CoMet: A Tool Using CUMM to Measure Unused Component Members

Hamdan Msheik[1], Alain Abran[1], Hamid Mcheick[2], Dimitrios Touloumis[1], Adel Khelifi[3]
Software Engineering Department, École de Technologie Supérieure, Université du Québec,
1100 Notre-Dame Ouest, Montréal (Québec) Canada, H3C 1K3
hamdan.msheik.1@ens.etsmtl.ca1, aabran@ele.etsmtl.ca1,
dimitrios.touloumis.1@ens.etsmtl.ca1

Department of Computer Science, Université du Québec à Chicoutimi, 555, Boulevard de l'Université, Chicoutimi (Québec) Canada, G7H 2B1
hamid_mcheick@uqac.ca2

Department of Software Engineering, ALHOSN University, Abu Dhabi, United Arab Emirates,
P.O. BOX. 38722
a.khelifi@alhosnu.ae3

Abstract

Software component technology has become a major pillar of the IT evolution. The benefits of this technology, such as reuse, enhanced quality and relatively short application development time, have been key drivers of its industrial adoption. However, in its progress towards maturity, component technology has suffered from a number of limitations, such as unused component members (data and functionalities). For instance, a reusable software component incorporates a set of members, a size-varying subset of which is actually used to satisfy the functional requirements of a particular software application. This means that a complementary subset of unused members will persist in the deployed application, where this subset provides no functional value to the host application. Furthermore, these unused members can consume memory and network resources and might compromise application integrity and/or security if they are exploited inappropriately. In this paper, we propose CoMet, a prototype tool which applies CUMM (Component Unused Member Measurement) method to measure unused component members (attributes and functionalities) and their usage percentages in a software application.

1. Introduction

Software component technology emergence has been directed towards tackling traditional problems, such as application complexity, parts reuse and the reduction of software development costs [1, 2]. Components are attractive because they exhibit long advocated software characteristics like modularity and cohesion. Moreover, their impact on software development has been described as another computing revolution on a par with those of stored programs and programming languages [1].

Although software components and object-oriented classes exhibit a number of differences, they have enough commonalities to the extent that they are used interchangeably. Jacobson [2] defines a component as a physical and replaceable part of a system which realizes, and conforms to, a set of interfaces. According to this definition, a complete application per se can be considered as a component, as can a single class. In this paper, the term component will therefore refer to: a complete software application, a software component (subsystem) or an ordinary class constructed using an object-oriented programming language.

While software components have been subject to continuous enhancement, they still suffer from certain limitations, in particular the presence of unused functionalities [3,15]. Typically, a component "owns" a set of functions satisfying specific functionalities in a particular software application. Whenever a new application is developed, and when this application reuses a particular component, it actually reuses subsets of the data and functionalities possessed by that component. In other words, the complementary subsets of the component's used data and functionalities are unused, and therefore their presence provides no value to the newly developed application. Consequently, these sets of unused functionalities inefficiently consume computing resources such as memory and network bandwidth. Moreover, these unused data and functionalities can be a source of side-effects which might compromise application integrity and security if this behavior is exploited inappropriately.

The evolution of component development is expected to rely heavily on software engineering principles and practices such as software measurement methods. For instance, we believe that the application of software measurement methods on components could contribute to their quality, popularity, exploitation and widespread industrial acceptance. In this respect, it is of interest to assess the extent of a component's unused members and to provide the measurement results to software engineers so that they can make informed decisions regarding the suitability of that component to their applications. In [4], a measurement method – CUMM (Component Unused Member Measurement – was proposed to calculate the number of unused members a component has. Up to now, using CUMM to calculate a component's unused members was a manual process, and obviously an awkward and time-consuming one.

In this paper, we propose CoMet as an experimental prototype tool which uses CUMM to measure a component's unused members and their usage percentages. In section 2, background information on component-based software engineering and software measurement is presented. The CUMM method is presented in section 3, and the CoMet tool in section 4. Section 5 briefly presents current and future work, and section 6 presents a summary and a discussion.

2. Background

2.1 Component-based software engineering

Software engineering processes have evolved through several programming paradigms over the relatively short history of software development. Since its emergence in the early '90s, component-based software engineering has become a major trend in software development processes.

One of the goals of software engineering is to take the software construction process to higher levels of automation [5] in order to reduce software complexity and increase reuse. This transfer to higher automation levels is realized through software development techniques such as component assembly and the use of compositional languages.

The component-based software engineering (CBSE) process is divided into two distinct processes [6, 7]: component engineering and application engineering. The first deals with the analysis and development of domain-generic and domain-specific components, while the latter deals with software application development by assembly, composition, integration and the plugging in of components like COTS (commercial off-the-shelf), as well as others developed in-house.

The Software Engineering Institute at Carnegie Mellon University [8] uses the expression CBSD to refer to the process of software development by the assembly and integration of software components. Essentially, CBSE and CBSD refer to more or less the same process. The focus of CBSE is the development of software by assembling and integrating COTS and other existing types of components, with an emphasis on composition rather than on programming [9]. It assumes that certain software parts are common to several software applications; therefore, it would be advantageous to reuse them for reasons such as [8] these:

- Better COTS quality and diversity
- Pressure to reduce development and maintenance costs
- Use of standards and open systems, and the emergence of integration mediators such as CORBA ORBs (Object Request Brokers)
- Increase in the enterprise's inventory of potentially reusable software components

2.2. Component-based software construction

Software construction can be considered as a subprocess which matches the implementation phase in the software development life cycle. According to the SWEBOK (Software Engineering Body of Knowledge) [14], software construction per se is a software engineering knowledge area which encompasses the activities of software coding, validation and unit testing. This software construction subprocess must be instantiated taking into account four general principles (reduction of complexity, anticipation of diversity, structuring for validation and the use of external standards), as well as the tools used by this subprocess such as compilers, code generators and development tools [14].

As software design breaks software down into smaller parts for construction, those parts are expected to comply with the general principles of software construction. Interestingly, component-based software construction meets these four general principles. For instance, components can reduce the complexity of an application through being modular, reusable parts available from specialized suppliers instead of being developed in-house. In addition, being reusable and replaceable, they will comply with the principle of anticipation of diversity. Furthermore, breaking down a software application into modular components will make it is easier to validate them. Finally, software components generally conform to component models (CORBA, EJB, .NET), and as such they abide by standards.

2.3 Software measurement

Software measurement can help in the evaluation of software quality attributes, so that better software development decisions can be made and better process control can be exerted. Moreover, the application of software measurement is needed in order to move activities in the software engineering process from a set of craft activities towards rigorous and well controlled engineering activities guided by rigorous measurement. In this respect, the IEEE [10] 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 the empirical domain to be mapped to a numerical domain. In other words, software objects are measured in terms of quantitative values rather than qualitative ones. Over the years, various software measures have been proposed and classified into three major classes: process measures, product measures and resource measures [11].

Unfortunately, the way in which the empirical quantitative values of software measures are mapped to semantically meaningful qualitative values is still problematic [12]. Research in software measurement is aimed at providing sound models, methodologies and measurement frameworks to the design and use of software measures. The objective of these models, methodologies and measurement frameworks is to establish software measures for the software engineering discipline in a way similar to that in which measures are established in other engineering disciplines [12, 13].

Traditionally, several software measurement methods have been defined in the form of mathematical formulas for deriving numerical values. The calculation results are then used in various types of models for evaluation and decision-making purposes. It is observed that few of these measurement methods have been defined according to well-defined measurement processes.

Therefore, to ensure the soundness of the CUMM design, we used the measurement process model suggested in [13] and illustrated in Figure 1. This high-level model is a four-step roadmap to be followed in the design and validation of software measurement methods. 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 numerical values. The third step requires the analysis, documentation and auditing of the measurement result. In the fourth step, the actual exploitation of the measurement result will be carried out.

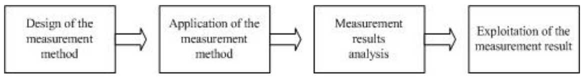

Design of the measurement method
Application of the measurement method
Measurement results analysis
Exploitation of the measurement result

Figure 1. Measurement process – high-level model [13]

3. Overview of the CUMM Method

3.1 The CUMM design

CUMM is a software measurement method which measures the number of software entity (component) unused members (attributes and functionalities), their usage percentages and their memory consumption in a host software application [4].

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

In addition, CUMM has been provided with ad hoc analysis models which use CUMM results to determine: a) the degree of generality of a software component's members, and b) the number of the component's unused members as a percentage.

When measuring the memory consumption of the elements (attributes and functionalities) of an entity, CUMM measures only the static memory consumed by the elements of an entity. In other words, CUMM does not calculate the amount of dynamic memory consumed by the objects created by an entity at runtime. The amount of an entity's consumed memory is the summation of the memory consumed by the binary code of that particular entity and the memory consumed by its aggregate classes in a recursive manner. More information on CUMM application and calculation can be found in [4] and in Box A.

The number of unused members (attributes or functions) is calculated by counting the number of times that member type is referenced in 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 of a component. 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 $u_f = |F|$.

Box A. CoMet Numerical Assignment Rules

3.2 Applying CUMM

The application of the method requires three substeps:

Substep 1: Gather the software documentation related to the entities subject to measurement. In the context of CUMM entities, they can be either topmost or inner entities, an inner entity being a component used by another, outer component which is referred to as the outer context of the entity. Documentation artifacts of an entity subject to measurement can be in the form of either the source or reflective binary code of: a) the entity's outer context, b) the entity itself, and c) recursively, any nested entities it owns.

Substep 2: Construct a software model by instantiating the CUMM generic metamodel (see Figure 2). 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.

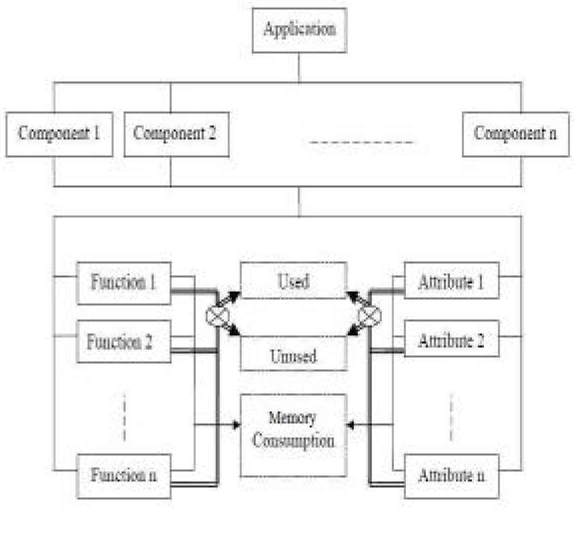


Figure 2. Generic metamodel representation of the application and its components

Substep 3: Apply 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.

4. CoMet (Component Measurement) Tool

4.1 Overview

The CoMet measurement process takes as input component binary code artifacts, performs measurement data collection, analysis and calculation, and provides the calculation results, as shown in Figure 3.

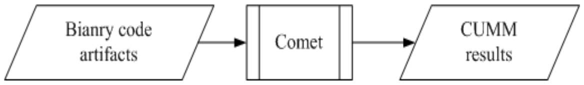


Figure 3. CoMet measurement process

4.2 CoMet environment

CoMet has been developed as a Java prototype tool to automate the CUMM measurement process. This automation considerably reduces the measurement effort that would otherwise be incurred had the measurement process been performed manually. CoMet is an experimental tool developed to measure components written in Java.

The current version of CoMet makes use only of the binary code artifacts of a software component to calculate the measurement results. There are several reasons for this: a) for certain components, especially proprietary ones, the only code artifact available for measurement is the binary code, b) it is relatively much easier to conduct measurement activities on the reflective binary code than on the source code, since the former is cleaner and more condensed, c) it is advantageous to reuse open source libraries to reduce the development effort required to build CoMet.

Internally, CoMet is developed as a collection of Java classes and reuses BCEL (Byte Code Engineering Library), an apache open source project for manipulating Java byte code [14].

4.3 CoMet realization

CoMet is designed with a GUI interface through which the user specifies the component to be measured and observes the measurement results reported when the calculation has been terminated (see Figure 6).

CoMet takes as input the name of the component to be measured (see top of Figure 6) and the binary code of the application which uses that component. The binary code of the application must be available on the classpath of CoMet. As output, it generates a report detailing the measurement results. The information presented in the report shows the number of unused members (attributes and methods) and the percentage of their usage, as shown in Figure 5.The CoMet analysis method is a two-part process – source analysis followed by measurement data extraction. During the execution of the first process, the component-referenced member types are indexed using a recursive depth-first transversal algorithm. When the recursive algorithm terminates, the analysis of member methods begins. The analysis of member method results generates a call graph and collects relevant measurement data. The second major step consists in: a) transforming the measurement data collected into meaningful measurement information through the application of the CUMM calculation formulas, and b) presenting the measurement to the user.

An object model (see Figure 4) was developed to represent the measurement data harvested from the indexing process to allow simplified subsequent transformation. In this model, classes and members are separated, so as to permit direct access and accelerate the look-up process during the member analysis. The model registry class acts both as a repository and a factory to prevent duplicate representation of identical data. In addition, the model was designed in a way which facilitates an eventual integration with a database for data persistence and optimization of subsequent analysis of data.

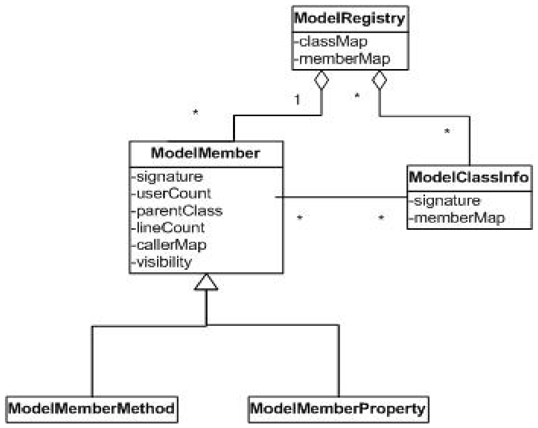


Figure 4. CoMet Object Model (simplified)

4.4 CoMet in action

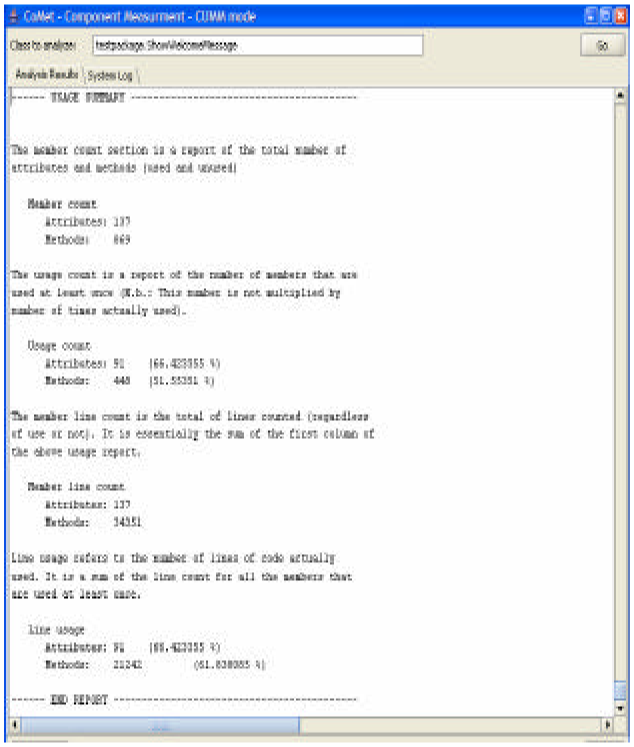
To illustrate the features of CoMet, a small example is used (see Figure 5) containing a component consisting of the class ShowWelcomeMessage with one data member and one method member. As shown in Figure 6, and, as unexpectedly as it might seem, CoMet calculated the total number of methods used by this component as 869. Obviously, this number of methods comes from: a) the inherited methods from the Object father class of all Java classes, b) the methods of all the classes (String, System, etc.) used by the ShowWelcomeMessage, and c) recursively, the methods inherited by those classes used by the ShowWelcomeMessage component. The ShowWelcomeMessage component in its current state makes use of 448 methods. Expressed as a percentage, the ShowWelcomeMessage component makes use of only 51.55 percent of the loaded methods. As for the number of data members of the ShowWelcomeMessage component, the same argument holds as for the method members.

```
public class ShowWelcomeMessage {
  static String message;

  Public static void main(String[] args) {
    message = new String("Hello
                   from CoMet!");

    System.out.println(message);
  }
}
```

Figure 5. Example


CoMet - Component Measurement - CUMM mode
Class to analyze:
testpackage.ShowWelcomeMessage
Go
Analysis Results
System Log

```
------ USAGE SUMMARY ----------------------------------------


The member count section is a report of the total number of
attributes and methods (used and unused)

  Member count
    Attributes: 137
    Methods:    869

The usage count is a report of the number of members that are
used at least once (N.b.: This number is not multiplied by
number of times actually used).

  Usage count
    Attributes: 91    (66.423355 %)
    Methods:    448   (51.55351 %)

The member line count is the total of lines counted (regardless
of use or not). It is essentially the sum of the first column of
the above usage report.

  Member line count
    Attributes: 137
    Methods:    34351

Line usage refers to the number of lines of code actually
used. It is a sum of the line count for all the members that
are used at least once.

  Line usage
    Attributes: 91    (66.423355 %)
    Methods:    21242        (61.838085 %)

------ END REPORT ----------------------------------------
```

Figure 6. CoMet measurement results

4.5 Current limitation

CoMet in its current state has a limitation as to the functionalities implemented so far to automate CUMM application. For instance, if there are component members used solely inside the body of a member method, they are not yet taken into consideration. Consequently, the CoMet calculated measurement results are not completely accurate in some instances. However, they do still provide a relevant indicator of the number of a software component's unused members and their usage in a software application.

5. Current and future work

Currently, our research is focused on addressing the implementation issue that leads to inaccurate results in certain instances, as mentioned in section 4.3. Now, a number of features can be added to CoMet, and in future work we are considering the addition of an implementation module to analyze a component member's byte code and to calculate the memory consumed by unused members.

Another possible enhancement would be to target the user interface, which could be useful in giving the user more control over: a) the components to be measured, b) the path-setting of the application that uses the component being measured, and c) the inclusion or exclusion, on an optional basis, of certain library components, such as Java API, so that they will be ignored during the measurement process.

Ultimately, we will consider conducting experimental measurement case studies covering a number of open source components to measure their members' percentage usage and memory consumption in different application settings.

6. Summary and conclusion

In this paper, we have presented CoMet, a tool to automate measurement of a software component's unused members according to the CUMM measurement method. The current version of CoMet provides a valuable indicator to users of CUMM. The results calculated by the measurement method have a cross-cutting impact on a number of ISO/IEEE standard [10] quality characteristics and subcharacteristics. For instance, unused functionalities could indicate that the security subcharacteristic of the application that makes use of these functionalities might be compromised. Similarly, the maintainability characteristic is impacted by the number of unused functionalities in terms of the effort that might be required when adapting and customizing the software components concerned.

7. References

[1] P. Maurer, Component-Level Programming, Pearson Education Inc., 2003.

[2] I. Jacobson, "Component-based Development with UML," 1998.

[3] M. S, Al-Hatali and H. G. Walton, "Smart Features for Compositional Wrappers," ICSR7 2002 – Workshop on Component-Based Software Development Processes, Austin, Texas, 2002.

[4] H. Msheik, A. Abran, H. Mcheik and P. Bourque, "Measuring Components Unused Functions," 14th International Workshop on Software Measurement (IWSM) IWSM-Metrikon 2004, Konigs Wusterhausen, Magdeburg, Germany, 2004, pp. 367-380.

[5] A. Abran, J. Moore, P. Bourque, R. Dupuis and L. Tripp, Guide to the Software Engineering Body of Knowledge – SWEBOK – 2004 version: IEEE-Computer Society Press, Los Alamitos, California, 2001.

[6] S. Ghosh, "Improving Current Component Development Techniques for Successful Component-Based Software Development", ICSR7 2002 Workshop on Component-based Software Development Processes, Austin, Texas, 2002.

[7] A. Rashid, "Aspect-oriented and component-based software engineering," IEEE Proceedings – Software, vol. 148, pp. 87-88, 2001.

[8] SEI, "CBS Overview." [Online]: http://www.sei.cmu.edu/cbs/index.html: Software Engineering Institute, 2003.
[9] C. P. Clements, "From Subroutines to Subsystems: Component-Based Software Development," in Component-Based Software Engineering: Selected Papers from the Software Engineering Institute,: IEEE Computer Society Press, Los Alamitos, California, 1996.

[10] IEEE, "Standard Glossary of Software Engineering Terminology," IEEE Standard 610.12, Computer Society, Los Alamitos, California, 1990.

[11] R. R. Dumke and A. S. Winkler, "Managing component-based software engineering with metrics," Fifth International Symposium on Assessment of Software Tools and Technologies, 1997.

[12] H. Zuse, A Framework of Software Measurement. Berlin, 1997.

[13] J.-P. Jacquet and A. Abran, "From Software Metrics to Software Measurement Methods: A Process Model," IEEE Third International Symposium and Forum on Software Engineering Standards (ISESS'97), Walnut Creek, California, 1997.

[14] Apache, [Online]: http://jakarta.apache.org/bcel/manual.html.

[15] H. Msheik, A. Abran and E. Lefebvre, "Compositional structured component model: handling selective functional composition," 30th Euromicro Conference on Component-Based Software Engineering, Rennes, France, IEEE 2004, pp. 74-81.