A New Software Maintenance Scenario Based on Refactoring Techniques

Gustavo Villavicencio
Universidad Católica de Santiago del Estero, 4200 Santiago del Estero, Argentina
gustavov@ucse.edu.ar

The problem

Some initial observations that triggered the current work:

- Some literature emphasizes that **understanding** and **efficiency** are opposite programming properties.
- Understanding is critical for **maintenance**, and efficiency for **running**.
- Many refactorings are **understanding-oriented refactorings** (renamed as **reverse refactorings**).
- Their inverses can be considered as **efficiency-oriented refactorings** (renamed as **forward refactorings**).

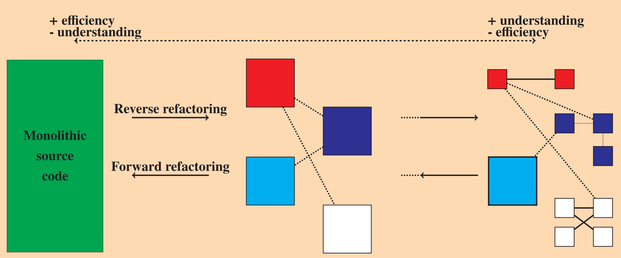
Can we propose more relevant and useful properties than **understanding** and **efficiency** to categorize refactoring techniques? It is almost the natural way to classify refactoring techniques. However, viewing refactorings from this perspective entails further consequences.

- Regarding the current state of the art in refactoring we want to highlight some aspects:
  - Refactoring is an activity performed in the context of many others: during code examination, understanding, design improving, maintenance, etc. That is, refactoring is a transversal activity to other ones. The lack of a well-organized application context makes refactoring a little confusing and/or a not well-organized activity, and so many refactorings are executed on the fly. Here we propose a framework which might provide some insight into to the application of refactoring techniques.
  - If the **refactor-to-understand** pattern [2, 7] is taken as a technique to understand programs, the information used throughout the application process must be properly saved [1] to be reused for new programmers (those who do not know the code being analyzed) faced with comprehension and maintenance activities.

An Initial Catalogue of Refactorings

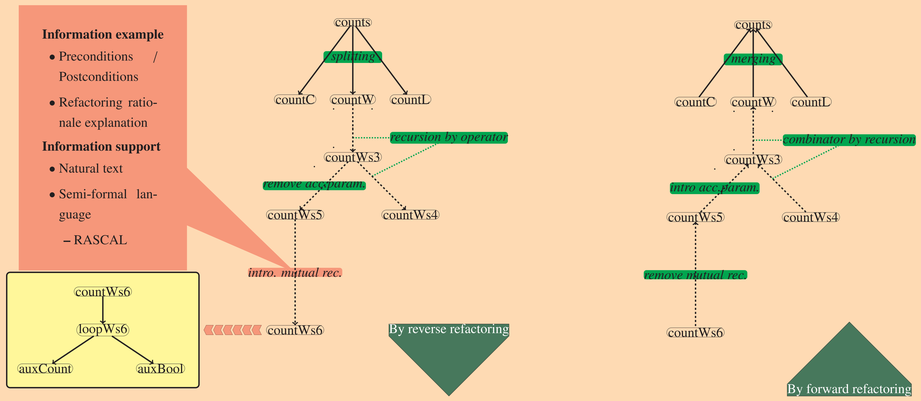
Reverse Refactorings	Forward Refactorings
Splitting refactoring	Merge refactoring
Removing accumulator parameter	Introducing accumulator parameter
General function abstraction	Function specification
Fission	Fusion
Folding	Inlining
Replacing recursion by combinator	Replacing combinator by recursion
Introducing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

A First Consequence



Sequences of refactoring

The next graphs show the sequences of reverse and forward refactoring respectively, applied to the source code example in the paper. The nodes represent a code artifact, and the arrows the refactoring applied. The arrows can be of two types: filled arrow represent **decomposition** and dashed arrow **transitions** between versions.



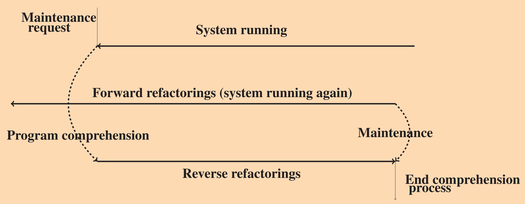
Both representations can be viewed as new "dynamic" representations for program comprehension. They resemble what in the other engineering disciplines are the **disassembly manuals**.

Open question

Up to now we do not have empirical evidence for stating that the comprehension process of programmers who do not know the code can be improved by providing information extracted from the refactoring process. The **refactor-to-understand** pattern improves the understanding of the programmer who is performing the refactoring process. The hypothesis supported here is that the comprehension process of those programmers who are not familiar with the code will be significantly improved by extracting the information from the reverse refactoring process and render it the proper way.

Further Consequence: A New Maintenance Scenario

Suppose that in the previous context, where more than one comprehensible version is available after the application of reverse refactorings, we want to use some of such versions to introduce updating instead of modifying the (monolithic) running version directly.

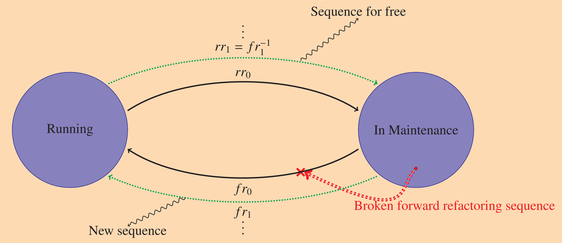


Open question

One immediately question arises: what is the most appropriate version to carry out a specific maintenance request? That is, if we have a more comprehensible version where it should be easier to introduce modifications in order to reply a specific request, and assume that we can reconstruct the (monolithic) efficient version after such modifications, why not using such version for maintenance? In the paper example, for instance, the required modifications have been introduced in the version generated by **removing accumulation parameter** refactoring, i.e. *countWs5* function. However, the same modification can also be introduced in the version generated by the application of **introduction mutual recursion** refactoring, i.e. *countWs6*. However, in the last case modifications are harder to handle since the critical sentences are now split in other two functions.

Synchronization Between Versions

In real working conditions we must relax the restriction in order to allow modifications to break the sequence of forward refactoring (those obtained by reversing the reverse refactoring) for restoring the system. Then a new sequence of forward refactoring must be set up. Interestingly, the next time the artifact has to be maintained the sequence of reverse refactorings to be applied is obtained by reversing the last sequence of forward refactoring.



Formalization

A lot of work has been done in other areas providing the foundation for the approach being described.

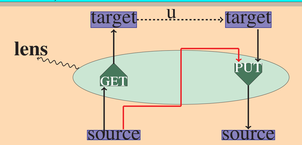
- **Program inversion**

In the same way compression-decompression or coding-decoding problems have been studied in the **program inversion** area, reverse and forward refactorings are also excellent candidates to be studied.

- **Composing refactorings**: In many refactorings, the application of another of them entails the previous application of other one. So, refactorings can be combined to set up sequences of refactorings. In our context such sequences can be constructed in both directions saving a lot of work.
- **Conditional transformation**: Going further on refactoring composition we can also construct composite refactorings that are reusable on arbitrary programs [6].

Bidirectional Transformations

Bidirectional transformations are mechanisms for maintaining the consistency of two (or more) related sources of information.



The well-known problem on **view updated** can be enunciated as follows. Let's take a **source** of information s, originally a database. A function **get** is a query such that a specific view is generated: $v = get(s)$. The function u updates the captured view v and generates a new one v'. The problem is how to propagate the changes in the view v' on s.

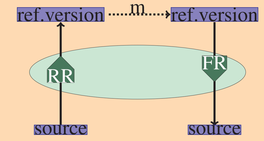
The purpose of bidirectional programs is kept consistency between source and target. With this aim, bidirectional languages model the forward and backward transformations under the same expression; that is, every expression can be read in both directions!

Broadly speaking, lenses are designed to guarantee three main properties [4]:

- Robustness. The modifications in the view are carried out without having to consider whether they are consistent with the underlying source.
- Lenses propagate view updates exactly to the source.
- When possible, lenses preserve any source of information that is not reflected in the view.

RR and FR as Bidirectional Transformations

We can fit the current maintenance scenario based on reverse and forward refactoring in the bidirectional transformation context as follows:



The maintenance **m** is performed on the refactored version obtaining a refactored version updated. The important issue here is how to model the forward transformation in order to inject the changes made in the refactored version into the monolithic and efficient source code.

Unlike the original view update problem where the target obtained by the get function application loses information, here the refactored version obtained is semantically equivalent to the original one. It means that each pair of reverse refactoring with its inverse can be designed as **bijective lenses**.

Feasibility

The most challenging aspect to solve is the put function that propagates the changes to the source code. Regarding the technology available to cope with such problem there are many successful experiences in the application of bijective lenses: **Janus** [8], **XSugar** [3], **biXid** [5].

Although the domain of refactorings is more complex than the previous ones, refactorings and forward refactorings specifically represent a sound and well-understood framework.

We require a language of **refactoring lenses** for:

- Designing pairs of reverse and forward refactoring. It would involve design of lenses for typical syntactic structures: sequences, conditional, iteration, recursion, and so on, depending on the language being handled. This aspect should be generalized to extend the spectrum of the lenses.
- Operators for combining, removing and inserting refactoring lenses in a sequence. Furthermore, extend such operators in order to handle sequences of refactoring as unit.
- Operator for refactoring lenses parallelization.

Benefits

- The proposed classification of refactorings provides:
  - A well-balanced environment for understanding and maintenance.
  - It can improve the application of refactorings.
  - It can encourage the development of new refactorings.
  - It can improve the design of refactoring tools, promoting and encouraging the development of new ones.
- Program understanding is improved by reverse refactorings.
- Reverse refactorings unveil the critical fragments making the introduction of modification easier.
- A new understanding-oriented dynamic view of the code artifact structure.
- The proposed approach emphasizes the **inside-out maintenance**, which means that:
  - There is a framework to introduce changes in a more systematic way.
  - Many changes can be automated: The critical modifications can only be introduced by the programmers, the rest must be automated.

References

[1] Andrew P. Black, Danny Dig, and Chris Parnin. Gathering refactoring data: a comparison of four methods. In *2th Workshop on Refactoring Tools*, Nashville, Tennessee,USA, 2008. ACM.

[2] Bart Du Bois, Serge Demeyer, and Jan Verelst. Does the "refactor to understand" reverse engineering pattern improve program comprehension. In *Ninth European Conference on Software Maintenance and Reengineering (CSMR'05)*, Manchester,UK, March 2005. IEEE.

[3] Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Dual syntax for xml languages. *Inf. Syst.*, 33(4-5):385–406, June 2008.

[4] John Nathan Foster. *Bidirectional Programming Languages*. PhD thesis, Department of Computer & Information Science, University of Pennsylvania, Pennsylvania, USA, 2009.

[5] Shinya Kawanaka and Haruo Hosoya. bixid: a bidirectional transformation language for xml. *SIGPLAN Not.*, 41(9):201–214, September 2006.

[6] Günter Kniesel and Helge Koch. Static composition of refactorings. *Sci. Comput. Program.*, 52(1-3):9–51, August 2004.

[7] Gustavo Villavicencio. Refactoring for comprehension. In *Draft Proceeding of the 8th. Trends in Functional Programming*, New York, USA, April 2007. Seaton Hall University.

[8] Tetsuo Yokoyama, Holger Bock Axelsen, and Robert Glück. Principles of a reversible programming language. In *Proceedings of the 5th conference on Computing frontiers*, CF '08, pages 43–54, New York, NY, USA, 2008. ACM.