Software Maintenance like Maintenance in Other Engineering Disciplines

Gustavo Villavicencio
Facultad de Matemática Aplicada, Universidad Católica de Santiago del Estero
4200 Campus de la UCSE, Santiago del Estero, Argentina
gustavov@ucse.edu.ar

ABSTRACT

Software maintenance exhibits many differences regarding how other engineering disciplines carry out maintenance on their artifacts. Such dissimilarity is caused due to the fact that it is easy to get a copy from the original artifact to be used in maintenance, and also because the flat dimension of the software text facilitates access to the components by simply using a text editor. Other engineering disciplines resort to different artifact 'versions' (obtained by dissassembling) where the introduction of modifications (previous comprehension) is easier. After which the artifact is reassembled. In software engineering this approach can be simulated by combining program transformation techniques, search-based software engineering technology and design attributes.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*restructuring, reverse engineering, and reengineering*

General Terms

Algorithm

Keywords

Software maintenance, software comprehension, refactoring

1. INTRODUCTION

In general, the term maintenance in software engineering has different implications from the same term in other engineering disciplines. For the latter, maintenance entails the activities performed to keep the artifact working by solving its deterioration caused by use and the passing of time. On the contrary, in software engineering use and time have no effect on the artifact due to its intangible nature. Likewise, the flat dimension of the software text makes its inspection extremely easy by simply using a text editor. Furthermore, getting a copy of the artifact to be used in maintenance while the original is in execution is not an option available in the other engineering sciences. This easiness can lead us to the false perception that software maintenance would not require some basic procedures that are part of the maintenance process in the other engineering sciences. Such easiness can also have influenced the 'tepid' approach for maintenance that prevails today.

On the other hand, other engineering sciences have neither the option to carry out maintenance on a copy of the artifact, nor to inspect it so easily. In those disciplines, access to a specific component may require the removal of others. Thus, *disassembling* (or *decomposition*) is a common activity during maintenance in the other engineering areas. Clearly, the aim of disassembling is to reach a 'version' of the artifact where the introduction of modifications is easier. In order to simulate this approach in software engineering, the physical disassembling of components from the other engineering disciplines can be replaced by program transformation techniques to generate subsidiary versions (or 'virtual versions') of the artifact. As in the other engineering disciplines, we require that these versions improve the conditions for comprehension and the introduction of changes, while keeping the original version for the subsequent reception of changes by automatically propagating such changes back.

We should note that such disassabling or decomposition used by older engineering disciplines is part of a *phase of preparation* of the artifact for maintenance. Furthermore, such preparation is specific, in the sense that it depends on the specific maintenance request. For instance, if during an aircraft maintenance we have the request for solving a failure in a video system, we may need to remove some plastic parts from the aircraft cabin but not components from the hydraulic systems. We can say, then, that the preparation phase is oriented by the maintenance request. In software engineering, there is not preparation phase as a response to a specific maintenance request. Given a maintenance request, the maintenance machinery tries to solve it by introducing the modification directly into the current version of the artifact, as it is, without making any previous arrangement of the source code to facilitate the changes.

Besides, in software maintenance the easiness for making changes is associated with the *design quality*. That is, a good design will allow the software artifact to be comprehensible and easy to maintain. Traditionally, the software engineering research effort has been focused on keeping a good design with the hope that it will be able to face all the future unpredictable maintenance requests. *We argue that*

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2666613

a good design is necessary for maintenance but not enough.

We want to highlight two design aspects which show that a good design does not guarantee a successful future maintenance. First, software maintenance entails two intrinsically related critical facets, i.e. comprehension and capacity for introducing changes (*modifiability*). Both are human-centered and the second one depends on the first. However, there is a factor affecting comprehension which can not be controlled or measured, the programmer *cognitive style*. Different programmers adhere to different cognitive styles and a good design can not necessarily cope with all of them. Since the capacity for introducing changes depends on comprehension, even a good design can not guarantee such capacity either.

The second 'obscure' aspect of the software design is its characterization by using properties and metrics. The appropriateness of a design is evaluated according to properties like efficiency, security, reusability, maintainability, etc. These properties are quantified by *metrics* that are usually gathered from the code. In the last few years *refactoring* has been one of the most extensively used technique for keeping the software design in good conditions. Unfortunately, such design properties are in continuous *tension* - while some of them are improved others can be affected [1]. The two properties in which this phenomenon is more evident are efficiency and comprehension. Many works have reported that an efficient algorithm is compact and difficult to understand, whereas a well-structured algorithm is easier to understand but inefficient. This circumstance has been better observed in the functional programming setting [7].

So, we can not state that by using refactoring we can improve design in all its aspects at the same time. Here we are not interested in measuring the degree of deterioration of a property as a consequence of improving another one, instead, we want to emphasize that such properties are in continuous tension. We sustain that such tension is caused because the artifact, after being delivered, is only viewed from one perspective. We mean that the current source code artifact in execution is exactly the same as the one in maintenance. But execution and maintenance are different *referential contexts*, demanding entirely different characteristics from the code (execution and maintenance are the two sides of the same coin, the artifact source code). Nevertheless, in the current software engineering belief there is no distinction between the artifact in maintenance and the artifact in execution, both versions are exactly the same and so, exhibit exactly the same attributes. This 'simplification' of reality might be due to the fact that we can get a copy of the artifact and easily explore it.

Thus, in this paper we propose a software maintenance model as in the other engineering disciplines, where an artifact version is functioning while an entirely different version is in maintenance. Such distinction is the consequence of the preparation phase for arranging the source code to solve a specific maintenance request. In the *execution referential context* we want to strengthen properties such as efficiency, security, etc., and diminish others such as comprehension and maintainability which are irrelevant in such conditions. On the other hand, in the *maintenance referential context* we want to strengthen properties like comprehension and maintainability and diminish efficiency, security, etc., which are irrelevant in these conditions. Consequently, in our view software artifact properties are *relative* to a specific referential context. In this way, the new maintenance model not only solves the tension problem among the software attributes, but also considers human factors during comprehension and maintenance. The latter issue is also settled by the model since the transformation process, applied on the artifact version in execution to obtain the version for maintenance, generates subsidiary versions that can be assigned to programmers with different cognitive styles.

Section 2 supplies a more detailed description of the main mechanisms that compose the model. The unresolved issues on refactoring which are important in our context are mentioned in section 3. Finally, section 4 details the conclusions.

2. A NEW MAINTENANCE MODEL

As we said, *execution* and *maintenance* are two *referential contexts* where a software artifact can be after being delivered. Current software engineering makes no distinction between the artifact in one context or in the other. But the needs in one context are different from those in the other, and so, why should we expect to obtain good results in both referential contexts with exactly the same code version? We propose an integral software maintenance model where the software attributes are emphasized according to the referential context considered.

The proposed maintenance model involves the two connected referential contexts (execution and maintenance) biased to keep *consistency* between them. When the maintenance process starts, the version of the current software artifact is in the execution referential context. Since the maintenance model must keep consistency, the artifact version in the maintenance referential context must be *semantically equivalent*. Additionally, as we have indicated previously, we also require that the artifact in maintenance strengthen comprehension and maintainability properties while minimizing efficiency, security, etc. The technology that can accomplish this transformation process is *refactorings* [9]. Once the comprehension is carried out and the changes are introduced into the subsidiary version in maintenance, the model consistency must be *restored* again. That means that the artifact version in execution must reflect the changes introduced to the version in maintenance. To this aim, the transformation process performed before is *reversed* in order to *propagate* the changes introduced to the maintenance version back to the version in execution. This description can be depicted graphically in figure 1.

The versions in maintenance are named *subsidiary versions* and will be generated by refactoring sequences named *reverse refactorings*. The name comes from *reverse engineering* which has among its aims the decomposition of the system in smaller pieces. In our view the aim is similar, not always, though. The *forward refactorings* will be the *inverse* of the reverse refactorings and they will be responsible for propagating the changes introduced to the selected subsidiary version back to the current version in execution.

2.1 Maintenance Request-Driven Transformation

As we indicated in section 1, software engineering lacks a preparation phase for arranging the source code in order to better solve an entering maintenance request. Traditional software engineering assumes that a good design can face all future unpredictable maintenance requests with the same efficiency. But we know that each request has its own char-

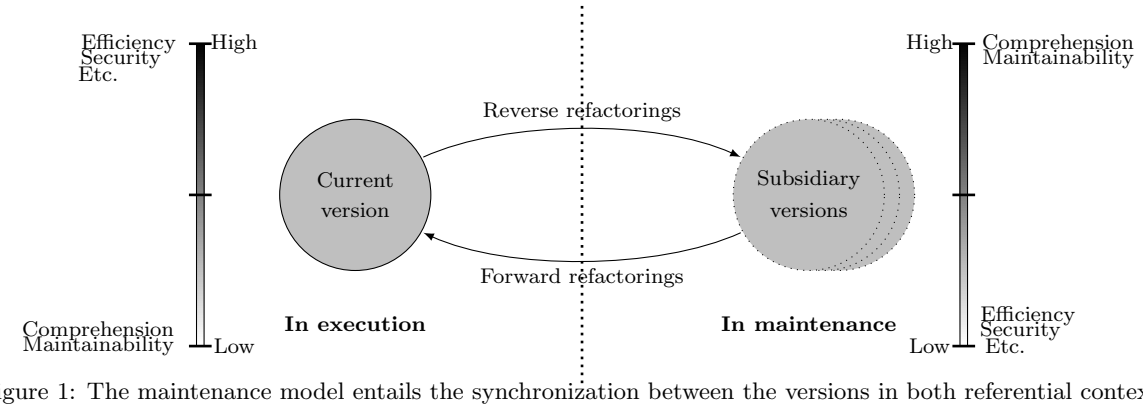

Figure 1: The maintenance model entails the synchronization between the versions in both referential contexts.

acteristics, and thus, it can be solved with different levels of effort according to how well the design accepts the modifications. In our proposal we deal with each maintenance request individually. That means that the generated source code arrangement depends on the current request, exactly like in the other engineering areas (remember the simple example on aircraft maintenance).

In the other engineering areas once there is a clue as to what component is failing, other components are removed to get access to the failing component. In our view a component is a statement or group of statements that can be identified as critical since a relevant data item can be used or updated there, for instance. After identifying the statement (or group of statements) the next step is to isolate it. The isolation of a statement can not be performed as the isolation of a physical component in other engineering areas, that is, by removing the surrounding components. We can simulate this 'fault-isolation process' by reducing the size of the procedure or function where the statement is located.

The isolation process will be supported by sequences of reverse refactorings as in figure 1. The reverse refactorings are refactorings in combination with *Search Based Software Engineering* (SBSE) technology [6]. Refactorings are responsible for carrying out the transformations while the search algorithms in SBSE are responsible for finding the best solution. To this aim, the search algorithms are equipped with a *fitness function*. In our context, using the *'metrics are fitness functions'* approach [5] we can define a metric to measure the size of the functional unit where the critical statement is placed. The definition of this metric should be similar to the *Component Balance* metric defined in [2], whose calculation is based on other two metrics, *system breakdown* (number of components) and *component size uniformity* (the relative size of the components). However, in our view we are interested in smaller components (methods, procedures, or functions) than those considered by [2]. Furthermore, the isolation process must be *parameterized* with the critical statement or group of statements.

2.2 Program Comprehension

The previous preparation phase based on reverse refactorings is focused on reducing the volume of information to be analyzed by the maintainer. Additionally, the application of sequences of reverse refactorings will generate intermediate versions until the critical statement is isolated in a functional unit with a required size. In this way, we avoid to overwhelm the maintainer with unnecessary information by focusing his attention on a smaller source code area.

However such process is not a typical 'zoom effect'. In order to reduce the size of the functional unit where the critical statement is located, the surrounding statements with which the critical statement keeps data and control dependencies must be arranged accordingly. That means that such statements, in turn, are grouped together in separated functional units, which are invoked by the functional unit where the critical statement is placed. Thus, the dependencies of the critical statements with the others will be reformulated in a more 'abstract' style that favours comprehension.

On the other hand, the intermediate versions generated before getting the expected results can also be useful. Clearly, they are a consequence of the intermediate steps performed by the refactoring sequence. These results can exhibit different code arrangements and different granularity levels on the generated functional units, which supply alternative perspectives on the code. On this matter we must remember what was said before regarding the programmers can adhere to different *cognitive style*. Thus, different code perspectives can be associated with different programmers during maintenance.

In [4] Harman introduces the idea of 'semantic lens' through which the programmer could view the algorithmic structure clearer. He also states that the current version in execution can be the most efficient, but the version (the semantic lens) viewed by the programmer executing comprehension, would be the one that better fits its cognitive style. In this view, the artifact version in execution must be semantically equivalent to those in maintenance. Harman's idea is a faithful description of part of our view. In our perspective, the 'semantic lenses' are not only useful for comprehension but also for introducing changes, and they can be propagated to the current version in execution.

2.3 Introducing and Propagating the Changes Back

At first sight, we might assume that the last generated artifact version, where the critical statement will be enclosed in the functional unit with the required size and 'calculated' by the search algorithm, would be selected for introducing the modification to solve the request at hand. However, it would not always be in this way. The maintainer can select a version with which he feels more comfortable when introducing the changes.

Whichever the version used to introduce the modifications, they must be propagated back to the current version in execution in order to keep the consistency of the maintenance model. As we have said, it will be carried out by the

sequence of forward refactorings as shown in figure 1. This sequence is the inverse of the sequence of reverse refactorings calculated as solution by the search algorithm. For this sequence its *precondition* must be calculated, which will be the *AND-sequence of preconditions* corresponding to each refactoring step. The precondition of the sequence of forward refactorings is the responsible mechanism for controlling the propagation process. Three cases can occur with the AND-sequence of preconditions after modifications are introduced into a subsidiary version:

1. The modifications have not affected the precondition of the sequence of forward refactorings and so, the whole sequence of forward refactorings can be performed.
2. The modifications have *partially* affected the precondition and so, only the active subsequence of the forward refactorings can be performed to propagate the change back.
3. The whole precondition is affected and thus the sequence of forward refactoring can not propagate the changes back to the current version in execution.

The first possibility has already be documented and exemplified in the functional setting where the observation of the attributes in tension seems clearer [11]. The first as well as the second case guarantee the changes propagation and the restoration of the current version in execution, and consequently, the maintenance model consistency. In the third case, instead, the modifications can not be propagated to the current version and so it can not be restored. A way to solve this situation would be to 'reverse' the fitness function and find a new sequence of forward refactorings. By reversing the fitness function we mean that instead of improving comprehension and maintainability, the opposite attributes such as efficiency, etc., should be improved. This will bring a new sequence of forward refactorings and, as a consequence, a new current version to the execution referential context. In such case we can not say that the maintenance model is consistent but that it *diverges* to an alternative solution.

3. ISSUES ON REFACTORING

Refactoring plays a critical role in the model in figure 1. We have explained that the fitness function, expressing the decomposition required, guides the transformation process for generating subsidiary versions. However, besides decomposition, we can consider *abstraction* as another criterion to follow. In this case, the aim is that the transformed source code fits a well-known *pattern* or *schema* which, in turn, will improve maintainability. Whatever criterion is considered, new refactorings must be designed. The design of new refactorings with specific orientation and their composition in sequences has been suggested by [8]. Other refactoring aspect to solve is its *genericity*, that is, the refactorings can not only be applied to different programming languages [10] but also to different domains. Finally, a refactoring facet that acquires importance in our context is the construction of a *bidirectional language* for defining refactorings in both directions. This possibility has been suggested in [3].

4. CONCLUSION

After a software artifact is delivered, traditional software engineering does not make any distinction between the source code in execution and the source code in maintenance. However, this 'laid-back' simplification has affected comprehension and maintenance, since they have not been supplied with the appropriate source code perspective. According to our view, the (bidirectional) transformation mechanisms supported by refactoring and SBSE algorithms, and guided by the design attributes can supply the appropriate perspectives required by the two referential contexts.

5. REFERENCES

[1] J. Bansiya and C. G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, Jan. 2002.

[2] E. Bouwers, J. P. Correia, A. van Deursen, and J. Visser. Quantifying the analyzability of software architectures. In *9th Working IEEE/IFIP Conference on Software Architecture (WICSA)*, pages 83–92, Boulder, Colorado, USA, June 2011. IEEE CS.

[3] K. Czarnecki, J. N. Foster, Z. Hu, R. Lämmel, A. Schürr, and J. F. Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective—GRACE meeting notes, state of the art, and outlook. In *ICMT2009 - International Conference on Model Transformation, Proceedings*, volume 5563 of *LNCS*. Springer, 2009.

[4] M. Harman. Search based software engineering for program comprehension. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, ICPC '07, pages 3–13, Washington, DC, USA, 2007. IEEE Computer Society.

[5] M. Harman and J. Clark. Metrics are fitness functions too. In *Proceedings of the Software Metrics, 10th International Symposium*, METRICS '04, pages 58–69, Washington, DC, USA, 2004. IEEE Computer Society.

[6] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43:833–839, 2001.

[7] Z. Hu, T. Yokoyama, and M. Takeichi. Optimizations and transformations in calculation form. In R. Lämmel, J. Saraiva, and J. Visser, editors, *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05)*, volume 4143 of *LNCS*, Braga, Portugal, July 2006. Springer-Verlag.

[8] G. Kniesel and H. Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, Aug. 2004.

[9] W. F. Opdyke. *Refactoring: A Program Restructuring Aid in Designing Object-Oriented Application Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[10] H. Schink, M. Kuhlemann, G. Saake, and R. Lämmel. Hurdles in multi-language refactoring of hibernate applications. In *Proceedings of the International on Software and Data Technologies (ICSOFT 2011)*, pages 129–134. SciTePress, 2011.

[11] G. Villavicencio. A new software maintenance scenario based on refactoring techniques. In *16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*, Zseged, Hungary, March 2012. IEEE.