Volume 8, No. 3, March – April 2017
ISSN No. 0976-5697
International Journal of Advanced Research in Computer Science
RESEARCH PAPER
Available Online at www.ijarcs.info

Understanding the compliance of Refactoring with Maintenance

Sandeep Bal
Research Scholar, IKG-PTU,
Jalandhar, Punjab, India

Sumesh Sood
Asst. Prof.,Department of Computer Applications,
IKG-PTU Campus, Dinanagar, Punjab, India

Abstract: It is a belief that it is harder to maintain a code than writing a new code. The software maintenance comes into picture after the code has been delivered. The purpose of maintenance is to remove bugs, improve performance and design and adding new features to the existing system. The process of Refactoring helps in improving software's internal structure without altering its external behaviour. It is also a part of Maintenance activity. Another belief is that Refactoring improves software design by making it more extensible and flexible. This paper presents a study of the case studies already performed in order to show the compliance of Refactoring activity with the Maintenance Phase.

Keywords: Software Maintenance, Refactoring, Quantification.

1. INTRODUCTION

There are many direct and indirect quality attributes of reusability like adaptability, completeness, maintainability, and understandability [1]. Refactoring helps in enhancing particularly maintainability and understandability [2], [3].
The idea is to analyze whether refactoring enhances reusability by analyzing its impact on some of the internal reusability metrics proposed and validated in the work of Dandashi et al. [1]. A restricted set of internal product attributes are used in order to assess reusability [4]. When you talk about the Refactoring Process, there are three important roles: the developer, the analyst, and the manager.
The developer analyzes the program source code to find which parts to be refactored in order to start the refactoring process. The analyst then examines and organizes these refactoring candidates in terms of the cost and effect. The developer is also responsible for the application of program refactorings, and validation of the functional equivalence before and after refactoring. Those refactorings are given by the analyst after he/she validates the cost and the effect of the refactorings.
After the developer has identified refactoring candidates, it is the analyst who selects appropriate refactorings. The analyst is also responsible for the cost and effect estimation. The estimation result is to be given to the manager who is responsible for the quality of the final product. Given the global strategy of the manager, the analyst deploys the strategy into program refactorings. Those refactorings are validated against the estimated cost
and effect, and then passed to the developer who is responsible for the implementation. Given the selected refactorings, the project manager must make strategic decisions taking the cost–effect aspect into consideration. At this stage, cost and effect estimations are two major key activities to make the decision. There are several traditional and effective quantitative methods to estimate the cost for software development including program modification. On the other hand, the effect estimation requires high–level knowledge of the program in question and its structure, and there are few methods to quantify the effect. It is very important to provide an appropriate strategy of refactoring because it will affect both the software development process and software itself. Proper cost–effect trade–off estimation and considerable prioritization are essential in this phase.
In order to perform best suited refactoring following are the three phases of Program refactoring Process:
1. Identification of refactoring candidates
2. Validation of refactoring effect
3. Application of refactoring

A quantitative evaluation method has been proposed [5] for measuring the maintainability enhancement effect of program refactoring. The comparison between the coupling before and after the refactoring of a program has been performed for evaluating the degree of maintainability enhancement.

2. EVALUATING THE IMPACT OF REFACTORING ON REUSABILITY

The approach proposed by Dandashi and Rine [1] is followed in [4] where two different sets of metrics are used:
• One for micro level measurements (measurements at a method level).
• And one for macro level measurements (measurements at a class level).

McCabe's cyclomatic complexity of a method [6] and the number of Java statements per method are used for the micro level measurements whereas the Chidamber and Kemerer (CK) set of object-oriented metrics [7] are used for the macro level measurements. To arrive at a measure for the whole class, the highest measure is used as a representative measure of the corresponding class measure.
The motivation for choosing this set of metrics is twofold: First, some of them such as the CK metrics are among the best-understood and validated metrics for object oriented systems and therefore we can be more confident in their expressiveness [8].Second, the tool we use for collecting these metrics is able to collect them in an automatic and non-invasive way - a fundamental requirement for data collection in an XP process [9]. Several empirical studies put the CK metrics into relationship with software quality and reusability. Li and Henry [10] for example show that the CK metrics are useful to predict maintainability. Basili et al.

[8] investigate the relationship between the CK metrics and code quality: Their findings suggest that 5 of the 6 CK metrics are useful quality indicators. However, such studies are rare in XP-like environments and they do not analyze how the evolution of the CK metrics during development is affected by refactoring. Table 1 summarizes the metrics used in this research as indicators for reusability.

Table 1: Internal Product Metrics

Metric Name	Level	Definition
MAX_LOC	Class	Maximum no. of Java statements of all methods in a class
MAX_MCC	Class	Highest McCabe's cyclomatic complexity of all methods in a class
CBO	Class	Coupling Between Object classes(CK)
LCOM	Class	Lack of Cohesion in Methods(CK)
WMC	Class	Weighted Methods per Class (CK)
RFC	Class	Response of a Class(CK)
DIT	Class	Depth of Inheritance Tree
NOC	Class	Number of Children

The approach analyzes the changes in the metrics mentioned in the table during development. A set of candidate classes which are considered for reuse are chosen at the first place. Afterwards, the daily changes in the reusability metrics for each class during development are noticed. The average of these daily changes with the change each class gains after refactoring are mentioned at the final step.
This approach helps in quantifying the impact of refactoring on reusability metrics compared to their overall evolution during development.

A bit more formally the method can be defined as follows.
Let Mi ∈ M={MAX_MCC, MAX_LOC, CBO, RFC, WMC, DIT, NOC, LCOM} be one of the reusability metrics listed in Table 1. The daily changes are averaged as the first step for each candidate class over the whole development period without including days when the class has been refactored. It is denoted as ΔM_i. N in equation below is the total number of development days; Δt is a time interval of 1 day and R is the set of all days during which developers have refactored a particular class.

$$\Delta M_i = \frac{\sum_{\substack{k=1 \\ k \notin R}}^{N} M_i(k \cdot \Delta t) - M_i((k-1) \cdot \Delta t)}{N}$$

ΔR_i and ΔM_i values are computed and compared in order to assess the improvement scope of reusability of a class while refactoring it. Here ΔR_i denotes the average of the daily changes of reusability metric M_i only for the days ($k \in R$) in which a class has been refactored. Either the negative value of ΔR_i or lower value than ΔM_i suggests that refactoring improves reusability metric M_i compared to its standard evolution during development.

The Case Study: This case study helps to analyze the promotion of ad-hoc reuse by refactoring in a software project developed using an agile, XP-like methodology [11].
To collect the metrics listed in Table 1, PROM [12] has been used which extracts a variety of standard and user defined source code metrics from a CVS repository. A checkout of the CVS repository is performed on daily basis where the values of the CK and complexity metrics are computed and stored in a relational database. It includes the daily evolution of the CK metrics, LOC and McCabe's cyclomatic complexity.
This study has been conducted on a commercial software project developed in Java at VTT in Oulu, Finland. The project delivers the required product, a production monitoring application for mobile, Java enabled devices on time and on budget. The development process followed a tailored version of the Extreme Programming practices [11]. The software consists of 30 Java classes and a total of 1770 Java source code statements (denoted as LOC).
The design of the developed system is based on the MVC pattern [13], the Broker architectural pattern and several standard design patterns described in [14].
Total five candidate classes denoted as A, B, C, D and E has been selected and computed in a first step the daily changes of the metrics for each of them omitting the days when they have been refactored. The average of these changes for all days in which a class has been refactored has been computed afterwards. Table 2(a) and 2(b) shows the results after calculating the following for each metric and candidate class: the average changes during development (without refactoring), ΔM_i, the average changes induced by refactoring.

Table 2**:** Average daily changes of reusability metrics in case of refactoring (ΔR) and development (ΔM).

Class	CBO		RFC		WMC		LCOM	
	ΔM	ΔR	ΔM	ΔR	ΔM	ΔR	ΔM	ΔR
A	0	0	0	1	1	-1	0	-1
B	1	-4	1	-4	0	0	0	0
C	1	0	2	-5	4	0	1	0
D	1	-1	1.4	-2	2	0	1	0
E	1	-1	3.5	-2	2	3	0	0

Class	MAX_MCC		MAX_LOC		DIT		NOC	
	ΔM	ΔR	ΔM	ΔR	ΔM	ΔR	ΔM	ΔR
A	0	-1	0	0	0	0	0	0
B	1	0	2	-2	0	0	0	0
C	3	0	6	-46	0	0	0	0
D	3	-2	0	0	0	0	0	0
E	1	0	10	-20	0	0	0	0

The interpretation of the numbers in Table 2 is straightforward:
1. A minimum of two reusability metrics for every candidate class can be seen improving significantly after it has been refactored (compared to the average evolution

during development). E.g. Class A and E both provide general interfaces to the user interface and database and they might/can be reused in a similar application.
2. On the other hand, the metrics related to inheritance and cohesion are not at all or only in a negligible way changed by the refactorings applied in the project because of not using deep inheritance hierarchies. Therefore, it is quite obvious that no refactoring dealing with inheritance has been applied (it was not necessary to restructure code due to complexity caused by inheritance).
3. The CBO and RFC metrics show the highest benefit of refactoring as they express the coupling between different classes and the complexity of a class in terms of method definitions and method invocations. It is believed that these two metrics are strong indicators for how difficult it is to reuse a class: A high value of RFC makes it difficult to understand what the class is doing and a high value of CBO means that the class is dependent on many external classes and difficult to reuse in isolation. Both situations prevent it from being easily reused. For three out of the five candidate classes refactoring improves significantly both the RFC and CBO values and as such clearly makes them more suitable for ad-hoc reuse.
4. Refactoring seems also to lower method complexity: In all the classes either the method with the maximum lines of code or the one with the highest cyclomatic complexity have gained a notably improvement after refactoring. Again, classes with less complex methods are easier to reuse.

3. A QUANTITATIVE EVALUATION METHOD FOR MEASURING THE MAINTAINABILITY ENHANCEMENT:

The refactoring process consists of three major subprocesses, which are the identification of refactoring candidates, the validation of refactoring effect, and the application of refactoring [5]. Following are some key aspects of these subprocesses:

1. Improvement Planning
It includes organization of refactorings and selection of refactorings to be applied after identifying the program points which are to be refactored i.e. to identify refactoring candidates.

2. Improvement Validation
It consists of three different validations with their own objective such as the verification of the functional equivalence before and after the refactoring at the developer level, the validation of the intended effect at the analyst level and the cost–effect trade–off at the manager level.

3. Improvement Execution
It includes the ordering of each refactoring according to the priority in terms of cost–effect trade–off by the analyst and the actual code modification by the developer. It is basically the implementation of refactoring to the target program.

Table 3: Status of Refactoring Process Support

Subprocess	Activity	Support
Planning	Bad-smell detection	Partly
	Bad-smell analysis	Partly
	Refactoring planning	Partly
Validation	Plan evaluation	No
	Refactoring validation	No
	Functional equivalence validation	No
Execution	Refactoring deployment	No
	Refactoring application	Partly

Some effective approach is required to support the Validation subprocess as it is visible in Table 3 also. The support in Validation subprocess will help the project managers for developing the software in enhancing their projects' maintainability efficiently.
The evaluation of refactoring effect to support the plan evaluation and refactoring validation (Validation Subprocess in Table 3) can be seen as follows:

Evaluate Refactoring Effect

3.1 Definition of Refactoring Effect

Following steps are taken in order to evaluate the refactoring effect:
1. Selection of an appropriate maintainability quantification metrics.
2. Measurement and comparison of metrics before and after refactoring.

There have been many distinguished previous works on the software maintainability quantification [8][15][16][17]. The maintainability of each part of a program is desired for refactoring purpose rather than the whole program. Following are the few aspects of maintainability of a program.
Coupling: Generally speaking, decreasing the coupling among modules enhance the system maintainability.
Cohesion: Modules of high cohesion are easier to maintain than those of low cohesion.
Size and Complexity: Simple and small modules are easy to maintain.
Description: Appropriate naming rule helps us to understand the program.

The focus here is on refactoring method that enhances the maintainability of a method in terms of coupling i.e. reduction of coupling among methods such as "Extract Method," "Extract Class," or "Move Method." The coupling metrics have been measured before and after those refactorings and the metrics values are compared to quantify those refactoring effects in terms of maintainability enhancement. Java and a method have been used as examples of a programming language and a unit of a part of a program respectively.

3.2 Definition of Coupling

The coupling between methods definition has been especially customized for object-oriented programming languages like Java where inter–class coupling coefficients and coupling type coefficients have been introduced and the coupling can be classified into the following three major categories:

3.2.1 Return value coupling

When method A uses a return value from method B, it is said that A has a return value coupling with B or else, when method A provides a return value to method B, it again is said that A has a return value coupling with B. The whole return value coupling of method A can be defined as a total of those related values. Hence the return value coupling of A, or $C_{rv}(A)$ has been defined as follows:

$$C_{rv}(A) = \sum_{m_\rho \in \rho(A)} K_{rv}(m_\rho) + \sum_{m_\sigma \in \sigma(A)} K_{rv}(m_\sigma),$$

where,

ρ(A): a set of methods whose return value is used by A,

$K_{rv}(m)$: return value coupling inter–class coefficient,

=1 when m is in the same class as A,

$=\kappa_{rv}>1$when m is in different class,

σ(A): a set of methods that use A's return value.

$K_{rv}(m)$ has been introduced to reflect that an inter–class coupling could be a greater maintenance obstacle more than an intra–class coupling. Hence $\kappa_{rv}(>1)$ coefficient is applied to inter–class parameter couplings.

3.2.2 Parameter coupling

When method A receives n parameters from method B, it is defined that A has n parameter coupling with B or else, when method A passes m parameters to method C, it can be defined that A has m parameter coupling with C. The whole parameter coupling of method A can be defined as a total of those related values. Hence parameter coupling of A, or $C_{pp}(A)$can be defined as follows:

$$C_{pp}(A) = \sum_{m_\zeta \in \zeta(A)} K_{pp}(m_\zeta) p_{m_\zeta} + \sum_{m \in \xi(A)} K_{pp}(m_\xi) p_{m_\xi},$$

where,

ζ(A): set of methods that invoke A,

ξ(A): set of methods that are invoked by A,

Kpp(m): parameter coupling inter–class coefficient,

= 1 when m is in the same class as A,

= κpp > 1 when m is in a different class,

pm: # of parameters of m.

Kpp(m) has been introduced for the same reason as Krv(m) has been introduced.

3.2.3 Shared variable coupling

When method A uses n class/instance variables in common with another method B, it is said that A has n shared variable coupling with B.

The whole shared variable coupling of method A can be defined as a total of those related values. Hence we define shared variable coupling of A, or $C_{sv}(A)$ as follows:

$$C_{sv}(A) = \sum_{m_\chi \in \chi(A)} K_{sv}(m_\chi) v_{m_\chi},$$

where,

χ(A): a set of methods that use class/instance variables in common with A,

$K_{sv}(m)$: shared variable coupling inter–class coefficient,

= 1 when m is in the same class as A,

$= \kappa_{sv} > 1$ when m is in a different class.

vm: number of class/instance variables appearing both method A and method m in common.

$K_{sv}(m)$ has been introduced for the same reason as $K_{rv}(m)$ has been introduced.

3.3 Combining Three Couplings

Three coefficients K_{Trv} , K_{Tpp}, and K_{Tsv} have been prepared for the parameter coupling, the shared variable coupling, and the return value coupling, respectively. It has been done for combining these three coupling metrics into one in order to evaluate the maintainability of a certain method. The following inequality relationship has been assumed among them:

$0<K_{Trv}\leq K_{Tpp} < K_{Tsv}$ $(K_{Trv} + K_{Tpp} + K_{Tsv} = 1)$

The whole coupling metrics value for the method A C_T (A) is then calculated in the following formula:

$C_T(A) = K_{Trv}C_{rv}(A) + K_{Tpp}C_{pp}(A) + K_{Tsv}C_{sv}(A)$

The Experiment

An old maintained software project written in C++ has been chosen to evaluate the theory. The maintainability of the software has been much deteriorated. Many bad–smells in terms of coupling metrics were found by applying Refactoring Assistant to the software. An interview was also conducted for the developers of the program to recognize the most serious problems in terms of maintainability. Five problems were identified afterwards and then refactoring has been applied to each of these five problems.

3.3.1 Case 1: Extract Method (1)

In this case, the original method indicated very high coupling with many other methods including the ones in other classes. The developer admitted that the method should be divided into at least two parts: one for manipulating instance variables to update the object's state and the other for collecting required information from other objects. Therefore the developer applied Extract Method refactoring.

3.3.2 Case 2: Extract Method (2)

This case also showed rather high inter–class parameter coupling and intra–class shared variable coupling. The solution has been applied anyway only to find that the refactored code was no more maintainable enough than the original one.

3.3.3 Case 3: Extract Class (1)

In this case, the original method indicated very high shared variable coupling with many other methods in the same class. The method dealt with four instance variables all of which performed very similar roles to one another. Actually

those four instance variables all together prescribe a certain state of the object. Hence the developer decided to introduce a new class to manage those four instance variables together with appropriate methods to manipulate them.

3.3.4 Case 4: Extract Method (3)

The method in question is a check method which traverses two instance variables. Those two instance variables are very similar to each other and so are the ways to traverse them. The developer therefore extracted a traverse method. The extraction separated the original method into two parts: one for the query part and the other for the traversing

.

3.3.5 Case 5: Extract Class (2)

The original method accessed twelve instance variables. The developer had realized that some encapsulation should be applied to those instance variables which access a number of other methods not only in the same class but in many different classes. A new class has also been introduced that deals with those instance variables and provides methods to access/update those values. As a result, the extracted class became responsible for the shared variable coupling and the method in question became almost free from the shared variable coupling.

Table 4: Refactoring Experiment Results

Case	Sbj.	Before	After(Effect)	Average(Effect)
1	B	10.4	2.8(7.6)	3.6(6.8)
2	C	12.1	9.0(3.1)	8.3(3.8)
3	B	25.2	9.0(16.2)	9.0(16.2)
4	B	26.4	1.7(24.7)	13.8(12.6)
5	A	126.0	26.0(100.0)	28.3(97.7)

Following coefficient value for the target system were chosen:

$$\kappa_{rv} = 1.5, \kappa_{pp} = 2.0, \kappa_{sv} = 3.0$$
$$K_{Trv} = 0.2, K_{Tpp} = 0.2, K_{Tsv} = 0.6$$

The result of the experiment can be seen in the Table 4. The subjective evaluations of the refactoring effectiveness by the developer is shown in column “Sbj”. The corresponding values are out of A, B, C which means:

A: considerably effective
B: somehow effective
C: not necessarily effective.

The coupling metrics value of the target method before and after the refactoring can be seen in the column “Before” and “After (effect)” respectively.

4. CONCLUSION

It can be concluded from the case study of internal metrics that Refactoring improves significantly important internal measures for reusability of object-oriented classes and it has a positive effect on reusability.

The second research method evaluated the refactoring effect in terms of the degree of maintainability enhancement of a target program.

5. REFERENCES

[1] Dandashi, F. (2002, March). A method for assessing the reusability of object-oriented code using a validated set of automated measurements. In Proceedings of the 2002 ACM symposium on Applied computing (pp. 997-1003). ACM.

[2] Du Bois, B., Demeyer, S., & Verelst, J. (2004, November). Refactoring-improving coupling and cohesion of existing code. In Reverse Engineering, 2004. Proceedings. 11th Working Conference on (pp. 144-151). IEEE.

[3] Ratzinger, J., Fischer, M., & Gall, H. (2005). Improving evolvability through refactoring (Vol. 30, No. 4, pp. 1-5). ACM.

[4] Moser, R., Sillitti, A., Abrahamsson, P., & Succi, G. (2006). Does refactoring improve reusability?. In Reuse of Off-the-Shelf Components (pp. 287-297). Springer Berlin Heidelberg.

[5] Kataoka, Y., Imai, T., Andou, H., & Fukaya, T. (2002). A quantitative evaluation of maintainability enhancement by refactoring. In Software Maintenance, 2002. Proceedings. International Conference on (pp. 576-585). IEEE.

[6] McCabe, T. J. (1976). A complexity measure. Software Engineering, IEEE Transactions on, (4), 308-320.

[7] Chidamber, S. R., & Kemerer, C. F. (1994). A metrics suite for object oriented design. Software Engineering, IEEE Transactions on, 20(6), 476-493.

[8] Basili, V. R., Briand, L. C., & Melo, W. L. (1996). A validation of object-oriented design metrics as quality indicators. Software Engineering, IEEE Transactions on, 22(10), 751-761.

[9] Disney, A. M., & Johnson, P. M. (1998, November). Investigating data quality problems in the PSP. In ACM SIGSOFT Software Engineering Notes (Vol. 23, No. 6, pp. 143-152). ACM.

[10] Li, W., & Henry, S. (1993, May). Maintenance metrics for the object oriented paradigm. In Software Metrics Symposium, 1993. Proceedings., First International (pp. 52-60). IEEE.

[11] Abrahamsson, P., Hanhineva, A., Hulkko, H., Ihme, T., Jäälinoja, J., Korkala, M., ... & Salo, O. (2004, October). Mobile-D: an agile approach for mobile application development. In Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (pp. 174-175). ACM.

[12] Sillitti, A., Janes, A., Succi, G., & Vernazza, T. (2003, September). Collecting, integrating and analyzing software metrics and personal software process data. In Euromicro Conference, 2003. Proceedings. 29th (pp. 336-342). IEEE.

[13] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (1996). {Pattern-oriented Software Architecture Volume 1}.

[14] Vlissides, J., Helm, R., Johnson, R., & Gamma, E. (1995). Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley, 49(120), 11.

[15] Bril, R. J., & Postma, A. (2001). An architectural connectivity metric and its support for incremental re-architecting of large legacy systems. In Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on (pp. 269-280). IEEE.

[16] Bengtsson, P. (1998, August). Towards maintainability metrics on software architecture: An adaptation of object-oriented metrics. In First Nordic Workshop on Software Architecture, Ronneby.

[17] Wake, S., & Henry, S. (1988, October). A model based on software quality factors which predicts maintainability. In Software Maintenance, 1988., Proceedings of the Conference on (pp. 382-387). IEEE.