2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications

Measurable Concepts for the Usability of Software Components

Thomas Scheller, eva Kühn
Institute of Computer Languages
Vienna University of Technology, 1040 Wien, Austria
Email: {ts,eva}@complang.tuwien.ac.at

***Abstract*—While usability has proven to be an important software quality attribute, its application to APIs is still rather uncommon. Available methods for measuring software usability show significant disadvantages when applied to APIs, like the need for test users and experienced evaluators. This makes it difficult to evaluate the usability of software components, as well as to compare different software components. An API usability measurement method is needed that is both machine-computable and objective.**

This paper takes a first step in the direction of such a measure by identifying measurable concepts for the usability of software components, and validating these concepts against existing studies and guidelines for usability and API design.

I. Introduction

The principle of Component Based Software Engineering (CBSE) has become more and more popular as a way for building high-quality software systems. By splitting software systems into smaller components, their complexity can be reduced, and already available components can be (re-) used. With this, however, the question arises how to identify components of high quality and suitability for a given problem. On the other hand, component creators need to know requirements for producing high-quality components.

Several important quality characteristics for software components have been identified in literature [1], [2], most of them based on or very similar to the software quality attributes defined in the ISO/IEC 9126 standard [3]. All of the models define *usability* as being an important quality attribute for software components: Developers need to have usability in mind when building their own software components, as well as when choosing between a number of externally available software components.

While measurement methods have been proposed for different software component quality attributes (like reusability) in the last years, only very few measurement approaches can be found for usability. Existing methods are often cost- and/or time-expensive, and need usability experts for evaluation. Their applicability often also hasn't been tested for the special area of software components, where not a *GUI* (graphical user interface) is used, but an *API* (application programming interface).

To make usability an applicable decision factor for choosing between different software components, as well as to give developers the possibility to evaluate the usability of their own components, the need arises for a simple and automated usability measurement method.

II. Existing Measures and Usability

For the term *usability* in general there are several definitions available [4], [5]. Only [5] deals with defining usability for software components, and therefore adopts the definition given by the ISO/IEC 9126 standard [3], saying usability is "The capability of the software component to be understood learned, used and attractive to the user, when used under specified conditions". Especially important is the last part of the definition, stating that a software product has no intrinsic usability, only a capability to be used in a certain context ("context of use").

There is a number of existing measures (or *metrics*) that can be applied to measure different aspects of software component complexity. In the following sub sections we give a short overview of them, and explain why they are not fit for measuring software component usability.

A. General Measures

There are several measures that can generally be applied to any piece of code, for example the *Cyclomatic Complexity* [6] and the *Halstead Length* [7]. Further, in the area of object-oriented programming languages many metrics have been introduced to evaluate object-oriented aspects of code structure, for example the metrics of Chidamber & Kemerer [8]. When applied to a software component it means that these measures evaluate aspects of the component's internal complexity. But for the user of a component it is not relevant e.g. how complex the component's overall class structure is, since the user only sees the component's API, which normally is a very small subset of the component's classes and interfaces.

B. Measures for Software Components

There is a number of metrics that have been introduced for measuring different aspects of software components. These metrics can be split into different categories:

- Component interaction [9], [10]
- Component reusability and customizability [11], [12]
- Component complexity [12], [13]
- Component usability [5]

For our purpose only the last category is of interest, which to our knowledge has only been treated in [5]. This paper

978-0-7695-4488-5/11 $26.00 © 2011 IEEE
DOI 10.1109/SEAA.2011.28

IEEE computer society

examines the question if a proposed set of measures that seem to logically relate to usability can be used to measure any sub characteristic of usability [3] with statistically significant relevance. It comes to the conclusion that most of the proposed measures are not significant, while some of the measures in combination show a statistical significance to a certain usability sub characteristic. One of the main reasons why none of the metrics in [5] alone could be related to usability can be seen in the fact that, although the authors clearly state that usability depends on the *context of use*, none of the measures respect this fact.

C. Usability Measures

When talking about usability, it is of course also interesting which general measurement methods exist in this field. Existing measures for the usability of software products can be split into three different categories [4], which are usability testing (e.g. *Thinking Aloud* [14]), usability inspection (e.g. *Heuristic Evaluation* [15]) and usability inquiry (e.g. *Surveys* [14]). Although most of these methods can generally be applied to every kind of "software product", their applicability for software component APIs has not been sufficiently researched. E.g. when using heuristic evaluation, it is likely that other heuristics will be needed for an API than for a graphical UI.

Other problems with existing usability measurement methods are: All methods are strongly dependent on their evaluators, which can lead to wide differences in evaluation results [16]. Also, the results are mainly suggestions for where improvements could be made, but are hard to quantify. These problems would make it difficult to compare results from different software components. Further, many usability tests require a fully functional product, making it hard for component developers to get early response for their software components. For many tests also a representative set of users is needed, and the results from all users must be recorded and evaluated, which makes them cost- and time-expensive.

III. Measuring Software Component Usability

A. Why to Measure

Several reasons why an automated and objective usability measure for software components would be important are:

- To provide the component developer with fast initial response to the APIs of his/her components
- To avoid costly changes in late development stages
- To be able to objectively compare the usability of different components
- To make usability evaluation possible also for developers that are unexperienced in this area, or cannot afford costly usability tests

Existing usability measures cannot fulfill theses points because of the problems mentioned in section II-C. Of course, just like software complexity measures cannot completely replace personal code reviews, such kind of automated usability measure can never completely replace a thorough usability investigation with human evaluators or tests with users. It will for example be difficult to check automatically if the class and method names fit to the language of the domain where the software component is used. An automated measure should nevertheless be able to identify a certain percentage of the usability problems existing within a software component, increasing the probability that the rest of the usability problems is discovered with subsequent usability tests and inspections.

B. A New Approach

According to [5], at least three measurable characteristics can be identified as being related to API usability: The *complexity of the problem*, the *complexity of the solution* and the *quality of the documentation*. Since a comparability between software components is only relevant to compare solutions for the same kind of problem, it can be assumed that all of them share the same *complexity of the problem*, so there is no particular need for evaluating this characteristic. Furthermore, while [5] shows that documentation plays an important role for usability, especially in earlier development phases it will not be fully available for evaluation. When a developer designs a component API, she will evaluate it by writing some usage examples against this API, but not already create its documentation. Therefore we will here focus on only one of these three characteristics, which is the *complexity of the solution* built with a given API and for a given scenario.

We propose that the *complexity of the solution* can further be split into the following measurable sub-characteristics:

1) Interface Complexity: Interface complexity describes the complexity of the parts of a component's API (including interfaces, classes, methods, fields, ...) that are used for implementing a given scenario. The fewer interfaces, classes, methods, ... a system developer must use, and the less complex they are, the better the usability of the software component will be. We believe that this is a main factor of software component usability, and that this cannot yet be evaluated by any existing software measure.

2) Implementation Complexity: Implementation complexity describes the complexity of the resulting code when implementing a given usage scenario. This is especially important for comparing components that are not equal in functionality. If a needed feature is not supported by a certain component, the system developer will need to implement it herself, leading to much higher implementation effort.

For example: For a remote messaging scenario, functionality for lookup and replication is required. From components A and B, both support lookup, but only B supports replication. Although A may have a lower interface complexity and therefore seem easier to use, the better component for the given scenario is B, because the need to implement replication on your own makes A much more expensive in the end.

There are already different existing measures that have proven to be very useful to show the complexity of code, like the ones mentioned in section II-A. These measures will need to be evaluated for their suitability to measure implementation complexity.

3) Setup Complexity: Setup complexity includes any action that needs to be taken so that a given API can be used and the developer is able to create software with it, for example programs that need to be installed and configured. Similar to the *quality of the documentation*, this kind of complexity is mainly important when different components are compared, and won't play a big role when a developer wants to evaluate the usability of his own component. To our best knowledge there exists no literature dealing with this kind of complexity.

C. Measuring Interface Complexity

In this paper, we want to concentrate on finding a way to measure one of these characteristics, namely *interface complexity*. While the other two characteristics are equally important, interface complexity alone already allows a usability evaluation in a limited area: It can be used by component developers to evaluate their interfaces, or to compare two components that are equal in functionality (so the implementation complexity doesn't matter) and only use different concepts in their APIs.

We claim that an API is more complex to use, the more of its *concepts* you need to use, and the more complex these *concepts* are. We define the term *concept* as one single action the developer needs to understand and take when using a certain software component, needed to fulfill the requirements given by a certain scenario (the *context of use*). This could for example be creating an instance of a certain class, or calling a certain method. For every concept it must then be defined how its complexity can be calculated.

Therefore, the actions that need to be taken to calculate interface complexity are: (1) extracting a list of all concepts used in a given scenario, (2) calculating the complexity for every one of these concepts, and (3) combining the calculated values to get a comparable interface complexity value for a given scenario and software component.

IV. Identifying Measurable Concepts for Interface Complexity

To be able to measure interface complexity, our main goal can now be identified as getting a list of all possible measurable concepts, and defining which properties influence the complexity of these concepts, so that in a later stage we will be able to create a measurement approach based on these properties for each concept.

A. Identification Approach

We have identified measurable concepts simply by taking a look at different available software components, implemented several scenarios with them and after that extracted a list of concepts that were needed for implementation. We want our measure to be applicable to modern object-oriented programming languages that are suited for component based development, and therefore concentrated on Java and .Net.

For each concept, the characteristics need to be identified that concern usability and must therefore be taken into consideration. Some characteristics can be easily acquired by programming experience and common sense, e.g. calling a method is more complex the more parameters it has. Other questions cannot be solved that easily, like how to rate the complexity of a concept in comparison to another one.

```
var foo = new FooClass();
foo.Bar = barValue;
foo.Use();
```

VS

```
var foo = new FooClass(barValue);
foo.Use();
```

Fig. 1. Comparison of object initialization variants as in [19] (first is better)

To help answer these questions we derive evidence from literature, as well as from code reviews with our research group and industrial research partners. Helpful resources include API usability studies [17], [18], [19], reports about problems with API design [20], [21], [22], [23], as well as API design guidelines [24], [25].

B. Identified Concepts

A summary of identified concepts is shown in table I. It shows for each concept a list of the identified measurable properties that can be used to evaluate usability. With the knowledge gained from the above mentioned resources we additionally extracted an estimated rating for the complexity of each concept in relation to the other concepts.

1) Instantiation: When using a component, often the first step is that a certain class must be instantiated. Many developers learn using a component by experimenting with it [25] – only after instantiation a developer can explore (non-static) members of the class. So it is especially important that classes that act as a starting point for using the component have a low barrier to entry by requiring a small number of constructor parameters. This is also proven by a study that researches the usability implications of requiring parameters in constructors [19], with the result that a create-set-call pattern (first creating an object, then setting fields, then calling methods) is more usable than, and preferred to, constructors with required parameters (see figure 1). We therefore can define the *number of constructor parameters* as an important property concerning usability for this concept. Further, when choosing which constructor to use, of course also the *number of constructor overloads* is important.

Concerning constructor parameters, not only the number of parameters is important, but also the complexity of each parameter, since not all parameters are equally easy or hard to understand. We will further investigate how to deal with this question in the subsequent section.

2) Method Call: One of the most typical actions when using a software component is calling a method. Often this is done right after instantiating an object, as for example shown in figure 1. This means that when the developer needs to call a method, he has already found the correct class, and is now exploring a very small area of the component (namely the contents of the class in question). For our measurement

TABLE I
SUMMARY OF IDENTIFIED MEASURABLE CONCEPTS

Concept	Measurable Properties Influencing Usability	Rel. Complexity
Instantiation	number and complexity of constructor parameters (incl. generic parameters), complexity of parameters rated by their self-explainability, number and complexity of constructor overloads	medium
Method Call	number and complexity of parameters (incl. generic and return parameters), complexity of parameters rated by their self-explainability, number and complexity of other members of the class, type of method (static or not)	low
Field Access	number and complexity of other members of the class, type of field (static or not)	low
Annotation	can be mapped to the concepts *Instantiation* and *Field Access*	medium - high
Implementation	number and complexity of interface members	high
Inheritance	number and complexity of abstract members of the base class, complexity of the base class constructor, number and complexity of constructor overloads	high
XML Config	number and complexity of xml elements and attributes, availability of XML schema	medium - high

approach this means that the complexity of this concept can be rated lower than for other concepts like *Instantiation*. An important exception to this are *static* methods, where the same problems exist as described in section IV-B1, where a developer first needs to identify an appropriate class, since the step of first instantiating the class is missing in this case. So the fact if the *method is static* is an important property.

Classes should not contain too many public members, since the developer will get confused by too much choice [25]. Therefore we can define the *number of other members of the class* as another important property for this concept.

Similar to calling a constructor, it is clear that a method also becomes more complex, the more parameters it has. Since not all parameters are equally easy or hard to understand, the question is how the complexity of each parameter can be rated best. We think that it is mainly depending on the fact if a parameter is *self-explaining* or not, meaning if it can be understood when looking at a method call. A simple example for this can be found in [25]:

```
Stream stream = File.Open("foo.txt", true, false)
```

The meaning of the first parameter can clearly be figured out from the method name as well as the parameter itself. But the call gives the reader no context to understand the meaning behind the parameters `true` and `false`. The call would be much more usable if it were to use *enums* instead:

```
Stream stream = File.Open("foo.txt",
    CasingOptions.CaseSensitive, FileMode.Open)
```

When judging by the parameter type, as it is done by other measures like in [13], the boolean value seems to be the simpler of the two. But the exact opposite applies when it comes to usability: The usability of the second method variant is higher, as the meaning of the method can be immediately understood when looking at the line of code.

To rate parameters by their self-explainability is an approach that has not yet been used in any other measures. The problem remains how to define a measurement approach for this. From guideline suggestions and the above example, it can be stated that especially enums have a good self-explainability. Other than that, if a parameter can be understood clearly also depends on the fact if the name of the method is able to explain this parameter. The chance for this is higher, the fewer parameters the method has. While we cannot yet give a clear definition about self-explainability, tests could be carried out in the future if this could be measured by looking at the *number of parameters*, and if the *parameter is an enum*.

3) Other Concepts: In contrast to basic concepts like *Instantiation* and *Method Call*, hardly any information is available concerning the usability of less used concepts like interface *Implementation*, class *Inheritance* and *XML Configuration*. We therefore tried to find similarities to the basic concepts, as well as to identify properties from programming experience within our research group. Nevertheless the results for these concepts as shown in table I are still preliminar and require further research, e.g. by conducting usability tests.

V. EXPLORATORY EVALUATION

For evaluation of the identified concepts and measurement approach, we let two students implement solutions for a number of simple scenarios in the area of distributed communication. The components were chosen from freely available .Net middleware solutions with a comparable set of features. For each combination of middleware and scenario, the students stated their opinion of the middleware's usability concerning the given scenario. Also, for each scenario the students rated the used middleware solutions in a range from best to worst.

In a second step, we extracted a list of used concepts for each middleware and scenario. For each implementation we then gave an estimated rating for each concept (according to their measurable properties, as well as their relative complexity compared to other concepts). The results were summed up into a single value for each implementation. These results showed that the concept rating correlated well with the rating the students gave, which we take as proof that, with a proper calculation method for the complexity rating for each concept, the usability of an API can reliably be measured.

An example for a rating by used concepts is shown in table II for a simple request/response messaging scenario using the *Windows Communication Foundation*. For *Instantiation* we gave a rating of 15-25, for *Method Call* (which is considered less complex than *Instantiation*) a rating of 5-15, for *Field Access* 2-12, according to the complexity rating and measurable properties as shown in table I.

TABLE II
EXAMPLE EVALUATION FOR MESSAGING SCENARIO USING WCF

Concept Type	Concept Name	Rating
Instantiation	new ServiceContractAttribute()	15
Instantiation	new OperationContractAttribute()	15
Instantiation	new ServiceBehaviorAttribute()	17
Field Access	ServiceBehaviorAttribute.InstanceContextMode	4
Instantiation	new ServiceHost(Type,Uri[])	20
Method Call	ServiceHost.AddServiceEndpoint(Str,Binding,Str)	12
Method Call	ServiceHost.Open()	8
Instantiation	new NetTcpBinding()	15
Instantiation	new ChannelFactory(Binding,String)	20
Method Call	ChannelFactory.CreateChannel()	7
Overall		**133**

The results of this evaluation are also supported by another test that was carried out with 17 students during a laboratory course, where each student had to solve a given problem with two different middleware components (the used middlewares differed for each student). Students afterwards gave their opinion on which of the two middleware components were easier to use. Afterwards we again extracted the used concepts and compared the results with the students' opinions. In all cases the results showed that the middleware preferred by the students also had a better usability rating when evaluated for the used concepts.

VI. CONCLUSIONS

This paper introduced a new measurement approach for the usability of software components. For this approach, we identified a number of measurable concepts as shown in table I and tried to verify these concepts by using existing reports, studies and guidelines on usability and API design. It thereby represents an important step towards an objective and automated measure for the usability of software components. For some aspects of software component APIs, it became clear that there is no literature available yet concerning their usability. In these cases we tried to find measurable properties through comparison with other concepts and from programming experience in our research group.

For future work, further research will be necessary concerning measurable properties for the identified concepts, especially for those where not much information could be found in literature. Therefore, usability studies will need to be carried out that are especially tailored to test certain concepts of API usage, as it was done in [18] and [19]. Studies will also show more clearly how concepts can be rated in relation to each other, and if things like the order of used concepts play a role for the overall usability. Finally, the identified concepts must be integrated into a measurement approach, using the identified measurable properties. This measurement approach will be tested against proven usability measurement methods.

ACKNOWLEDGEMENTS

The work is funded by the Austrian Government under the program FIT-IT (Forschung, Innovation und Technologie für Informationstechnologien), project 827571 AgiLog – Middleware technologies to reduce complexity for agile logistics.

REFERENCES

[1] A. Alvaro, E. Santana de Almeida, and S. Romero de Lemos Meira, "A software component quality framework," *SIGSOFT Softw. Eng. Notes*, vol. 35, pp. 1–18, January 2010.

[2] P. Botella, X. Burgus, J. P. Carvallo, X. Franch, and C. Quer, "Using quality models for assessing cots selection," in *In Proc. of WER02*, 2002, pp. 263–277.

[3] ISO/IEC, *ISO/IEC 9126. Software engineering – Product quality*. ISO/IEC, 2001.

[4] E. Folmer and J. Bosch, "Architecting for usability: a survey," *Journal of Systems and Software*, vol. 70, no. 1-2, pp. 61 – 78, 2004.

[5] M. F. Bertoa, J. M. Troya, and A. Vallecillo, "Measuring the usability of software components," *J. Syst. Softw.*, vol. 79, pp. 427–439, March 2006.

[6] T. J. McCabe, "A complexity measure," in *Proc. of the 2nd Int. Conf. on Software engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 407–.

[7] M. H. Halstead, *Elements of software science (Operating and programming systems series)*. Elsevier, 1977.

[8] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, pp. 476–493, June 1994.

[9] N. S. Gill and Balkishan, "Dependency and interaction oriented complexity metrics of component-based systems," *SIGSOFT Softw. Eng. Notes*, vol. 33, pp. 3:1–3:5, March 2008.

[10] L. Kharb and R. Singh, "Complexity metrics for component-oriented software systems," *SIGSOFT Softw. Eng. Notes*, vol. 33, pp. 4:1–4:3, March 2008.

[11] H. Washizaki, H. Yamamoto, and Y. Fukazawa, "A metrics suite for measuring reusability of software components," in *Proceedings of the 9th International Symposium on Software Metrics*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 211–.

[12] E. S. Cho, M. S. Kim, and S. D. Kim, "Component metrics to measure component quality," in *Proceedings of the Eighth Asia-Pacific on Software Engineering Conference*, ser. APSEC '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 419–.

[13] A. Sharma, R. Kumar, and P. S. Grover, "Empirical evaluation and critical review of complexity metrics for software components," in *Proc. of the 6th WSEAS Int. Conf. on Softw. Eng., Parallel and Distributed Systems*. Stevens Point, Wisconsin, USA: WSEAS, 2007, pp. 24–29.

[14] J. Nielsen, *Usability Engineering*. San Francisco: Morgan Kaufmann, 1994.

[15] E. T. Hvannberg, E. L.-C. Law, and M. K. Lárusdóttir, "Heuristic evaluation: Comparing ways of finding and reporting usability problems," *Interact. Comput.*, vol. 19, pp. 225–240, March 2007.

[16] M. Hertzum and N. E. Jacobsen, "The Evaluator Effect: A Chilling Fact About Usability Evaluation Methods," *Int. Journal of Human-Computer Interaction*, vol. 15, no. 1, pp. 183–204, 2003.

[17] B. Ellis, J. Stylos, and B. Myers, "The factory pattern in API design: A usability evaluation," in *Proc. of the 29th Int. Conf. on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 302–312.

[18] J. Stylos and B. A. Myers, "The implications of method placement on API learnability," in *Proc. of the 16th ACM SIGSOFT Int. Symposium on Foundations of Software Engineering*, ser. SIGSOFT '08/FSE-16. New York, NY, USA: ACM, 2008, pp. 105–112.

[19] J. Stylos and S. Clarke, "Usability implications of requiring parameters in objects' constructors," in *Proceedings of the 29th international conference on Software Engineering*, ser. ICSE '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 529–539.

[20] M. P. Robillard, "What makes APIs hard to learn? answers from developers," *IEEE Softw.*, vol. 26, pp. 27–34, November 2009.

[21] M. Henning, "API design matters," *Queue*, vol. 5, pp. 24–36, May 2007.

[22] S. G. McLellan, A. W. Roesler, J. T. Tempest, and C. I. Spinuzzi, "Building more usable apis," *IEEE Softw.*, vol. 15, pp. 78–86, May 1998.

[23] S. Clarke, "Measuring API usability," *Dr. Dobb's Journal*, vol. 29, pp. S6–S9, 2004.

[24] J. Tulach, *Practical API Design: Confessions of a Java Framework Architect*, 1st ed. Berkely, CA, USA: Apress, 2008.

[25] K. Cwalina and B. Abrams, *Framework design guidelines: conventions, idioms, and patterns for reusable .net libraries*, 1st ed. Addison-Wesley Professional, 2005.