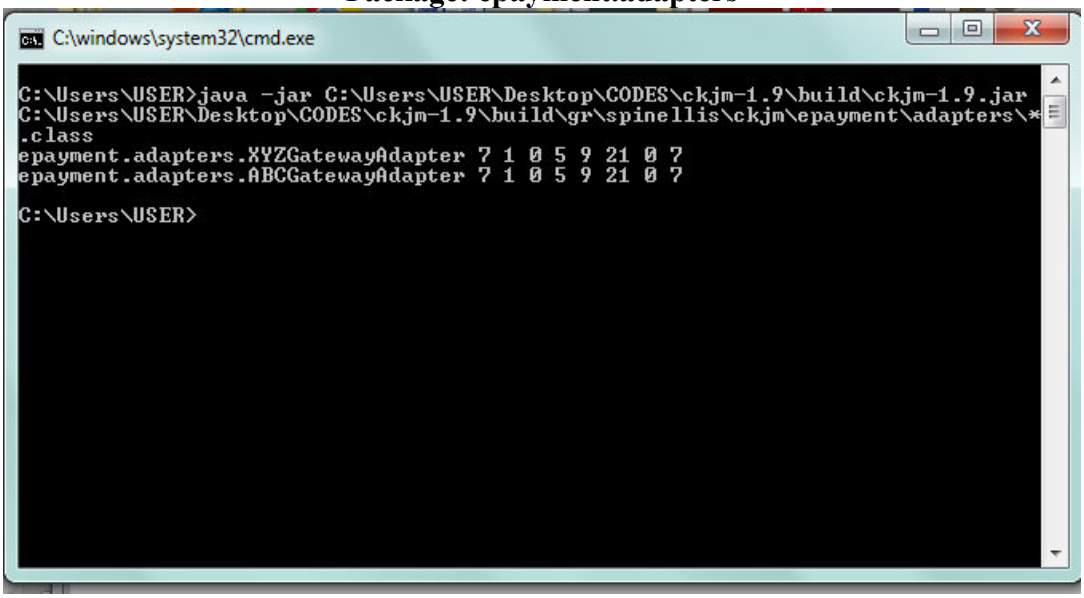

Figure 2.: Showing WMC, DIT, NOC, CBO, RFC, LCOM, Ca, And NPM Of The Package Epayment.Adapters.

Table 1:The classes and metrics of test case1

		METRICS							
PACKAGES	CLASSES	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM
Epayment.Adapters	XYZ Gateway	7	1	0	5	9	21	0	7
	ABCGateway	7	1	0	5	9	21	0	7
Epayment.Framework	IPaymentCommand	1	1	0	3	1	0	1	1
	IgatewayAdapter	6	1	0	3	6	15	2	6
	IPaymentResponse	2	1	0	0	2	1	3	2
	IPaymentRequest	6	1	0	0	6	15	2	6
	AbstractPaymentCommand	3	1	0	5	4	1	0	2
	paymentException	1	3	0	0	2	0	3	1
Epayment.Commands	Sales command	2	0	0	5	5	1	0	2
	CaptureCommand	2	0	0	5	5	1	0	2
	AuthorizeCommand	2	0	0	5	5	1	0	2
	VoidSaleCommand	2	0	0	5	5	1	0	2
	CreditCommand	2	0	0	5	5	1	0	2
Epayment.Processor	PaymentProcessorConfig	2	1	0	2	4	1	1	2
	PaymentProcessor	5	1	0	5	14	8	1	4
Epayment.Response	PaymentResponse								

Table 2: showing the instability of test case 1.

Package name	Ca	Ce	Instability
epayment.adapters	0	2	1
epayment.processor	1	2	0.6
epayment.commands	0	5	1
epayment.response	0	1	1
epayment.request.	0	1	1
epayment.framework	3	0	0

Table 3: showing the classes and their metrics of test case2

		METRICS							
PACKAGES	CLASSES	WMC	DIT	NOC	CBO	RFC	LCOM	Ca	NPM
Kirk.Analyzer	Analyzer	2	1	0	5	6	1	0	2
	Configuration	3	1	0	0	13	1	1	2
Analyzer.Textui	XMLUISummary	20	1	0	6	67	58	0	5
	JarAnalyzertask	8	0	0	3	21	10	0	7
	Summary	2	1	0	0	2	1	3	2
	DotSummary	13	1	0	4	43	66	0	5
Analyzer. Framework	JarCollection	7	1	0	1	17	21	1	7
	JarBuilder	1	1	0	2	1	0	0	1
	JarPackage	5	1	0	1	5	10	1	5
	CollectionDecorator	8	1	0	2	16	0	0	8
kirkk.jar	JarMetrics	5	1	0	0	5	10	1	5
	JarClass	5	1	0	0	5	10	1	5
	Jar	18	1	0	2	18	153	3	18

Table 4: showing the instability of test case 2.

Package name	Ca	Ce	Instability
kirkk.analyzer	1	5	0.83
Kirkk.framework	4	4	0.5
kirkk.bcel	1	5	0.83
kirkk.bcelbundle	1	7	0.88
kirkk.jar	2	4	0.67
kirkk.textui	0	7	1

Discussion of Results

The Coupling between objects of test case 1is high, it is not desired while that of test case 2 is low and desired. The lack of Cohesion of both test cases is lower, so there is higher similarities between the methods in the class. When the RFC of test case 1 is higher, there is a probability that the classes are fault prone while that of test case 2 is low. The Lesser NOC of test case 1 indicates there is no reusability while test case 2 has high reusability. The larger WMC of both test cases, indicates that there is a chance that the classes are fault prone.

Conclusion

Based on the analysis carried out, we conclude that coupling and cohesion analyzer can tell the stability of a java application by investigating the relationships between the efferent coupling and afferent coupling, it also shows how software metrics which quantify the internal complexity of a design can be used to characterize it's external quality. It is important to note that the Coupling Analyzer *calculates* the metrics from the code appearing in the compiled byte code files. This can serve as a benchmark for java programmers and maintainers in assessing the quality of their products before final release.

References

[1] Albrecht, A.J.(1979) Measuring Application Development. In *IBM Applications Development Joint SHARE/GUIDE symposium*, Monterey California, USA, pp 83-92.

[2 Constantine L. L., Stevens W.P, and Myers G.J.(1974) Structured Design. *IBM Systems Journal,* **13**(2):115-139.

[3] Diomidis S.(2002). **Tool writing: A forgotten art?** http://www.spinellis.gr/pubs/jrnl/2005-IEEESW-TotT/html/v22n4.html)

4] Fenton, N.E. and. Neil, M (1999) Software Metrics: Successes, Failures and New Directions. *The Journal of Systems and Software*, **47**:149-157.

[5] Gonzalez. R.R.(1995.) A Unified Metric of Software Complexity: Measuring Productivity, Quality and Value. *The Journal of Systems and Software*, **29**(1):17-37,

[6] Linda H. R..(1998). **Applying and interpreting object oriented metrics** (http://www.stc-online.org/cd-rom/1998/slides/p7lrosenberg.PDF

[7] Lorenz, M. and Kidds , J.(1994) *Object-Oriented Software Metrics*. Object-Oriented Series, Englewood Cliffs, USA Prentice Hall

[8] McCabe, T.(1976). A Software Complexity Measure.*IEEE Transactions on Software Engineering*, **2**(4):308-320,

[9] Myers. G. (1974) **Reliable Software Through Composite Design**. Mason and Lipscomb

[10] Robert C. M, (2003) **Agile Software Development**, Pearson Education Inc Publishers, New York, USA.

[11] Shooman, M.L.(1983) **Software Engineering***: Design, Reliability and Management*. McGraw Hill, New York, USA.

[12] Shyam R. C. and Chris F. K.(1994) **A metrics suite for object oriented design. IEEE Transactions on Software Engineering,** 20(6):476–493, (http://dx.doi.org/10.1109/32.295895))