2009 Fourth International Conference on Software Engineering Advances

Strengthening Refactoring:

Towards Software Evolution with Quantitative and Experimental Grounds

Sérgio Bryton
QUASAR/VALSE Team, CITI
Departamento de Informática, FCT
Universidade Nova de Lisboa
Monte da Caparica, Portugal
sergio.bryton@gmail.com

Fernando Brito e Abreu
QUASAR/VALSE Team, CITI
Departamento de Informática, FCT
Universidade Nova de Lisboa
Monte da Caparica, Portugal
fba@di.fct.unl.pt

Abstract: *Refactoring is a process meant to improve the internal quality of software systems. However, while on one hand, the guidelines for this delicate process are still empirical and qualitative, on the other hand, software product metrics often indicate that this process has the opposite results. Also, there is a lack of evidence regarding improvements on maintainability due to refactoring. This means that this process, although widely acknowledged as one of the best software practices, is difficult to deploy within large scale software systems, and can be better grounded. To address these challenges, we propose a method for refactoring with quantitative and experimental grounds. Upon the consolidation of this method, we will build the necessary blocks to implement and validate it.*

Keywords: *Software Design; Quality Analysis and Evaluation Techniques; Software Engineering Tools and Methods; Software Quality Tools; Review and Audit.*

I. INTRODUCTION

A. Overview

Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure, with the purpose of minimizing the chances of introducing bugs [1]. However, this technique still presents some fragilities:

1) *Qualitative, inefficient and error-prone process;*
2) *Lack of quantitative evidence of the effects of refactoring on product quality.*

Take, for instance, the following heuristic, proposed by Fowler [1] to detect the *Long Method* bad smell:

"Whenever we feel the need to comment something, we write a method instead."

This kind of guidelines is mostly subjective, thus not repeatable, prone to errors and not automatable [15, 16]. Another evidence of refactoring fragilities occurs when it is reported that although people use refactoring to improve the quality of their systems, metrics indicate that this process often has the opposite results [2].

B. A refactoring example

To better understand the problem, we will use one of Fowler's refactoring examples, the "Video Store", which is a program to calculate and print a statement of a customer's charges at a video store [1].

In this program, to justify the need for refactoring, some design flaws are pointed out [1], like these in the *statement()* method[1] from the *Customer* class:

1) *The* statement() *method is too long;*
2) *The* statement() *method does too much;*
3) *Many of the things done by the* statement() *method should be done by other classes.*

Requirements evolution also points out [1] to the consequences of not performing the refactoring:

1) *If we want a method to produce an* HTML *statement, none of the behavior of the* statement() *method can be reused, leading to the implementation of a similar* htmlStatement() *method;*
2) *If the charging rules change, both (*statement() *and* htmlStatement( )*) methods have to be changed;*
3) *Changing the way movies are classified, will affect both the way renters are charged for movies and the way that frequent renter points are calculated, that is both (statement() and htmlStatement()) methods will have to be changed consistently.*

As we see, the refactoring decision process is empirical and qualitative in nature, which may lead to errors, hampers its usage in large scale software systems and will hardly be repeatable.

To analyze the impact of this refactoring on maintainability, we collected a set of software product metrics, all defined in [16], from both versions of this example, with a tool [14], to try to find evidence of the claimed product quality improvements.

[1] The initial and final versions of the *statement()* method can be found in appendix A.

978-0-7695-3777-1/09 $26.00 © 2009 IEEE
DOI 10.1109/ICSEA.2009.90

From Figures 1 and 2 we can see that the *statement()* method has improved, after the refactoring, regarding size and complexity.

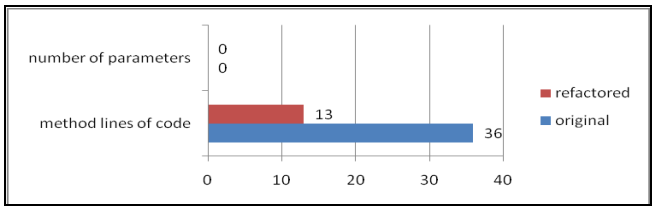

Figure 1: Size metrics for the *statement()* method from the *Customer* class

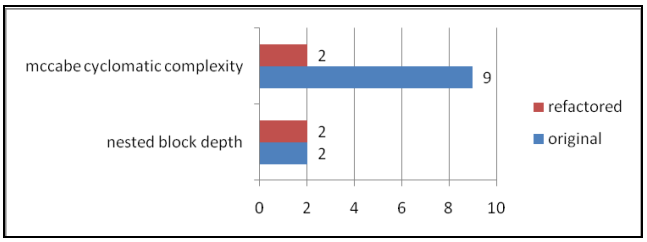

Figure 2: Complexity metrics for the *statement()* method from the *Customer* class

From Figures 3, 4 and 5 we can see that the *Customer* class did not have improvements regarding size. On the contrary, the *method lines of code* and the *number of methods* have increased. Regarding complexity and cohesion, this class records minor improvements after refactoring. Notice that with almost the double of the *number of methods*, the *method lines of code* practically remains unaltered, which means that, in spite of having more methods, these are shorter. Even though the *mean cyclomatic complexity* is half the original, the *number of methods* has doubled, therefore this does not mean necessarily that the methods are less complex, as the results of the *weighted methods per class* confirms.

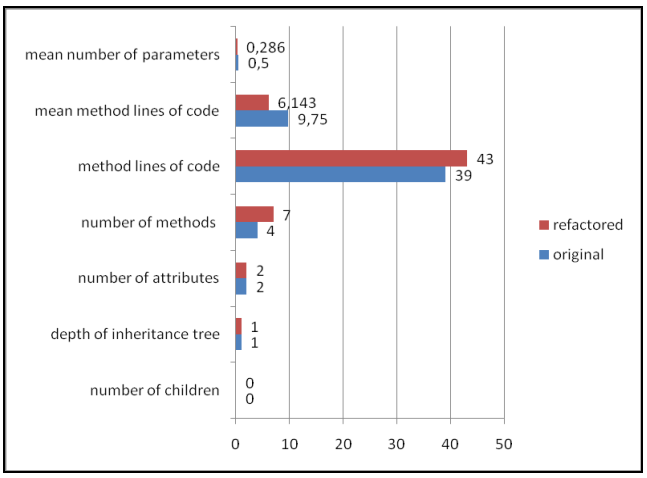

Figure 3: Size metrics for the *Customer* class

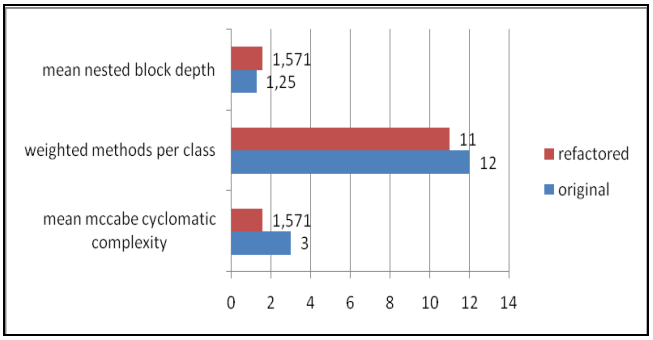

Figure 4: Complexity metrics for the *Customer* class

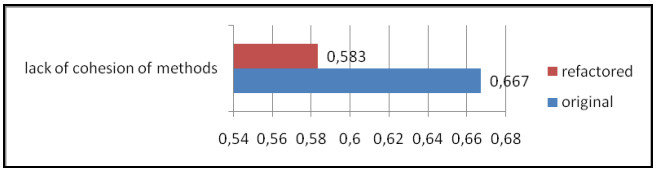

Figure 5: Cohesion metrics for the *Customer* class

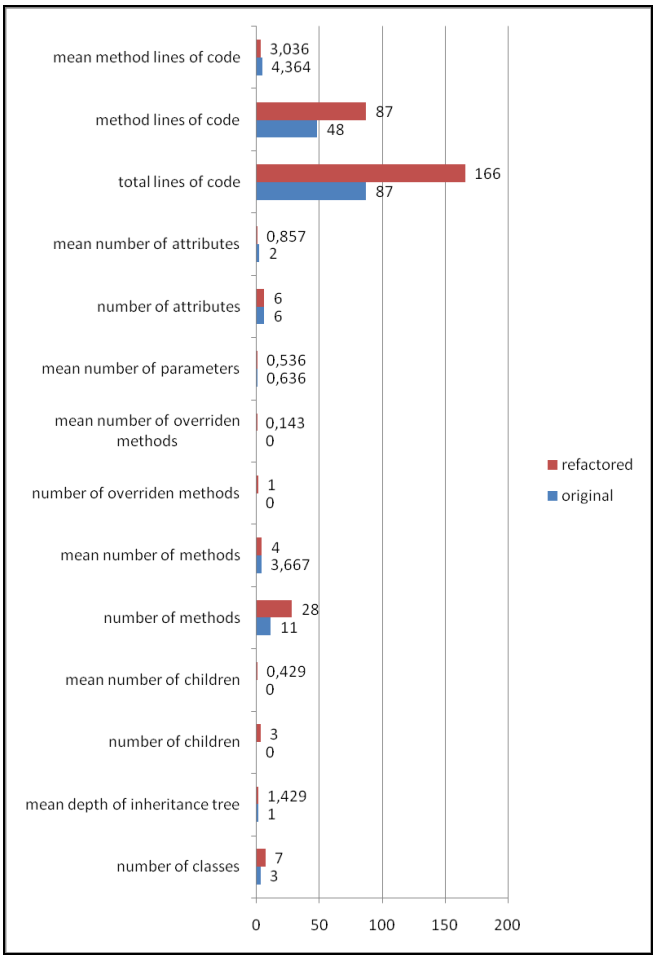

Figure 6: Size metrics for the *Video Store* program

From Figures 6, 7 and 8 we can see that the *Video Store* program has practically doubled in size, considering the *number of classes*, the *number of methods* and the *total lines*

of code. Another result is that of the *mean method lines of code*, which is slightly smaller in the refactored version. However, since this version has almost the double of the number of methods, in average, the methods of the refactored version are not smaller than those of the original version. Regarding cohesion, in spite of *the mean lack of cohesion of methods* being half of the original, since the *number of methods* practically doubled, the cohesion benefits in the overall program are irrelevant. Finally, the *mean cyclomatic complexity* is practically the same in both versions, meaning that the methods in both versions have similar complexities. This is corroborated by the *weighted methods per class*, which is almost the double of the original version, and coherent with the *number of methods* in the refactored version, which is also practically the double of the original version.

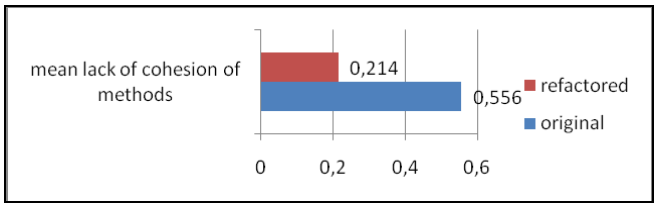

Figure 7: Cohesion metrics for the *Video Store* program

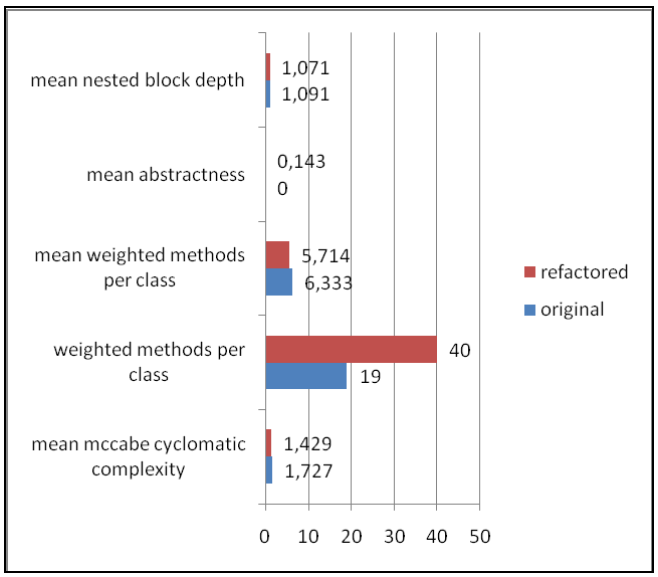

Figure 8: Complexity metrics for the *Video Store* program

Considering the results obtained, there is no evidence that the internal software quality has improved. On the contrary, it is generally worse in size, complexity and cohesion.

Since the process which lead to these results [1] was empirically sound and performed by an expert, either these metrics are not appropriate to evaluate the internal quality of software systems, namely the changes introduced by refactoring, or there is a price to pay for maintainability in cohesion, complexity and size[2]. However, to the best of our knowledge, there is no evidence of the maintainability benefits resulting from refactoring and, therefore, we cannot evaluate this tradeoff.

To summarize, this research intends to provide a solid contribution towards mitigating some of the refactoring fragilities presented above, thus making it a stronger process.

The remainder of this paper is organized as follows. Section 2 will present a synthesis on the state of the art; section 3 will present the research objectives and approach; in section 4 the current work and preliminary results will be described, followed by section 5 where the work plan and implications are described and by section 6 where some preliminary conclusions are drawn.

II. State of the Art

Several refactoring catalogues have been proposed, being the most widely accepted those from Fowler [1] and Kerievsky [5]. There are also some works regarding refactoring from OOP to AOP, such as those from Laddad [6] and Monteiro [7]. All of these approaches are of qualitative nature and lack adequate empirical validation.

Stroggylos et al. [2] question if refactoring improves the product quality, based on metrics results.

Simon et. al. agree with the difficulty of identifying where to apply each refactoring, and propose object-oriented cohesion-based metrics to solve this problem. However, they do not assess the quality improvements after the refactoring [9] and cohesion itself is not enough, as can be concluded from the previous example.

Tahvildari proposed a taxonomy for design flaws, a reengineering strategy [11], and a framework to detect design flaws and re-engineer them [12, 13] for object-oriented systems using, among other, classical modularity metrics. The possibility of refactoring by using design patterns or aspects is not addressed in her work.

Mens [8], Simon [9], Naji [10] and Tahvildari [11,12,13] corroborate the opinion that metrics can identify potential refactorings and estimate the refactoring effect.

In spite of the research done so far, the relation between code smells, refactorings, and their effects on the internal product quality and maintainability has a lot of room to progress.

III. Research Objectives and Approach

A. Research Objectives

The objective of this research is to provide quantitative and experimental grounds to the refactoring process, namely regarding code smells detection and refactoring selection, as well as provide evidence of its effect on the internal product quality.

The main expected contributions of this research are:

[2] Coupling was not measured since the only available metrics in the used tool required more than one package, which was not the case (*Afferent and Efferent Coupling*).

(1) *A refactoring method (QBR: Quantitative-Based Refactoring) with quantitative and experimental grounds, depicted in figures 9, 10, 11 and 12;*

(2) *A catalogue of adequate metrics for code smells detection;*

(3) *A compilation of the actual refactoring catalogues, including the alternatives for Design Patterns and Aspect-Oriented Programming, amended by quantitative evidence of each refactoring effect on software product internal quality;*

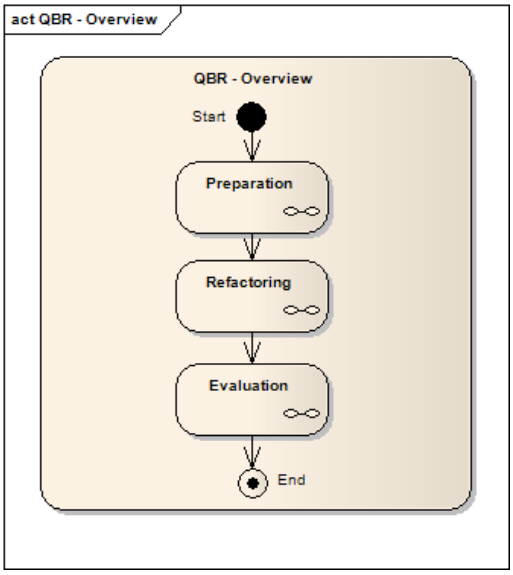

Figure 9: QBR Overview

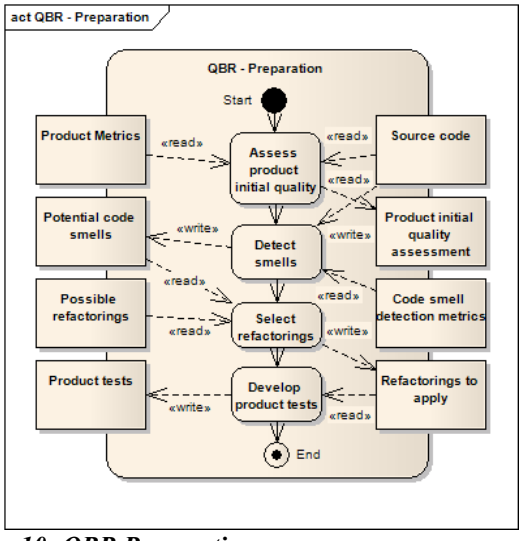

Figure 10: QBR Preparation

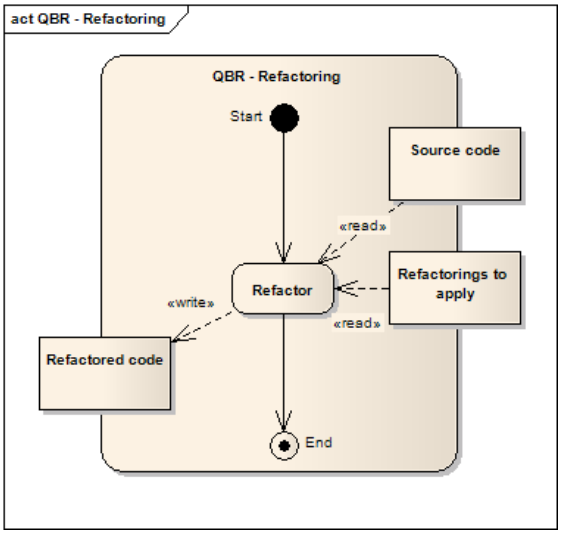

Figure 11: QBR Refactoring

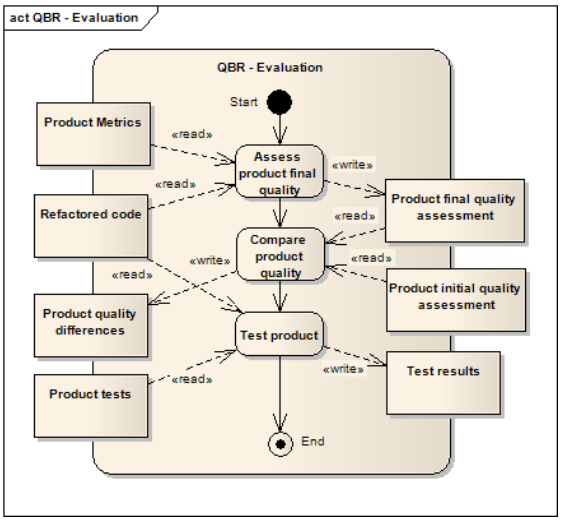

Figure 12: QBR Evaluation

B. Research Approach

The classical scientific method will be followed, since it is appropriate for software engineering research [3, 4].

First, an evaluation of the state of the art will be conducted; then the QBR method and the remainder contributions will be developed and validated with experiments and statistical methods.

To develop the catalogue for code smells detection metrics, we intend to use statistics, like the Binary Logistic Regression, to define a mathematical model for each code smell, based on product metrics and expert's opinion. Each model will then be used to detect code smells based on probabilities.

Each code smell can be refactored according to several alternatives (aspects, design patterns, etc). The objective will be finding which is the best alternative for each smell. We intend to conduct experiments to find statistical evidence on the benefits of each alternative to allow conclusions generalization. The outcome will be a rank of alternatives for

refactoring based on the probabilities of improving the code best.

IV. Current Work and Preliminary Results

A lot of significant references have already been collected and these seem to underline the actuality and relevance of this research.

Experiments and several statistical methods have already been successfully used to study software quality properties like modularity [18].

The QBR method is an evolution and generalization of the MORe method to software properties other than modularity [15].

The data presented here, regarding the refactoring effects on the internal product quality, is a first step towards the usage of metrics for code smell detection.

The Binary Logistic Regression has already been used successfully [16] to detect the *Long Method* bad smell [1], and a model has been established for it. However, further experimentation is required to generalize the results.

V. Work Plan and Implications

The work plan is divided into four phases, which are expected to take place between September 2009 and September 2012, when the PhD dissertation of the first author is expected to be delivered.

Phase 1 (6 months): Aims at obtaining theoretical and practical preparation. The main activity here will be an intensive literature review. The main areas of research will be (i) refactoring code-smells, (ii) catalogues, (iii) quantitative based refactoring, (iv) experimental software engineering and related areas like statistics.

Phase 2 (18 months): The research contributions will be implemented and validated (i) QBR (2 months); (ii) Code smells detection metrics catalogue (7 months); (iii) Amended refactoring catalogue with evidence of refactoring effects on software quality characteristics (9 months).

Phase 3 (3 months): Production, submission, review and presentation of papers with intermediate results, to obtain validation feedback from international research peers.

Phase 4 (9 months): Writing and reviewing of the PhD dissertation chapters.

VI. Conclusions

The refactoring process, in spite of being empirically solid, reveals some fragilities, like those discussed herein (based upon an example), that hampers its widespread adoption.

The outcome of this research expects to strengthen the refactoring process, by contributing to the mitigation of the discussed fragilities, by means of quantitative and experimental arguments.

By providing the grounds to make refactoring a more efficient process and granting quantitative evidence on its effects on product quality characteristics, we believe to be delivering a valuable contribution towards software evolution.

References

[1] Martin Fowler, Kent Beck, John Brant, and William Opdyke, Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999

[2] Konstantinos Stroggylos and Diomidis Spinellis. Refactoring: Does it improve software quality?. In B. Boehm, S. Chulani, J. Verner, and B. Wong, editors, 5th International Workshop on Software Quality. ACM Press, May 2007.

[3] Marvin V. Zelkowitz, Dolores R. Wallace: Experimental validation in software engineering. Information & Software Technology 39(11): 735-743 (1997)

[4] W. Tichy, N. Harbermann, and L. Prechelt. "Future directions in software engineering". ACM SIGSOFT, Software Engineering Notes, 18(1):35–48, 1993.

[5] Joshua Kerievsky, "Refactoring to Patterns", Addison-Wesley, 2004

[6] Ramnivas Laddad, "Aspect-Oriented Refactoring", Addison-Wesley, 2006

[7] M.P. Monteiro, J.M. Fernandes, Towards a Catalogue of Refactorings and Code Smells for AspectJ. Transactions on Aspect-Oriented Software Development (TAOSD), A. Rashid, M. Aksit (Eds.), Springer LNCS vol. 3880/2006, p. 214 – 258. ISSN: 0302-9743. DOI: 10.1007/11687061

[8] Tom Mens, Tom Tourwé, "A survey of software refactoring", IEEE Transactions on Software Engineering Volume 30, Issue 2, Feb 2004 Page(s): 126 – 139, 2004

[9] Frank Simon , Frank Steinbrückner , Claus Lewerentz, Metrics Based Refactoring, Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, p.30, March 14-16, 2001

[10] Naji Habra, Miguel Lopez, "On the use of measurement on Software Restructuring", in Proceedings of the International ERCIM Workshop on Software Evolution, 2006

[11] Ladan Tahvildari and Kostas Kontogiannis, in Proceedings the 7th European Conference on Software Maintenance and Reengineering (CSMR'03), 2003

[12] Ladan Tahvildari, "Quality-Driven Object-Oriented Reengineering Framework, Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04), 2004

[13] Mazeiar Salehie, Shimin Li, Ladan Tahvildari, "A Metric-Based Heuristic Framework to Detect Object-Oriented Design Flaws, in Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06), 2006

[14] F. Sauer. "Metrics." Internet: metrics.sourceforge.net, Jul. 8, 2005 [accessed on May 2, 2009].

[15] Sérgio Bryton and F. Brito e Abreu, "Modularity-Oriented Refactoring", in Proceedings the 12th European Conference on Software Maintenance and Reengineering (CSMR'08), 2008

[16] Sérgio Bryton and F. B. e Abreu, "Removing Subjectivity from Bad Smell Detection", unpublished, 2009.

[17] Henderson-Sellers, B. "Object-Oriented Metrics, measures of Complexity", Prentice Hall, 1996

[18] Sérgio Bryton. "Modularity Improvements with Aspect-Oriented Programming". MSc Dissertation, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2008. Internet: http://www-ctp.di.fct.unl.pt/QUASAR/Resources/publications.htm

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration<Rental> rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
                double thisAmount = 0;
                Rental each = (Rental) rentals.nextElement();
                //determine amounts for each line
                switch (each.getMovie().getPriceCode()) {
                case Movie.REGULAR:
                        thisAmount += 2;
                        if (each.getDaysRented() > 2)
                                thisAmount += (each.getDaysRented() - 2) * 1.5;
                        break;
                case Movie.NEW_RELEASE:
                        thisAmount += each.getDaysRented() * 3;
                        break;
                case Movie.CHILDRENS:
                        thisAmount += 1.5;
                        if (each.getDaysRented() > 3)
                                thisAmount += (each.getDaysRented() - 3) * 1.5;
                        break;
                }
                // add frequent renter points
                frequentRenterPoints ++;
                // add bonus for a two day new release rental
                if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
                                &&
                                each.getDaysRented() > 1) frequentRenterPoints ++;
                //show figures for this rental
                result += "\t" + each.getMovie().getTitle()+ "\t" +
                String.valueOf(thisAmount) + "\n";
                totalAmount += thisAmount;
        }
        //add footer lines
        result += "Amount owed is " + String.valueOf(totalAmount) +
        "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints)
        +
        " frequent renter points";
        return result;
}
```

Listing 1: Initial version of the statement() method from the Customer class

```
public String statement() {
        Enumeration<Rental> rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";
        while (rentals.hasMoreElements()) {
                Rental each = (Rental) rentals.nextElement();
                //show figures for this rental
                result += "\t" + each.getMovie().getTitle()+ "\t" +
                String.valueOf(each.getCharge()) + "\n";
        }
        //add footer lines
        result += "Amount owed is " +
        String.valueOf(getTotalCharge()) + "\n";
        result += "You earned " +
        String.valueOf(getTotalFrequentRenterPoints()) +
        " frequent renter points";
        return result;
}
```

Listing 2: Final version of the statement() method from the Customer class