# Structuring Software Reusability Metrics for Component-Based Software Development

Danail Hristov, Oliver Hummel, Mahmudul Huq, Werner Janjic
Software Engineering Group
University of Mannheim
Mannheim, Germany
e-mail: {dhristov, hummel, shuq, janjic}@mail.uni-mannheim.de

*Abstract* — **The idea of reusing software components has been present in software engineering for several decades. Although the software industry developed massively in recent decades, component reuse is still facing numerous challenges and lacking adoption by practitioners. One of the impediments preventing efficient and effective reuse is the difficulty to determine which artifacts are best suited to solve a particular problem in a given context and how easy it will be to reuse them there. So far, no clear framework is describing the reusability of software and structuring appropriate metrics that can be found in literature. Nevertheless, a good understanding of reusability as well as adequate and easy to use metrics for quantification of reusability are necessary to simplify and accelerate the adoption of component reuse in software development. Thus, we propose an initial version of such a framework intended to structure existing reusability metrics for component-based software development that we have collected for this paper.**

***Keywords- Software Reusability; Software Reusability Metrics; Component-Based Software Development.***

## I. INTRODUCTION

The idea of software reuse [1] is not new: its roots date back to 1968 when McIlroy has presented his seminal work on reusable components [2] at the NATO Software Engineering Conference in Garmisch, Germany. However, there has only been limited practical experience with reuse until the late 1980s, when large-scale reuse programs were adopted by companies, mainly in the US (e.g., by IBM and Hewlett Packard [3]) and Japan (e.g., by Toshiba and Fujitsu); these efforts have also pushed forward the research in the 1990s, and in turn created a growing interest in systematic software reuse and reuse programs for organizations at that time [4]. After the turn of the millenium, widely available broadband internet connections and the raise of the open source movement have clearly created opportunities for broader inter-organizational reuse [5] and resulted in a huge amount of source code and components that is freely available [6]. Even the recently rising popularity of agile methodologies and practices has created numerous interesting ideas on how to facilitate reuse in that context [7,8]. However, as stated by Frakes and Kang, there are still numerous open issues to be solved [9] – including a better understanding of reusability.

Before being able to go into more details on the challenges tackled in this paper, we have to clarify what is not in its scope since reuse is a broad concept that cannot only be applied on components, but also on numerous other artifacts necessary for software development. Such artifacts are, for example, design structures [3,11,12], architectures [1,11,12], or even documentation [12]. However, the results presented in this paper will focus on software components, in both source code and binary form or in other words on the white- and black-box reuse of these software building blocks. In white-box reuse, the source code is available to the developer and can be changed before it is integrated into a new context, while in black-box reuse this is not the case and therefore only a component's interface (containing the public methods and attributes) and the documentation are visible [13]. Services in Service Oriented Architectures (SOA) [14] are conceptually certainly similar, but research on service reusability could not be considered in this publication due to space limitations. Black-box reuse probably tended to be the more facilitated approach in the past [15], however, the wide availability of search engines for open source software [7] has certainly brought the possibility of white-box reuse back into the center of interest .

It has been observed that software reuse research is rather scattered than focused and consecutive [9]. It is also evident that – though the software industry has developed massively in last decades – the paradigm of component reuse is still facing numerous issues and hence lacking adoption from practitioners. One of the central impediments that prevent efficient and effective reuse today is the difficulty to determine which artifacts are best suitable to solve a particular problem in a certain context and how easy it will be to reuse them. In other words, no comprehensive framework describing the reusability of software and structuring appropriate metrics exists in literature so far. Nevertheless, a good understanding of software reusability as well as adequate and easy to use metrics for its quantification are crucial in order to facilitate the adoption of reuse in software development. The focus of the research presented in this paper is on the reusability of software components in an ad-hoc reuse scenario. By "ad-hoc reuse scenario" we mean the spontaneous decision of a developer to use a component repository or search engine [7], indexing e.g. open source software not specifically built for being reused, in order to search for a component that might match the given requirements. This type of software reuse is probably one of the most promising that does not require larger upfront investments as for example the creation of a software product line [17] or a planned reuse program [4]. Nevertheless, it is likely that the insights gained from this work can be transferred to the latter areas where a measure for judging the reusability of software artifacts is also important.

Hence, in this paper we are distilling an initial framework for structuring software reusability metrics in component-based software development based on a comprehensive survey of metrics proposed in the literature. Its remainder is organized in the following order: in Section II we discuss the general concept of software reusability before we turn to the current state of the art in reusability metrics in Section III. In Section IV we propose our novel software reusability framework for the context of ad-hoc reuse approaches. The paper is finally concluded with a summary of its findings and an outlook on potential future work in Section V.

## II. REUSE & REUSABILITY

Literature provides a significant number of definitions for software reuse; probably the most popular one was published by Krueger [1] who has defined software reuse as the use of *"existing software artifacts during the construction of a new software system"*. Software reuse can be embraced in several different ways: on the one hand, it can be practiced in a structured and controlled manner inside organizations, where software artifacts are systematically designed for reuse when created according to a software reusability policy [12]. In this case, the artifacts can be usually reused within a particular domain, which is nowadays well-known as the paradigm of software product line engineering [4,9]. It is based on the presumption that most software systems are not new but rather a variation (or improvement) of already existing systems in a domain [9]. For such software reusability policies to succeed, they have to be systematic [18] and well planned. Thus, such a scenario of reuse is also known as planned reuse which indicates that it requires up-front investments by the organization implementing it, for example, designing the software for potential future reuse, establishment of libraries of reusable components, etc. [3,19]. Such systematic software reuse has been the central topic of a substantial amount of research papers (as, e.g., [12]).

Another interesting scenario is ad-hoc reuse [3, 19]: in this case, the artifacts for reuse are taken from generic libraries or search engines. This usually happens on an individual basis (i.e., per developer) and not per project or company. Here, the role of the libraries and retrieval mechanisms is of high importance [20]. With the fast growth of the World Wide Web and the possibility to store and retrieve large amounts of data online, it has become much easier to distribute reusable assets over the Internet even between organizations [21,22]. According to the body of existing literature, practicing this kind of reuse in software development can bring substantial benefits to organizations as well as the developers [9,19,23,24]. The most widely expressed and discussed benefits of reuse are: a productivity and quality increase, easier maintainability and higher portability.

An important prerequisite for every kind of component reuse is a *"repository for storing reusable assets, plus an interface for searching the repository"* [12]. In case of planned reuse, companies need to implement and maintain an internal repository of reusable components to store the assets produced and keep them ready for reuse [23]. For ad-hoc reuse, the components are usually stored in a publicly available library, accessible over the Internet (cf. e.g. [7] for an overview). However, the issue with the efficient retrieval of components suitable for reuse is still not completely solved. However, our screening of literature did not uncover sufficient empirical evidence of practitioners and organizations actually experiencing the expected benefits. Therefore, it should not be concluded that the mere availability of prerequisites for reuse alone will increase productivity and quality in software development already. The characteristics of the available artifacts also have to be taken into consideration.

As for software reuse, a large number of definitions for *software reusability* can be found in existing literature, e.g., Kim and Stohr [19] have defined software reusability as *"a measure for the ease with which the resource can be reused in a new situation"*. It is important to distinguish between software reuse and reusability as the former is focused on the practice of reuse itself while the latter tries to make the potential of artifacts for being reused measureable. Poulin [25] has stated in this context that knowing what makes software reusable can help us learn how to build new reusable components and to identify potentially useful (and thus reusable) modules in existing programs. The literature lists several characteristics of software, which are believed to determine reusability and are therefore repeatedly referenced in research papers [8,13,20,25]. Such factors are: adaptability, complexity, composability, maintainability, modularity, portability, programming language, quality, reliability, retrievability, size and understandability. Furthermore, the reusability of a component in a certain context should be comparable to the reusability of other – potentially functionally equivalent – software components in the same context. However, most of the existing research is rather incoherent and only covers one or a few of these aspects so that to our knowledge there is no publication that has tried to bring all these aspects together in a single model.

## III. EXISTING REUSABILITY DEFINITONS

In order to get an impression of reusability definitions available in the literature, we performed a systematic literature review that identified a number of articles proposing quantitative metrics for assessing the reusability of software. For that purpose, Google Scholar, IEEE Xplore, ACM Digital Library, Citeseer and Springer Link were searched with the keywords "software reusability". Titles and abstracts of delivered publications were read in order to determine whether they could contribute to the aim of our study, which resulted in a total set of 73 papers that were deemed worthwhile to be more closely read and investigated. In general, we found that some of the metrics described were newly developed solely for the purpose of measuring reusability, while others were modified or adapted from other areas (such as maintainability measurement) and have not been initially developed with reusability in mind. For the sake of practicality, we have separated the results of this survey, i.e., the discovered metrics, into two categories: one for white-box (allowing to look into the code of the components) and one for black-box (where usually merely interface and documentation of a component are available) reusability. This separation helps to distinguish the different nature of metrics for these two paradigms. Within these two categories that are presented in the following two subsections, the results of previous research are presented in chronological order to illustrate the development of reusability measurement. Due to limited space, we can only briefly describe most of these contributions; the interested reader is referred to the original sources for more detailed information.

## A. *White-Box Reusability*

As early as in 1991, Caldiera and Basili [26] have defined three main (but still relatively abstract) attributes for assessing the reusability of components – reuse costs, functional usefulness and quality of components. These attributes were determined by factors, which are directly or indirectly measured by classic software metrics such as Halstead's Volume [27], McCabe's Cyclomatic Complexity [28] or other metrics such as Regularity and Reuse Frequency [26]. Volume was used in order to estimate two important attributes, namely reuse costs and quality of components. The Cyclomatic Complexity was used to assess all three reusability attributes introduced before,. Regularity was used to assess the former two attributes, while Reuse Frequency was merely used to assess functional usefulness.

Seven years later, Barnard [29] has suggested a composite metric for reusability of object-oriented software, which was derived from two empirical experiments. As foundation, again a variety of readily available software metrics have been used. Based on the experiments, those metrics that were related best to reusability have been selected (with corresponding confidence intervals). In order to come up with this relation, classes from C++, Java and Eiffel libraries have been considered in the experiments, assuming that classes in libraries are more reusable. Barnard's metric suite is focused on the Simplicity, Genericity and Understandability of classes' interfaces, methods and attributes.

Around the same time, Mao et al. [30] have investigated the effects of inheritance, coupling and complexity on the reusability of classes in object-oriented software. Two years later, Lee and Chang [31] proposed another set of metrics for measuring the reusability and maintainability of object-oriented software. The determining criteria here are complexity and modularity. The corresponding metrics are Internal and External Class Complexity (for complexity), and Class Cohesion and Class Coupling (for modularity).

In 2001, Cho et al. [32] have suggested metrics for component complexity, customizability, reusability and reuse. Component Reusability is determined by the functionality that the software components provide for their domain: it is the ratio between the number of interface methods in the component that provide commonality functions in the its domain, and the total number of interface methods in the component. The more commonality functions a component provides, its reusability is considered higher. Additionally to this metric, Cho et al. have suggested metrics for Component Customizability. They state that if the possibility to customize a component is not given, the reusability is low, since developers cannot adapt components for their purpose. Customizability is determined by the metric Component Variability, which is defined as the ratio of the number of customization methods to all methods in a component's interfaces.

Also in 2001 Etzkorn et al. [33] have published a model capturing reusability of object-oriented legacy software. They suggest a comprehensive metric suite covering different aspects of the reusability of individual classes. It is defined as the sum of metrics for Modularity, Interface Size, Documentation and Complexity of a class, each equally weighted.

Four years later, Bhattacharya and Perry [34] have stated that the usefulness of a software component depends not only on its internal characteristics, but also on the context in which it should be integrated. Therefore, they suggested reusability metrics measuring how well a component fits in a predefined architectural context. The prerequisite is that the (potentially reusable) components are adapted to the architectural description of the target system, which includes a description of the services needed by the system. They have proposed two metrics for measuring software reusability, namely Architecture Compliance and Component Characteristics. The Architecture Compliance metric is measured by three different sub-metrics: Architectural Component Service Compliance Coefficient, Architectural Component Attribute Compliance Coefficient and Architectural Component Behavior Compliance Coefficient. A higher value for the Architecture Compliance metrics indicates a more reusable component in a given context. The Component Characteristics metric measures the compliance of a component with regards to the data and functionality requirements of all attributes and services in the architectural description.

In 2008, Gui and Scott [35] have suggested revised formulas for established coupling and cohesion metrics in order to measure the reusability of Java components. They are proposing to measure to which extent classes are coupled together and to which extend their methods are cohesive. Additionally, they have considered transitive relationships and finally defined two metrics for measuring software reusability, based on their own versions of Coupling and Cohesion. The authors admit that additional determinants of a components' reusability exist; however they are not considered in their paper. Only very recently, Gill and Sikka [36] have proposed five new metrics for better assessing reuse and reusability in object-oriented software development. The metrics are Breadth of Inheritance Tree, Method Reuse per Inheritance Relation, Attribute Reuse Per Inheritance Relation, Generality of Class and Reuse Probability.

### B. *Black-Box Reusability*

To our knowledge, the first set of metrics for measuring the reusability of black-box components was proposed by Washizaki et al. [16] in 2003. They proposed to consider three main factors that are expected to affect reusability: understandability, adaptability and portability. Through an empirical analysis of Java Beans components, the authors have established thresholds for each proposed metric. For measuring the overall understandability, the metrics Existence of Meta-Information and Rate of Component Observability are defined. Rate of Component Customizability measures adaptability, while Self-Completeness of Component's Return Value and Self-Completeness of Component's Parameter measure portability.

In 2004, Boxall et al. [37] have proposed that the Understandability of a software component's interface is a major quality factor for determining reusability. To measure this, they have defined a set of metrics, including values such as Interface Size, Identifier Length or Argument Count. The authors have selected 12 components from different software systems in C and C++ to empirically validate their metrics and developed simple tools to automatically calculate them. The derived values have (merely) been compared against the expert knowledge of the authors judging the reusability of these components. Consequently, the authors have stated that more empirical research is necessary.

Again one year later, Rotaru et al. [24] have identified adaptability, composability and complexity of software components as determinants for their measure of reusability. The composability of a component is determined by the complexity of its interface. Adaptability is the ability of a component to handle environmental changes. Although a preliminary metric specification is given for all three aspects, the authors have stated that an empirical calibration is necessary to better understand its effects.

Only recently (in 2009), Sharma et al. [38] have proposed an Artificial Neural Network (ANN) approach to assess the reusability of software components. The authors have considered determining reusability by four factors: Customizability, Interface Complexity, Portability and Understandability. However, only customizability was quantitatively evaluated so far. The other three factors are only to be assessed qualitatively, i.e. merely ranked on a relative scale by experts.

### C. *Open Issues*

For the metric suites reviewed and presented in this section so far, the most evident shortcoming beyond their quite limited scope is that almost all lack a sufficient empirical validation of their prediction capabilities. Their expressiveness originates mainly from expert opinions and evidence mainly derived from small case studies so that it seems that research on reusability is largely underrepresented in empirical software engineering research so far. There may be many reasons for this situation: one possible explanation is that there is no agreement in the research community which software characteristics provide a sufficient basis for determining software reusability and which metrics for these characteristics are sufficient. In other words, there is no common understanding of what software reusability is and how it can be best measured, yet.

Furthermore, as can be seen in the metric sets we presented, they mainly focus on the technical characteristics of software, which may be inconsistent with the expectations of practitioners. Therefore, a more holistic approach to defining and measuring reusability is needed. We intend to address these issues in current research in the next section. Following the Goal-Question-Metric (GQM) approach [39], an initial proposal for a more structured and analytically justified reusability model will be developed. Thereby, the understanding of software reusability can be improved, which on the one hand should encourage researchers to invest more effort in empirically validating this (and similar) models and on the other hand should give practitioners the confidence they need for measuring reusability "in real life".

## IV. STRUCTURING REUSABILITY METRICS

In this section, the reusability requirements for software components will be explained and structured in a reusability requirements model. For this purpose the well-accepted Goal-Question-Metric (GQM) paradigm [39], for deriving appropriate measurements and metrics for software reusability will be used. Following this approach, the first step is to define the goal of this research work. It can be expressed as follows:

*Improve the reusability assessment of software components in an ad-hoc software reuse scenario from the developer's point of view.*

The purpose here is to *improve*, the issue is *reusability assessment*, the objects are *software components* and the viewpoint is that of the developer. Another element – that of the context (*ad-hoc reuse*) is added to the goal definition. It is not explicitly defined in the GQM approach, but it is important for the research presented in this paper and will be relevant for the further elaboration. The next step foreseen in the GQM approach is to define the questions resulting from the goal stated above. They can be expressed as follows:

- which are the requirements to the software components that can determine their reusability in an ad-hoc reuse scenario?
- which are the characteristics of the software component that determine their reusability in an ad-hoc reuse scenario?

Obviously, these questions are not trivial and it is not possible to give an answer to them directly through identifying the appropriate metrics and hence a more sophisticated elaboration is needed in this case. Therefore, it is helpful to look at the following common ad-hoc reuse scenario (sometimes also called opportunistic reuse [3]) to better identify the needs of their users: usually, a developer (e.g., a software developer) would start thinking about the possibility to reuse a software component when he or she receives a task to develop certain functionality in the software system that she or he is working on. He or she would have two possibilities – (1) to develop this functionality from scratch or (2) to reuse already existing code that provides as much of this functionality as possible.

It is also necessary to stress that (2) would be a valid option only if there were no managerial, organizational or other company-internal obstacles preventing people from reusing code [29] that might come from a 3rd party and of course must be accessible in some kind of repository [17]. Looking into these two alternatives, the developer is likely to choose alternative (2) – the reuse of a software component – if the expected effort (or the corresponding costs) in case (2) is less than the effort (or costs) in case (1) [11]. The costs in (1) could be derived by monetizing the effort invested in the short term and middle/long term activities performed by the developer, and the costs in (2) could be derived by the effort invested in the short term and middle/long term activities (such as searching, integrating and testing a component) of the developer plus possible royalties that need to be paid to the creator or vendor of the component.a

To summarize, reusability requirements can be divided into functional and non-functional requirements as presented in the following. The functional requirement is that, in order to be considered for reuse in a particular context, the component(s) need to provide the functionality requested by the developer. There are two possibilities to address this requirement. The first possibility is to consider this a "hard" , i.e. a "yes or no" requirement. This means that a component would be considered for reuse only if it fully satisfies all needs specified by the developer. The second possibility is to consider the functional requirement as "soft". In this case, also components that do not fully satisfy the requested and specified functionality are included in the consideration set. In this case, the developer has to change/adjust the functionality of the component before reusing it, or to find a workaround, which clearly also influences the perceived degree of reusability.

The non-functional requirements, which determine the reusability of a software component, can be derived from the ad-hoc reuse scenario and the factors affecting the decision on reuse just explained and can be structured as follows:

- Fast and easy to retrieve: in order to consider a software component for reuse, it has first to be found by the developer. The easier and faster a component can be retrieved, the less effort it will take to reuse it.
- Licensed for reuse in the particular context: If the component is readily approved for reuse in the context of the developer, she or he can directly implement it and thus save time. If there are any legal concerns, they have to be clarified and settled first, which will require additional effort.
- Usability: software components, which are more usable from the viewpoint of the developer, will be preferred. This can have several dimensions: satisfying quality of the software component, easy to understand how it is built and structured, guaranteed maintenance of the component in the future etc.
- Inexpensive: if a component is too expensive, this will increase the overall costs (or their equivalent effort expressed in the developer's overall effort) and thus make the reuse alternative unattractive.
- Easy to adapt to a new usage context: The easier the component can be adapted to the context of the developer, the less effort it will take to implement it.

Overall it can be said that the better a software component meets these requirements, the higher is the probability that the developer will select it for reuse.

### *A. Core Elements of Reusability*

Our reusability measurement model does not aim on identifying new characteristics of software components determining reusability while rejecting the existing ones, but rather focuses on structuring them better and identifying a common superset of these characteristics to determine the reusability of software components. The match between reusability requirements and characteristics is obviously an *n-to-m* relationship. This means that one characteristic can address many requirements, and in the same time one requirement can be addressed through various characteristics. Therefore, the core measurement model is defined based on the following characteristics distilled from the reusability models presented in the last section:

- **Availability:** the availability of a software component can determine how easy and fast (or hard and slow) it is to retrieve it, this is not to be confused with the opertational availability often used in the context of long-running (server) systems.
- **Documentation:** a good documentation can make the software component more reliable since it makes it easier to understand. Furthermore, it should contain the legal terms and conditions and thus make clear if it is licensed for reuse in the context of the developer or if any legal issues may arise.
- **Complexity:** the complexity of a software component determines how usable it is (i.e., if it possesses satisfying quality, if it is easy to understand and to maintain) and how easy it is to adapt the software component in the new context of use. The rationale behind this is that if there are two components which provide the same functionality (which is the prerequisite for assessing their reusability), then a lower component complexity would mean that functionality is implemented more efficiently. Thus, it is likely that the implementation of this functionality in the component is of higher quality, is easier to understand by the developer and easier to maintain in the future, and it will be easier to adapt in a new context.
- **Quality:** the quality of the component directly determines how usable it is in a given context. The quality of a component is regarded as a characteristic which describes how good it fulfils its requirements and also how error- and bug-free it is. This can include a number of sub-characteristics, e.g . whether it often crashes when it is used, whether it is thoroughly tested and whether it provides suitable test cases to be tested when integrated in a new context.
- **Maintainability:** The maintainability of a software component directly determines how usable it is for reuse. After the integration into the new system, the

component should be able to adjust to the changes in the system along with its evolution (e.g., in future versions of the system). This can be facilitated through appropriate methods that the component provides to the developer or simply through providing changeable source code of the component (this is of course only possible for white-box components).

- **Adaptability:** This characteristic directly determines how easy it is to adapt a software component to the context of the developer. Otherwise, the availability of compatible adapters will increase the ease of adapting, compared to components for which such adapters have to be developed from scratch. Apart from the programming language, the design of the component and the availability of appropriate methods and interfaces to modify, adapt and bind the component to the software landscape of the developer are of high importance.
- **Reuse:** The actual reuse of the component can also be used to infer how usable and how easy it is to adapt it. The amount and frequency of reuse, especially in contexts similar to that of the developer can serve as reference points and she or he may select the component with the higher amount and frequency of reuse.
- **Price:** the price of the software component determines how expensive it is to reuse.

An illustrative overview of the elements influencing reusability measurement as just discussed is presented in the following figure, the concrete metrics contained there are briefly discussed afterwards as well as potentially necessary distinctions between white- and black-box reuse (i.e. for source and binary components).

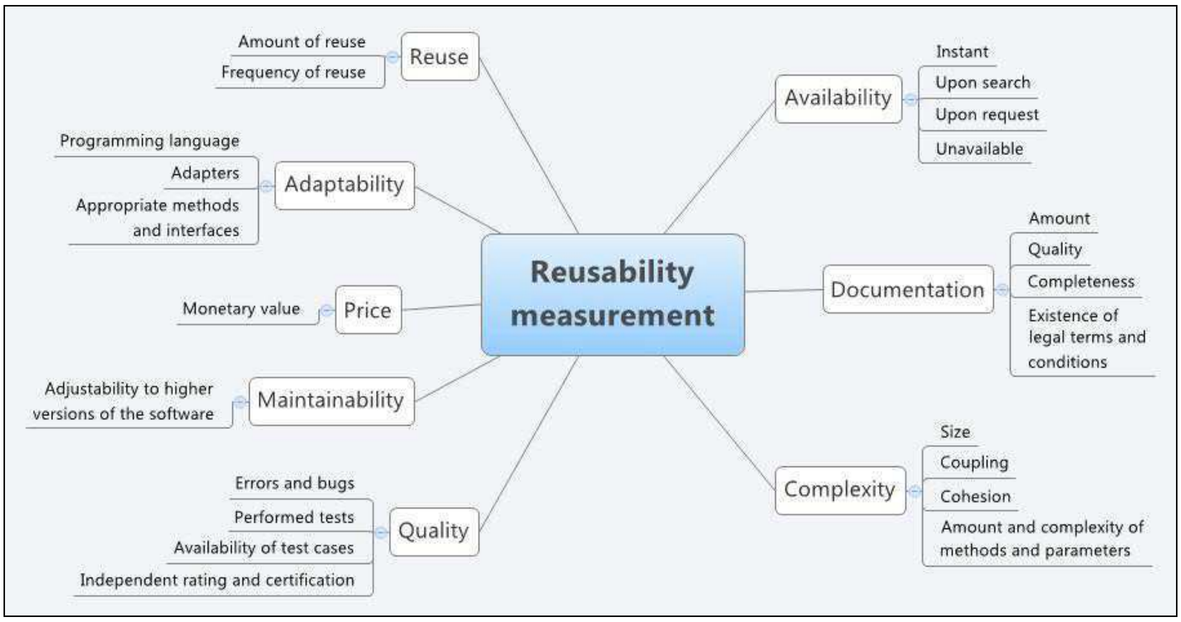

Figure 1. Factors influencing reusability measurement.

In order to calculate the reusability *(R)* of a software component *(c)* in the context of the developer *(c)*, the individual parts of the measurement model have to be quantified through metrics first, and then these metrics have to be aggregated in a reusability calculation model. Based on the characteristics introduced above, it can be defined as follows:

$$Rc_c = w_1 \cdot Avail + w_2 \cdot Doc + w3 \cdot Compl + w_4 \cdot Qual + \\ w5 \cdot Maint + w_6 \cdot Price + w_7 \cdot Adapt + w_8 \cdot Reuse \quad (1)$$

$w_1$ - $w_8$ are weights and the rest are composite metrics for the attributes from the reusability measurement model. To facilitate the comparison of the reusability of different components in the same context, these values should be adjusted to a common scale, e.g., normalized to the range [0..1] since this is common for software metrics, but not always done for the metrics presented before. The values of the weights determine the importance of each characteristic of the component for its reusability and have to be determined empirically or through expert opinion. Their sum has to be equal to 1 (because of the normalization). As to the other metrics, their absolute values should be calculated first and then normalized such that a minimal and a maximal value need to be found for each metric [38]. These values can be absolute min/max values found by analytical methods or empirical values derived from the components in a large enough consideration set.

## B. *Concrete Metrics Proposals*

Based on the insights gained from surveying numerous reusability definitions, we have been able to distill the following preliminary suggestions for concrete reusability metrics within each characteristic as presented below:

- **Availability**: a generic, qualitative and subjective metric can be used. The alternative values are: *instant*, *search with automatic aid* (e.g., an online library), *search manually* (e.g., via manually screening software systems for suitable components), upon *request* and *not available*. These values have to be placed on an ordinal scale (by, e.g., an expert – therefore the metric is subjective) and normalized (as all other metrics) to fit in the overall calculation of reusability.
- **Documentation**: to be determined by four different attributes*: amount*, *quality*, *completeness* of documentation and *availability of appropriate legal terms* and conditions. The amount is a generic, objective and quantitative metric that can be measured through its size, e.g. in kilobytes (kB). The quality is a generic, subjective and qualitative metric that can be measured on an ordinal scale (from, e.g., poor to very good) set by an expert. The same applies to the completeness. The existence of legal terms and conditions is a boolean metric: either this information is provided, or it is not.
- **Complexity**: The complexity of software is a widely researched topic and numerous metrics have been suggested in the literature. Therefore, it makes sense to use some of them for assessing the complexity of software components. The complexity intended here is a composite metric of the size of the component (e.g., in Lines of Code (LOC), excluding the LOC containing only documentation, i.e. comments) and complexity metrics for the classes, methods and parameters of the component, as well as their coupling and cohesion. It should be noted that the application of these metrics will be different for white-box and black-box components.
- **Quality**: generally, it is difficult to assess the quality of code in software engineering. This may often be subjective and inaccurate. In the narrow understanding of quality in this first version, it can be assessed via four attributes: the *number of bugs*, the *number of tests performed* (and their coverage and outcome), *availability of test cases*, provided along with the component, and an *independent rating* and certification. The first two attributes can only be collected through the lifetime of the component and may not be available. They are generic, objective and quantitatively measurable values. The availability of test cases is a context-based, subjective and qualitative metric. The best option would be to provide ready-to-use test cases which fit to the testing environment of the developer. An ordinal scale set by an expert seems reasonable here. A rating can be provided by experts or by other developers who have already reused the component ("wisdom of the crowd"). Such a rating can be an ordinal value, which is subjective, context-based and qualitative.
- **Maintainability**: The difference between maintainability and adaptability of a component is basically the perspective, the former is more concerned with the source code of the component while the latter is focused on its interface. Otherwise, the idea of both is how easy it is to adjust the component to a new context and hence, metrics related to the maintainability of the component are also included in the adaptability metric below. Therefore, the preliminary maintainability metric presented here will include only one additional aspect: the availability of the source code (as available for white-box components). A boolean metric is thus sufficient for this calculation. In the long run it makes sense to incorporate more detailed characteristics such as changeability etc. from the maintainability research community. However, the effect of the metrics used there (such as LOC, Cyclomatic Complexity, Volume etc.) are not yet well understood so that their impact on reusability is also not clear.
- **Price**: A generic, objective and quantitative metric, expressed through a predefined currency (conversions between currencies are possible).
- **Adaptability**: one important aspect of the adaptability is the programming language, and another is the availability of appropriate methods and interfaces for adapting the component [40]. The first aspect is context-dependent, subjective and qualitative. The possible values are: same programming language of the component and the context of the developer, different language with available and suitable component adapters, different programming language with no available suitable adapters. Again, these values have to be placed on an ordinal scale by an expert, while considering the similarity of the programming languages (e.g., it may be easier to adapt a C component to C++ then to Java). The second aspect should be addressed by some of the metrics in chapter 3 – the adaptability metric Rate of Component Customizability (RCC) from the metric suite of Washizaki et al. [16] seems useful in this case. The different applicability of adaptability metrics to white-box and black-box components has to be considered here, since the latter lack the possibility to change their code.
- **Reuse**: can be determined by the amount and frequency of reuse. Both are generic, objective and quantitative metrics. The amount is the overall

number of reuses of the component, and the frequency is the number of reuses for a certain period of time (e.g., the last week, month, year etc., or since it is available).

As stated before, these metrics are currently only suggestions for quantifying the reusability measurement and could be used as a starting point for quantitative empirical research. Their completeness and accuracy in measuring the reusability characteristics have to be empirically validated and synchronized with insights from other communities (such as those investigating software quality, complexity or maintainability).

### *C. Discussion*

This chapter presented a holistic analytical approach for assessing the reusability of software components in an ad-hoc reuse scenario. The literature survey, which was carried out in the beginning of this research, did not identify any other publication that has conducted such extensive analysis of this topic. The GQM approach was used as a formalization technique for the analysis in order to increase its expressiveness. It was argued that, in order to identify appropriate and reasonable metrics, a reusability requirements model and a reusability measurement model have to be defined first. This also corresponds to the guidelines of the GQM approach.

Clearly, the major drawback of this analysis is the missing empirical validation of the proposed measurements and metrics, since the usefulness and practicability of the suggested models can only be proven by conducting empirical case studies and statistically significant tests using real-life data from existing libraries for reusable components. Therefore, the next logical step would be to implement a reusability model based on these metrics by following the guidelines provided in the previous subsections, and to adjust the calculation model until it satisfies the needs of practitioners. It is also possible that the calculation model has to be adjusted for different implementation scenarios – e.g., for implementation in a company-internal reuse library and a freely available online library. The metric suites described in chapter III should be the first source to look into when searching for alternatives or extensions.

Additionally, empirical investigations need to establish thresholds that the reusability values of the components have to beat. These thresholds should reflect the effort that a developer is willing to invest in the case of developing the functionality in-house (similar to the idea of Halstead's Effort metric [16]). However, this is a non-trivial task, since the abstract and technical characteristics of the software component (included in the reusability measurement model) need to be translated into time, cost or effort values (e.g., "How much effort will be needed to reuse a component in a particular context, if its complexity has the value *X*?"). Obviously, a lot of further research is needed in this area, but since this clearly creates a considerable amount of effort (that has not even been spent for most other software metrics so far), we are forced to limit ourselves on merely sketching the idea of this model for the time being.

Another issue arises from the white-box and black-box nature of the components. If there is a mix of such components in the consideration set, it might become difficult to compare their reusability. This is an issue of the granularity of measurement – the object of measurement for some of the component's characteristics (e.g., the complexity or adaptability) is different. In the white-box case, these metrics can be calculated on the basis of the whole component, and in the black-box case, they can be calculated based only on the parts of the components made available to the developer – usually the interfaces with their methods and attributes. Further research is needed to define guidelines for comparing the reusability of white-box and black-box components.

In general, we believe that the reusability models defined here will bring more clarity and better structure to reusability research and have the potential to become a new starting point when it comes to assessing reusability. Once this field has made further progress, it becomes more likely that practicing reuse of software components will increase and be more efficient.

## V. CONCLUSION & FUTURE WORK

In this paper we have surveyed the current state of research on software reusability: available metrics for the reusability assessment of code in the object-oriented and component-based software development were presented and evaluated. Moreover, a revised comprehensive definition for the reusability of software components was proposed by following a structured analytical approach.

We have identified that the reusability of a software component is a context-specific characteristic that can vary in different application scenarios. This is an important aspect, which affects the definition of reusability and the implementation of its measurement. The reusability of a software component depends on various non-functional characteristics while fulfilling functional requirements (i.e. providing the desired functionality) is a prerequisite for assessing the reusability at all. It has become evident that reusability is a highly complex characteristic and its quantitative assessment is a non-trivial problem (as is the case for most other software quality characteristics). It may not be possible to fully automate the calculation in the near future so that human intervention may always be necessary.

The proposed software reusability measurement models and metrics in this paper still lack empirical validation so that is a logical next step in reusability research that should be addressed in the future. Moreover, the specific issues of setting practical threshold values when assessing reusability and comparing reusability of different components have to be addressed by researchers. Otherwise, it will be difficult to implement the model in practice. If future research succeeds to overcome these open issues in determining software reusability, it is likely that it will be a major step towards a wider adoption of the component reuse paradigm by both academia and business, which in turn can be seen a cornerstone for the further improvement of recently spreading software search engines and component marketplaces.

## REFERENCES

[1] C.W. Krueger, Software reuse, ACM Computing Surveys, vol.24, June 1992, pp.131-183.

[2] M.D. McIlroy, Mass produced software components, Software Engineering: Report on a conference by the NATO Science Committee, Garmisch, Germany, 7th to 11th October 1968, Naur, P., Randell, B., Eds., NATO Scientific Affairs Division, Brussels, Belgium, 1969, pp.138–150.

[3] R. Prieto-Diaz, Status report: software reusability, IEEE Software, vol.10, May 1993, pp.61-66.

[4] M. Griss, Software reuse: From library to factory., IBM Systems Journal, vol. 32, 1993, pp. 548--566.

[5] O. Hummel and C. Atkinson, Using the Web as a Reuse Repository, Reuse of Off-the-Shelf Components, Lecture Notes in Computer Science, vol. 4039, Springer, 2006, pp.298-311.

[6] A. Ampatzoglou, K. Apostolos, G. Kakaronzzos, and I. Stamelos, An Empirical Evaluation on the Reusability of Design Patters and Software Packages, Journal of Systems and Software, vol. 86, Dec. 2011, pp. 2265-2283.

[7] O. Hummel, W. Janjic, and C. Atkinson, Code Conjurer: Pulling Reusable Software out of Thin Air, IEEE Software, vol.25, Sep./Oct. 2008, pp. 45-52.

[8] G. Kakarontzas and I. Stamelos, Component Recycling for Agile Methods, Sev-enth International Conference on the Quality of Information and Communications Technology (QUATIC), 29 Sept. 2010 – 2 Oct. 2010, pp.397-402.

[9] W.B. Frakes and K. Kang, Software reuse research: status and future, IEEE Transactions on Software Engineering, vol.31, July 2005, pp.529-536.

[10] T.C. Jones, Reusability in Programming: A Survey of the State of the Art, IEEE Transactions on Software Engineering, vol.SE-10, Sept. 1984, pp.488-494.

[11] W.B. Frakes and C. Terry, Software reuse: metrics and models, ACM Computing Surveys, vol. 28, June 1996, pp.415-435.

[12] A. Sharma, R. Kumar, and P.S. Grover, A Critical Survey of Reusability Aspects for Component-Based Systems, World Academy of Science, Engineering and Technology, vol. 33, 2007, pp.35-39.

[13] A. Khoshkbarforoushha, P. Jamshidi, and F. Shams, A metric for composite service reusability analysis, Proceedings of the 2010 ICSE Workshop on Emerging Trends in Software Metrics (WETSoM '10), ACM, New York, USA, 2010, pp.67-74.

[14] T. Elr, Service-Oriented Architecture: Concepts, Technology, and Design, Prentice-Hall, 2005.

[15] H. Washizaki, H. Yamamoto, and Y. Fukazawa, A Metrics Suite for Measuring Reusability of Software Components, Proceedings of the 9th International Symposium on Software Metrics (METRICS '03), IEEE Computer Society, Washington, DC, USA, 2003, pp.211-223.

[16] P. Clements and L. Northrop, Software Product Lines: Practices and Patterns, Addison-Wesley, 2002.

[17] W.B. Frakes and S. Isoda, Success Factors of Systematic Reuse, IEEE Software, vol. 11, Sep./Oct. 1994, pp.14-19.

[18] Y. Kim and E.A. Stohr., Software reuse: survey and research directions, Journal of Management Information Systems - vol.14, March 1998, pp.113-147.

[19] D. Merkl, Self-Organizing Maps And Software Reuse (book chapter), Compu-tational intelligence in software engineering, Pedrycz, W., Peters, J.F. (eds.), Singapore, World Scientific, 1998, pp.65-95.

[20] O.P. Rotaru and M. Dobre, Reusability Metrics for Software Componenents, ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'05), Washington DC, USA, 2005, pp.24-31.

[21] J. Poulin, Measuring software reusability, Proceedings of the International Conference on Software Reuse: Advances in Software Reusability, 1-4 Nov. 1994, pp.126-138.

[22] G. Caldiera and V.R. Basili, Identifying and qualifying reusable software components, IEEE Computer, vol.24, Feb. 1991.

[23] M. Halstead, Elements of Software Science, Amsterdam: Elsvier North-Holland, Inc., 1977.

[24] T.J. McCabe, A Complexity Measure, IEEE Transactions on Software Engineering, vol. 2, Sept. 1976, pp. 308-320.

[25] J. Barnard, A new reusability metric for object-oriented software, Software Quality Journal, vol. 7, Jan. 1998, pp.35-50.

[26] Y. Mao, H. Sahraoui, and H. Lounis, Reusability Hypothesis Verification using Machine Learning Techniques: A Case Study, Proceedings of the International Conference on Automated software engineering, IEEE, 1998, pp.84-93.

[27] Y. Lee and K.H. Chang, Reusability and maintainability metrics for object-oriented software, Proceedings of the 38th annual on Southeast regional con-ference (ACM-SE 38), ACM, New York, NY, USA, 2000, pp.88-94.

[28] E.S. Cho, M.S. Kim, and S.D. Kim, Component Metrics to Measure Component Quality, Proceedings of the Eighth Asia-Pacific on Software Engineering Con-ference (APSEC '01), IEEE Computer Society, Washington, DC, USA, 2001, pp.419-426.

[29] L.H. Etzkorn, W.E. Hughes Jr., and C.G. Davis, Automated reusability quality analysis of OO legacy software, Information and Software Techn., vol.43, 2001, pp. 295-308.

[30] S. Bhattacharya and D.E. Perry, Contextual reusability metrics for event-based architectures, Intern. Symp. on Empirical Software Engineering, 17-18 Nov. 2005, pp.459-468.

[31] G. Gui and P.D. Scott, New Coupling and Cohesion Metrics for Evaluation of Software Component Reusability, Proc. of the Intern. Conf. for Young Computer Scientists, 2008, pp.1181-1186.

[32] N. Gill and S. Sikka, Inheritance Hierarchy Based Reuse & Reusability Metrics in OOSD, International Journal on Computer Science and Engineering (IJCSE), vol.3, June 2011, pp.2300-2309.

[33] M.A.S. Boxall and S. Araban, Interface Metrics for Reusability Analysis of Components, Australian Software Engineering Conference (ACWEC'04), Melbourne, Australia, 2004, pp.40-50.

[34] A. Sharma, P.S. Grover, and R. Kumar, Reusability assessment for software components, SIGSOFT Software Engineering Notes, vol.34, No.2, February 2009, pp.1-6.

[35] V.R. Basili, G. Caldiera, and H.D. Rombach, The Goal Question Metric Ap-proach, Encyclopedia of Software Engineering, vol.1, New York, John Wiley & Sons, Inc., Sept. 1994, pp.528-532.

[36] S. Becker, A. Brogi, I. Gorton, S. Overhage, A. Romanovsky, and M. Tivoli, Towards an engineering approach to component adaption. In Architecting Systems with Trustworthy Components, Springer, 2006, pp. 193–215.