Integrating GQM and Data Warehousing for the Definition of Software Reuse Metrics

Marco Vieira, Henrique Madeira
University of Coimbra / CISUC
Coimbra, Portugal
mvieira@dei.uc.pt
henrique@dei.uc.pt

Sérgio Cruz, Marco Costa
Critical Software, SA
Coimbra, Portugal
scruz@criticalsoftware.com
mcosta@criticalsoftware.com

João Carlos Cunha
Polytechnic Institute of Coimbra / CISUC
Coimbra, Portugal
jcunha@isec.pt

Abstract—**Software reuse is the practice of using existing artifacts (code, architecture, requirements, etc.) in new projects. The advantages of using previously developed software in new projects are easily understood. However, reusing artifacts is usually done in an *ad-hoc* and incipient way, requiring an important effort of adaptation, so developers frequently prefer to develop components from scratch. In this paper we present a strategy that is being adopted by Critical Software, a medium-sized company, to promote software reuse. This strategy starts by assuming that the success of software reuse is dependent on the ability of measuring its advantages. We have thus proposed the use of the Goal-Question-Metric (GQM) technique, extended with Data Warehousing data model design concepts to extract a set of reuse-specific metrics for measuring the gains of reuse. We show that it is very easy to measure the productivity improvement due to code reuse, by simply measuring or estimating the efforts of developing a component for reuse, integrating it a new artifact, and developing this artifact, built with reusing the component.**

Keywords-software reuse; metrics; GQM; Data Warehousing

I. INTRODUCTION

It is widely accepted that metrics are essential in software engineering as an instrument to measure quality, or at least to monitor in what degree goals previously defined are actually achieved. "*If you cannot measure something, you cannot really understand it*" (Lord Kelvin) – is a well-known aphorism that emphasizes the need for effective software metrics to characterize a given software development process and product.

Opportunistic and non-systematic reuse is an omnipresent aspect of the software development process, as adapting/reusing previous solutions at all levels of software development (requirements, architectures, code, tests, documentation, etc.) is an inherent facet of the mental model of software engineers and programmers. Unfortunately, evolving from *ad-hoc* and incipient forms of reuse into systematic reuse approaches has proven to be a very hard problem [1]. In order to reduce development costs and improve software quality, many organizations have undertaken initiatives to foster software artifacts reuse. However, even though it is well accepted that software reuse is a key path, most reuse initiatives typically fail. Some key conditions that contribute to this situation are:

- Developing and maintaining software artifacts for reuse requires additional effort and cost for certification, upgrades, documentation, etc.
- Integrating new versions of software artifacts into already existing products may represent an additional cost on software maintenance.
- Reusing requires a huge knowledge about the reusable artifacts available, and developers seldom trust on the quality of artifacts developed by other parties.
- Building and maintaining a repository of software artifacts is complex, and requires a continuous effort.
- Reuse gains are not easy to measure and understand as relevant **metrics are typically not available**.

A key aspect is that software companies frequently fail on successfully promoting reuse. In fact, even well designed software reuse initiatives supported by convenient platforms, fail if the developers do not understand the advantages. In practice, if metrics are very important to characterize and manage the software development process, they are absolutely indispensable to monitor a reuse strategy [2].

In this paper we discuss how reuse metrics were defined at Critical Software SA (CSW), a medium-sized company that is currently recognized as CMMI Level 5 [3]. We present the problem of measuring reuse in the software development process and propose extending the Goal-Question-Metric (GQM) technique [4][5] with Data Warehousing (DW) data model design concepts [6] to allow the definition of a small set of representative metrics that satisfy the company's needs.

GQM is a systematic approach to specify goals and select metrics of interest and to design and develop a software metrics program. The idea is to use the goals and metrics steps of GQM as an input to define a multidimensional data structure. This data structure is a typical multidimensional schema such as those used for decision support in data warehousing and on-line analytical processing (OLAP) environments. Based on this, we extract a set of reuse-specific metrics for measuring productivity improvement, considering the reuse of code.

This paper is organized as follows. Section II addresses software metrics, providing basic definitions and discussing typical problems. Section III discusses the integration of GQM and DW. Section IV proposes a set of reuse metrics for characterizing productivity improvement. Finally, Section V concludes the paper and provides directions for future work.

II. SOFTWARE METRICS: BASIC CONCEPTS

The main goal of a measurement process is to characterize attributes or features of the entities under evaluation, where

metrics are used to quantify the attributes of interest. Metrics can be directly observable quantities, normally called **raw metrics** (e.g., number of: lines of code, of requirements, of documentation pages, of staff-hours, of tests, of defects, etc.), or can be **derived from raw metrics** (e.g., lines of code per staff-hour, defects per thousand lines of code, etc.). Derived metrics are also called **indicators**, especially when they are complex and represent a value with a clear meaning for the improvement of the process/entity under measurement (e.g., the evolution of a metric over time).

In general, metrics related to software products (i.e., not related to the process of developing software) can be divided in two groups: **structural metrics** that reflect the structure of the product and **functional metrics** that are focused on the functionality provided by the program. Typical structural metrics are program size and program complexity, which can be described by well-known raw metrics such as lines of code and cyclomatic complexity. The classic functional metric is function points. In general, structural metrics are easy to collect but they are not necessarily meaningful, as it is possible to find alternative programs for the same functionality with quite different size and complexity. Functional metrics, on the other hand, are normally much more complex and difficult to collect.

Metrics are generally not perfect, as they often allow a limited characterization of the attributes of the entity under evaluation. One of the aspects that has clear impact on metrics' precision (or simply imperfection) is the subjective nature of many software metrics.

Subjective metrics are dependent on both the object that is being measured and the viewpoint from which the measurement is taken, that may vary with the person involved in data collection. For example, readability of a requirement is a metric inherently subjective, as it is difficult to assure a consistent evaluation of such metric along time and people involved.

Objective metrics, on the other hand, depend only on the object that is being measured and can often be collected in an automatic or semi-automatic way, which greatly simplifies the process of gathering the measurements. Classic examples of objective metrics include lines of code and staff hours required to accomplish a given task. Even so, collecting objective metrics normally assumes the adherence to a predefined set of rules or standards to assure consistent and correct measurements (e.g., coding standards are necessary to assure a consistent measurement of lines of code).

Although the objective/subjective nature of metrics has a clear impact on **metrics precision**, it is worth noting that even objective metrics cannot fully characterize the related attributes in most of the cases. For example, if the goal is to characterize the structural attribute "program size", we may use one or more metrics such as lines of code, number of unique operators, number of unique operands, number of occurrence of operators, number of occurrence of operands, etc. Each of these metrics alone, or even all of them combined, just conveys an approximate characterization of the attribute "program size".

Another classic problem related to software metrics is the **cost** associated to the collection of the metrics and possible impact on the process/entity under measurement. As in general there is a variety of possible metrics to characterize a single attribute of interest, an important issue is to **identify the minimum set of metrics that can convey the best information possible** on the attributes of interest, with the minimum costs and interference in the process/object under measurement.

In summary, software metrics are direct or derived quantities that are collected from the entity under measurement and characterize a given attribute or set of attributes of this entity. Metrics normally characterize the related attribute in an incomplete way, either because metrics are usually not rich enough to fully describe the attributes, or because of the subjective nature of some metrics. The collection of the data always represents a cost, and may interfere with the process/entity under measurement, requiring great care in the selection of the set of metrics.

III. METRICS SELECTION APPROACH: INTEGRATING GQM AND DW

The very first question in any measurement program is **deciding what to measure**. This is an essential step and, unfortunately, the answer to this question is far from being trivial. Very often metrics are selected based on what is easy to collect or what looks interesting, without a clear reasoning that justifies the choice of metrics and without a plan for their utilization.

The classic Goal-Question-Metric (GQM) approach, proposed by Victor Basili [4], is a systematic way to select metrics of interest, and to design and develop a software metrics program. GQM considers three levels of interest in the measurement process: 1) conceptual level (**goal**) that states quality goals defined for the entity under measurement; 2) operational level (**question**) based on set of questions used to characterize the attributes of the entity under measurement deemed as relevant to achieve the elicited goal; and 3) quantitative level (**metric**) consisting of a set of metrics needed to answer every question in a quantitative way. Possible objects of measurement considered in the GQM approach are:

- **Products**: artifacts and documents such as requirement, specifications, designs, programs, test suites, etc., produced in the software development process.
- **Processes**: activities of the software development process such as interviewing, specifying, designing, coding, etc., which are normally associated with time.
- **Resources**: items such as personnel, hardware, software, office space, etc., that are used by processes to accomplish their outputs.

The hierarchical approach of GQM is normally described as a six-step process:

1. **Identify the goals**, normally expressing productivity or quality targets, which can be applied to several levels, from project level to the entire company.
2. **Generate questions** able to characterize the goals in a quantifiable way.
3. **Specify the metrics** needed to answer to the questions.

4. **Create the mechanism and tools** to gather metrics.
5. **Collect, validate, and analyze the data** (metrics) meant to provide real-time feedback to engineers and project managers, allowing the timely introduction of corrective measures.
6. **Analyze the data** to assess in what degree goals have been achieved and identify future improvements.

The big advantage of the GQM approach is that metrics are defined with clear goals in mind and goals are identified considering the object under measurement and the measurement purpose and perspective. Nevertheless, GQM and its variants and extensions have also some limitations and drawbacks that must be taken into account when designing a metrics program:

- **The number of metrics tends to be high**, as a consequence of the several questions needed to express a given goal. Furthermore, in many cases there are several possible metrics for each question, which may increase the cost and impact of collecting data.
- Goals, questions, and metrics tend to assume a many-to-many type of relationship that may originate **redundant metrics** that increase the cost of collecting data and may originate inconsistencies in the measurements (i.e., answer to a given question may vary according to the metrics used).
- **Metrics may not be easily mapped to the related questions** and goals, which may induce some bias, particularly in the case of subjective metrics.

Fig. 1 (adapted form [4]) illustrates the hierarchical approach of GQM and provides a graphical image of the problems mentioned above, such as the trend to originate a high number of metrics and the many-to-many relationships. Nevertheless, in spite of these difficulties, GQM (and the different variations of this technique) is the most commonly used approach to select software metrics and to design software metrics programs.

Our proposal to select the set of metrics is to use the GQM approach as first step of the analysis for the definition of the metrics. In other words, the GQM approach is used to identify the metrics requirements in the form of goals and questions. However, instead of trying to identify metrics that provide answers to the questions (in a direct or in an indirect way), as happen in the GQM approach, we use the goals and questions steps of GQM as an input to define a multidimensional data structure. This data structure is a typical multidimensional schema such as the ones used for decision support in data warehousing (DW) environments [7]. Actually, software metrics are the basic inputs of the decision support mechanisms for the improvement of software development process, thus identifying metrics using an approach inspired on the DW design process seem pretty intuitive.

In data warehousing the data is organized according to a multidimensional model, which includes two kinds of data: facts and dimensions. **Facts** are numeric or factual data that represent a specific business or process activity and each **dimension** represents a different perspective for the analysis of the facts. Each dimension is described by a set of attributes. Normally, data warehouses store the data in a

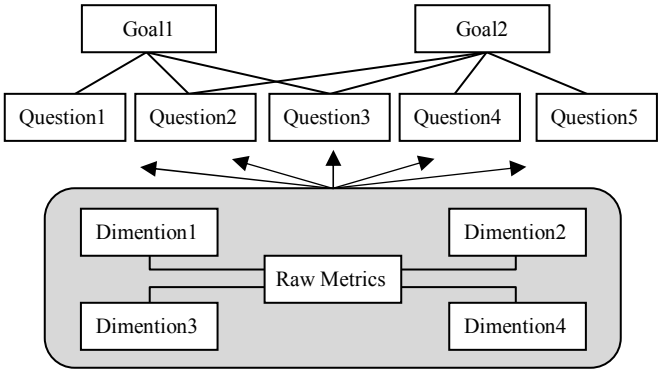

Figure 1. Data warehousing approach to define software metrics.

relational database [8]. That is, the (logical) multidimensional model is implemented as database tables in a data schema composed by a facts table surrounded by several dimensions tables connected to the facts table by the classic relational integrity mechanism of primary key-foreign key. This kind of data schema is known as star schema [7], as the central fact table surrounded by dimensions resemblances a star shape.

Using the DW & OLAP paradigm to define software metrics it is easy to conclude that facts correspond to the raw metrics and the dimensions represent the different viewpoints of analysis. For example, staff-hours is a fact (metric) while product, project, business unit, employee, etc., represent possible dimensions. One of the advantages of using the DW & OLAP paradigm is that the number of facts (i.e., raw metrics) tends to be reduced to the minimum number possible, as more elaborated metrics are obtained by the aggregation of basic facts. That is, unlike the GQM approach that tends to define new metrics for each question, using the proposed approach the answers for new questions are extracted from a small number of raw metrics through aggregation functions. This has several advantages:

- Reduces the number of metrics, with positive effect on the cost of gathering the metrics.
- Makes the mapping between the metrics and the goals less evident, as the metrics represent very low-level facts about the object under measurement. This is relevant to avoid bias in collecting the metrics, as many of them require human intervention to be recorded.

Fig. 2 represents the proposed approach to define software metrics using the GQM technique and DW concepts.

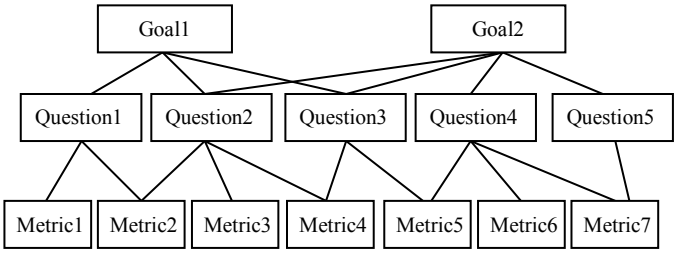

Figure 2. GQM approach.

IV. EXAMPLE: REUSE METRICS FOR MEASURING PRODUCTIVITY IMPROVEMENT

This section discusses general reuse goals that we have defined for CSW and proposes reuse-specific metrics, taking into account that our focus is primarily on the reuse of code. As proposed, the GQM approach is used as guideline for the identification of goals and related questions, and multidimensional analysis is used to guide us to the final set of metrics.

Although there are always a large number of goals (sometimes contradictory) associated to software development processes, considering the reutilization perspective we can divide the goals in two groups:

1. Goals related to the **impact of reuse on the process** and on its outcomes (the software products), namely improve productivity, reduce time-to-market/time-to-deploy, and improve software reliability.
2. Goals **related to the reuse process** such as increase of reuse and maximize return on investment (ROI).

In the rest of this section we focus on **productivity improvement** and identify the set of metrics required to answer all the possible questions related to that goal. The problem is to identify the smallest set of raw metrics that allow us to measure the impact of reuse on productivity. Some general questions related to the goal of improving productivity at CSW are:

- What is the impact of reuse on projects' productivity?
- What is the impact of code reuse on engineering units' productivity?
- What is the impact of reuse on the company's productivity?

A. Metrics for productivity in code reuse

Considering the **reuse of code**, it is necessary to define different types of code: 1) **new code**: lines of code developed specifically for a given object[1]; 2) **reused code**: lines of code reused, picked-up from the reutilization repository, and used without changes; and 3) **adapted code**: lines of code reused from the reutilization repository that required adaptations.

In order to measure productivity (with or without considering reuse or code adaptation) we need to characterize the objects and the effort required to produce them. This leads to two sets of metrics: **object metrics**, including **size** and **complexity**, and also the **effort** required for building each object. Size and complexity can be combined in order to give a measure of the **volume of the code**, which is more meaningful that just size or complexity alone. Although several combinations are possible, a reasonable approach is to consider that volume is proportional to the size and increases logarithmically with complexity, leading to the following formula [9]:

$$volume = size \times log(complexity) \quad (1)$$

Productivity (*prod*) considering code production[2] is then obtained by the following formula:

$$prod = \frac{volume}{effort} \quad (2)$$

Assuming that software artifact A was developed with the reuse of component C, the impact on productivity (*prod_impact*) related to reuse is simply calculated as the percentage of increment in productivity as compared with not adopting reuse ($prod_{A\_without\_reuse}$), as shown in the next formula:

$$prod\_impact = \frac{prod_A - prod_{A\_without\_reuse}}{prod_{A\_without\_reuse}} \times 100\% \quad (4)$$

To calculate this productivity impact, it is necessary to measure the effort to build artifact A ($effort_A$), and estimate the effort that would be necessary to build A without reusing C ($effort_{A\_without\_reuse}$), by using the formula (5), where $effort_{C\_integrate}$ (which can be estimated by the developers) represents the effort of integrating (i.e., reusing) component C in artifact A, and $effort_C$ represents the effort that would be necessary to build component C without reuse concerns:

$$effort_{A\_without\_reuse} = effort_A - effort_{C\_integrate} + effort_C \quad (5)$$

By checking several CSW projects, we have verified that the code volume with and without reuse is nearly the same. We have thus assumed that $size_A$ and $size_{A\_without\_reuse}$ are similar deducting, from formulas (4), (5) and (2), that:

$$prod\_impact = \frac{effort_C - effort_{C\_integrate}}{effort_{A\_with\_reuse}} \times 100\% \quad (6)$$

In the case we have to adapt the code of the reused module, the productivity impact can be calculated in the same way as in (6), where $effort_{C\_integrate}$ would represent the effort of adapting component C.

If the effort spent in the development of the reusable component was not recorded (e.g., because the component was not developed at CSW) it is possible to estimate the effort required to build such component by using average/typical productivity values for the technology/language used in the development of the code. That is, by knowing the size and complexity of the component (this can be measured using complexity tools) it is possible to estimate the effort of building the reusable component. In practice, the description of each reusable component available in the reuse repository must include an attribute stating the effort needed (or estimated) to build the component, which can be used to facilitate the calculation/estimation of productivity gains due to the reuse of that component.

The previously defined metrics are suitable for assessing the impact of reuse on the productivity of projects and engineering units, but insufficient if considering the company's productivity. At this level, it is necessary to take into account the extra effort required to produce reusable code, which is more expensive than normal code as it is obligatory to consider generic aspects of reusable objects. Thus, the effort needed to build a reusable component C is given by the sum of two terms: the effort needed to build the component C as a

[1] The word "object" is used to designate software at any level of granularity and includes entities such as classes, modules, components, programs, etc.

[2] The IEEE 1045 Standard for Software Productivity Metrics considers that coding effort includes both the code and the documentation.

specific component (i.e., not meant to be reused) plus the overhead due to the reusable aspects of C, as shown in the following formula:

$$effort_{C\_for\_reuse} = effort_{C} + effort_{C\_preparation\_for\_reuse} \quad (7)$$

The term $effort_{C\_preparation\_for\_reuse}$, representing the coding effort overhead due to reuse, although difficult to measure in a direct way (because the extra code needed to make C reusable tends to be mixed with the normal code of C), should be relatively easy to measure with acceptable precision by programmers.

A. Metrics to assess productivity improvement due to reuse

From the discussion above we can conclude that for reusing code we just need one basic raw metric: the **effort** required for building the object. From this very small set of metrics we will derive all the metrics needed to answer all the questions related to productivity in code reuse.

One important aspect is that the proposed set of metrics assumes that the effort spent in the software development process is always mapped to a tangible object. This has a clear impact on project management, as it is necessary to state explicitly the set of objects expected as output of each work package. In the case of work packages that do not lead to tangible objects, such as management, it is necessary to define virtual objects to assure consistency of the metrics structure.

The grain used to describe the objects is completely flexible. That is, it is possible to use fine grain code component such as classes or high-level components such as modules or programs. What is necessary is that the size and complexity of the object, and the effort needed to build it, are recorded for analysis.

Fig. 3 shows a simplified representation of the multidimensional structures needed to store the proposed metrics and some examples for possible records. The date is a common dimension in Data Warehousing that allows analyzing data over time (not mandatory in the present case). The two links between the facts table (effort) and the object dimension represent the target of the intervention and, in case of integration, the object in which a given object was integrated (column "Integrated in object" in the table). The examples in the table show, for instance, that component *compC* was built in the scope of project *Prj1*, and reused later on in artifact *artifactA*, for project *Prj2*, requiring some integration effort. In the meanwhile, *compC* had to be prepared for reuse, which required some preparation effort, spent by the engineering unit Unit1 (which, at CSW, consists of a Reuse Business Unit, currently being created).

Dimensions					**Facts**
Engineering unit	***Project***	***Object***	***Intervention type***	***Integrated in object***	***Effort***
Unit2	Prj1	compC	build		30
Unit1		compC	prep. reuse		8
Unit2	Prj2	compC	integrate	artifactA	3
Unit2	Prj2	artifactA	build		40
Unit2	Prj3	compC	adapt	artifactB	8

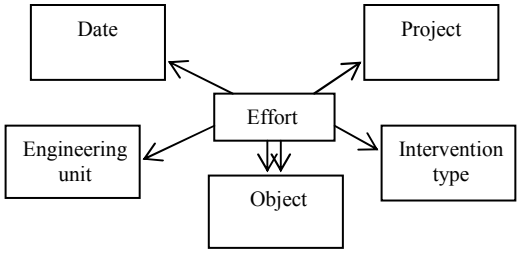

Figure 3. Multidimensional structure to store proposed metrics.

V. Conclusions and Future Work

This paper presented an approach to promote software reuse that is based on the ability of measuring the improvements derived from reusing software artifacts. The proposed approach extends the Goal-Question-Metric (GQM) technique with Data Warehousing concepts, allowing extracting a set of reuse-specific metrics and defining a database multidimensional model to store data.

As an example, we show that it is possible to measure the productivity improvement due to code reuse, by simply measuring or estimating the development effort of the reused component, its integration, and the new artifact built with the reuse of a component.

The proposed approach is being used at Critical Software. Future work includes the definition of additional metrics (e.g., related to reuse in different phases of the development lifecycle), the instantiation of the multidimensional models in a real Data Warehouse to store data, and the use of Online Analytical Processing to obtain useful information.

References

[1] Ivar Jacobson, Martin Griss, and Patrik Jonsso, "Software Reuse: Architecture, Process and Organization for Business Success", Addison-Wesley Professional, 1997.

[2] Jorge Mascena, Eduardo Almeida, Silvio Meira, "A Comparative Study on Software Reuse Metrics and Economic Models from a Traceability Perspective", IEEE International Conference on Information Reuse and Integration (IEEE IRI), Las Vegas, USA, 2007.

[3] Software Engineering Institute, "CMMI for Development, Version 1.2", Carnegie Mellon University, August 2006.

[4] Victor Basili, Gianluigi Caldiera, and Dieter H. Rombach, "The goal question metric approach", Encyclopedia of Software Engineering. Wiley, 1994.

[5] R. van Solingen and E. Berghout, "Goal-Question-Metric Method", McGraw-Hill, Ed. McGraw-Hill, 1999.

[6] Chauduri, S. and Dayal, U., 'An overview of data warehousing and OLAP technology', SIGMOD Record, Vol. 26, No. 1, pp.65–74, 1997.

[7] Ralph Kimball and Margy Ross, "The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling (Second Edition)", Ed. J. Wiley & Sons, Inc, 2002.

[8] Ramakrishnan, R., and J. Gehrke, "Database Management Systems", 3rd ed., McGraw Hill, 2002.

[9] W. M. Evangelist, "Software complexity metric sensitivity to program structuring rules", Journal Systems and Software, Vol. 3, Issue 3, 1983.