Opportunities and challenges in deriving metric impacts from refactoring postconditions.

Bart Du Bois
Lab On ReEngineering
Universiteit Antwerpen
Middelheimlaan 1, B-2020 Antwerpen, Belgium
bart.dubois@ua.ac.be

Abstract

Refactoring – transforming the source-code of an object-oriented program without changing its external observable behaviour – is a restructuring process aimed at resolving evolution obstacles. Currently however, the efficiency of the refactor process in terms of quality improvements remains unclear. Such quality improvement can be expressed in terms of an impact on OO metrics. The formalization of these metrics is based on the same constructs as refactoring postconditions. Therefore, in this position paper, we elaborate on a research approach to derive Object-Oriented metric impacts from refactoring postconditions, in order to provide qualitative guidelines on the application of specific refactorings to resolve quality deficiencies.

1 Introduction

Current knowledge about the refactoring process can be divided into *analysis* and *resolution*. The building blocks of resolution by refactoring consists of a set of refactoring operations (Fowler, 1999) with pre- and postconditions (Opdyke, 1992; Roberts, 1999; Tichelaar, 2001; Ó Cinnéide, 2001). These conditions express properties of the object model either before or after the refactoring has been applied. Research on the use of these building blocks has been targeted mostly towards the introduction of design patterns (Tokuda and Batory, 2001; Ó Cinnéide, 2001; Kerievesky, 2004). Contributions on analysis of software systems in the context of the refactoring process include descriptions of anti-patterns characterising potential refactoring opportunities (Brown et al., 1998) and metric descriptions of design flaws (Marinescu, 2001; Muraki and Saeki, 2002; Ratiu et al., 2004).

Little research has been performed on binding the analysis and resolution results together. (Sahraoui et al., 2000) analyzed the impact on inheritance and coupling metrics of composite transformations. (Tahvildari et al., 2003) evaluated design patterns from the perspective of non-functional requirements and modeled this in a soft-goal interdependency graph for maintainability. While both works provide practical guidelines on how to improve certain quality indicators, the level of information feedback is very coarse grained.

We propose to evaluate the contribution of smaller refactorings on model properties that can be composed in Object Oriented metrics. Our main hypothesis is that refactorings can be evaluated by their impact on those internal software properties which are highly related with external properties. More specific, we argue that the application of a refactoring should be based on it's improvement regarding specific aspects of complexity, coupling and cohesion.

Therefore, we need a single language in which to specify both internal software metrics and refactorings in order to clarify how the application of the latter can improve the value of the former.

This paper is organized as follows. Section 2.1 provides an overview of the information available in refactoring postconditions. Section 2.2 elaborates on the composition of this information into Object Oriented metrics. Section 2.3 describes how the impact of a refactoring on model properties can be derived (2.3.1), and how these impacts can be combined in an impact on Object-Oriented metrics (2.3.2).

2 Research approach

We elaborate on the derivation of metric impacts from refactoring postconditions by first describing the results of past research on which our approach is based.

2.1 Refactoring postconditions

In describing postconditions of composite refactorings, postconditions of primitive restructurings are combined using techniques such as chaining and set iteration (Ó Cinnéide, 2001). The postcondition of a primitive restructuring itself is described as a set of updates to *analysis functions*, and the postcondition is computed by concatenating these function updates.

Analysis functions are functions and predicates in the first-order predicate calculus on top of a metamodel that extracts information from an object model. They are used in the pre- and postcondition to express properties of the object model respectively before and after the refactoring has been applied. As we will discuss in section 2.2, these analysis functions form the main vocabulary of the language used to describe both refactorings and metrics.

Cinnéide worked out an example to illustrate the rules of chaining and set iteration, in which he composed pre- and post condition of the EncapsulateConstruction composite refactoring according to specific guidelines. The resulting post-condition has been included in Table 1. Composing such postconditions is non-trivial and requires a lot of effort, yet this only needs to be done once.

$\forall e : ObjectCreationExprn, classCreated(e) = product, containingClass(e) = creator \bullet \exists m : Method$ such that
$createsSameObject' = createsSameObject[(constructorInvoked(e), m)/true]$
$nameOf' = nameOf[m/createP]$
$defines' = defines[(creator, m)/true]$
$\forall e : ObjectCreationExprn, classCreated(e) = product, containingClass(e) = creator,$
$nameOf(containingMethod(e)) \neq createP\bullet$
$containingMethod' = containingMethod[e/m]$

Table 1. Postcondition of the EncapsulateConstruction(Class creator, Class product, String createP) refactoring. function[a/b] can be read as function(a) = b. The distinction between the state of an analysis function before and after a refactoring is indicated with an accent.

While these composition guidelines pave the way for formalizing other composite refactorings, we strongly believe that doing so for even the most widely known refactorings described by (Fowler, 1999) will not be limited to chaining the already formalized primitive refactorings, but will also include formalizing additional primitive refactorings to fill in the gaps.

Therefore, one major challenge is to identify a set of primitive refactorings that minimizes the need for *patching* the postconditions of composite refactorings with other information than the chaining or set iteration of the postconditions of primitive refactorings. Possible pathways to identify such a set are analysing the informal descriptions provided by Fowler, or alternatively, applying reverse engineering techniques on existing refactoring tools, in order to detect similarities or duplicates between the implementations of multiple refactorings.

Secondly, for each written postcondition, it is possible to write an equivalent one. Much in the same way as mathematical equations can be simplified, postconditions also can be rewritten to make it easier to interpret specific aspects. Therefore, another challenge lies in finding criteria on which to evaluate the suitability of refactoring postconditions to derive useful information from the context of Object-Oriented metrics. Further issues regarding this challenge will be discussed in section 2.3.1.

2.2 Composing metrics

The concept of analysis functions is introduced in other works with synonims like auxilary definitions (Muraki and Saeki, 2002) or library functions (Baroni, 2002). The concepts used in these works are very similar, and can be summarized as consisting of set operators ($\in$, $\bigcup$, $\bigcap$, /, #), logical operators ($\forall$, $\exists$, $\neg$, $\vee$, $\wedge$) and some variant of set notation ($\{x \in A | P(x)\}$ to denote the set of elements X in A satisfying $P(x)$). Equivalent notations are used in other measurement formalizations such as (Abreu and Carapuca, 1994; Moore, 1996; Briand et al., 1999; Reißing, 2001).

Evidently, the metamodel used in these analysis functions is crucial for their expressiveness. Ultimately, the analysis functions dictate which information can be used in order to compose higher level metrics. For example, if the metamodel used does not hold any information about conditionals, it is impossible to compose metrics such as Cyclomatic Complexity on top of the model. In this context however, we are only interested in those metrics which are affected by refactorings, and therefore, are constructed upon the same metamodel elements as the refactoring postconditions. Therefore, we argue that the choice of metamodel should depend on the ease and completeness by which refactoring postconditions can be expressed using its model elements.

Analysis functions are used to extract information about a software system, f.e. $classOf(Constructor/Method/Field\ a)$, which returns the class to which the given constructor/method/field belongs. Shorthand notations for such predicates make it easier to read expressions, as for example $a \in c$ instead of $classOf(a) = c$. Based on these analysis functions and a set of mathematical operators (+,-,*,/, $\sum$ etc.), Object-Oriented metrics can be defined as higher order functions. As an example, Table 2 depicts the formalization of the Information Flow based Coupling as defined by (Y. S. Lee et al., 1995) and formalized in (Briand et al., 1999).

$ICP^c(m) = \sum_{m' \in PIM(m)/\left(M_{NEW}(c) \bigcup M_{OVR}(c)\right)} \left(1 + \#Par(m')\right) * NPI(m, m')$
$ICP(c) = \sum_{m \in M_I(c)} ICP^c(m)$

Table 2. Composition of the Information Flow based Coupling (IFC) metric for a class from the lower level IFC of a method, where the latter is composed of analysis functions.

Not only does such a formalization unambiguously define the calculation of the metric value, it also indirectly identifies the model elements on which the calculation of the metric value is dependent. In the case of the ICP metric, the related model elements are methods with their parameters, and polymorphic method invocations between them.

This formal specification of the model elements which the metric uses form the targets towards which we will derive refactoring postconditions. This is the key to the research approach described in this position paper.

2.3 Deriving impacts

The impact of a refactoring on a metric is dependent on the scope of calculation of the metric value. For example, the IFC metric (Table 2) can be calculated for the class on which EncapsulateConstruction is applied (Table 1), or on the clients of this class. While the derivation of the particular impact is the same for all scopes, its result is not. The difference lies in the focus on the specific instances of model element associated with the scope. In the PullUpMethod refactoring for example, it is possible to distinguish between the subclass and the superclass roles, and each represents a particular scope. Therefore, the number of different scopes in a refactoring affect the complexity of deriving the impact on a metric.

As the analysis functions in which the IFC metric is expressed are different from those used in the EncapsulateConstruction refactoring postcondition, we have to map these equivalent functions in this illustration. For more information on the semantics of these analysis functions, we direct the reader to the origins of the metric formalization (Briand et al., 1999) and the postcondition specification (Ó Cinnéide, 2001).

Deriving the impact of a refactoring postcondition on a particular metric therefore consists of (a) translating the refactoring postcondition in terms of the analysis functions used in the metric formalization (2.3.1); and (b) composing the impacts on the analysis functions into an impact on the metric (2.3.2).

2.3.1 Translation into metric analysis functions

As not to overwhelm the reader with irrelevant details, we adhere to the guidelines of Cinnéide and specify the impact of a refactoring only on those analysis functions which are affected by the refactoring. Table 3 illustrates the translation of the postcondition of the EncapsulateConstructor refactoring into a specification that incorporates the specific analysis functions used in the formalization of the IFC metric.

The table formally specifies that all polymorphic method invocations to the constructor will be replaced with invocations to the creation method. The parameters of this factory method are equal to the parameters of the constructor.

To key to this translation is the specification of the relationships between the analysis functions used in the postcondition, and those used in the metric formalization.

$\forall meth : Method, PIM[(meth, constr)/true] \bullet PIM' = PIM[(meth, constr)/false], PIM[(meth, crMeth)/true]$
$M'_{NEW} = M_{NEW}[(creator, crMeth)/true]$
$Par' = Par[createMeth/Par(constr)]$
In which $crMeth : Method \wedge nameOf[crMeth/createP]$ and $constr : Constructor \wedge classOf(constr) = creator$

Table 3. Derivation of impact on analysis functions for IFC metric of EncapsulateConstruction(Class creator, Class product, String createP) refactoring.

To extend the postcondition with specific information (the analysis functions used in the metric formalization), we had to enrich the metamodel used by Cinnéide in order to express the changes on the parameter-level. This again points out the need for a metamodel which is sufficiently expressive for both refactoring postconditions and Object-Oriented metrics.

2.3.2 Composing the impacts

Once the impact of a refactoring on the analysis functions used in a metric formalization is expressed in terms of more specific postconditions (Table 3), we can combine these specific postconditions to derive the impact of the refactoring on the composite metric. We use the $\Delta metric$ notation to express the difference between the metric value after applying the refactoring and before applying the refactoring.

Table 4 demonstrates the composition of analysis function impacts into a metric impact on all classes but the creator class. The same derivation can be used for the scope of the creator class, with the same result (coincidentally).

$\forall c : Class, nameOf(c) \neq creator, \forall m \in M_I(c), PIM[(meth, constr)/true] \bullet$
$PIM'(m)/\Big(M'_{NEW}(c) \bigcup M'_{OVR}(c)\Big)$
$= \Big((PIM(m)/\{constr\}) \bigcup \{createP\}\Big)/\Big(M_{NEW}(c) \bigcup M_{OVR}(c)\Big)$
$\Rightarrow \Delta ICP^c(m) = -\Big(1 + \#Par'(constructor)\Big) * NPI(m, constructor) + \Big(1 + \#Par'(createP)\Big) * NPI(m, createP)$
And since $Par'(createP) = Par(constructor)$
$\Rightarrow \Delta ICP^c(m) = 0$
$\Rightarrow \Delta ICP(c) = 0$

Table 4. Composition of the impacts on the IFC metric of EncapsulateConstructor(Class creator, Class product, String createP) refactoring for all classes but the creator class.

The result of this derivation is that for all classes excluding the creator class the EncapsulateConstructor refactoring does not affect the Information Flow based Coupling (analogue derivation for the creator class leads to the same result). This corresponds to our intuition, as all information flow based coupling to the constructor is replaced with coupling to the create method.

More generally, this derivation approach will be able to derive the following conclusions about the impact of a refactoring on a metric:

- 0: the refactoring will never affect the metric value for the specific scope. I.o.w., $\Delta metric \in [0, 0]$
- +: the refactoring potentially increases the metric value for the specific scope. I.o.w., $\Delta metric \in [0, +\inf]$
- -: the refactoring potentially decreases the metric value for the specific scope. I.o.w., $\Delta metric \in [-inf, 0]$
- ?: in the general case, it is undecidable how the refactoring will impact the metric value for the specific scope. I.o.w., $\Delta metric \in [-inf, +inf]$. However, it is always possible to express a conditional impact

The reason that sometimes it is undecidable how the refactoring will impact the metric, is that there is unsufficient information to pin down in which direction the metric will be affected. This challenge can be tackled by introducing assumptions on the context in which the refactoring will be applied. Such assumptions can be specified under the form of conditional expressions. For example, when deriving the impact of the ExtractMethod(Set(Statements) stmts, Set(Method) fromMethods,

String newMethod) on the Weighted Methods per Class metric (WMC), the following conditional can formalize assumptions about the specific context: $if(\#fromMethods > 1)then\Delta(WMC) \in [-inf, 0]else\Delta(WMC) \in [0, +inf]$.

In other words, deriving the impact of a refactoring on a metric depends on (a) the scope in which the metric is to be calculated; and (b) sometimes also on the specific context in which the refactoring is applied. Currently, it is unclear to which extent these conditions will be necessary. Ideally, we would like to express the impact as generally as possible, that is, for as many contexts as possible.

However, conditional impact descriptions provide more detailed qualitative feedback on specific applications of the refactoring. Therefore, they specify in which situations to apply a refactoring in order to optimize its improvement regarding a specific quality attribute.

3 Related work

(Casais, 1992) formalized an algorithm to restructure class hierarchies in terms of changes to a formal object model. While Casais clearly described the specific object model change for each step of the algorithm, unlike in the work of (Ó Cinnéide, 2001), we were not able to find a postcondition for the complete algorithm. That is why we based our examples on Cinnéide's work.

(Sahraoui et al., 2000) uses Object-Oriented metric thresholds in rules for restructuring software systems. To prescribe refactorings to design flaws, they provide the impact of a number of transformations on these metrics. However, the paper provides no clues as to how this impact was calculated. Therefore, the main difference with their work is that we explicitly focus on an open and traceable derivation of these metric impacts. The derivation is open, as it allows to derive impacts of refactorings on other metrics by reusing derived impacts on lower level analysis functions. The derivation is traceable, as the cause and effect relationship between specific aspects of a refactoring postcondition and specific parts of a metric formalization are made explicit in our approach.

4 Conclusions and future work

In this position paper, we proposed to derive metric-specific refactoring postconditions in order to provide qualitative feedback on the application of refactorings. We presented both the opportunities and the challenges of our approach. Summarizing, while the major challenges of this approach are (a) to identify a sound set of primitive refactorings and rewriting postconditions; and (b) to extend them to incorporate specific information needed for metrics.

This research targets the impact of refactoring on internal quality attributes. In parallel, we experimentally test hypothesis regarding the impact of refactoring on external quality indicators in the context of maintainability (i.e. maintenance task duration and correctness). By linking the results of both branches together, in the future we will have better insights which specific internal quality indicators tend to strongly correlate with external quality indicators.

References

Abreu, F. B. and Carapuca, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. In *Proc. 4th Int'l Conf. Software Quality*.

Baroni, A. L. (2002). Formal definition of object-oriented design metrics. Master's thesis, Vrije Universiteit Brussel and Ecole des Mines de Nantes, Belgium.

Briand, L. C., Daly, J., and al. (1999). A unified framework for coupling measurement in object-oriented systems. *IEEE Trans. Software Engineering*, 25(1):91–121.

Brown, W. J., Malveau, R. C., McCormick, III, H. W., and Mowbray, T. J. (1998). *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc.

Casais, E. (1992). An incremental class reorganization approach. In Madsen, O. L., editor, *Proc. ECOOP '92*, volume 615 of *Lecture Notes in Computer Science*. Springer-Verlag.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley.

Kerievesky, J. (2004). *Refactoring To Patterns*. Addison-Wesley.

Marinescu, R. (2001). Detecting design flaws via metrics in object-oriented systems. In Li, Q., Meyer, B., Pour, G., and Riehle, R., editors, *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS 39)*, volume 1, pages 173–182. IEEE Computer Society.

Moore, I. (1996). Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings Int'l Conf. OOPSLA '96*, ACM SIGPLAN Notices, pages 235–250. ACM Press.

Muraki, T. and Saeki, M. (2002). Metrics for applying gof design patterns in refactoring processes. In *Proceedings of the 4th international workshop on Principles of software evolution*, pages 27–36. ACM Press.

Ó Cinnéide, M. (2001). *Automated Application of Design Patterns: a Refactoring Approach*. PhD thesis, University of Dublin, Trinity College.

Opdyke, W. (1992). *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign.

Ratiu, D., Ducasse, S., Gîrba, T., and Marinescu, R. (2004). Using history information to improve design flaws detection. In *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR 2004)*.

Reißing, R. (2001). Towards a model for object-oriented design measurement. In e Abreu, F. B., editor, *Proc. 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 71–84.

Roberts, D. (1999). *Practical Analysis for Refactoring*. PhD thesis, University of Illinois at Urbana-Champaign.

Sahraoui, H. A., Godin, R., and Miceli, T. (2000). Can metrics help to bridge the gap between the improvement of oo design quality and its automation? In *Proc. International Conference on Software Maintenance*, pages 154–162.

Tahvildari, L., Kontogiannis, K., and Mylopoulos, J. (2003). Quality-driven software re-engineering. *J. Syst. Softw.*, 66(3):225–239.

Tichelaar, S. (2001). *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern.

Tokuda, L. and Batory, D. (2001). Evolving object-oriented designs with refactorings. *Automated Software Engg.*, 8(1):89–120.

Y. S. Lee, B. L., Wu, S. F., and Wang, F. (1995). Measuring the coupling and cohesion of an object-oriented program based on information flow. In *Proc. Int'l Conf. on Software Quality*.