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

The reverse engineering pattern Refactoring to Understand has been proposed as an alternative to the so-called Read to Understand pattern. Although the first has been reported as more effective than the second, it is only true for the programmer performing the refactoring process. That is, the code understanding is only gained by the programmer carrying out the refactoring. In this paper, we propose to refactor in a specific direction, i.e. comprehension oriented. Thus, by a systematic comprehension-oriented refactoring, we plan to construct alternative versions of the code that allows new code readers to analyze the software artifact from different perspectives.

1 Introduction

Refactoring [Opdyke1992, Fowler] has emerged as a technique for transforming the source code while keeping its behavior. Usually, such transformations are oriented to improve some quality aspects of a software component such as *extensibility*, *maintainability*, *efficiency*, *complexity*, etc., [Mens and Tourwe2004, Mens et al.2007]. Specifically, here we are focused on improving *comprehension* only; therefore, we argue that a particular set of refactoring must be applied and/or refactoring sequences with that specific direction must be designed.

It has been held by [Bois et al.2005] that the reverse engineering pattern Refactor to Understand is better than the Read to Understand pattern in many aspects. However, the comprehension gained is only for the programmer performing refactoring, and it is neither represented or saved to be reused. To solve this problem we propose to refactor a software artifact in a systematic way for generating multiple artifact versions. In this way, such versions become useful resources for comprehension by providing different artifact perspectives. Furthermore, comprehension is emphasized since the refactoring is *comprehension-oriented*. That means that the applied refactorings and/or the sequence of refactorings should show some specific behaviour, i.e. *decomposition* (*disassambling*) or *abstraction*.

We focus on comprehension since it has been identified as the major factor during software maintenance [von Mayrhauser and Vans1995]. Thus, our purpose is to generate artifact versions that provide further insight for comprehension, by applying comprehension-oriented refactorings. However, in doing so such versions lose *efficiency*. The observation that comprehension and efficiency are opposite program properties is not introduced in this paper since such phenomenon has already been documented in other literature [Hu et al.2006, Tullsen2002]. At the same time, it means that by reversing the comprehension-oriented sequence the efficiency can be restored in the generated versions. For now, we call *efficiency-oriented refactorings* to the inverse of the comprehension-oriented refactorings. Afterwards we will explain that the efficiency-oriented refactorings (in the current context) are not *undoing* the comprehension-oriented refactorings.

As we have argued, comprehension is critical for maintenance, an expression that can be complemented by “efficiency is critical for running”. Such sentence can be combined with the kinds of refactorings identified before for drawing the scenario shown in figure 1.

By this graphic we want to indicate that the comprehension carried out during maintenance is executed on artifact versions which are different from the

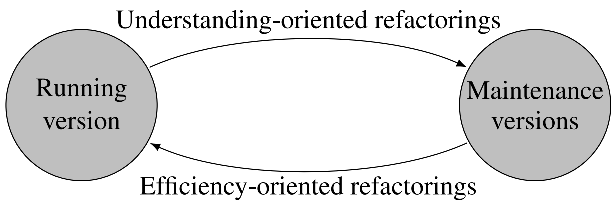

Understanding-oriented refactorings
Running version
Maintenance versions
Efficiency-oriented refactorings

Figure 1: Synchronization between artifact versions.

one that is running, and that such versions are generated by understanding-oriented refactorings. This scenario is new for software maintenance since in this discipline the version that is running is the same as the one that supports the maintenance (including comprehension) activities. However, figure 1 depicts exactly how other engineering disciplines operate on their artifacts: The compact, complex, and monolithic artifact is disassembled in smaller pieces to have a more detailed perspective. That is, the maintainer (who is also performing comprehension before introducing any change), resorts to an entirely different artifact "version" to execute his activities.

On the contrary, in software engineering the running version remains static and unchanged throughout maintenance (and comprehension). By using refactoring techniques we can simulate, in software engineering, how other engineering disciplines operate on their artifacts. That is, we can generate by understanding-oriented refactorings multiple artifact versions that provide useful insight for comprehension.

It is worth mentioning that figure 1 is more general and can involve assumptions that we leave aside in this paper. Specifically, we refer to the efficiency-oriented refactorings arc, which enclose a more "radical" meaning: The modifications made during maintenance, on a properly selected version, can be propagated back to the running version [Villavicencio2012]. Such issue is not developed here. In this work, the mentioned arc simply denotes that efficiency can be restored by reversing the understanding-oriented refactorings.

The rest of this paper is organized as follows: Section 2 briefly describes some facts observed in the literature that justify this work; section 3 presents the strategy together with an example from the functional programming area; section 4 deals with future challenges, and finally section 5 with the conclusions.

2 Background

In the literature on refactoring and program comprehension there are evidence that some issues have not been investigated in depth or from another perspective. This work tries to cover some of such “holes” and they will be treated shortly here.

2.1 A refactoring can improve a quality aspect while damaging other/s

During refactoring there are program properties in continuous *tension*, comprehension and efficiency are properties involved in such phenomenon. Implicit in the *maintainability* quality factor, stipulated by the standard ISO 25000, is how comprehensible a software artifact is. At the same time, *efficiency* quality factor involves how fast an artifact is. However, some literature [Tullsen2002, Hu et al.2006] has explicitly recognized that comprehension and efficiency are two opposed programming properties. Implicitly, [Bois and Mens2003, Kataoka2006] have accepted the same fact. Therefore, if during a refactoring process we are trying to improve maintainability (comprehension), we could, at the same time, damage efficiency. In OOP, *extract method refactoring* is an example of refactoring improving comprehension but affecting efficiency. On the contrary, *inline method refactoring* is the inverse of the previous refactoring that improves efficiency by damaging comprehension. It can be argued that such damages are insignificant but here we are not quantifying them but trying to classify the refactorings according to those opposite properties.

2.2 The question ’what am I refactoring for’ leads to a scheduling of refactoring

Refactoring is an intuitive and planning-lacking activity [Murphy-Hill et al.2009]. So it can be assumed that this is due to the fact that refactoring is frequently performed in a problem-driven way, i.e. the problem to solve states the refactoring to apply. Instead, here we propose an *objective-driven refactoring*, i.e. led to enhance comprehension. An specific orientation of refactoring favors the identification of the refactorings to be used, their schedule, and so their implementation in an automatic support. It also favors the definition of new refactorings from the scratch since, as shown in section 3, we know the characteristics they should have.

2.3 The knowledge of refactoring must be saved to be reused

On the other hand, refactoring is a complex, expensive, and time-consuming activity [Murphy-Hill and Black2008] Therefore, it is important to collect the knowledge acquired during its execution in order to be reused in other activities such as comprehension and maintenance. Regarding this subject, we sustain that the schedule of understanding-oriented refactorings and the results generated by their application are valuable pieces of knowledge. Reverse refactorings and their scheduling can be promoted by expanding their applicability to all the languages in the domain [Verbaere2008, Hills et al.2012] (language-independent refactorings). In this way, the gathered knowledge becomes critical in transforming programs in a comprehension-oriented direction.

3 Comprehension by Refactoring

Based on the previous observations we propose to extend the reverse engineering pattern Refactor to Understand in the following way:

1. The refactorings applied must be comprehension-oriented. To be cataloged as such, a refactoring o sequence of refactorings must show some of the following characteristics:

    (a) *Disassembling* or *decomposition*: The refactoring or sequence of refactorings must lead to a successive fragmentation of the code.

    (b) *Abstraction*: The refactoring or sequence of refactorings can lead to an instantiation of a *generic pattern*. This pattern can be a language or domain specific pattern.

2. According to the selected understanding-oriented refactorings plus the new ones constructed from the scratch following the previous guidelines, a scheduling of refactorings must be defined. So, we will be able to refactor in a specific direction and in a systematic way. Thus, the reverse refactoring schedule outlines a solid support for comprehension since the user can know a priori the orientation of refactoring, the specific refactoring or sequence of refactorings applied, the result generated, and the conditions under which each transformation is performed.

Some further explanations on these aspects are required. Disassembling and abstract have a close relation with the concepts of *chunks* and *beacons* respectively, observation that has also been made in [Bois et al.2005]. Afterwards, in the example, we will see concrete cases of these relationships.

Regarding item 2 we must consider that the disassembling or abstraction effects can be obtained by a single refactoring or by a sequence of refactorings. It not only depends on the current code arrangement but also on the design of the refactoring sequence. Further, a refactoring sequence can generate intermediate versions that are less relevant for comprehension before getting a more useful version where apply disassembling or abstraction. However, the generation of these “irrelevant” intermediate versions can be necessary for getting a final version exhibiting disassembling or abstraction.

An important issue to highlight is that we are not interested in the *history of refactoring* of a code artifact. Instead, we are interested in generating all the possible versions of the artifact in a systematic and automatic way, from the most comprehensible but less efficient to the most efficient but less comprehensible.

With that objective in mind we classify those refactorings with the previous characteristics under the name of understanding-oriented refactorings and their corresponding inverse ones under the name of efficency-oriented refactoring. However, to synchronize with the concepts of *reverse* and *forward engineering* [Biggerstaff1989], we have renamed the first ones as *reverse refactorings* and the second ones as *forward refactorings* [Villavicencio2012]. So, for instance, *splitting refactoring* in the reverse refactoring group has *merge refactoring* as its inverse in the forward refactoring group, and vice-versa. Although some literature has noted there are refactorings with inverses, here we are going further by linking such refactorings with two opposite programming properties.

It could be argued that not all refactorings can be cataloged according to the properties proposed here. But not all refactorings are used in all areas where this technique is applied. Thus, the refactorings used in *refactoring to patterns* [Kerievsky2004] are not equally useful in *refactoring to aspects* [Hannemann et al.2003]. Therefore, we sustain the idea of providing programmers with the ability to define their own reverse refactorings from the scratch. The design of new refactorings with a specific aim has already been introduced in other literature [Kniesel and Koch2004, Verbaere2008, Hills et al.2012].

Figure 2, which is a more detailed view of figure 1, shows what are the expected effects of the previous classification on a monolithic artifact. Note how some artifact parts can be further refactored than others. As we have already indicated it is similar to how other engineering disciplines operate during maintenance.

Although further experimentation is required, the high level refactorings (in classes or methods, for instance) as well as the low level refactorings (in local definitions) are useful in this context. But, in general, we can sustain that fine-grained refactorings deliver more semantic information, which is a helpful element for comprehension.

Finally, and following the parallelism with other engineering disciplines, we must say that by following this approach, we want to construct an automatized support for generating the *disassembling manual* of a software artifact. Further, and as we have suggested in the figure 1, we would always keep this manual updated (automatically) after each maintenance phase.

3.1 An Example

Suppose that after some analysis we have cataloged (in the functional programming area) some refactorings [Brown2008] as shown in table 1. Suppose also that we have implemented, in a tool, a schedule of reverse refactorings shown in figure 3 (only a subset of the transformations in table 1 is considered), where filled dots are refactorings and the arcs or transitions are the information passed amongst them.

The dotted box is the *replace recursion by operator refactoring* that is composed of two simpler refactorings. Figure 4 shows an incomplete view of the components generated by the application of the previous schedule of sequence to a monolithic and complex artifact (the box on the left). In this figure we start with the Haskell code which calculates the number of characters, lines and words from an input string (function *counts*). At the beginning, we apply *splitting refactoring* on each of the first three elements of the tuple returning as result function *loops*.

Consequently we obtain three new subcomponents:

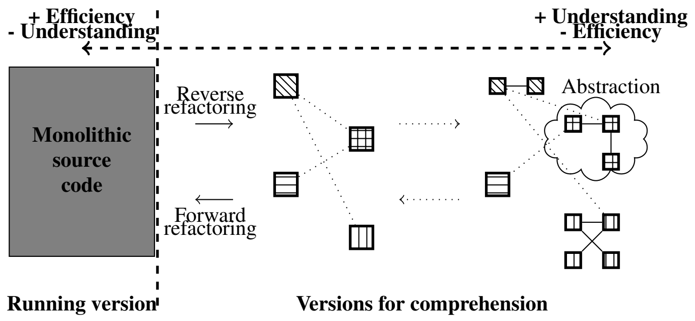


Figure 2: Application of reverse and forward refactorings.

Reverse Refactorings	**Forward Refactorings**
Splitting refactoring	Merge refactoring
Removing accumulator parameter	Introducing accumulator parameter
General function abstraction	Function specification
Fission	Fusion
Folding	Inlining
Replacing recursion by combinator	Replacing combinator by recursion
Introducing mutual recursion	Remove mutual recursion
Removing memoization	Memoization

Table 1: An incomplete catalog of reverse and forward refactorings.

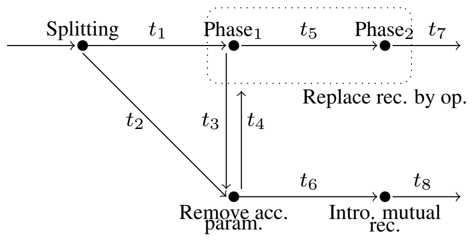


Figure 3: Example of schedule of reverse refactorings.

1. The first one encapsulates all the calculations related to the *c* variable in the tuple. After that, we apply the following sequence of refactorings: *remove accumulation parameter* and *replace explicit recursion by generic operator*. The complete transitions according to the schedule in figure 3 are: t_2, t_4, t_7. Only the final version is shown in figure 4, we omit the intermediate version due to space reasons.

2. The second component encapsulates the calculations related to the *w* variable in the tuple. In this case, the rest of the sequence of refactorings is: *first phase of the replace recursion by operator refactoring*, *remove accumulation parameter refactoring* and *introduction of mutual recursion refactoring*. Thus, the complete transitions are: t_1, t_3, t_6, t_8. Here again, the intermediate versions generated are omitted.

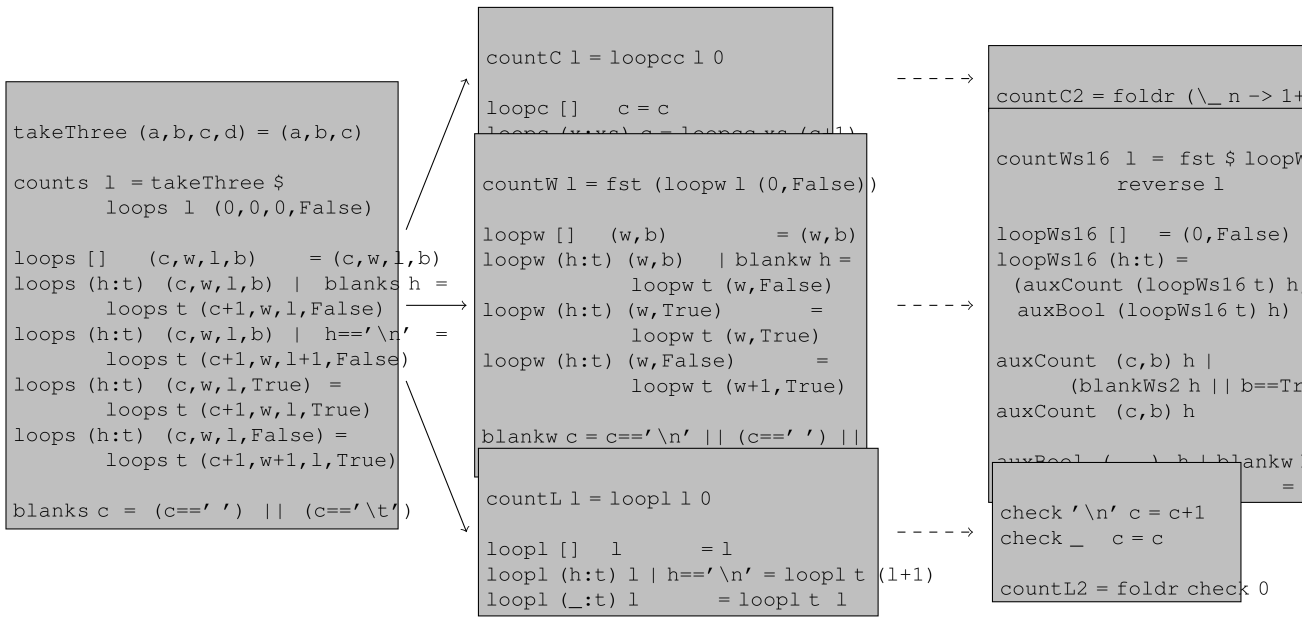
3. The third subcomponent encapsulates the computations related to the *l* variable. The rest of the sequence of refactorings after *splitting refactoring* is the same as the first component:*remove accumulation parameter*

and *replace explicit recursion by generic operator*. So, the transitions are as in item 1. As before, the intermediate version generated is omitted.

In this example, all the applied refactorings are cataloged as reverse refactorings. Regarding the phases of *replace recursion by operator refactoring*, $phase_1$ is reached by transitions t_1 and t_4, but the latter aims at the dotted box meaning that both phases are performed. Other refactorings used here can be decomposed in smaller phases but it is irrelevant in the current example.

Note how in the case of the first and third components, the last versions generated are expressed in terms of a *generic operator* (the abstraction in figure 2) that is in the standard Haskell libraries, i.e. *foldr*. Therefore, by knowing how this operator works, we can understand these components. In these cases, a function call to itself with a inductive structure (a list) as parameter acts like a beacon invoking a recursive schema.

Regarding the second component we have calculated a further decomposition, isolating the calculations of a integer variable (*auxCount*) and the calculations of a boolean variable (*auxBool*) in separate functions. It is an example of what is pointed out in section 3: the application of a refactoring sequence can



```
takeThree (a,b,c,d) = (a,b,c)

counts  l  = takeThree $
        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')
```

```
countC l = loopcc l 0

loopc []    c = 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==' ') ||
```

```
countL l = loopl l 0

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

```
countC2 = foldr (\_ n -> 1+
```

```
countWs16  l  =  fst $ loopW
          reverse l

loopWs16 []   = (0,False)
loopWs16 (h:t) =
 (auxCount (loopWs16 t) h
  auxBool (loopWs16 t) h)

auxCount  (c,b) h |
      (blankWs2 h || b==Tr
auxCount  (c,b) h
```

```
check '\n' c = c+1
check _   c = c

countL2 = foldr check 0
```

Figure 4: Result of the application of schedule of reverse refactorings from figure 3.

generate intermediate versions less useful for comprehension. In this case, the last version exhibiting disassembling is more useful than the previous ones, but the previous ones are necessary for introducing disassembling. Observe here, that the refactoring has chunked the code in separate funcions: *aux-Count* operating like a *counter* function and *auxBool* operating like a *switch* function. Then, these two lower chunks are elements for composing a higher level chunk, i.e. *word_counter*.

So, the last versions calculated for each component and all the intermediate versions (not shown here) will provide different perspective of the respective component. The programmer performing comprehension can select such version that better fits its *programming style* [Mohan et al.2004]. We consider that providing different code arrangements is a helpful resource for comprehension since the same artifact can be viewed from different perspectives, and also, because different programmers can agree with different programming styles. Furthermore, on each representation obtained we can apply visualization tools [Lemieux and Salois2006] for instance, or other kind of comprehension tools to get further insight.

3.2 Forward Refactoring is not Undoing a Reverse Refactoring

It may not be necessary to explain, but when reversing the refactorings just applied, i.e. applying forward refactorings, we must reconstruct the original monolithic artifact. We should also note that it has been assumed that the monolithic artifact from which we started the process is the most efficient and less comprehensible, and also the version currently running. However, this will not always be the case, and by forward refactorings we could calculate more efficient and less comprehensible versions.

Consequently, a mechanism searching for a sequence of refactorings to be applied on the artifact in hand must consider that the selected sequence might be a subsequence of a longer one. However, it has further consequences. Consider the sequence of refactorings in figure 5, where the arcs are the refactorings and the dots their output. Each r_i represents a reverse refactoring.

$$\begin{tikzcd}
\bullet \arrow[r, "r_0"] & \circ \arrow[l, dotted, bend left, "r_0^{-1}"] \arrow[r, "r_1"'] & \bullet \arrow[l, dashed, bend right, "u_1"'] \arrow[r, "r_2"'] & \bullet \arrow[l, dashed, bend right, "u_2"'] \arrow[r, "r_3"'] & \bullet \arrow[l, dashed, bend right, "u_3"']
\end{tikzcd}$$

Figure 5: Application of a sequence (or subsequence) of reverse and forward refactorings.

Let's assume also, that r_1 refactoring has been detected as the refactoring to apply on the artifact in hand. So, by applying the subsequence r_1, r_2, r_3 we obtain the most comprehensible versions. But, what occurs with the reverse refactoring r_0 ? If the forward refactorings are simply implemented by *undoing* (u_i) the reverse refactorings, u_0 will never exist since r_0 has not been executed. Therefore, for executing the inverse of r_0, forward refactorings must be implemented as the real inverses of reverse refactorings: $r_i^{-1}(r_i(P)) = P$. Although the version obtained after applying r_0^{-1} might not be useful for comprehension, as we have said at the beginning of this section, we could calculate more efficient versions than the original one by forward refactorings, in fact, by forward refactorings modeled in the specific way as we explain below.

4 Further Directions

Many research lines are opened from this entirely new approach for program comprehension specifically related to the automatic support required: what analysis must be carried out to gather information for scheduling reverse refactorings?, how must information like the one in figure 3 be represented?, which mechanisms (algorithms) must be used to select the most appropriate sequence of reverse refactorings?, how can this strategy be integrated with the technology already developed in program comprehension?, etc. Although some of these matters have already been treated in [Mens et al.2007] for instance, the fact that here we are focused on improving comprehension, force us to consider aspects like disassembling and abstraction in the refactoring schedule.

However, the more exciting challenge might be in the integration of this comprehension approach in a maintenance environment. Although this comprehension approach can be built in any maintenance strategy, a question arises (almost naturally) that outlines a new software maintenance scenario: Given a maintenance request and after performing comprehension following the approach described here, instead of introducing the modifications in the running version like the traditional software maintenance, why not make the modifications in the most suitable version (like other engineering disciplines) and then, propagate them back (automatically) to the running version? It recalls the *Laplace transform* immediately since the “solution” is founded in a subsidiary representation and then translated back into the original representation. A first approach to the problem of propagating changes in this context has been suggested in [Villavicencio2012], but an alternative one can be considered, i.e. that emerging from the *bidirectional transformations* area [Foster2009, Foster et al.2012]. The scenario depicted in figure 1 is exactly the same environment where bidirectional transformations operate trying to keep the synchronizations between artifact versions.

5 Conclusions

This paper proposes a way for extending the reverse refactoring pattern Refactor to Understand so that the refactoring knowledge (figure 3) and the result generated by its application (figure 4) can be reused for comprehension (and maintenance). This entails the construction of an automatic support for transforming programs in an understanding-oriented way. Basically, the idea is to simulate how other engineering disciplines operate on their artifacts during maintenance, i.e. the maintainer resorts to entirely different artifact “version” for getting better comprehension before introducing any change.

References

[Biggerstaff1989] Biggerstaff, T. (1989) , *IEEE Computer* **22(7)**, 36

[Bois et al.2005] Bois, B. D., Demeyer, S., and Verelst, J. (2005) , In *Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, Manchester,UK: IEEE

[Bois and Mens2003] Bois, B. D. and Mens, T. (2003) , *Describing the Impact of Refactoring on Internal Program Quality*, Technical report, Institute of Computing Science, Faculty of Science, Université de Mons

[Brown2008] Brown, C. (2008) , *Ph.D. thesis*, University of Kent, Canterbury, Kent, England

[Czarnecki et al.2009] Czarnecki, K., Foster, J. N., Hu, Z., Lämmel, R., Schürr, A., and Terwilliger, J. F. (2009) , In *ICMT2009 - International Conference on Model Transformation, Proceedings*, Vol. 5563 of *LNCS*, Springer

[Foster2009] Foster, J. N. (2009) , *Ph.D. thesis*, Department of Computer & Information Science, University of Pennsylvania, Pennsylvania, USA

[Foster et al.2012] Foster, N., Matsuda, K., and Voigtländer, J. (2012) , In *Spring School on Generic and Indexed Programming (SSGIP 2010), Oxford, England, Revised Lectures*. (J. Gibbons ed.), Vol. 7470 of *LNCS*, pp. 1–46, Springer-Verlag

[Fowler] Fowler, M., *Refactoring Home Page*, `http://www.refactoring.com`

[Hannemann et al.2003] Hannemann, J., Fritz, T., and Murphy, G. C. (2003) , In *Proceedings of the 2003 OOPSLA workshop on eclipse technology eXchange*, eclipse '03, pp. 74–78, New York, NY, USA: ACM

[Hills et al.2012] Hills, M. A., Klint, P., and Vinju, J. J. (2012) , In *Proceedings of the 5th Workshop on Refactoring Tools 2012.* (P. Sommerlad ed.), pp. 40 – 49, Rapperswil, Suisse: ACM

[Hu et al.2006] Hu, Z., Yokoyama, T., and Takeichi, M. (2006) , In *Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'05).* (R. Lämmel, J. Saraiva, and J. Visser eds.), Vol. 4143 of *LNCS*, Braga, Portugal: Springer-Verlag

[Kataoka2006] Kataoka, Y. (2006) , *Ph.D. thesis*, Information Science and Technology of Osaka University, Osaka, Japan

[Kerievsky2004] Kerievsky, J. (2004) , *Refactoring to Patterns*, Pearson Higher Education

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

[Lemieux and Salois2006] Lemieux, F. and Salois, M. (2006) , In *Proceedings of the 2006 conference on New Trends in Software Methodologies, Tools and Techniques (SoMeT 2006)*, pp. 22–47, Amsterdam, The Netherlands, The Netherlands: IOS Press

[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

[Mens et al.2007] Mens, T., Taentzer, G., and Runge, O. (2007) , *Software and System Modeling* **6(3)**, 269

[Mens and Tourwe2004] Mens, T. and Tourwe, T. (2004) , *IEEE Transactions on Software Engineering* **30(2)**, 126

[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

[Murphy-Hill et al.2009] Murphy-Hill, E., Parnin, C., and Black, A. P. (2009) , In *Proceedings of the 31st International Conference on Software Engineering (ICSE ' 09)*, pp. 287–297, Washington, DC, USA: IEEE Computer Society

[Murphy-Hill and Black2008] Murphy-Hill, E. R. and Black, A. P. (2008) , In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pp. 421–430

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

[Rech and Ras2004] Rech, J. and Ras, E. (2004) , In *The First International Workshop on Software Quality (SOQUA 2004)*. (S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert eds.), Vol. 58 of *LNI*, GI

[Tullsen2002] Tullsen, M. A. (2002) , *Ph.D. thesis*, Yale University

[Verbaere2008] Verbaere, M. (2008) , *Ph.D. thesis*, Wolfson College, Oxford University

[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