Software Quality Journal, 27. 4 (2019): 1505-1530

**DOI:** http://dx.doi.org/10.1007/s11219-019-09455-4

# Analysis and Measurement of Internal Usability Metrics through Code Annotations

Maximilian Schramme
Technische Universität München

Department of Business Informatics
Munich, Germany

José A. Macías **(*)**
Universidad Autónoma de Madrid

Computer Engineering Department
Madrid, Spain

**(*) Corresponding Author (j.macias@uam.es)**

## ABSTRACT

Nowadays, usability can be meant as an important quality characteristic to be considered throughout the software development process. A great variety of usability techniques have been proposed so far, mostly intended to be applied during analysis, design and final testing phases in software projects. However, little or no attention has been paid to the analysis and measurement of usability in the implementation phase. Most of the time, usability testing is traditionally executed in advanced stages. However, the detection of usability flaws during the implementation is of utmost importance to foresee and prevent problems in the utilization of the software and avoid significant cost increases. In this paper, we propose a feasible solution to analyze and measure usability metrics during the implementation phase. Specifically, we have developed a framework featuring code annotations that provides a systematic evaluation of the usability throughout the source code. These annotations are interpreted by an annotation processor to obtain valuable information and automatically calculate usability metrics at compile time. In addition, an evaluation with 32 participants has been carried out to demonstrate the effectiveness and efficiency of our approach in comparison to the manual process of analyzing and measuring internal usability metrics. Perceived satisfaction was also evaluated, demonstrating that our approach can be considered as a valuable tool for dealing with usability metrics during the implementation phase.

## KEYWORDS

Internal Software-Product Quality, Usability Metric, Code Annotation, User-Centered Development, Human-Computer Interaction.

## ACKNOWLEDGEMENTS

This work was partially supported by the Madrid Research Council (P2018/TCS-4314).

## 1. Introduction

Usability evaluation comprises techniques and methods that have been used for more than two decades (Card et al. 1999). Most commonly used techniques are principally aimed at formative evaluations in early development stages and late summative assessments before the installation phase. Actually, there is a huge number of existing techniques that can be applied in early prototyping and late evaluation of mature products (Nielsen and Molich 1990; Wharton 1992). There is, however, a major lack of usability techniques that can be applied during the implementation phase, where important changes to functional and non-functional aspects of the software product and its usability take place, influencing the usability *in-use* of the software overall. Companies justify the main reasons for this problem in the necessity of counting on additional resources and the developer's mindset (Otkjær et al. 2008). However, collecting quality indicators (such as those that can be found in the code) is highly demanded by companies (Tomas et al. 2013).

Code-based usability evaluation provides strong benefits such as the revision of usability problems during the implementation phase, the reduction of usability testing effort and the improvement of maintainability in the long term, enabling to control the degradation in quality that happens due to software aging (Oliveira et al. 2014). Some of the most common methods used to detect usability problems in the coding phase are inspections (IEEE 2008), including informal, walkthroughs and technical reviews that are intended to manually analyze the code in order to detect defects. Inspections, which are principally carried out by experts, are commonly used to measure specific metrics and detect certain problems related to different software quality issues.

In fact, the inspection of usability problems during the implementation phase comprises a challenging task. This is principally due to the lack of explicit high-level information needed to discern usability objectives and related requisites in the source code. Another reason is the lack of specific tools, which results in limited support for programmers in terms of usability measurement during the implementation, in contrast to the vast number of resources available for measuring usability in other phases of the development process (Veral and Macías 2019; Tullis and Albert 2013).

According to the aforementioned drawbacks, we have proposed the following general research questions to conduct out research:

- RQ1. Are there previous approaches and works focusing on usability measurement during the implementation phase?
- RQ2. Is it possible to create a framework to automatically calculate usability metrics, as defined in ISO/IEC 25023, from the source code?
- RQ3. Can the framework crated be considered as effective, efficient and satisfactory for the final user?

To give answers to the above research questions, we have carried out an extensive research of previous work, concluding that no specific approaches exist that bring a clear solution to the problems mentioned. In this sense, our contribution is aimed at providing a framework for usability measurement during the implementation phase of a software project. This way, our framework provides code-annotation facilities to codify and automatically process internal usability metrics embedded in the code, obtaining measurements at compile time. To carry out this task, our approach features internal metrics – i.e., those concerning the software product's implementation aspects. More concretely, our approach is inspired by the internal usability metrics included in ISO/IEC 25023 (ISO 2016), a current and revised version of ISO/IEC TR 9126-2 (ISO 2002) and ISO/IEC TR 9126-3 (ISO 2003). These metrics are intended to measure internal values that can be used as early indicators to predict usability before the subsequent phases of testing and installation, increasing also the maintainability of the software overall. We argue that internal measures are important since they represent valuable clues for forestalling further problems that influence the values of corresponding

*external* and *in-use* measures (ISO 2001), decreasing the number of usability tests that are usually achieved once the product has been fully developed.

In a nutshell, the main contributions of our work are the following:

- A set of annotations that can be inserted in the source code to create meta-information intended to address internal metrics related to non-executable product quality during the implementation phase.
- A framework where technical users, such as programmer analysts, programmers, usability inspectors and so on, can use the aforementioned annotations to analyse and inspect quality issues related to usability. Although our approach is independent of the human resources planning for a given project, it enables the participation of different roles according to their expertise in usability (e.g., programmers with knowledge in usability, usability engineers and inspectors, etc.) in order to assume annotation and inspection tasks depending on the implementation process chosen.
- An annotation processor that automatically processes the meta-information in form of annotations created by users. The processor calculates and reports internal metrics related to usability. This is automatically accomplished, thus metrics are calculated in real time when the user modifies the code or creates new meta-information.

This paper is structured as follows. Section 2 presents the research carried out to find previous works. Section 3 presents our approach in detail. Section 4 presents the evaluation of our approach and the discussion of results. Finally, Section 5 reports on conclusions and future work.

## 2. Related Work

In order to look for previous works and give an answer to RQ1, we conducted a literature review. We utilized the following bibliographical resources: ACM Digital Library, Springer, IEEE Xplore Digital Library and Google Scholar. The search pattern utilized was:

(“usability” or “usable” or “user-centered” or “user-centred”) and

(“metric” or “measure” or “quality” or “ISO”) and

(“code” or “coding” or “implementation”)

This way, we initially identified a total of 70 papers. Then we carried out a screening, selecting papers related to code-based metric measurement, preferably related to usability. After screening, we finally obtained 15 partially related papers.

In general, the problem that we identified during the screening of the selected papers is that there is a high amount of works focusing on automatically computing different source code measures such as complexity, size and so on (Jabangwe et al. 2015; Tomas et al. 2013; Jabangwe and Šmite 2012; Singh et al. 2010; Kanellopoulos et al. 2010; Briand and Wüst 2002), but there are few works related to usability measures. Other works found are mainly based on developed metrics or usability metrics, but according to an existing conceptual model as part of a model-driven approach (Ammar et al. 2013; Fernandez et al. 2012; Panach et al. 2011; Feuerstack et al. 2008). Also, other existing papers are mainly based on computing external usability metrics (Lettner and Holzmann 2011; Bailey et al. 2009; Seffah 2006) or even on evaluating the product *in-use* (Clemente and Macías 2010). Besides, we identified previous works focused on quality characteristics in several application domains, but not specifically related to internal characteristics. This is the case for the work proposed by Carvalho et al. (2017), which is mainly focused on discovering quality characteristics and measures that should be taken into account in the evaluation of ubiquitous systems. Other works are focused on the Web to identify data-oriented quality characteristics (Han 2017; Orehovački et al. 2013), proposing specific web metrics (Fraternali et al. 2002) or utilizing specific tools to evaluate *in-use* metrics (Brajnik 2000).

However, there is a lack of papers investigating on usability metrics related to the source code of a software application. A paper by Memon (2009) reports on techniques of reverse engineering for the automated usability evaluation of a GUI (Graphical User Interface). This allows to automate usability evaluations but only when the application includes a GUI. Another shortcoming found was the utilization of the word “metric" in the area of software development, where it is very common to find this term related to the size and complexity of the software rather than being related to usability. This way, we found papers based on metrics that are not specifically related to end-user usability, but to the understandability of the program by the developer.

Another related paper, written by Dubey and Rana (2011), is focused on usability estimation in software systems, using diverse object-oriented metrics to relate usability with software architecture. However, it mentions but does not further investigate the metrics suggested by ISO/IEC 9126 (ISO 2001) or ISO/IEC 25023. Anyway, the approach is quite technical and not specifically oriented to usability.

All in all, the literature review verified the lack of existing proposals related to the analysis and measurement of usability based on source code. This provided an answer to research question RQ1 and a challenge to carry through our research.

## 3. Our Approach

In order to give an answer to RQ2, we propose a framework for the analysis and measurement of usability through the source code. To carry out this task, we have based on the ISO/IEC 25023 internal usability metrics to create a set of custom annotations combined with an annotation processor that can be used in Java (Oracle 2018) implementations. Based on these annotations, which can be inserted in the source code, the annotation processor computes the metrics and provides automatic advising in real time. Our framework can be applied under a role-based approach, where different technical users, such as programmer analysts, programmers, usability inspectors and so on, can use the annotations to codify and inspect quality issues related to usability, splitting up responsibilities for the different annotation tasks if desired.

### 3.1. Proposed Method to Measure Internal Usability

As stated in ISO/IEC 25023, the measurement of internal metrics requires from experts to carefully review and calculate each of the values in order to evaluate the internal usability of the product by analyzing the code manually. This task is principally achieved by technical members of the project team or quality staff, according to the standard specification. However, as modern project teams today are composed of persons with different competencies and background, usability analysis and evaluation can be assumed by different roles.

We propose a method that conceptually operates as follows:

1) First, the roles in charge of annotations should be defined. In general, our approach is independent of any human resources planning, which is out of the scope of our research. In general, only the role of a usability inspector (or similar) is necessary to fully annotate the code previously generated by programmers and analyze the resulting metrics. However, different roles can participate in code annotations and further analysis, as our approach can be used under incremental and iterative programming and inspection processes as desired. This facilitates the management and interpretation of the values for each metric in order to support decision-making, which may imply to compare the results with the usability objectives of the software, thus addressing usability issues during the implementation phase by splitting up responsibilities. For instance, programmers and usability engineers can work together (i.e, programmers can write the code and usability engineers can deal with annotations, parameters and results) to get a first version of the annotated code and analyze the resulting metrics to improve and inspect the code in next refinement steps.

2) Second, the source code has to be annotated. This enables the calculation of the internal usability metrics as specified in ISO/IEC 25023. This is achieved by means of the annotation facilities that certain programming languages and environments provide. We have based on Java annotation facilities in order to implement our annotation processor. However, the proposed annotations and parameters can be used in other programing languages as long as they directly or indirectly provide annotation facilities (i.e., using comments to include the annotations and creating an application to process such comments).
3) Third, annotations are automatically processed to obtain the resulting metrics with no need to run any test. We have developed an annotation processor that is in charge of automatically processing the annotations in the code and calculate the metrics at compile time (i.e., with no need to execute the code), providing numerical results for the internal usability metrics, in terms of percentage values, according to the calculations and optimal values specified in ISO/IEC 25023. This facility is automatically triggered when any change in the code occurs, which supports iterative revision of the code that can be further improved according to the results observed through the metrics.

### 3.2. ISO/IEC 25023

In order to carry out our approach, we have based on the specification of internal metrics included in ISO/IEC 25023. This standard is part of ISO/IEC 25000 (ISO 2005), which is known as SQuaRE – System and Software Quality Requirements and Evaluation. ISO/IEC 25000 comprises an attempt to gather and improve previous quality models. The standard was specifically designed to replace ISO/IEC 9126 and ISO/IEC 14598 (ISO 2006). More specifically, ISO/IEC 25023 is concerned with the measurement of system and software product quality, providing a set of quality measures for each characteristic and sub-characteristic defined in ISO/IEC 25010 (ISO 2011), as well as specific explanations of how to apply such measures. Coming from ISO/IEC TR 9126-2 and ISO/IEC TR 9126-3, the new ISO/IEC 25023 provides three different kinds of metrics. On the one hand, internal metrics are directly related to the source code but not to the execution of it, therefore they are affected by the quality of the development process. On the other hand, external metrics require the software to be executed, and they are applicable to the running application. They depend on the internal software quality and affect the quality *in-use* of the software product (Bevan 2009). Finally, in-use metrics are applied when the software is fully developed and can be tested under real conditions. In-use metrics always depend on the context of use.

However, and contrary to ISO/IEC 9126, ISO/IEC 25023 features some changes related to the calculation of the metrics, presenting also a slightly different structure. In general, this new standard combines both external and internal metrics. Nevertheless, we are principally interested in internal metrics involving usability. This way, we have based on the 6 quality sub-characteristics related to the usability characteristic that can be found in ISO/IEC 25023. These 6 sub-characteristics comprise a total of 12 metrics including 22 parameters required for the calculations. In summary, all this information is shown in Appendix A1.

All in all, the information provided by ISO/IEC 25023 is too general, and the metrics included are barely described to be of any benefit to usability engineers (Bevan et al. 2016). This way, the metrics provided by the standard, as well as the suggested parameters for the calculations, have to be interpreted and further detailed for specific utilization through the source code. This would provide strong benefits such as the revision of usability problems in the implementation phase, the reduction of usability testing effort and the increase of the source code maintainability in the long term. Our approach provides these benefits for usability engineers, proposing a semi-automatic mechanism to obtain metric values using meta-information through code annotations. With that in mind, the proposed mechanism will be developed in the following sections.

### 3.3. Code Annotations

In order to annotate to code with meta-information intended to automatically calculate internal usability metrics as specified in ISO/IEC 25023, we have utilized Java annotations (Oracle 2018). Java annotations were first introduced as a part of Java version 5, and now they are used in many popular technologies like JavaBeans, JDBC and Javadoc. Java annotations are widely used in Java programming as they allow enriching the code with meta-information that can be later evaluated by a developed compiler plugin called annotation processor. Annotation facilities can be found in other programing languages and environments. However, Java programing language and Eclipse development environment (IBM 2018) are widely used for software development, and we have based on both technologies to develop our approach. Java provides native annotations as part of the programming language. The default annotation processor is triggered during the compilation of the program code, and it is able to analyze the annotations, display warnings and generate additional source code.

In order to include meta-information as code annotations to automatically calculate internal usability metrics, we have developed the following method:

1) First, using the original definitions appearing in ISO/IEC 25023, we have classified each metric according to the parameters needed for its calculation. This way, Table 1 depicts an enumeration of the metrics used in our approach, together with a comprehensive description and the parameters needed for the corresponding code annotation.

| # Metric | Metric Name | Comprehensive Description | Parameter for Code Annotation |
|---|---|---|---|
| 1 | Integrity of Description | This metric helps ensure that potential users will understand the capability of the product after reading the product description –i.e., users can understand whether the software product is suitable for their intended use and that it can be used for particular tasks. This way, code functions have to be adequately described along the whole project in order to be consistent with user requirements. | A function can be annotated using a parameter that can take the value true or false depending on whether the function is intended to be described as understandable in the product description or not according to expert criteria. |
| 2 | Demonstration Capability | This metric is used to identify the demonstration capability of a code function –i.e., demonstration steps showing how the product is used, also including facilities for wizards. This facilitates the interaction with functional elements. | A function can be annotated using a parameter that can take the value true or false depending on whether the function has demonstration capability or not according to expert criteria. |
| 3 | Evident Function | This metric is useful to identify whether users are able to identify specific functions by interacting with the interface elements –i.e., navigating through the interface options or menus. This facilitates the validation | A function can be annotated using a parameter that can take the value true or false depending on whether the function is evident to user or not according to expert criteria. |

| | | | |
|---|---|---|---|
| | | of user requirements and tasks. | |
| 4 | Completeness of User Documentation and/or Help Facility | This metric can be used to identify the completeness of documentation, help facility or both. This facilitates the identification of functionality that is completely documented or not, which eases the user interaction with the software product. | A function can be annotated using a parameter that can take the value true or false depending on whether the function is completely described in the user documentation and/or help facility or not according to expert criteria. |
| 5 | Operational Error Recoverability | This metric can be used to identify functionality that tolerates user errors, which is essential to assure usability, ensuring that software controls user errors and responds accordingly. | A function can be annotated using a parameter that can take the value true o false depending on whether the function is implemented with user error tolerance or not according to expert criteria. |
| 6 | Operational Consistency | This metric is useful to ensure consistency –i.e., same operations behave the same way in different parts of the software, which guarantees a consistent interaction when the same operation appears in the software product. | A function or operation can be annotated using a parameter that can take the value true or false depending on whether it behaves the same way to similar operations in other parts of the system or not according to expert criteria. |
| 7 | Message Clarity | This metric helps identify the clarity of user messages. For instance, clear error messages enable the user to know how to recover from errors. As messages are important interaction elements, those should be clear and well composed enough to fulfill their purpose. | A message can be annotated using a parameter that can take the value true or false depending on whether the message provides clear explanations or not according to expert criteria. |
| 8 | Customizing Possibility | This metric is used to identify customizable elements in the code. This helps assess the flexibility of the software functionality to be configured according to user criteria. Customizable functionality facilitates user interaction according to the user's needs. | A function or operation can be annotated using a parameter that can take the value true or false depending on whether it can be customized by user during operation or not according to expert criteria. |
| 9 | Input Validity Checking | This metrics can be used to identify interactive input functionally that allows validity checking –i.e., functionality that checks whether the user introduces the required data or not. This enables controlling the interaction, providing user with the necessary support to input information into the | An input user interface element can be annotated using a parameter that can take the value true or false depending on whether this element checks for valid data or not according to expert criteria. |

| | | | |
|---|---|---|---|
| | | software product. | |
| 10 | Avoidance of Incorrect Operation | This metric enables to identify fault tolerance. More specifically, it is useful to identify functionality to avoid critical and serious failures, as well as data damage, caused by incorrect operations. This is necessary to avoid the interruption of user interaction due to fatal errors. | A function or operation can be annotated using a parameter that can take the value true or false depending on whether it avoids incorrect operational patterns according to expert criteria. |
| 11 | Appearance Customizability of User Interface | This metric is used to identify user interface elements that can be customized in appearance. This facilitates the customization of the user interface depending on the user's criteria (i.e., look and feel). | A user interface element can be annotated using a parameter that can take the value true or false depending on whether this element can be customized in appearance or not according to expert criteria. |
| 12 | Physical Accessibility | This metric helps identify user interface elements that can be customized for universal access. This is an important issue to ensure that all users can interact with the software product. | A user interface element can be annotated using a parameter that can take the value true or false depending on whether this element can be customized for access by users with physical handicaps or not according to expert criteria. |

**Table 1.** Metrics used in our approach, together with a comprehensive description and the parameters needed for the implementation through code annotations

As shown in Table 1 some metrics, such as 1 and 2, need to take into account code functions. However other metrics, such as 9 and 11, are intended to process UI (User Interface) elements. Finally, the metric number 7 calculates on the number of messages. This classification is shown in Figure 1. It is worth noting that some metrics, such as 8 and 10, appear in two different categories (i.e., Function and Operation) as they are related to parameters involving functional and operational information for the calculation.

2) Second, we have created annotation types for the metric classification previously described. The identification for each annotation type begins with the letters `ME`, which means Measurement, followed by the name of the annotation type. This way, we have created the following 4 annotation types (see Figure 1):

- `MEFunction`
- `MEMessage`
- `MEOperation`
- `MEUIElement`

3) Third, we have created annotation parameters as inputs for each annotation type according to the parameters needed to calculate each related metric (as described in Table 1).

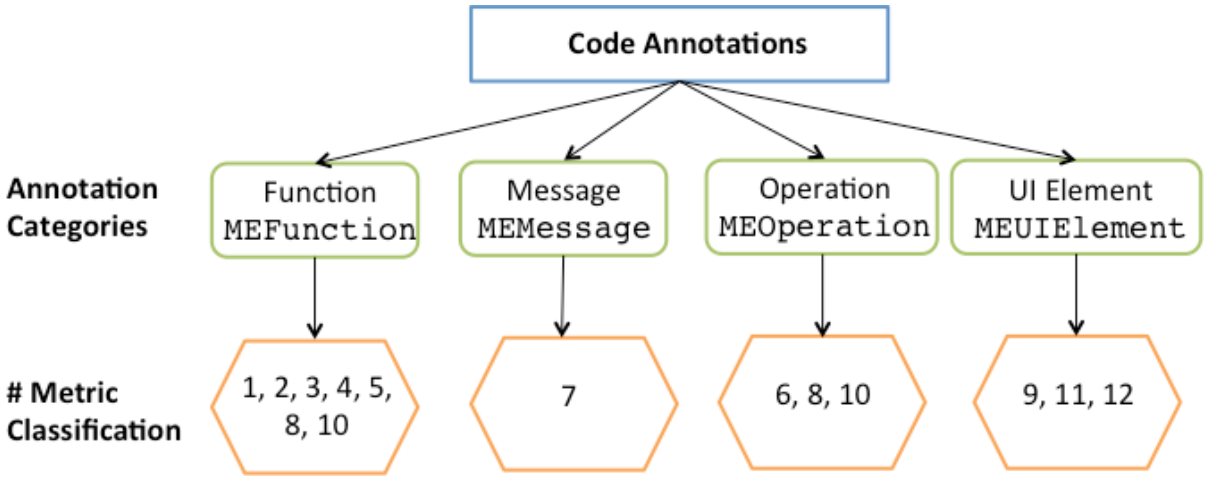


**Fig. 1** Types of annotations and classification of metrics

Since the terms function, message, operation and UI element are not clearly defined in the ISO standard (Abran et al. 2003), we defined those terms in the context of our work:

- Function: represents a functional piece of code that has several inputs and generally one output. In the case of Java, a function can be meant as a method.
- Message: represents an interaction between the user and the system that is initiated by the system.
- Operation: represents an interaction between the user and the system that is initiated by the user.
- UI Element: represents a single functional entity in the user interface of the application that the user can interact with.

As for annotation parameters, they are defined according to the nature of the measure and the parameters needed to calculate each metric. This way, and according to the information appearing in Table 1, we have considered parameters that can take the value `true` (when the parameter is satisfied) or `false` (when the parameter is not satisfied). Table 2 presents the parameters associated to each type of code annotation and metric (according to the information presented in Figure 1).

| Annotation Parameter to be Used in the Code | Related Annotation Type | Related # Metric |
|---|---|---|
| `isDescribedAsUnderstandableInProductDescription` | `MEFunction` | 1 |
| `hasDemonstrationCapability` | `MEFunction` | 2 |
| `isEvidentToUser` | `MEFunction` | 3 |
| `isCompletelyDescribed` | `MEFunction` | 4 |
| `isImplementedWithUserErrorTolerance` | `MEFunction` | 5 |
| `hasInconsistentBehavior` | `MEOperation` | 6 |
| `hasClearExplanations` | `MEMessage` | 7 |
| `canBeCustomizedByUser` | `MEFunction, MEOperation` | 8 |
| `hasCheckForValidData` | `MEUIElement` | 9 |
| `avoidsIncorrectOperationalPatterns` | `MEFunction,` | 10 |

| | | |
|---|---|---|
| | MEOperation | |
| canBeCustomizedInAppearance | MEUIElement | 11 |
| canBeCustomizedForUsersWithHandicaps | MEUIElement | 12 |

**Table 2.** Annotation parameters to be used in the code, together with the related annotation types and metrics.

The number of functions, messages, operations and user interface elements needed to measure the metrics is automatically calculated by the annotation processor when processing `MEFunction`, `MEMessage`, `MEOperation` and `MEUIElement` annotations, respectively. In the same way, some of the parameters needed to measure the metrics, as they originally appear in ISO/IEC 25023 (see Appendix A1), are automatically calculated by the annotation processor. This is the reason why we have considered in Table 1 only one parameter per metric, facilitating the codification to users. For instance, in the measurement of metric 9 (input validity checking), a function or method has to be annotated as an input UI element checking or not for valid data (parameter A). However, our approach automatically computes the total number of input UI elements that could check for valid data (parameter B), which provides with the other value needed to finally calculate the metric as X = A/B, having the user to take into account only a single parameter (`hasCheckForValidData`), as specified in Table 2.

The following Java code represents an example that includes some of the annotations and parameters, appearing in Table 2, to calculate metrics. Let us suppose that we have two methods, called `showMenuOptions` and `createCustomTextField`. The former is used to show the (specific level) options of a visual menu, whereas the latter is used to create a customized input text field in the UI:

```
@MEFunction (isEvidentToUser=true)
public void showMenuOptions(int i);
@MEUIElement (hasCheckForValidData=true,
              canBeCustomizedInAppearance=true)
public void createCustomTextField(String name);
```

In the above fragment of Java code, the two methods, `showMenuOptions` and `createCustomTextField`, are annotated with `MEFunction` and `MEUIElement` annotations, respectively. In the case of `showMenuOptions`, the annotation is parameterized with the parameter `isEvidentToUser`, which is related to the measurement of metric 3, whereas the method `createCustomTextField` is parameterized with parameters `hasCheckForValidData` and `canBeCustomizedInAppearance`, which are related to the measurement of metrics 9 and 11, respectively. In all cases, the parameters are set to `true`, which means that the parameters are satisfied, denoting, for the case of `showMenuOptions`, that the user is able to identify specific functions by navigating through the menu. Similarly for the case of `createCustomTextField`, the parameters denote that the method creates input text fields that enable validity checking and can be configured according to user criteria. Therefore, depending on the total number of functions and UI elements, a measure for metrics 3, 9 and 11 will be automatically calculated by the annotation processor.

Together with the customized annotation processor, we provide with auto-complete facilities in order for the different users that interact with our framework to annotate the code easily, thus helping in reducing the cognitive burden. Users only have to bear in mind the four ME annotation types, as the system triggers automatically a list intended to help complete the corresponding parameters according to each annotation type.

### 3.4. Role-Based Annotations

The main aim of our work is to propose a mechanism for measuring internal usability metrics with minimal effort. According to standard ISO/IEC 25023, the process of measuring internal usability metrics is traditionally carried out by reviewers and inspectors that inspect the source code in order to find the clues needed to calculate the metrics manually.

In our approach, the only role required to deal with the metrics could be a technical usability inspector with expertise in source-code review. This way, and once the software code is in an advanced stage, it could be delivered to the usability inspector in order for her/him to annotate the code with the proposed annotations and analyze the resulting values of the metrics. However, this task can be also supported by the help of a programmer who can create initial annotations in the code depending on the characteristics of each function.

Although tasks and roles assignment is not the focus of our work, our approach can be utilized under a role-based paradigm. In fact, modern project management today comprises team members who are in charge of different tasks involving usability measurement. Based on this concept, we include an example of role-based annotation that allows assigning the creation and analysis of the different types of annotations according to specific roles during the implementation phase.

This mechanism can be useful for sequential workflows or even for inspection meetings, where annotations can be arranged cooperatively and/or in a distributed way using SCM (Software Configuration Management) tools that allow programing and annotating collaboratively. This allows team members to be assigned different tasks and play more than one role according to the project plan.

As an example, we propose the following division of annotation responsibilities. It is worth noting that this is only an instance (but not the only one) that may vary depending on the project size and the roles required. In fact, the project team might be interested only in a subset of metrics.

More specifically, we propose the following roles in our example:

- Programmer Analyst: This is an intermediary role between functional analyst and programmer. S/he is concerned with technical requirements tasks but also with programming duties. This role is especially appropriate to deal with annotations related to requirements inspection at code level, as most usability metrics are influenced by functional and non-functional requirements.
- Usability Inspector: This role is in charge of inspecting the resulting code and verifying that it meets the usability requirements specified and then programmed.

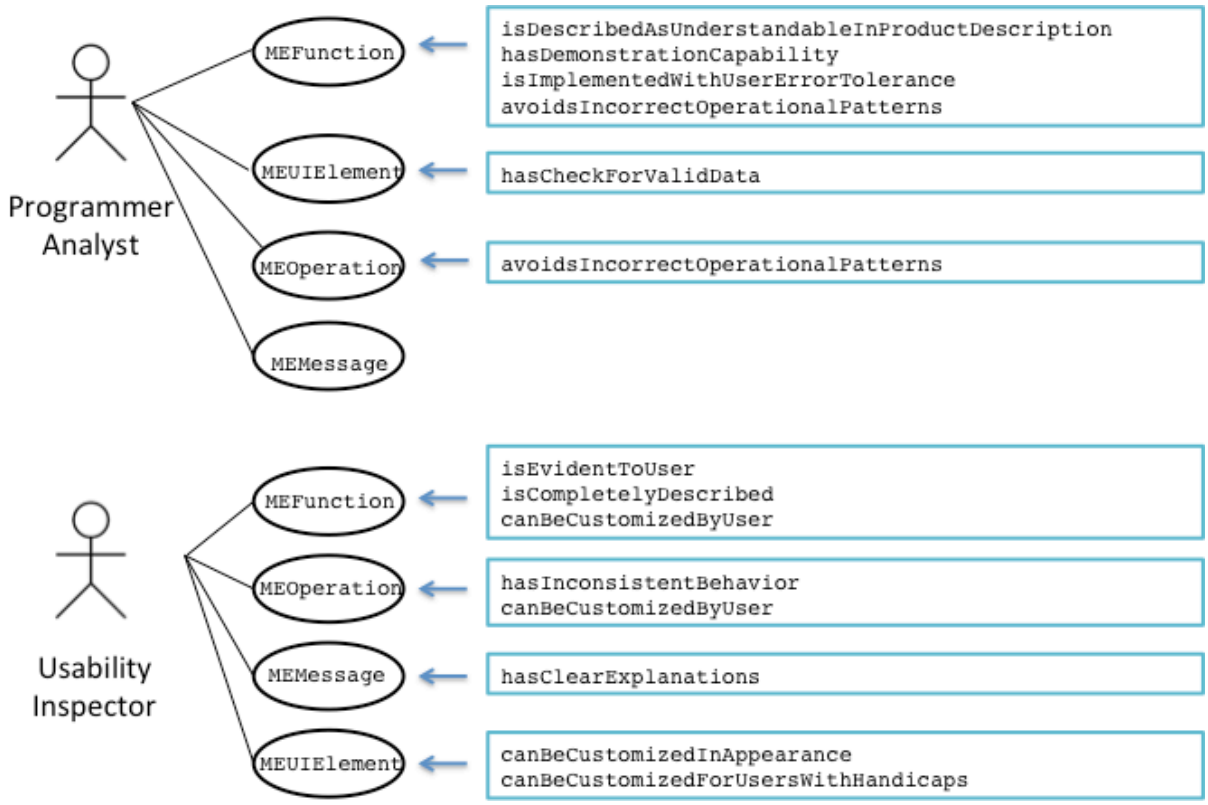


**Fig. 2** Example of different roles dealing with the annotation types and corresponding parameters

Figure 2 presents an example that splits up annotation tasks, including the different annotations (see Figure 1 and Table 2) and the associated roles dealing with each annotation type and parameter. Considering Figure 2, an example of annotation workflow could be the following:

1) First of all, the programmer analyst is in charge of programming the code and creating all the principal annotation types (`MFunction`, `MEOperation`, `MEMessage` and `MEUIElement`) for every function in the source code. In addition, this role would annotate some specific parameters related to functional aspects of the usability metrics, such as error tolerance, validity checking and other specific parameters involving functional and operational usability issues, such as incorrect operational patterns and demonstration capability.
2) Then, the usability inspector is in charge of creating specific parameters for the principal annotations created by the programmer analyst. In this case, the usability inspector would deal with annotation parameters involving ergonomic issues embedded in the code, such as whether a function is evident to user, can be customized for users with handicaps and so on.

In this example, the programmer analyst creates the code together with the basic structure of annotations and parameters related to functional issues, whereas the usability expert creates the main parameters related to ergonomic aspects (combined with non-functional usability concerns) that have a straight impact on the usability (and accessibility) of the system. In general, all functions can be annotated, independently of their level, as long as they have impact on the usability of the software. This will mostly depend on the requirements and the project team strategy

Our contribution also enables an iterative and incremental approach, where annotations can be drafted and modified depending on the evolutionary versioning of the code, as the calculation of the metrics is automatic and the visual effect is immediate in order to revise the code or even improve usability requirements with the help of other team roles. All in all, and depending on the background, the usability expert should finally review all the code to set up and analyze the metrics that are more related to human factors.

### 3.5. Annotations Processor

In order to process the aforementioned annotations and parameters, we have developed a customized annotation processor that can be installed in the Eclipse development environment.

At compile time, the annotation processor recognizes all the annotation types, the parameters and their location in the code. By counting the occurrences of the annotation elements, the annotation processor is able to compute the metrics according to ISO/IEC 25023, displaying warning messages for elements that do not meet the syntax. In Eclipse environment, the compilation is achieved every time a change occurs in the code. This means that the annotation processor is automatically triggered every time a change is made, thus providing updated information about the metrics, which is transparent to the user. This offers a significant advantage compared with directly using other approaches such as Java Reflection API, which is the alternative way of obtaining information about annotations in Java.

The architecture of the annotation processor and also its implementation is predefined by the way Java evaluates annotations. The annotation definition contains the structure of the annotation types. The approach can be seen as a compiler plugin related to the Java API for annotation processing. Before including the annotations in the source code of the application, the developer needs to add the customized annotation processor as a compiler plugin. Many development environments such as Eclipse or Netbeans (Sun 2018) support this native implementation.

In order to calculate the metrics, we rely on the structure of projects, packages and compilation units defined by the JDT Core Component API. This allows to analyze the structure of the application based on a tree, and thus explore specific annotations as the ones defined previously.

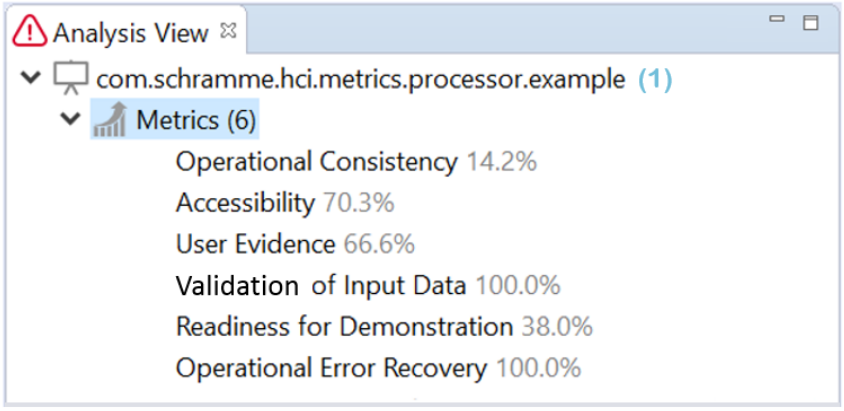


**Fig. 3** Metric analyzer plugin for Eclipse displaying 6 different metrics for an specific Java project

Figure 3 depicts the Analysis View window, appearing in the upper-right corner of an Eclipse project and showing an example of the resulting metrics for a given project. The metrics node allows an overview of all the metrics that can be evaluated based on the annotations inserted in the code. In this case, 6 measures are visualized, corresponding to metrics 2, 3, 5, 6, 9 and 12 in Table 1. Results are calculated as percentage values in order to easily show the compliance percentage for a given metric. For instance, the Accessibility result (metric 12 in Table 1) is indicating that 70.3% of the UI elements included in the source code can be customized for access by users with physical handicaps. The annotation processor is designed in a way that allows developers to work with an arbitrary set of metrics depending on the project's characteristics.

We propose this annotation processor to be used in the implementation phase of a project. The immediacy of results makes this approach helpful to be considered in collaborative environments (i.e., using SCM tools) or even stand-alone setups, where modifications can be instantaneously applied and visualized thanks to the automatic triggering related to changes in the source code.

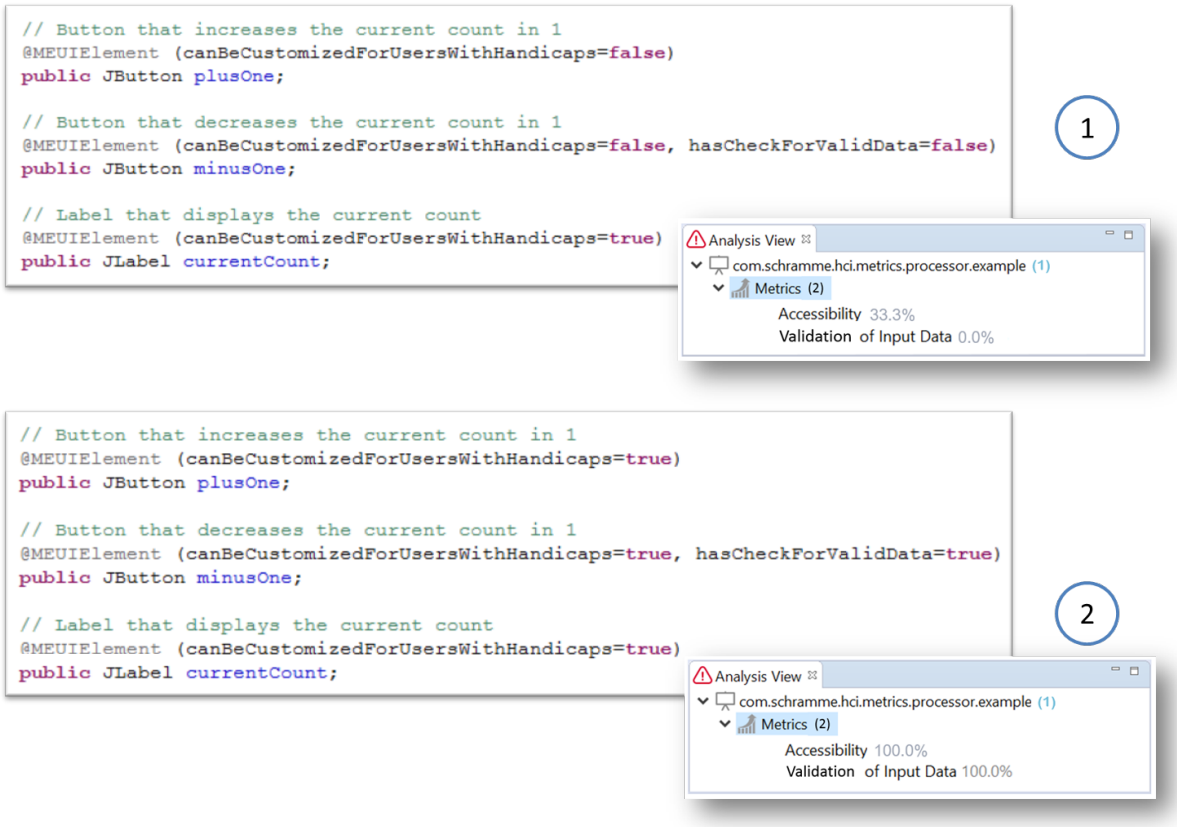


**Fig. 4** An example of the same code fragment together with the corresponding metric measures (1) and the results perceived after modifications (2)

Figure 4 presents a code fragment comprising 3 methods that has been annotated with `MEUIElement`, also including parameters to study the accessibility (metric 12 in Table 1) and the validation of input data (metric 9 in Table 1):

- Accessibility (Physical Accessibility) => X = A / B, where A represents the number of UI elements that can be customized for access by users with physical handicaps, and B represents the total number of UI elements. The closer to 1 the better physical accessibility.
- Validation of Input Data (Input Validity Checking) => X = A / B, where A represents the number of UI elements that check for valid data, and B represents the number of UI elements that could check for valid data. The closer to 1 the better.

As shown in the situation marked with 1 (at the top of Figure 4), methods `plusOne` and `minusOne` have the parameter `canBeCustomizedForUsersWithHandicaps` set to `false`, whereas method `currentCount` has the same parameter set to `true`. In addition, method `minusOne` has the parameter `hasCheckforValidData` set to `false`. This situation indicates that the measures for the aforementioned metrics are (see Figure 4):

- Accessibility: X = 1 / 3 = 33.3%
- Validation of Input Data: X = 0 / 1 = 0.0%

In the situation 2 (at the bottom of Figure 4), after an improvement in the source code, the person in charge has changed the metric parameters to `true` in all cases, which implies the automatic update of metric calculations, resulting now in (see Figure 4):

- Accessibility: X = 3 / 3 = 100.0%
- Validation of Input Data: X = 1 / 1 = 100.0%

This means that all annotated UI elements and methods satisfy accessibility and input verification metrics at 100% level (the better possible), respectively.

All in all, the reported mechanism helps answer RQ2, showing that internal usability metrics, as defined in ISO/IEC 25023, can be automatically calculated from the source code using the framework developed. This enables technical users to easily inspect the usability, also facilitating decision-making in order to improve the code accordingly.

## 4. Evaluation

In order to corroborate RQ3, we have carried out a controlled evaluation with real users. The main objective of this evaluation is twofold. On the one hand, we want to demonstrate that, when evaluating code-based internal usability metrics, our framework is more effective and efficient than using the traditional and manual inspection process. On the other hand, we want to demonstrate that our approach provides suitable values of usefulness and satisfaction.

### 4.1. Procedure and Method

Our evaluation has been carried out following a two-step procedure. In the first part, we asked recruited participants to calculate internal usability metrics with and without the proposed annotations framework. In the second part, we asked the participants who used our proposed framework to fill in a questionnaire, as it will be detailed down below.

Recruited participants were divided into two different groups: a control group carrying out the measurement of internal usability metrics manually, and an experimental group carrying out the same task but using our framework. In the experimental group, participants were given a previous short talk of 20 minutes about the objective of the task, the framework including the developed annotations and the relationship with the corresponding metrics. By contrast, participants in the control group were given only a previous short talk of 10 minutes about the objective of the task. We provided participants with the same Java code of 100 lines in a single file that comprises a total of 10 functions and user interface elements. Participants in each group where asked to calculate the metrics in the code as specified in ISO/IEC 25023. Measures comprising effectiveness and efficiency were assessed for every user. This way, a between-subject design was accomplished in order to obtain independent measures of the mentioned variables.

After finishing the task, we asked participants in the experimental group to complete a questionnaire in order to measure the perceived satisfaction with our framework.

### 4.2. Variables and Research Questions

We have considered the method –i.e., manually or automatic with our framework, as the independent variable to study, according to the group division in which each participant was enrolled (i.e., control or experimental). As for dependent variables, we have considered the effectiveness –i.e., the extent to which participants are able to carry out the principal task of measuring the metrics, and the efficiency –i.e., average time in minutes spent by participants in measuring the metrics. Also, we considered usefulness, satisfaction, ease of use, and ease of learning as dependent variables. To obtain such values, users were asked to fill in the USE questionnaire (Perlman 2015; Lund 2001), which helps measure user perceived satisfaction in-use by means of the 4 aforementioned variables (Tullis and Albert 2013). USE questionnaire includes 30 questions grouped into 4 different categories: 8 questions for measuring usefulness, 11 questions devoted to measure ease of use, 4 questions used to measure ease of learning, and 7 questions for measuring overall satisfaction. Responses were assessed in a Likert scale ranging from 1 to 7, where 1 implies “strongly disagree” and 7 “strongly agree”. Results were normalized in order to obtain percentage values. Questions were worded for the intended subject of evaluation –i.e., the framework proposed.

We established specific research questions for the evaluation, which are related to RQ3 described in Section 1:

- RQ3.1. Do the users perform the measurement of internal usability metrics more efficiently and effectively using our framework than using the manual process?
- RQ3.2. Do the users consider the proposed framework as satisfactory?

These research questions will be validated through the results obtained from the evaluation. Specifically, to answer RQ3.1 we expect to get higher efficiency and effectiveness values in the experimental group than in the control one. Additionally, to answer RQ3.2 we expect to obtain percentage values over 75% for usefulness, ease of use, ease of learning and overall satisfaction. In fact, 75% represents a positive benchmark level to indicate agreement when responding to the different questions in a 1-7 Likert scale. A normalized average value of 75% represents a number between 5 and 6 (i.e., between agree and very agree), which can be considered as an acceptable measure for these dependent variables.

### 4.3. Apparatus

The evaluation task was accomplished using 2 laptop computers with similar technical characteristics: a MacBook Air with a 1.8GHz dual-core Intel Core i5, 8 GB of onboard memory, 13.3-inch LED-backlit glossy widescreen, and macOS Sierra operating system. Recruited participants accomplished the task using Eclipse Oxygen 4.7 IDE for Java Developers, with Java SE 9, loaded and ready to be used in every evaluation. One of the laptops, utilized by the users in the experimental group, included the annotation processor and facilities, which was already installed in the Eclipse environment. The other laptop, used by the users in the control group, did not include the annotations framework. In both cases, the source code was already loaded in order to save time. All evaluations were carried out in a research laboratory of our institution.

### 4.4. Participants

We recruited 32 participants with ages ranging from 25 to 35 (M=32.5, SD=2.3). They were 10 women and 22 men. All participants had at least knowledge, for more than 5 years on average, on the specific topics required –i.e., Java software inspections, software quality and usability evaluation, being also familiar with the usability metrics included in ISO/IEC 25023. As for the provenance, all participants were recruited from the university environment –i.e., instructors and researchers on software engineering and human-computer interaction, and professionals from software development companies working in the topics required on a regular basis. This way, we initially carried out a random process to assign the half of the participants to the control group (16) and the other half (16) to the experimental one, but ensuring also to counterbalance the groups to be as homogeneous as possible in terms of gender, age, knowledge, years of experience and provenance.

The experimental group, including 16 users, is the more relevant for the evaluation of the annotations framework. In order to justify this sample size, we have based on the binomial probability. This way, we expect to identify problems that impact 17% or more users with a 95% probability of observing these problems in the evaluation. This way, the number of users to test can be calculated as Log (1-0.95) / Log (1-0.17) ≈ 16 users. It is worth noting that the discovery rate can be considered as high (95%), and an impact percentage of 17% enables to find complex problems. In fact, a problem-impact percentage among 30%-60% implies problems affecting a great deal of users (i.e., coarse-grain errors), whereas reducing this figure to a more restrictive percentage (10%-20%) helps find a higher number of problems, and more specifically those being more difficult to find (i.e., less obvious problems). This tradeoff would help find most important problems, so we think that a sample size of 16 is adequate, given the typology of the problems that we expect to observe in the evaluation (Tullis and Albert 2013; Sauro 2012; Hwang and Salvendy 2010; Faulkner 2003; Dix et al. 2004; Nielsen and Landauer 1993).

### 4.5. Parameters Summary

Table 3 depicts the most important settings of the evaluation carried out.

| Parameter | Value |
|---|---|
| Subjects | 32 participants randomly divided into two groups: half of the participants were in the control group, and the other half in the experimental one. |
| Main hypotheses related to research questions | Evaluation is based on RQ3 that is twofold. On the one hand, RQ3.1 is used to corroborate that users perform the measurement of internal usability metrics more efficiently and effectively using our framework than using the manual process. On the other hand, RQ3.2 is used to corroborate that the users consider the proposed framework as satisfactory. |
| Independent variable | Method to measure the metrics: manually or automatic according to the group division. |
| Dependent variables | Effectiveness and efficiency, as well as those obtained from USE questionnaire: usefulness, satisfaction, ease of use and ease of learning. |
| Statistics utilized | T-test, with a previous Shapiro-Wilk test to corroborate normality, is used to compare effectiveness and efficiency in both groups. Mean, min, max, SD, median and CI (95%) were also calculated. The same descriptive statistics were used to represent USE questionnaire results, including also the Cronbach alpha value for each variable. |
| Validation criteria | Higher efficiency and effectiveness values in the experimental group than in the control one. Also, values above 75% for usefulness, ease of use, ease of learning and satisfaction. |

**Table 3.** Summary of evaluation parameters

### 4.6. Results and Discussion

As for the first part of the study, a t-test with a significance level of 95% for independent means was carried out in order to compare effectiveness and efficiency values in both groups. Data normality was ensured using a Shapiro-Wilk test. Results corroborated that group differences for the proposed measures can be considered as significant at $p\text{-}value < 0.05$ for both measures.

| | Control Group | | Experimental Group | |
|---|---|---|---|---|
| Measure | Effectiveness (%) | Efficiency (minutes) | Effectiveness (%) | Efficiency (minutes) |
| Mean | 76.08 | 71.16 | 96.05 | 35.09 |
| Min | 55.11 | 35.23 | 89.20 | 24.12 |
| Max | 92.51 | 102.45 | 100.00 | 55.20 |
| SD | 11.79 | 16.64 | 3.37 | 9.82 |
| Median | 79.47 | 76.88 | 96.99 | 34.26 |
| CI (95%) | 5.77 | 8.15 | 1.65 | 4.81 |

**Table 4.** Descriptive statistics on effectiveness and efficiency for each group

As depicted in Table 4, control group obtained a 76% average value for effectiveness when measuring internal usability metrics manually, whereas the experimental group obtained 96%. This means that

the users were 20% more effective using our framework. This difference was mainly due to metric miscalculations. Effectiveness was normalized and measured by comparing the results obtained from users with the expected results pursued –i.e., the number of metrics correctly calculated (right quantitative results desired for each metric according to the code's characteristics). In addition, efficiency value was also normalized, denoting a significant reduction of average time (more than 50%) in the case of users using our framework. Additionally, lower statistical values were obtained in the experimental group for standard deviation and confidence interval 95%, denoting that the mean values obtained in the experimental group can be considered as an accurate indicator in this case.

Results obtained demonstrate that users in the experimental group carried out the analysis and calculation of metrics more effectively and efficiently than users in the control group. In general, users in the experimental group took advantage of the automatic calculations and the facilities for code annotation provided by the framework, rather than using paper sheets, desk calculators and other resources (e.g., spreadsheets) utilized by users in the control group. On the other hand, no major problems were found during the interaction with the framework. It is worth mentioning, however, that some users missed graphical facilities, such as icons and tool bars, in order to have a more pleasant interaction experience, instead of bearing in mind the main annotation tags. All in all, most users appreciated the automatic calculation of the metrics and the immediacy of the results obtained. Comments received will be considered as future work.

As for the second part of the study, a reliability analysis of the variables used to measure perceived satisfaction was carried out. As shown in Table 5, we obtained values $\alpha > 80\%$ for all variables. In general, Cronbach's alpha values above 70% are considered as acceptable.

| Measure (%) | Usefulness | Ease of Use | Ease of Learning | Overall Satisfaction |
|---|---|---|---|---|
| Mean | 89.08 | 83.06 | 86.01 | 87.17 |
| Min | 74.67 | 64.20 | 72.39 | 72.94 |
| Max | 100.00 | 94.33 | 97.52 | 100 |
| SD | 8.28 | 8.96 | 8.08 | 7.48 |
| Median | 89.66 | 85.69 | 86.63 | 89.07 |
| CI (95%) | 4.05 | 4.39 | 3.96 | 3.66 |
| Cronbach alpha | 82.20 | 83.10 | 85.00 | 82.30 |

**Table 5.** Statistics for each variable related to perceived satisfaction

In addition, high average percentage values were obtained for all the satisfaction variables studied (see Table 5): 89% usefulness, 83% ease of use, 86% ease of learning and 87% overall satisfaction. In general, most users agreed that the framework comprises a useful tool to measure internal usability metrics through the implementation phase.

Results obtained helped answer the proposed research questions. This way, RQ3.1 can be answered in the affirmative as users carried out the measurement of internal usability metrics more efficiently and effectively using our framework in comparison with the manual process. Additionally, RQ3.2 can be also answered in the affirmative as users perceived a high satisfaction using our approach. Average values obtained for perceived satisfaction in-use are above 83% in all cases, while the minimum value considered for validation was 75%.

All in all, results obtained helped answer RQ3 affirmatively, and thus affirm that the framework proposed to systematize the measurement of internal usability metrics results effective, efficient, and satisfactory for the final user.

### 4.7. Threats to Validity

Main threats are principally related to the between-subject design, which comprises several concerns that must be understood and addressed. In general, between-subject designs require a representative number of participants to generate reliable data. In our case, we have divided the total number of

participants into two groups, which may be seen as a threat. However, data normality was checked in both groups to satisfy t-test initial conditions. In addition, fishing and error rate problems are minimized by fitting the t-test initial conditions and avoiding testing many different hypotheses to find a significant effect, reducing the alpha rate (or type I error) by avoiding to repeatedly test the data and pre-specifying the outcomes of interest. On the other hand, we have considered a representative sample size for the experimental group, which is the most important and representative in order to test perceived satisfaction with the framework proposed. Sample size has been justified in order to find a high percentage or problems that might be representative of the utilization of our framework with real users. Another issue related to the between-subject design is the assignment bias, which causes skewed data results and leads to false conclusions. In our case, we prevent this problem by initially carrying out a random process to assign participants to each group, but also counterbalancing the assignments to have groups as homogeneous as possible. Other drawbacks with respect to the between-subject design are related to generalization and individual variability. Admittedly, we considered groups of users with background on (at least) code inspections, usability evaluation and quality, which are the main target for our approach, but also with different ages, gender and provenance. This helps minimize the risk of generalization problems in order to extrapolate the result to broader groups, and also the individual variability by reducing the lack of homogeneity. In addition, the arbitrary time devoted for the initial short talk in the experimental group may be considered as a threat, however this was an interactive talk that most users also utilized to clarify missing concepts, and where the evaluator ensured that all important issues were made clear enough.

Another alternative would have been considering a within-subject design, asking all users to measure internal usability metric with and without our framework. However, we wanted to test effectiveness and efficiency variables in an independent way, saving time by testing both groups simultaneously, and avoiding carryover effects that can negatively affect the evaluation. For instance, asking all users to utilize both evaluation methods –i.e., with and without the framework, may cause that the first test influences the other, arising effects of previously acquired knowledge that may affect the results obtained.

## 5. Conclusions

Quality is an important concern in software development. Quality models, such as ISO/IEC 25000, comprise an interesting approach to address software quality according to diverse characteristics, such as usability, that can be measured through different metrics. In general, a great deal of work has been focused on external and in-use quality metrics. However, less attention has been paid to internal quality measures related to the source code. Internal quality metrics are important as they are related to other usability measures. In general, usability is affected by functional and non-functional requirements, thus it should be measured at the source-code level as well in order to assure usability during all the development process (Cayola and Macías 2018, Sánchez and Macías 2017).

In general, internal usability metrics are analyzed and measured using a manual process by means of code inspections and reviews. This can be time-consuming and costly in terms of the resources needed. In this paper, we propose an approach to improve the manual process, providing users with parameterized annotations that our processor computes automatically, showing the measurement of the metrics at compile time. This allows the development team to save time in analyzing the usability of the code, which can also help compare the results with the usability objectives of the software, enabling to split up responsibilities among different team members. In addition, our approach increases the maintainability of the source code and the software overall by enabling meta-information. Our approach is based on internal usability metrics as they appear in ISO/IEC 25023, but we have contributed by developing technical aspects to get a feasible code implementation.

Our framework has been tested with 32 real users through a controlled evaluation, providing promising results involving effectiveness and efficiency in comparison with the manual inspection process. In addition, the evaluation has provided acceptable results in terms of satisfaction,

demonstrating that users perceived our approach as useful, easy to use and learn and satisfactory. Therefore, initial research questions have been affirmatively answered, demonstrating that internal usability metrics, as defined in ISO/IEC 25023, can be automatically calculated from the source code using annotations, being the proposed solution effective, efficient and satisfactory for users.

As for limitations, the proposed solution is mainly intended for Java code. However, annotations can be developed in other programming languages such as Python, .NET and C#, maintaining the same annotations and parameters proposed in our approach, but probably developing specific annotation facilities for each programming language. This would imply to create or extend the annotation processor where applicable, and proposing annotation alternatives (i.e., using code comments) if the programing language does not provide straight facilities to deal with annotations. All in all, our research is an attempt to provide an acceptable solution to the measurement of internal usability metrics. This is not a definitive solution, but a contribution that can be useful to broaden the research in this area, which may lead to explore other programming languages and development environments, and addressing other internal quality characteristics and metrics.

With respect to future work, we expect to carry through a more interactive annotation process (using icons and tool-bar facilities) in order to increase automation and reduce the cognitive burden (Macías 2008; Macías and Castells 2003). Another interesting issue is how internal metrics can predict external and in-use metrics, determining the overall usability of a software product, and thus connecting internal usability metrics with those related to external and in-use quality measurements. Also, we expect to apply our approach to a real project in order to obtain feedback and improve the framework further.

**Conflict of Interest**: The authors declare that they have no conflict of interest.

## References


Abran, A., Khelifi, A., Suryn, W., & Seffah, A. (2003). Usability meanings and interpretations in ISO standards. *Software Quality Journal*, 11(4), 325-338.

Bailey, R. W., Wolfson, C. A., Nall, J., & Koyani, S. (2009). Performance-based usability testing: Metrics that have the greatest impact for improving a system's usability. *In Proceedings of the International Conference on Human Centered Design*.

Ammar, L. B., Trabelsi, A., & Mahfoudhi, A. (2013). Dealing with Usability in Model-Driven Development Method. *In Proceedings of the International Conference on Enterprise Information Systems*.

Bevan, N. (2009). Extending quality in use to provide a framework for usability measurement. *In Proceedings of the International Conference on Human Centered Design*.

Bevan, N., Carter, J., Earthy, J., Geis, T., & Harker, S. (2016). New ISO standards for usability, usability reports and usability measures. *In Proceedings of the International Conference on Human-Computer Interaction*. Springer, Cham.

Brajnik, G. (2000, June). Automatic web usability evaluation: what needs to be done. In *Proc. Human Factors and the Web, 6th Conference*.

Briand, L. C., & Wüst, J. (2002). Empirical studies of quality models in object-oriented systems. *Advances in computers*, 56, 97-166.

Card, S., Newell, K., & Moran, T. (1999). *The Psychology of human-computer interaction.* Lawrence Erlbaum Associate.

Borges, C.R, & Macías, J.A. (2010). Feasible database querying using a visual end-user approach. *In Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems*.

Carvalho, R.M., de Castro Andrade, R.M., de Oliveira, K.M. et al. (2017). Quality characteristics and measures for human–computer interaction evaluation in ubiquitous systems. *Software Quality Journal* 25, 743-795.

Cayola, L, & Macías, J.A. (2018). Systematic guidance on usability methods in user-centered software development. *Information and Software Technology*, 97, 163-175.

Dix, A., Finlay J.E., Abowd, G.D., & Beale, R. (2004). *Human-Computer Interaction*. Prentice-Hall.

Dubey, S. K., & Rana, A. (2011). Usability estimation of software system by using object-oriented metrics. *ACM SIGSOFT software Engineering notes*, 36(2), 1-6.

Faulkner, L. (2003). Beyond the five-user assumption: Benefits of increased sample sizes in usability testing. *Behavior Research Methods* 35, 379-383.

Fernandez, A., Insfran, E., Abrahão, S., Carsí, J. Á., & Montero, E. (2012). Integrating usability evaluation into model-driven video game development. *In Proceedings of the International Conference on Human-Centred Software Engineering*.

Feuerstack, S., Blumendorf, M., Kern, M., Kruppa, M., Quade, M., Runge, M., et al. (2008). Automated usability evaluation during model-based interactive system development. *In Proceedings of the Engineering Interactive Systems*.

Fraternali, P., Matera, M., & Maurino, A. (2002). WQA: an XSL framework for analyzing the quality of web applications. In *Proc. of IWWOST* (Vol. 2).

Han, W.M. (2017). Evaluating perceived and estimated data quality for Web 2.0 applications: a gap analysis. *Software Quality Journal*, https://doi.org/10.1007/s11219-017-9365-7.

Hwang, W., & Salvendy, G. (2010). Number of people required for usability evaluation: the 10±2 rule. *Communications of the ACM*, 53, 130-133.

IBM (2018). Eclipse Development Framework. http://www.eclipse.org. Accessed 11 November 2018.

IEEE (2008). *IEEE 1028 Standard for software reviews and audits*.

ISO (2001). *ISO/IEC 9126-1:2001. Software engineering- Product Quality – Part 1: Quality model*.

ISO (2002). *ISO/IEC TR 9126-2:2002. Software Engineering – Product Quality – Part 2: External Metrics*.

ISO (2003). *ISO/IEC TR 9126-3:2003. Software Engineering – Product Quality – Part 3: Internal Metrics*.

ISO (2005) *ISO/IEC 25000. Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE).*

ISO (2006). *ISO/IEC 14598:2006. Information Technology – Software Product Evaluation*.

ISO (2011). *ISO/IEC 25010:2011. Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models – Quality Model Division*.

ISO (2016). *ISO/IEC 25023:2016. Systems and Software Engineering – Systems and Software Quality Requirements and Evaluation (SQuaRE) – Measurement of System and Software Product Quality*.

Jabangwe, R., Börstler, J. & Petersen, K. (2015). Handover of managerial responsibilities in global software development: a case study of source code evolution and quality. *Software Quality Journal*, 23, 539-566.

Jabangwe, R., & Šmite, D. (2012). An exploratory study of software evolution and quality: Before, during and after a transfer. *In Proceedings of the IEEE International Conference on Global Software Engineering*.

Kanellopoulos, Y., Antonellis, P., Antoniou, D., Makris, C., Theodoridis, E., Tjortjis, C., et al. (2010). Code quality evaluation methodology using the ISO/IEC 9126 standard. *International Journal of Software Engineering and Applications*, 1, 17–36.

Lettner, F., & Holzmann, C. (2011). Usability evaluation framework: automated interface analysis for android applications. *In Proceedings of the International Conference on Computer Aided Systems Theory*.

Lund, A.M. (2001) Measuring usability with the USE questionnaire. *Usability Interface* 8, 3-6.

Macías, J.A. (2008). Intelligent Assistance in Authoring Dynamically Generated Web Interfaces. *World Wide Web*, 11, 253-286.

Macías, J. A., & Castells, P. (2003). Dynamic web page authoring by example using ontology-based domain knowledge. In *Proceedings of the 8th international conference on Intelligent user interfaces*. ACM.

Memon, A. M. (2009). Using reverse engineering for automated usability evaluation of gui-based applications. *In Proceedings of the Human-Centered Software Engineering*.

Nielsen, J., & Molich, R. (1990, March). Heuristic evaluation of user interfaces. *In Proceedings of the SIGCHI Conference on Human factors in Computing Systems*.

Nielsen, J., & Landauer, T.K. (1993). A Mathematical model of the finding of usability problems. In *Proceedings of the Conference on Human Factors in Computing Systems*.

Oliveira, P., Valente, M. T., & Lima, F. P. (2014). Extracting relative thresholds for source code metrics. *In Proceedings of the Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering*. IEEE.

Oracle (2018). *Java Development Kit*. http://www.oracle.com/technetwork/java. Accessed 11 November 2018.

Orehovački, T., Granić, A., & Kermek, D. (2013). Evaluating the perceived and estimated quality in use of Web 2.0 applications. *Journal of Systems and Software*, 86, 3039–3059.

Otkjær Ba, J., Nguyen, K., Risgaard, P., & Stage, J. (2008). Obstacles to usability evaluation in practice: a survey of software development organizations. *In Proceedings of the Nordic Conference on Human-Computer Interaction*.

Panach, J. I., Condori-Fernandez, N., Vos, T., Aquino, N., & Valverde, F. (2011). Early usability measurement in model-driven development: Definition and empirical evaluation. *International Journal of Software Engineering and Knowledge Engineering*, *21*(03), 339-365.

Perlman, G. (2015). User Interface Usability Evaluation with Web-Based Questionnaires, http://garyperlman.com/quest/quest.cgi?form=USE. Accessed 11 November 2018.

Sánchez, E., & Macías, J.A. (2017). A set of prescribed activities for enhancing requirements engineering in the development of usable e-Government applications. *Requirements Engineering*, https://doi.org/10.1007/s00766-017-0282-x.

Sauro, J. (2018). *MeasuringU*. https://measuringu.com. Accessed 11 November 2018.

Seffah, A., Donyaee, M., Kline, R. B., & Padda, H. K. (2006). Usability measurement and metrics: A consolidated model. *Software Quality Journal*, *14*(2), 159-178.

Singh, Y., Kaur, A., & Malhotra, R. (2010). Empirical validation of object-oriented metrics for predicting fault proneness models. *Software Quality Journal*, 18, 3–35.

Sun (2018). *NetBeans Development Framework*. https://netbeans.org. Accessed 11 November 2018.

Tomas, P., Escalona, M. J., & Mejías, M. (2013). Open source tools for measuring the Internal Quality of Java software products. A survey. *Computer Standards & Interfaces*, *36*(1), 244-255.

Tullis, T., & Albert, W. (2013). *Measuring the User Experience*. Morgan Kaufmann.

Veral, R., & Macías, J.A. (2019). Supporting User-Perceived Usability Benchmarking Through a Developed Quantitative Metric. *International Journal of Human-Computer Studies*. 122, 184-195

Wharton, C. (1992). Cognitive walkthroughs: instructions, forms and examples. *Technical Report CU-ICS-92-17*, University of Colorado.

## Appendix

**A1.** Quality sub-characteristics related to the usability characteristic, as specified in ISO/IEC 25023, together with the set of 12 metrics considered in our approach, including also a brief description, calculations and optimal values

| Quality Sub-Characteristic | Metric | Description | Calculation | Optimal Value |
|---|---|---|---|---|
| Appropriate Recognizability | Integrity of Description | Proportion of functions that are described as understandable in the product description | X = A/B, where A represents the number of functions described as understandable in the product description, and B represents the total number of functions | The closer to 1 the better |
| | Demonstration Capability | Proportion of functions that have demonstration capability | X = A/B, where A represents the number of functions that have demonstration capability, and B represents the total number of functions requiring demonstration capability | The closer to 1 the more capable |
| Learnability | Evident Function | Proportion of functions that are evident to user | X = A/B, where A represents the number of functions that are evident to user, and B represents the total number of functions | The closer to 1 the better |
| | Completeness of User Documentation and/or Help Facility | Proportion of functions that are completely described in the user documentation and/or help facility | X = A/B, where A represents the number of functions completely described, and B represents the total number of functions | The closer to 1the more complete |
| Operability | Operational Error Recoverability | Proportion of functions that can tolerate user errors | X = A/B, where A represents the number of functions implemented with user error | The closer to 1 the more recoverable |

| | | | | |
|---|---|---|---|---|
| | | | tolerance, and B represents the number of functions requiring the tolerance capability | |
| | Operational Consistency | Proportion of operations that behave the same way to similar operations in other parts of the system | X = A/B, where A represents the number of instances of operation with inconsistent behavior, and B represents the total number of operations | The closer to 0 the more consistent |
| | Message Clarity | Proportion of messages that are self-explanatory | X = A/B, where A represents the number of implemented messages with clear explanations, and B represents the number of messages implemented | The closer to 1 the more clear |
| | Customizing Possibility | Proportion of functions that can be customized by user during operation | X = A/B, where A represents the number of functions that can be customized by user during operation, and B represents the number of functions requiring the customization capability | The closer to 1 the better customizability |
| User Error Protection | Input Validity Checking | Proportion of items that provide checking for valid data | X = A/B, where A represents the number of input items that check for valid data, and B represents the number of input items that could check for valid data | The closer to 1 the better |
| | Avoidance of Incorrect Operation | Proportion of functions implemented with | X = A/B, where A represents the number of | The greater the better incorrect |

| | | | | |
|---|---|---|---|---|
| | | incorrect operation avoidance capability | functions implemented to avoid incorrect operational patterns, and B represents the number of incorrect operation patterns to be considered | operation avoidance |
| User Interface Aesthetics | Appearance Customizability of User Interface | Proportion of user interface elements that can be customized in appearance | X = A/B, where A represents the number of interface elements that can be customized, and B represents the total number of interface elements | The closer to 1 the better |
| Accessibility | Physical Accessibility | Proportion of elements that can be customized for access by users with physical handicaps | X = A/B, where A represents the number of interface elements that can be customized for access by users with physical handicaps, and B represents the total number of interface elements | The closer to 1 the better physical accessibility |