Universidad Católica de Santiago del Estero

Facultad de Matemática Aplicada

Technical Report TR-1013

Technical Report

Gustavo Villavicencio

Campus de la UCSE
gustavov@ucse.edu.ar

Abstract

Current software maintenance is performed on the running version of the system. However, such version is not always the most suitable for supporting a specific maintenance request. So, successive maintenance applied on an inadequate version gradually diminishes the system. In this paper we sustain that alternative versions of the system can be generated by a systematic refactoring, so that the most suitable is selected for supporting a maintenance request. After maintenance, the introduced modifications are propagated back to the running version. In this maintenance scenario, the main challenge is how to keep the *synchronization* between the version selected for maintenance and the running version.

1 Introduction

Software maintenance is usually performed on the running system version. Whenever there is a coming maintenance request, the maintenance machinery is activated in order to address it. After the modifications, a new running version is delivered and then used for future maintenance. Therefore, the same system version is used for applying every kind of maintenance activities.

Contrarily, other engineering disciplines disassemble their artifacts for maintenance, i.e. they resort to an entirely different "version" of the system to perform maintenance. Moreover, the disassembled version selected to carry out maintenance depends on the maintenance request in hand. In aerospace engineering, for instance, the disassembling required for maintaining the air-conditioning system in an aircraft is different from that required for maintaining the video system. That is, different "aircraft versions" are generated by disassembling according to the maintenance request. Thus, the maintenance request fixes the artifact version on which to carry out the maintenance. Differently, in software engineering only one version, i.e. the running version, has to undergo all the changes.

Furthermore, the major factor during software maintenance is its *comprehension* [von Mayrhauser and Vans1995]. As only one artifact version (the running version) has to undergo all the maintenance activities, and because it is not always the most suitable for receiving a specific maintenance request, comprehension decreases after the modifications. As a consequence, future maintenance becomes more difficult. Figure 1 shows that understanding diminishes as new artifact versions are generated after successive maintenance phases.

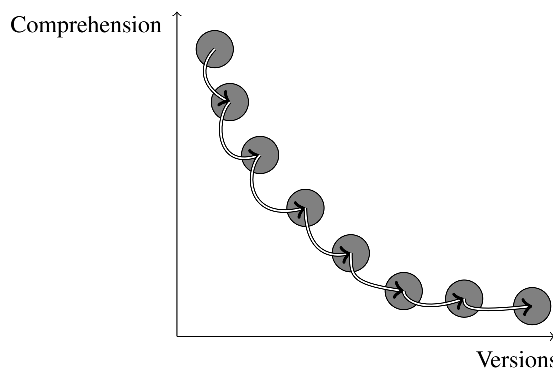

Figure 1: Traditional software maintenance process.

To improve this situation we propose to calculate, by *refactoring* [Opdyke1992],

alternative artifact versions after each maintenance phase. Thus, when a new maintenance request arrives, the most suitable from those alternative versions is selected. Then, the changes made are propagated back from the version where they were introduced to the running version by reversing the original sequence of refactorings. This scenario is depicted in figure 2.

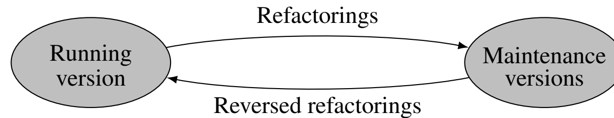

Figure 2: Synchronization between artifact versions.

However, the changes introduced must not violate the conditions that guarantee that, by reversing the original sequence of refactorings, these changes be propagated back to the original version. From a graphical perspective, this involves introducing a new axis in figure 1, i.e. that with the sequence of the alternative artifact versions. Therefore, in this scenario the main problem to solve is how to keep the *synchronization* between the version that receives the modification and the running version.

Some of these ideas have been presented in [Villavicencio2012]. In this paper, we provide an entirely different approach for improving the understanding of this strategy, and the challenges we are faced with. This document is organized as follows: An introductory example is developed in section 2 to show the main ideas. The new perspective on software maintenance is presented in section 3 and the conclusions in section 4.

2 A Motivating Example

Consider the Haskell program [Bird1998] in figure 3 counts the characters, lines and words from an input string. Suppose that this code is currently running and we want to change the mechanism for detecting words. The actual mechanism recognizes a word as a string composed by non-white symbols ended in a white symbol (space, tab, or newline). The new word-detection mechanism recognizes a word as made up of alphabetical characters ending in a white space. This example has also been used in [Gallagher and Lyle1991] for other purposes.

Assume also, that from this running code we calculate a sequence of new artifact versions by refactoring. By applying slicing or *splitting refactoring* [Brown and Thompson2007] we obtain a first of such versions shown in figure 4. Although slicing allows focusing attention on specific code areas by

```
counts l = loops l (0,0,0,False)

loops []    (c,w,l,b)           = (c,w,l,b)
loops (h:t) (c,w,l,b) | blanks h = loops t (c+1,w,l,False)
loops (h:t) (c,w,l,b) | h=='\n' = loops t (c+1,w,l+1,False)
loops (h:t) (c,w,l,True)        = loops t (c+1,w,l,True)
loops (h:t) (c,w,l,False)       = loops t (c+1,w+1,l,True)

blanks c = (c==' ') || (c=='\t')
```

Figure 3: The running Haskell code.

```
countCC l = loopcc l 0

loopcc []     c = c
loopcc (x:xs) c = loopcc xs (c+1)
```

(a) Slice on c.

```
countW l = fst (loopw l (0,False))

loopw []    (w,b)               = (w,b)
loopw (h:t) (w,b)
| blankw h = loopw t (w,False)
loopw (h:t) (w,True)
= loopw t (w,True)
loopw (h:t) (w,False)
= loopw t (w+1,True)

blankw c = c=='\ n' || (c==' ') || (c=='\ t')
```

(b) Slice on w.

```
countL l = loopl l 0

loopl []    l            = l
loopl (h:t) l | h=='\n'  = loopl t (l+1)
loopl (_:t) l            = loopl t  l
```

(c) Slice on l.

Figure 4: Slices generated from the running code.

reducing the volume of lines to analyze, further improvements on the conditions for maintenance (including comprehension) can be gained arranging the code so as to facilitate the introduction of modifications. Thus, in this example, according to the maintenance request in hand, the slice on *w* is of interest, but a most suitable code version can be obtained. Therefore, we continue the transformation process on the slice on *w* by performing the next sequence of refactoring: *replace explicit recursion by operator*, *remove accumulation parameter*, and finally *splitting refactoring* again. The results of this process are shown in figure 5.

In this way, different artifact versions that improve the conditions for comprehension have been obtained, and so the programmer can focus the attention on the one that better fits his *programming style* [Mohan et al.2004]. Further insights can also be gathered by using on these versions the automatic tools already developed for comprehension.

Regarding the maintenance request, we note that there are two main modifications to introduce: remove the boolean variable since it will not be necessary anymore, and change the conditions for detecting words which require not only verify the actual character in the input but also the upcoming. According to this, we consider that the slice version in figure 5c is the most appropriate for introducing such modifications. From there, we remove the second element of the output tuple of the *loopws6* function, and we change the guard in function *auxCount*. This sequence of modification is shown in figure 6. Unfortunately, a *mapping* between a plan for maintenance and a set of code arrangements, as described before, is a process based on knowledge and experience and would not be easily automatized.

After the modifications, we want to reconstruct the original program by reversing the process performed until now. Thus, we noted that after the modifications one function operating in mutual recursion had disappeared and the code arrangement returned to the configuration in figure 5b. It means that the reversing process has to start at this point. Therefore, we perform *introduction accumulator parameter refactoring* and then, we also reverse the previous refactoring performed at the beginning, i.e. we perform *introduction of explict recursion refactoring*.

Reversing the original sequence of refactorings after the modifications is feasible since the modifications have not affected the *preconditions* related to the refactorings. Another aspect to take into account is that such inversion does not imply that the reversed refactoring sequence is obtained by *undoing* the original sequence. Clearly, it is because the introduced modifications preclude such alternative.

So, at this point we have recovered the updated slice on *w*. The last step in

```
countWs3 l = loopWs3 l (0,False)

loopWs3 []    (cw,_)    =  cw
loopWs3 (h:t) (cw,bool) =  loopWs3 t (isWord
(cw,bool)  h)

isWord (w,b)     h | blankWs h = (w,False)
isWord (w,True)  h             = (w,True)
isWord (w,False) h             = (w+1,True)

countWs4 l  =  fst $ foldl isWord (0,False) l
```

(a) Replacing explicit recursion by operator.

```
countWs5 l  =  fst $ loopWs5 $ reverse l

loopWs5 []     =  (0,False)
loopWs5 (h:t)  =  isWord  (loopWs5 t)  h
```

(b) Removing accumulator parameter from *loopWs3*.

```
countWs6  l  =  fst $ loopWs6 $ reverse l

loopWs6 []     =  (0,False)
loopWs6 (h:t)  =  (auxCount (loopWs6 t) h,
                   auxBool (loopWs6 t) h)

auxCount  (c,b) h | (blankWs2 h || b==True) = c
auxCount  (c,b) h
= c+1

auxBool  (_,_)  h | blankWs2 h  = False
auxBool  (_,_)  h               = True
```

(c) Splitting *isWord* function and arranging the parameters accordingly.

Figure 5: Versions of the slice on *w* generated by a sequence of refactorings.

```
loopWs6 []     =  0
loopWs6 (h:t)  =  auxCount (loopWs6 t) h

auxCount  c h | blankWs2 h = c
auxCount  c h              = c+1
```

(a) The calculations related with the boolean variable are removed from the version in 5c .

```
loopWs6 []        = 0
loopws6 (h:[])    = 1
loopWs6 (h1:h2:t) = auxCount (loopWs6 (h2:t)) h1 h2

auxCount c h1 h2 | isSpace(h1) && isAlpha(h2) = c+1
auxCount c _  _
= c
```

(b) The condition of the detecting-word mechanism is changed from 6a.

Figure 6: Sequence of changes on the slice in figure 5c.

the reversing process is *merging* this slice with the other two: slice on *c* and on *l*. The result is shown in figure 7. The result of merging refactoring is far from being obtained by a tool like **HaRe** [Huiqing Li2008]. However an equivalent result can be obtained thus if we further refactor the slice in figure 7b before merging, in order to unify the pattern sets by moving the pattern conditions to the right hand sides.

In this way we have restored the original and monolithical program after updating a refactored version on the slice on *w*. *We hold the idea that human intervention should be limited to the two-step modifications shown in figure 6; and that the transformation process for getting the final slice version in figure 5c as well as the opposite process for propagating the changes back to the running version can be automatized*.

3 A Three-Dimensional View on Software Maintenance

According to the previous example, a new perspective on maintenance arises where a new axis is incorporated to the traditional perspective shown in figure 1, i.e. sequences of semantically equivalent versions of an artifact. Figure 8 shows this new multidimensional space where the planes are the successive maintenances phases, the circles represent artifact versions, and the upward and downward arrows are opposite refactorings.

```
loopWs7 []         w = w
loopWs7 (h:[])     w = w+1
loopWs7 (h1:h2:t) w = loopWs7 (h2:t) (auxCount w h1 h2)
```

(a) Introducing accumulator parameter in updated slice.

```
loopWs9 []
w                              = w
loopWs9 (h:[])
w                              = w+1
loopWs9 (h3:h4:t) w | isSpace(h3) && isAlpha(h4)
                        = loopWs9 (h4:t) (w+1)
loopWs9 (h3:h4:t) w      = loopWs9 (h4:t) w
```

(b) Introducing explicit recursion in 7a.

```
loop [] (c,l,w)                      = (c,l,w)
loop (h:[]) (c,l,w)
= (c+1,l,w+1)
loop (h:t) (c,l,w)      | h=='\n'
                          = loop t (c+1,l+1,w)
loop (h3:h4:t) (c,l,w) | isSpace(h3) && isAlpha(h4)
                          = loop (h4:t) (c+1,l,w+1)
loop (h3:h4:t) (c,l,w)
= loop (h4:t) (c+1,l,w)
```

(c) Merging the slices in figures 7b, 4a, and 4c.

Figure 7: The reversing process after updating slice on *w*.

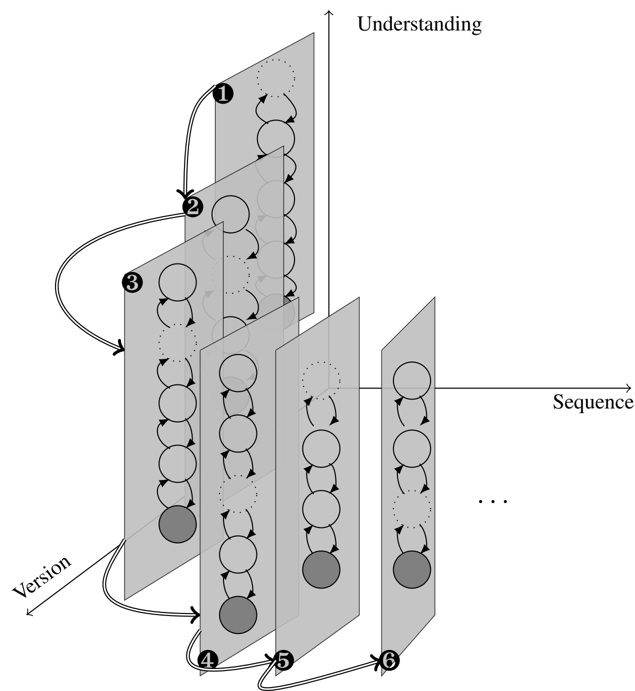

Figure 8: A three-dimensional view of the maintenance process.

The first version of the system is the "seed" which, we assume, is monolithic and efficient (the filled bottom circle on the first plane, figure 3 in the example). From here, an initial sequence of semantically equivalent versions is generated by refactorings (the upward arrows, figures 4 and 5 in the example). Whenever there is an incoming maintenance request the most suitable version is selected (the dotted circle on the first plane, figure 5c in the example) to apply the modifications. After the modifications (figure 6), the changes are propagated back to the running version by reversing the previous sequence of refactorings (the downward arrows, figure 7 in the example), and thus, a new "seed" is generated (the filled bottom circle in the second plane, figure 7c). Clearly, the *consistency* of the artifact versions above the one taken for performing the maintenance must also be checked. It paves the way for a future maintenance (including comprehension) phase.

3.1 Sequence of refactorings

By a sequence of refactorings we mean a sequence of behavior-preserving transformations where each transformation step generates a new meaningful arrangement to the user. "Meaningful arrangement", in turn, means that a refactored version provides additional insight (functional or structural) to that provided by the previous one. It is different from the *composite refactorings* [Opdyke1992] since these are composed of *primitive refactorings*. If we consider this last concept, we must say that we are interested in sequences of composite refactorings.

At the same time, we require that each transformation be meaningful according to the two opposite programming properties, i.e. *comprehension* and *efficiency*, the former intrinsically related to the maintenance version (the upward arrows in figure 8) and the latter to the running version (the downward arrows in figure 8). Refactorings improving comprehension are generally represented by transformations that *decompose* the artifact, or parts of the artifact, in smaller pieces. Another alternative for improving comprehension would be that the code arrangement fits in a design pattern previously defined (*foldl* in figure 5a, for instance). Thus, two are the criteria for selecting this kind of refactorings and/or sequence of refactorings: *decomposition* and *abstraction* (such concepts are closely related to the concepts of *chunk* and *beacon* respectively, well-known in the program comprehension area). On the contrary, efficiency-oriented refactorings improve efficiency usually by assembling fragments into more monolithic and complex ones, or removing the use of auxiliary structures taken from some library.

According to the aforementioned criteria, two are the benefits for consider-

ing alternative artifact versions generated by a systematic refactoring: 1) The conditions for comprehension are improved and, 2) the programmer can select the most amenable version for applying the maintenance request.

On the other hand, an analysis of sequential dependencies has to be carried out to schedule the appropriate order in which refactorings must be applied. This order is relevant since the application of a specific refactoring could preclude the application of others. Besides, there are sequences of refactorings that can be atomically executed and so must be regrouped together. As a result, a careful *schedule of refactoring* must be required [Liu et al.2007, Liu et al.2008]. To be effective, such schedule must be *program independent* [Kniesel and Koch2004].

3.2 The Update Problem

On the other hand, to avoid *undoing* refactorings, the application of a refatoring sequence must be preceded by the verification of the related *condition*. Such condition is the composition of the *preconditions* of refactorings that makes up the sequence. That is,

$$\underbrace{c_1, c_2, ..., c_n}_{\text{Join precondition}} \rightsquigarrow r_1, r_2, ..., r_n \tag{1}$$

where c_i is the precondition related to the refactoring r_i. Similarly, to denote the reversing sequence we will have $c_n^{'}, c_{n-1}^{'}, ..., c_1^{'} \rightsquigarrow r_n^{-1}, r_{n-1}^{-1}, ..., r_1^{-1}$.

In order to propagate the changes back to the original artifact version, they should not have added, deleted, or modified any program elements that influence the truth of $c_n^{'}$. However, this constraint can be relaxed and allow the restoration of the artifact by means of the rest of the refactoring sequence that remains executable. In the example, for instance, we have four preconditions and, after maintenance, the first in the reversed sequence becomes false since the *inWord* function can not be reconstructed by the related inverse. That is $\cancel{c_4^{'}}, c_3^{'}, c_2^{'}, c_1^{'} \rightsquigarrow \cancel{r_4^{-1}}, r_3^{-1}, r_2^{-1}, r_1^{-1}$. However, the remaining active refactoring subsequence can still restore the original artifact version with the modifications. In general, the modifications can break the joint precondition at any point even at the beginning, which will preclude the application of any refactoring from the reversed sequence. In that case, we must define (after maintenance) an entirely new refactoring sequence (as defined by equation 1) that will be used in subsequent maintenance (including comprehension) phases.

4 Conclusion

The present software maintenance scenario is based on one artifact version that must accept all kinds of changes. On the contrary, this paper proposes multiple artifact versions for supporting the maintenance activities (including comprehension). This approach, as shown in the example, encourages an *inside-out maintenance style* (similar to other engineering disciplines) where the programmer focuses the attention on the fragments requiring human intervention, and the rest of changes can be propagated back automatically.

References

[Bird1998] Bird, R. (1998) , *Introduction to Functional Programming* , Series in Computer Science, Prentice-Hall International, 2nd edition

[Brown and Thompson2007] Brown, C. and Thompson, S. (2007) , In *Draft Proceedings of the 19th International Symposium on Implementation and Application of Functional Languages (IFL 2007)*

[Gallagher and Lyle1991] Gallagher, K. B. and Lyle, J. R. (1991) , *IEEE Transactions on Software Engineering* **17(8)**, 751

[Huiqing Li2008] Huiqing Li, Claus Reinke, S. T. (2008) , *The Haskell Refactorer*

[Kniesel and Koch2004] Kniesel, G. and Koch, H. (2004) , *Sci. Comput. Program.* **52(1-3)**, 9

[Liu et al.2008] Liu, H., Li, G., Ma, Y., and Z., S. W. (2008) , *IET Software* 2(5)

[Liu et al.2007] Liu, H., Li, G., Ma, Z., and Shao, W. (2007) , In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pp. 489–492, New York, NY, USA: ACM

[Mohan et al.2004] Mohan, A., Gold, N., and Layzell, P. (2004) , In *11th Working Conference on Reverse Engineering (WCRE 2004)*, Delft, Netherlands: IEEE CS

[Opdyke1992] Opdyke, W. F. (1992) , *Ph.D. thesis*, University of Illinois at Urbana-Champaign

[Villavicencio2012] Villavicencio, G. (2012) , In *16th European Conference on Software Maintenance and Reengineering (CSMR 2012)*, Zseged, Hungary: IEEE

[von Mayrhauser and Vans1995] von Mayrhauser, A. and Vans, A. M. (1995) , *IEEE Computer* **28(8)**, 44