Measuring Components' Unused Members

Hamdan Msheik, Alain Abran, Hamid Mcheick, Pierre Bourque
Software and IT Engineering Department, École de Technologie Supérieure,
1100 Notre-dame Ouest, Montréal (Québec), Canada, H3C 1K3
hamdan.msheik.1@ens.etsmtl.ca, aabran@ele.etsmtl.ca, pbourque@ele.etsmtl.ca
Department of Computer Science, Université du Québec à Chicoutimi
555, Boulevard de l'Université, Chicoutimi (Québec), Canada, G7H 2B1
hamid_mcheick@uqac.ca

Abstract:

Currently, components technology represents a major step in the evolution of software technology as a whole. Although it has been undergoing continuous enhancement, this technology suffers from a number of limitations: in particular, components' unused functionalities. For instance, a software component incorporates a set of functions of which a size-varying subset is actually used to satisfy the functional requirements of a particular software application. Consequently, a subset of unused functionalities will persist in the deployed application. This subset of unused functionalities provides no functional value to the hosting application. Furthermore, these unused functionalities consume memory and network resources and might compromise application security if they are exploited inappropriately. In this paper, we propose CUMM (Components' Unused Member Measurement), a method to measure components' unused members (attributes and functionalities), and their memory consumption inside a software application. Furthermore, we present a set of analysis models which use the results of the CUMM to determine percentages of unused members as well as the degree of generality of a component's members.

Keywords

Measurement, Components' Unused Members Measurement, metrics, memory consumption

1 Introduction

Software components have emerged as an important paradigm to address several traditionally known problems such as complexity, reuse and reduction of software development costs [1, 2]. The importance of components in software development has led some to describe the use of components as another computing revolution on a par with those of stored programs and programming languages [3]. The use of components is aimed at achieving better reuse and at

reducing the complexity of developed applications and the efforts expended on development. Furthermore, components are attractive since they go hand in hand with long advocated software design practices and characteristics such as modularity and cohesion.

Software components have several commonalities with object-oriented classes, to the point that they are sometimes undistinguishable from them. Jacobson in [4] defines a component as a physical and replaceable part of a system that realizes and conforms to a set of interfaces. According to this definition, a complete application per se can be considered a component. In this paper, the word component refers to: a complete software application, a software component (subsystem) or a ordinary class constructed using an object-oriented programming language.

Even though software components have been undergoing continuous enhancement, they still suffer from the limitation of unused functionalities [5]. Typically, a component possesses a set of functions which satisfy specific functionalities in a particular software application. When reused in a different application, several component functionalities are unused and therefore provide no value in the application context in which they are used. These unused functionalities consume computing resources, such as memory and network bandwidth, inefficiently. Furthermore, these unused functionalities might compromise application security if they are exploited inappropriately. In this paper, memory consumption refers to what the static code occupies in memory, and not memory consumed by the dynamic creation of objects during runtime.

The unused functionalities exhibited by a software component in particular application contexts are caused by the tendency of software components to be coarse and large-grained. In such components, the set of useful and required functionalities provided by a particular component varies according to the particular software application context. Typically, during the development of software application families, considerable effort is expended on the wrapping, adaptation and customization of the functionalities of components shared by the various constituent applications.

In this paper, we propose CUMM (Component Unused Member Measurement), a method to calculate the number of a component's unused attributes and functionalities and their memory consumption. Furthermore, we present a set of statistical formulas which make use of the measurement method result to calculate: a) the percentages of unused functionalities and their memory consumption on a per component and a per application basis, and b) the degree of a component's functional and attribute generality.

We begin section 2 by presenting a background of software measurement. Next, we present the CUMM method and a set of statistical formulas which derive from the results of the CUMM method. We present in section 4 an example

which shows how to apply the CUMM method. Finally, we conclude in section 5 with a summary and a discussion.

2 Background

Software measurement can help evaluate software quality attributes and in making better decisions and controlling software and its development process. In this respect, the IEEE [6] defines software engineering as: "*The application of a systematic, disciplined quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software*".

Software measurement theory allows for the mapping of the empirical domain to a numerical domain. In other words, software objects are measured in terms of quantitative values. Several software measures have been proposed over the years, but the problem lies in the way the measures' empirical quantitative values are mapped to semantically meaningful qualitative values [7]. Research efforts have been concentrated on providing sound models, methodologies and measurement frameworks for the design and use of software measures so that these measures are established for a software engineering discipline in a similar way to the measures used in other engineering disciplines [7, 8].

Traditionally, and by analogy to the way other engineering disciplines define measurement methods, several software measurement methods have been defined in the form of mathematical formulas that lead to the calculation of numerical values. The results of calculations are then used in various types of models for evaluation and decision-making purposes. It is observed that fewer of these measurement methods have been defined according to well-defined measurement processes.

Therefore, to define the proposed CUMM method on a sound basis, we resorted to the measurement process model suggested in [8], which is illustrated in Figure 1. This high-level model sets up a four-step road map to be used in the design and validation of software measurement methods. According to [8], the first step requires the "*definition of the measurement method objectives, design and selection of the metamodel for the objects to be measured, the characterization of the concepts to be measured, and the definition of the numerical assignment rules*".

The second step requires the construction of the metamodel using the appropriate software documentation and the application of the measurement method to calculate the resulting numerical values. The third step requires the analysis, documentation and auditing of the measurement result. In step four, the actual exploitation of the measurement result will be carried out.

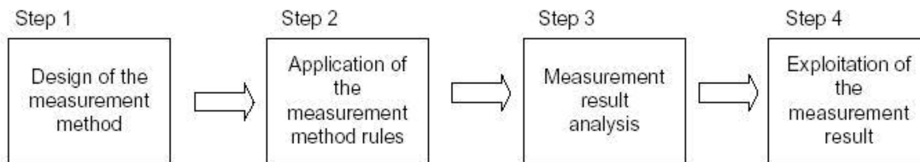

Figure 1: Measurement process – high-level model [8]

3 Overview of the CUMM Method

The CUMM method measures the number of a software entity's unused members (attributes and functionalities), and their memory consumption inside a software application. Furthermore, we present a set of analysis models that use the results of the CUMM to determine the degree of generality of a software entity's members, as well as percentages of its unused members.

The CUMM method applies to software entities which are components constructed using object-oriented programming languages. An entity measured by the CUMM method can be: i) a complete software application treated as a whole and single component, ii) a subsystem component, or simply iii) an ordinary object-oriented class considered in the context of the CUMM method as whole component.

The CUMM method measures the static memory consumption of the elements of an entity. Put differently, the CUMM method does not calculate the memory consumed by an entity's dynamic objects created at runtime. An entity's memory consumption is the recursive summation of the memory consumed by the binary code of the entity and its aggregate classes in a recursive manner.

Based on the measurement process model in Figure 1, the development of the CUMM method is carried out according to the following steps:

3.1 Step 1: Design of the CUMM Method

The design of the CUMM method follows 4 substeps:

Substep 1: Definition of the objectives

The objectives of the CUMM method are to measure, within an enclosing entity context: i) the number of the software entity's unused members (attributes or functionalities), and ii) the memory consumption of the entity's unused members. This enclosing entity context might be an outer component context or an application context. In the CUMM method context, an application is an

aggregation of one or more components in a recursive manner. The application with its aggregate components can be considered as one component per se. The measurement of unused members of a component within an enclosing entity context refers to the unused members of the component itself and the unused members of its nested component set in a recursive manner. It is important to note that a component's inherited members are actually implicit members of that component, and therefore they are all treated uniformly by the CUMM method. The intended users of this method are developers, architects and project managers; however, other stakeholders can use the measurement method results for control and decision-making purposes.

Substep 2: Design and selection of the metamodel

The CUMM method must permit the measurer to measure an entity's unused attributes and functionalities and their memory consumption in a quantifiable manner. As suggested by the measurement process model in Figure 1, to measure an entity, a metamodel of that entity must be designed or selected. A CUMM measurable entity can be instantiated according to the generic entity metamodel given in Figure 2. As depicted, this metamodel does not necessarily imply the real physical or logical composition relationship of an application and its component set. In practice, applications are aggregates of components, which in turn can be aggregates of other nested components.

Substep 3: Characterization of the concept to be measured

The measurement of unused attributes or functions of an entity is calculated based on the measurable subcharacteristics of the measured entity. The generic metamodel shown in Figure 2, in which the members of a measured entity are characterized on two relevant CUMM method bases: use basis and memory consumption basis. For instance, an entity member can be either used or unused exclusively. Similarly, an entity member consumes memory resources whether used or unused. A component member or functionality is considered unused if it has never been referenced, either in the code of its enclosing entity context or in its nested and aggregate components.

Substep 4: Definition of the numerical assignment rules

The numerical assignment rules permit the calculation of: a) the number of unused members of a component, and b) the memory occupied by these unused members.

The number of an unused member (attribute or function) is calculated by counting the number of times that member type is referenced within the code of the measured entity context. The summation of the number of members having a zero reference value is effectively the number of unused members. According to the component member being measured, the unit of the measurement result is "attribute/per component", denoted "ac", or "function/per component" denoted "fc".

The memory consumed by unused members of a particular component is calculated as follows. For the component's attribute members, the summation is made of the memory consumed by each attribute's type reference size. Similarly, for the component's functional members, the summation is made of the memory consumed by each unused function. The measurement result unit is a "byte".

Expressed in mathematical formulas, the numerical assignment rules for the number of a component's unused members are defined as follows:

Let A be the set of a component's unused attributes and $u_a \in \mathbb{N}$ be the number of unused attributes, then $u_a = |A|$.

Let F be the set of a component's unused functionalities and $u_f \in \mathbb{N}$ be the number of unused functionalities, then $u_f = |F|$.

Similarly, the numerical assignment rules for the memory consumed by unused members are defined as follows:

Let m_{ai} be the memory consumed by the i-th unused attribute reference size of a component. Then, t_{ma} is the total memory consumed by the unused attribute elements in A and is calculated as

$$t_{ma} = \sum_{i=1}^{|A|} m_{ai}$$

Let m_{fi} be the memory consumed by the i-th unused functionality of a component. Then, t_{mf} is the total of memory consumed by the unused functionality elements in F and is calculated as

$$t_{mf} = \sum_{i=1}^{|F|} m_{fi}$$

3.2 Step 2: Application of the CUMM Method

The application of the measurement method requires three substeps:

Substep1:

Gathering the software documentation related to the entities subject to measurement. The documentation artefacts of the entity subject to measurement can be either the source or reflective binary code of: the entity outer context, the entity itself and, recursively, its nested entities.

Substep2:

Constructing the software model by instantiating the generic metamodel. The software model is constructed by instantiating the generic metamodel in Figure 2 taking as input the documentation artifacts gathered in substep 1. The constructed model leads to the identification of the measurable characteristics of the entity subject to measurement.

Substep 3:

Applying the numerical assignment rules. The application of the numerical assignment rules makes it possible to assign quantifiable values to the measurable characteristics of the entity subject to measurement, and eventually to calculate the measurement results.

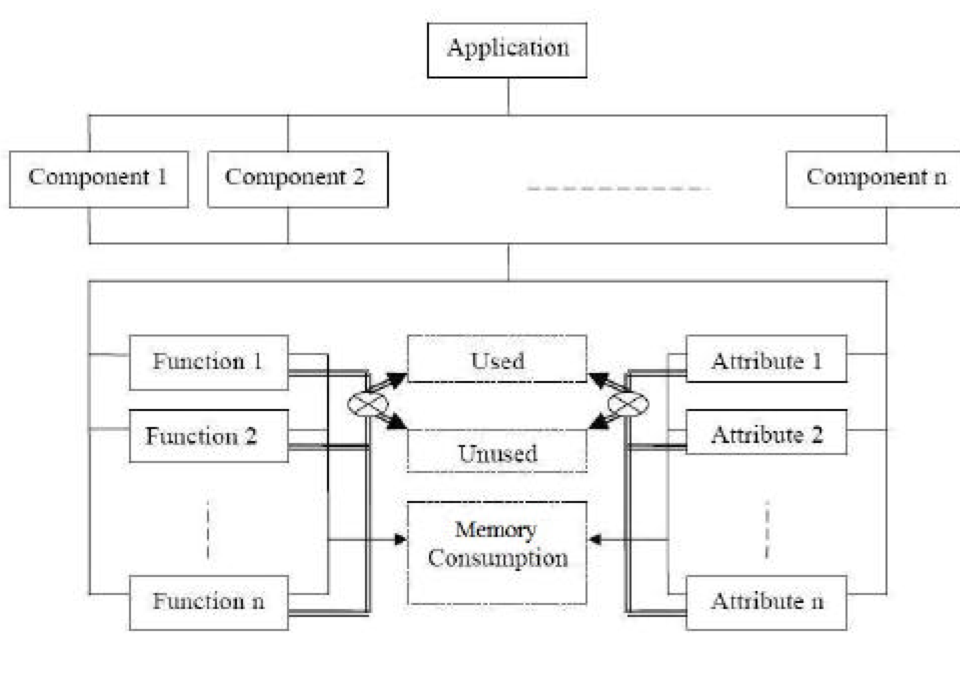

Figure 2: Generic metamodel representation of the application and component entities

3.3 Step 3: Derived statistics of the measurement result

When the results are ready, they must be documented in a presentable format. Furthermore, different types of analysis and derived statistics can be calculated out of the result. For instance, the following derived statistics for the CUMM method are potentially useful and provide value to the user of the CUMM method:

1. Measuring the percentage of unused attributes of a component: This is done by using the result calculated by applying the measurement method on unused attributes and by calculating the total number of attributes.
2. Measuring the percentage of unused functionalities of a component: This is done by using the result calculated by applying the measurement method on unused functionalities and by calculating the total number of functionalities.
3. Measuring the percentage of unused attributes' memory consumption of a component: This is done by using the result calculated by applying the

measurement method to measure the memory consumption of unused attributes and by calculating the total memory consumed by used attributes.

4. Measuring the percentage of unused functionalities' memory consumption of a component: This is done by using the result calculated by applying the measurement method to measure the memory consumption of unused functionalities and by calculating the total memory consumed by used functionalities.
5. Measuring the degree of a component's attribute generality of a component: This measures the degree of a component's attributes generality with respect to the set of applications that makes use of this component. The result of this measure depends on the percentage of unused attributes of a component and on the number of applications which use the component. The degree of a component's generality is a percentage calculated by summation of the percentages of unused attributes of a component in each application where the component is used, and then dividing by the number of these applications.
6. Measuring the degree of a component's functional generality of a component: This measures the degree of a component's functional generality with respect to the set of applications that make use of this component. The result of this measure depends on the percentage of unused functionalities of a component and on the number of applications which use the component. The degree of a component's generality is a percentage calculated by summation of the percentages of unused functionalities of a component in each application where the component is used, and then dividing by the number of these applications.

3.4 Step 4: Exploitation of the result

Finally, the results can be exploited to exercise the desired control and to make appropriate decisions. The results give indicators to the users of CUMM so that appropriate actions based on objective observations can be taken. For instance, functional optimization can be performed to enhance the performance of the application using the component. In addition, changes to the architecture, design and implementation of the component can also be considered.

4 Example

To illustrate the applicability of the CUMM method, we use a simple example (see Table 1) which consists of a small application that prints a welcome message. The application consists of the class `ShowWelcomeMessage` and makes use of the `java.lang.String` class of Java API [9] to construct a string object that contains the welcome message and prints the value of the object on the screen. This application in itself can be considered as a component in the context of the CUMM method.

In this example, we are mostly interested in step 2, which illustrates the application of the CUMM method, and step 3 which elaborates on derived statistics of the measurement results.

```
public class ShowWelcomeMessage {
    static String message;
    public static void main(String[] args) {
        message = new String("Good morning Everybody!");
        System.out.println(message);
    }
}
```

Table 1: Simple application that prints a welcome message

Step 2: Application of the CUMM method

Assumptions

For simplifying the computation, we assume in this example that:

1. The number of lines of code of a member is equivalent to its memory consumption.
2. Every attribute or function line of code consumes one byte, which is not true in real life since this relation depends on the type of attribute or the machine code instructions corresponding to a line of code in a function.
3. When counting the lines of code of a component, empty lines and comments are ignored.

Formatted: Bullets and Numbering

Substep 1:

This step requires the gathering of documentation artifacts to be used during the measurement process. The `ShowWelcomeMessage` application contains two components: `ShowWelcomeMessage` and String. Therefore, to apply the CUMM method, the source or binary code of those two components is needed. The source code for the `ShowWelcomeMessage` is shown in Table 1. The source code for the String component is taken from Sun's Java SDK 1.4.2 [9] and is not shown in this paper for obvious reasons.

Substep 2:

This step requires the construction of the software model by instantiating the generic metamodel shown in Figure 2 and using as input the documentation artifact gathered in substep 1. The software model for the `ShowWelcomeMessage` application is illustrated partially in Figure 3 and Table 2. To characterize the memory consumption of an unused member, in this example, we will not compute the memory consumed by each member, since this is a little complex and requires analysis of the class binary code, a task which is better done by an

automatic tool. Instead, for each method we will use the number of Java code lines to give us an approximate indication of the amount of memory consumed by unused members. It is important to mention that the class String contains a relatively large number of methods, since it is among the foundation library classes and which provides convenient methods for a variety of situations and applications. In other words, when it comes to real applications, several methods in this class are used.

Substep 3:

To obtain the measurement results, we use the information presented in Figure 3 and Table 2. The numerical rules are applied as follows.

The Component `String` contains 11 constructors, 54 functions and 7 attributes

a. The number of unused attributes u_a = 4 ac (attribute per component)

b. The number of unused functionalities u_f = 63 fc

Formatted: Bullets and Numbering

c. The total of memory consumed by the unused attributes $t_{ma} = \sum_{i=1}^{|A|} m_{ai} =$ 4 bytes

d. Based on the assumptions mentioned above, the total of memory consumed by the unused functionalities, $t_{mf} = \sum_{i=1}^{|F|} m_{fi}$ = 562 bytes

The Component `ShowWelcomeMessage` contains 1 function and 1 attribute of type **`String`** which is a nested component.

a. The number of unused attributes u_a = 4 ac

b. The number of unused functionalities u_f = 63 fc

Formatted: Bullets and Numbering

c. The total of memory consumed by the unused attributes $t_{ma} = \sum_{i=1}^{|A|} m_{ai} =$ 4 bytes

The total memory consumed by the unused functionalities $t_{mf} = \sum_{i=1}^{|F|} m_{fi}$ = 562 bytes

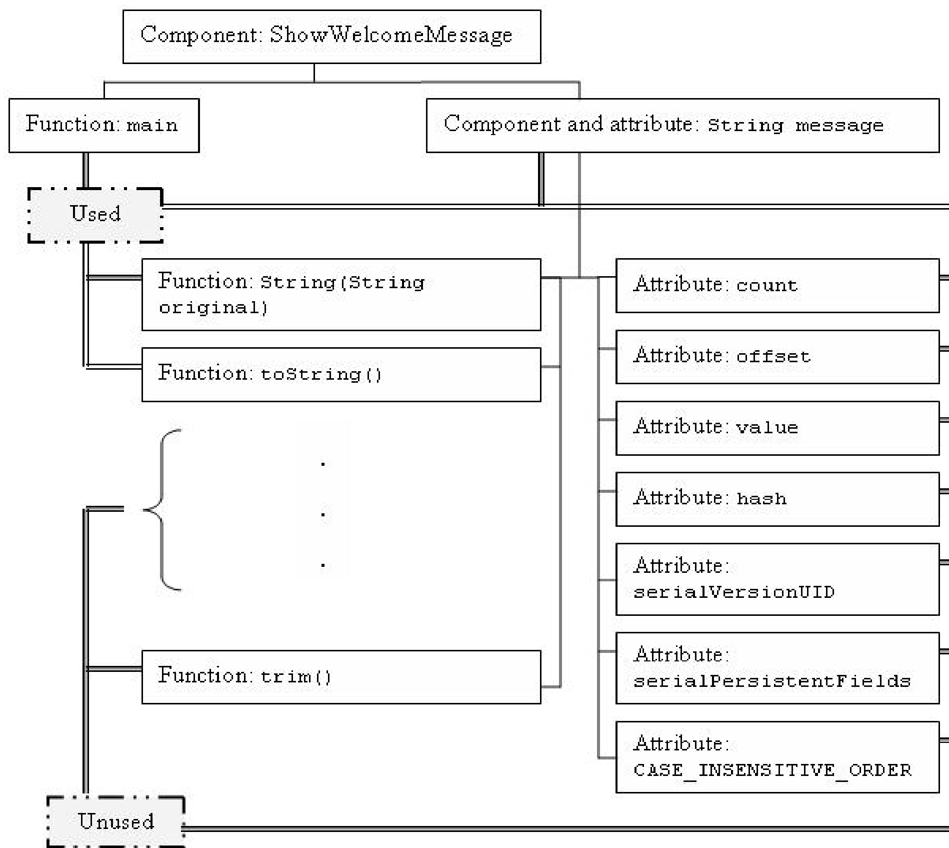

Figure 3: Metamodel instance for the `ShowWelcomeMessage` application

Step 3: Calculation of derived statistics of the measurement results of the `String` component

1. Percentage of unused attributes of the `ShowWelcomeMessage` component
   a. unused attributes
   b. 3 used attributes in the String component + 1 in the `ShowWelcomeMessage`

   The percentage of unused attributes = (4 * 100) /(4 + 4) = 50% .

2. Percentage of unused functionalities of the `ShowWelcomeMessage` component
   a. 63 unused functions
   b. 2 used functions in the String component + 1 in the `ShowWelcomeMessage`

   The percentage of unused functionalities = (63 * 100)/(63 + 3) = 95.45%.

Class `ShowWelcomeMessage`		
Methods	Used	Lines of code
`Main`	yes	4
Attributes		
`Message`	yes	1
Class `String`		
Methods	Use d	Lines of code
`public String()`	No	3
`public String(String original)`	Yes	10
`public int compareTo(String anotherString)`	No	30
`public boolean equals(Object anObject)`	No	21
`public String toString()`	Yes	3
...	…	…
Attributes		
`private int count`	Yes	1
`private int hash`	No	1
`private int offset`	Yes	1
`private char value[]`	Yes	1
...	…	…

Table 2: Members of the classes `ShowWelcomeMessage` and `String` and their number of lines of code

3. Percentage of unused attributes memory consumption of the `ShowWelcomeMessage` component
    a. 4 unused attributes
    b. 3 used attributes in the String component + 1 in the `ShowWelcomeMessage`

    The percentage of memory consumption of unused attributes = (4 * 100) /(4 + 4) = 50%.

4. Percentage of unused functionalities memory of the `ShowWelcomeMessage` component
    a. 63 unused functionalities which use 562 lines of code
    b. 2 used functionalities which use 13 lines of code + 1 in the `ShowWelcomeMessage` and which uses 4 lines of code

    The percentage of memory consumption of unused attributes = (562 * 100)/(562 + 17) = 97.06%.

5. Degree of a component's attribute generality of the `ShowWelcomeMessage` component

    Unavailable, since the `ShowWelcomeMessage` is not used in by other components.

6. Degree of a component's functional generality of the `ShowWelcomeMessage` component. Similar to 5.

5 Summary and discussion

In this paper, we have proposed a measurement method to measure the number of unused attributes and functionalities of a software component. Furthermore, we provided a set of derived statistics to analyze the measurement results with respect to certain aspects related to the unused attributes and functionalities per component and per application. The results given by the measurement method have a cross-cutting impact on a number of ISO/IEEE standard [6] quality characteristics and subcharacteristics. For instance, unused functionalities could indicate that the security subcharacteristic of the application which makes use of these functionalities might be compromised. Similarly, the memory consumption by unused attributes and functionalities indicates in turn an impact on the efficiency characteristic. In the same vein, the maintainability characteristic is impacted by the number of unused functionalities as to the efforts that might be required when adapting and customizing the concerned software components.

Finally, to better support the calculation of the measurement method result, we intend in the future to develop a tool to automate the calculation tasks. Furthermore, empirical studies can be conducted to evaluate the degree of unused functionalities, attributes and their static memory consumption on a number of components used in various products.

References

1. J. Li, "A Survey on Microsoft Component-based Programming Technologies," Concordia University, Montreal 1999.
2. SEI, "CBS Overview." [Online]: http://www.sei.cmu.edu/cbs/index.html: Software Engineering Institute, 2003.
3. P. Maurer, *Component-Level Programming*: Pearson Education Inc., 2003.
4. I. Jacobson, "Component-based Development with UML," 1998.
5. M. S. Al-Hatali and H. G. Walton, "Smart Features for Compositional Wrappers," presented at ICSR7 2002 Workshop on Component-based Software Development Processes, Austin, Texas, 2002.
6. IEEE standard glossary of software engineering terminology," in IEEE Std 610.12 1990, 1990.
7. H. Zuse, *A Framework of Software Measurement.* Berlin, 1997.
8. J.-P. Jacquet and A. Abran, "From Software Metrics to Software Measurement Methods: A Process Model," presented at Third International Symposium and Forum on Software Engineering Standards (ISESS'97), Walnut Creek, CA, 1997.
9. Sun, "Java 2 Platform SE v1.4.2." [Online]: http://java.sun.com/j2se/1.4.2/docs/api/, 2004.

Formatted: Bullets and Numbering