Describing the impact of refactoring on internal program quality

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

Tom Mens*
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
tom.mens@vub.ac.be

Abstract

The technique of refactoring – restructuring the source-code of an object-oriented program without changing its external behavior – has been embraced by many object-oriented software developers as a way to accommodate changing requirements. The overall goal of refactoring is to improve the maintainability of software. Unfortunately, it is unclear how specific quality factors are affected. Therefore, this paper proposes a formalism to describe the impact of a representative number of refactorings on an AST representation of the source code, extended with cross-references. We elicitate how internal program quality metrics can be formally defined on top of this program structure representation, and demonstrate how to project the impact of refactorings on these internal program quality metric values in the form of potential drifts or improvements.

1 Introduction

Refactorings are software transformations that restructure an object-oriented program while preserving its behavior [9, 16, 17]. The key idea is to redistribute attributes and methods across the class hierarchy in order to prepare the software for future extensions. If applied well, refactorings improve the design of software, make software easier to understand, help to find bugs, and help to program faster [9].

However, the impact of a particular refactoring on the software quality varies. Some refactorings raise the level of abstraction (at the expense of increasing the program complexity), others may reduce the complexity (at the expense of decreasing the performance, for example), etc. Therefore, our goal is to provide techniques and tools for software developers to help them maintain and improve program quality through refactorings. The use of metrics in achieving this goal is advocated in [6].

This paper takes a first step towards this goal, by proposing a formalism for describing the impact of refactorings on program structure. Our representation of the program structure is borrowed from [13], which uses an abstract syntax tree representation of the source-code, extended with cross-references to model type references, method calls, accesses, updates and inheritance links. This abstract syntax tree representation allows us to reason about program structure in terms of nodes interconnected with edges. The fact that dependencies between program entities are explicit in this representation makes it easier to reason about the impact of refactorings from a quality perspective.

Object-oriented program quality metrics are typically used as internal quality factors [3]. Defining these metrics in terms of the entities of the extended tree representation allows formal descriptions of structural changes on the tree (eg. refactorings) to be projected into impacts on the particular metrics. In this way, the integration of the formal description of refactorings and the formal definition of a representative set of object-oriented program quality metrics provides *a-priori* feedback on the impact of any application of a particular refactoring on any particular internal program quality metric.

The goal of this mechanism is to construct (once and only once) refactoring impact tables. Such information facilitates refactoring trade-offs, in that they make explicit which internal program quality metrics are affected when the refactoring

*Tom Mens is a Postdoctoral Fellow of the Fund for Scientific Research - Flanders (Belgium)

would be applied. In other words, the contribution of this work is to make explicit the *quality drift* caused by the application of refactorings.

This paper is structured as follows. Section 2 proposes the refactorings and case study we have selected for our experiments. Section 3 introduces our extended tree representation of the program structure. Section 4 shows how we can describe the impact of refactorings on the program structure. Section 5 uses this to analyse the impact of refactorings on object-oriented program quality metrics, and discusses the current limitations and their solutions. Section 6 discusses related work, and section 7 concludes.

2 Preliminaries

2.1 Selected refactorings

Fowler's catalogue [9] lists seventy-two object-oriented refactorings and since then many others have been discovered. To demonstrate the possibility of reasoning about refactoring in terms of their impact on program structure, we apply our formalism to a number of selected refactorings: ExtractMethod, EncapsulateField and PullUpMethod. These refactorings are quite typical for the categories of refactoring strategies they belong to - respectively Composing Methods, Organizing Data and Dealing with Generalization - which are among the most popular refactoring strategies in todays refactoring tools [IntelliJ IDEA, Eclipse, Together]. Hence they may serve as representatives for the complete set of primitive refactorings.

ExtractMethod extracts part of a method and factors it out into a new method.
EncapsulateField encapsulates public attributes by making them private and providing accessors. In other words, for each public attribute a method is introduced for accessing (getting) and updating (setting) its value, and all direct references to the attribute are replaced by calls to these methods.
PullUpMethod moves identical methods from subclasses into a common superclass.

2.2 Source code example

The example that we will use for our experiments consists of a simple Java package containing 4 classes: *Packet*, *Machine* and two subclasses *Workstation* and *PrintServer*. It is part of the implementation of a Local Area Network simulation (LAN). Although the code is in Java, other implementation languages could serve just as well, since we restrict ourselves to representing core object-oriented concepts only. For more information about this example, see [13].

```
01 public package LAN {
02  public class Machine {
03   public String name;
04   public Machine nextMachine;
05   public void accept(Packet p) {
06     System.out.println(name
07                   + " is accepting "
08                   + nextMachine.name);
09     this.send(p); }
10   protected void send(Packet p) {
11     System.out.println(name
12                   + " is sending "
13                   + nextMachine.name);
14     this.nextMachine.accept(p); }
15   }

16  public class Packet {
17   public String contents;
18   public Machine originator;
19   public Machine addressee;
20   }
```

```
21  public class PrintServer
22             extends Machine {
23   public void print(Packet p) {
24     System.out.println(p.contents); }
25   public void accept(Packet p) {
26     if(p.addressee == this)
27       this.print(p);
28     else super.accept(p); }
29   }

30  public class Workstation
31              extends Machine {
32   public void originate(Packet p) {
33     p.originator = this;
34     this.send(p); }
35   public void accept(Packet p) {
36     if(p.originator == this)
37       System.err.println("no dest");
38     else super.accept(p); }
39   }
40  }
```

We will describe the three refactor operations by example, and discuss their intuitive impact on quality.

ExtractMethod can be performed to extract the println-statement from methods $Machine.accept$ and $Machine.send$ to a separate method $log(String)$. To do this, we have to introduce a new method log in the $Machine$ class, with a single parameter. We can then move the println-statement from the $accept$ or $send$ method, and replace the String literal with a reference to the parameter. Thereafter, we have to replace the println-statement in both the $accept$ and $send$ methods with a method call to the log method, passing to it the appropriate String literal. Intuitively, performing this refactoring generalises the println-statement, which increases code reuse and reduces the impact of changes. When the output format needs changes, we will only need to change the log body, and not the $accept$ or $send$ methods.

EncapsulateField can be performed in order to shield the attribute $Machine.nextMachine$ from direct references. This causes the introduction of two methods in class $Machine$: a 'getter' which accesses the attribute and returns it to the caller and a 'setter' which takes the new value of the attribute as a parameter and updates the attribute. The attribute itself is made private. Intuitively, shielding the attribute from direct references reduces data coupling. This allows the attribute to change its data format without affecting clients using the attribute value.

PullUpMethod can be applied to $PrintServer.print(Packet)$, so that future subclasses of $Machine$ such as a FileServer class can reuse this code. This requires the introduction of a method in superclass $Machine$, to which the body of the $PrintServer.print$ method will be moved, and the removal of the former $PrintServer.print$ method. Intuitively, pulling up a method generalises specific behaviour, making it possible for subclasses to reuse and specialise the behaviour.

While the impact of these example applications of refactor operations is dependent on the situation in which they are applied, the following section will explain how we can set up a formalism to describe this impact in a generic way, for all situations.

3 Representing the program structure

The way in which we represent software is very straightforward: the source code is transformed into an abstract syntax tree representation extended with cross-references.

3.1 Abstract syntax tree representation

The program syntax tree directly reflects the natural containment relationship. A system contains packages. A package contains classes (or recursively another package). A class contains attributes and methods. A method contains expressions, local attributes and parameters.

All the nodes in the abstract syntax tree have a specific type: **S**(ystem), **PA**(ckage), **C**(lass), **M**(ethod), **A**(ttribute), (actual) **P**(arameter or return value), **L**(ocal variable), **E**(xpression).

Figures of the AST of the example have been omitted as the concept is well known in the context of software engineering.

3.2 Abstract Syntax Tree extensions

On top of this abstract syntax tree representation, we need to superimpose extensions to represent cross-reference relationships between software entities (such as class inheritance, method calls, attribute accesses and updates, type information). These are represented by *edges* between the corresponding tree nodes.

As for nodes, the superimposed edges have a specific type: **i**(nheritance), (method) **c**(call), (attribute or local variable) **a**(ccess), (attribute or local variable) **u**(pdate) and **t**(ype). The enrichment of the AST representation provided by these cross-references makes it easier to reason about code from the context of refactoring.

3.3 Program structure notation

We will now introduce a number of notations in terms of the program structure representation that are needed in the remainder of the paper to enable us to describe the impact of refactor operations.

Let r be an AST node, S_i a set, ν a node type, and ϵ a regular expression of edge types which adheres to the standard regular expression rules used in the popular UNIX grep-tool.

$\overline{S_i}$ denotes the complement of S_i.

$T(r)$ denotes the set of all nodes in the subtree with root *r*.

$ET(c, \epsilon)$ denotes the set of all edges incident to $T(c)$ of type ϵ, which are part of the Extended Tree.

$ET(c, \epsilon)_{inc}$ denotes those edges of $ET(c, \epsilon)$ which only have their target node in $T(c)$.

$ET(c, \epsilon)_{out}$ denotes those edges of $ET(c, \epsilon)$ which only have their source node in $T(c)$.

$ET(c, \epsilon)_{int}$ denotes those edges of $ET(c, \epsilon)$ which have both their source *and* target node in $T(c)$.

$\#S_1$ denotes the number of elements in set S_1.

$\nu(S_1)$ denotes the set of all nodes of type ν contained in the set S_1.

$S_1 \xrightarrow{\epsilon} S_2$ denotes the set of edges of type ϵ whose source node belongs to node set S_1 and whose target node belongs to node set S_2.

$target(S_1 \xrightarrow{\epsilon} S_2)$ denotes the set of target nodes of the given edge set.

$\Delta(S_i)$ denotes the change in S_i due to the application of a refactoring.

For example, $\#$**A**(*T(Machine)*) denotes the number of **A**-nodes (i.e., attributes) recursively contained in the subtree with root *Machine*. Table 1 describes the AST representation of the *T(Machine)* subtree, including our extension.

As another example, $\#ET(r, [au])_{out}$ denotes the number of **a**- and **u**-edges (attribute accesses and updates) from a node in the subtree of *r* to a node outside the subtree of *r*.

Table 1 describes the structure of the AST extension for the subtree T(*Machine*) by counting the superimposed cross-reference edges. Empty fields in the table indicate the impossibility of those occurences of those specific edges. For example, **i**-edges will never occur inside a class tree since inheritance only makes sense between two different classes. We will describe the AST and its extension subsequently. These AST descriptions (as illustrated in Table 1) will play a vital role in the formalism explained in the next section.

Class $Machine$ contains two methods and three attributes (implicit *this* attribute). Each of the two methods has one actual parameter, and one local variable (temporary string-variable). The number of expressions is irrelevant for the purposes of this paper, yet also provided on the left of Table 1.

Classes $Workstation$ and $PrintServer$ derive from $Machine$, represented by two incoming inheritance references. Attribute $name$ and the local variable in each method are of type $String$, which together with the two method parameters of type $Packet$ brings the number of outgoing type references to five. The only internal type reference is due to attribute $nextMachine$ of type $Machine$. Each method of $Machine$ calls the addition operator twice, println once, and the other method of $Machine$, which together makes eight (2x4). The $accept$ and $send$ method perform six and five internal accesses respectively, and each a single outgoing access. No updates are performed. This summarises the AST extension on the right of Table 1.

type	$\Delta T(c)$	type ϵ	$\#ET(Machine,\epsilon)_{int}$	$\#ET(Machine,\epsilon)_{inc}$	$\#ET(Machine,\epsilon)_{out}$
M	2	**i**		**2**	**0**
A	3	**t**	**1**	**2**	**5**
P	2	**c**			**8**
L	2	**a**	**11**		**2**
E	24	**u**	**0**	**0**	**0**

Table 1. Description of the AST (on the left, in terms of node types) and cross-references (on the right) of class $Machine$.

4 Describing the impact of refactorings on program structure

In this section, the impact of our selected refactorings on program structure is described. We split up the effect description in an impact on the AST representation, and an impact on its cross-references (from now on all together called the extended tree representation).

ExtractMethod($setE$:Set(E),m_1:M, m_2:String, c_1:C)

First, ExtractMethod introduces a new method m_2 in class c_1, for which possibly a number of actual parameters and a return value are required. A return value for m_2 is necessary when exactly one (multiple is not allowed) local variable or actual parameter of m_1 was updated by one of the expressions of $setE$. We calculate the number of new actual parameters of m_2 as the number of accessed local variables or actual parameters of m_1, even though Fowler [9] indicates not to create actual parameters for those local variables or actual parameters of m_1 whose first reference is an update (and are therefore immediately overwritten). The elaboration would contribute little to the overall goal of this paper and is left as an exercise for the enthousiastic reader. Consequently, we introduce no new local variables for method m_2.

The set of expressions $setE$ is copied to the new method m_2, and replaced inside m_1 by a method call to m_2 (1 **E**-node), with an actual parameter for each of the n accessed local variables or actual parameters of m_1 (n **E**-nodes). In case a local variable or actual parameter of m_1 was updated in $setE$, an extra expression is required to update that local variable or actual parameter (1 **E**-node). This summarises the impact on the AST of class c_1 as described on the left in Table 2.

Second, the introduction of new actual parameters and return value for method m_2 causes the introduction of cross-references to the types of those parameters, being either class c_1 itself - introduces an internal type reference - or another class - introduces an outgoing type reference. Naturally, ExtractMethod causes m_1 to call m_2 adding an extra internal call reference. Passing the arguments for the method call and returning the return value causes an increase of the number of internal access and update references, which summarises the right part of Table 2.

A superficial observation might lead to the interpretation that the application of ExtractMethod makes the program structure more complex in terms of our extended tree representation. Yet, these descriptions consist of both copying the expressions to the new method and thereafter replacing the set of expressions. Multiple applications of ExtractMethod on identical sets of expressions therefore only cause the former step to be performed once, while removing duplicate code by performing the latter step multiple types, as can be illustrated by extracting the println-statement from both $Machine.send$ and $Machine.accept$ in the example. Performing the last step multiple times removes code duplication.

type	$\Delta T(c_1)$
M	1
A	0
P	$\#target(T(setE) \xrightarrow{[au]} T(m_1))$
L	0
E	$1 + \#target(T(setE) \xrightarrow{a} T(m_1))$ $+\#target(T(setE) \xrightarrow{u} T(m_1))$

ϵ	$\Delta ET(c_1, \epsilon)_{int}$	$\Delta ET(c_1, \epsilon)_{out}$
t	$\#target(T(setE) \xrightarrow{[au]t} \{c_1\})$	$\#target(T(setE) \xrightarrow{[au]t} \overline{T(c_1)})$
c	1	0
a	$\#target(T(setE) \xrightarrow{a} T(m_1))$	0
u	$\#target(T(setE) \xrightarrow{u} T(m_1))$	0

Table 2. Impact of $ExtractMethod(setE, m_1, newMethod, c_1)$ refactoring on class c_1.

EncapsulateField(c_1:C,$attr$:A,$getter$:String,$setter$:String)

First, EncapsulateField introduces a *setter* and *getter* method, which respectively updates and accesses the attribute *attr* inside class c_1. Therefore, two methods and two parameters (actual parameter for setter and return value for setter) are added. As each new method consists of one expression (access or update), two expressions are added. This summarises the impact on the AST of class c_1, as described on the left in Table 3.

Second, the creation of the two new methods introduces a type edge from the actual parameter of the *setter* and one from the return value of the *getter* method to the type attribute, which can be either class c itself or another class. Then all former accesses and updates to the attribute $attr$ are replaced respectively by parameterless method calls to the *getter* and method calls to the *setter* method with the new value as an actual parameter. Finally, the *getter* method will update the return value with an access to the attribute, and the *setter* method will update the attribute with an access to the actual parameter.

type	$\Delta T(c)$
M	2
A	0
P	2
L	0
E	2

ϵ	$\Delta ET(c_1, \epsilon)_{int}$	$\Delta ET(c_1, \epsilon)_{inc}$	$\Delta ET(c_1, \epsilon)_{out}$	$\Delta \overline{ET(c_1, \epsilon)}$
t	$2 * \#(\{attr\} \xrightarrow{t} \{c_1\})$	0	$2 * \#(\{attr\} \xrightarrow{t} \overline{T(c_1)})$	0
c	$\#(T(c_1) \xrightarrow{[au]} \{attr\})$	$\#(\overline{T(c)} \xrightarrow{[au]} \{attr\})$	0	0
a	$-\#(T(c_1) \xrightarrow{a} \{attr\}) + 2$	$-\#(T(c) \xrightarrow{a} \{attr\})$	0	$-\#(\overline{T(c)} \xrightarrow{a} \{attr\})$
u	$-\#(T(c_1) \xrightarrow{u} \{attr\}) + 2$	$-\#(T(c) \xrightarrow{u} \{attr\})$	0	$-\#(\overline{T(c)} \xrightarrow{u} \{attr\})$

Table 3. Impact of $EncapsulateField(c_1, attr, getAttr, setAttr)$ refactoring on class c.

PullUpMethod($setM$:Set(M), $setC$:Set(C), c_s:C)

As PullUpMethod impacts subclasses c_i (with identical methods m_i) and superclass c_s, we will describe each of them separately and begin with the impact on the superclass.

First, PullUpMethod introduces a method m_s in superclass c_s with actual parameters and return value identical to those of m_i. The complete body of one of the identical methods m_i of subclasses c_i is copied to method $c_s.m_s$, which includes the local variables and expressions. This summarises the impact on the AST of superclass c_s, as described at the top of Table 4.

Second, the copying of these expressions causes all cross-references to be copied as well, consisting of type references, method calls, accesses and updates. This also means that former edges of the subclass method towards the superclass become internal edges of the superclass, and former edges from the subclass method towards other classes to become outgoing edges of the superclass. This impact is described at the bottom of Table 4.

The impact on any subclass c_i is identical, being the opposite of the impact on the superclass. The identical subclass methods m_i are removed, causing all actual parameters, local variables and expressions to be removed as well. Analogue, internal cross-references of $c_i.m_i$ are erased. The remainder of Table 5 describes the transformation as explained for the superclass.

Concluding, this section described the impact of the three refactorings on our extended AST representation of the source code. In the next section, we will introduce the formalisation of object-oriented program quality metrics on top of the extended tree representation.

type	$\Delta T(c)$
M	1
A	0
P	$\#P(m_i)$
L	$\#L(m_i)$
E	$\#E(m_i)$

ϵ	$\Delta ET(c_s,\epsilon)_{int}$	$\Delta ET(c_s,\epsilon)_{inc}$	$\Delta ET(c_s,\epsilon)_{out}$	$\Delta\overline{ET(c_s,\epsilon)}$
t	$\#(T(m_i) \xrightarrow{t} \{c_s\})$	$-\#(T(m_i) \xrightarrow{t} \{c_s\})$	$\#(T(m_i) \xrightarrow{t} (\overline{T(c_i)} \backslash T(c_s)))$	$-\#(T(m_i) \xrightarrow{t} (\overline{T(c_i)} \backslash T(c_s)))$
c	$\#(T(m_i) \xrightarrow{c} \{c_s\})$	$-\#(T(m_i) \xrightarrow{c} \{c_s\})$	$\#(T(m_i) \xrightarrow{c} (\overline{T(c_i)} \backslash T(c_s)))$	$-\#(T(m_i) \xrightarrow{c} (\overline{T(c_i)} \backslash T(c_s)))$
a	$\#(T(m_i) \xrightarrow{a} \{c_s\})$	$-\#(T(m_i) \xrightarrow{a} \{c_s\})$	$\#(T(m_i) \xrightarrow{a} (\overline{T(c_i)} \backslash T(c_s)))$	$-\#(T(m_i) \xrightarrow{a} (\overline{T(c_i)} \backslash T(c_s)))$
u	$\#(T(m_i) \xrightarrow{u} \{c_s\})$	$-\#(T(m_i) \xrightarrow{u} \{c_s\})$	$\#(T(m_i) \xrightarrow{u} (\overline{T(c_i)} \backslash T(c_s)))$	$-\#(T(m_i) \xrightarrow{u} (\overline{T(c_i)} \backslash T(c_s)))$

Table 4. Impact of $PullUpMethod(setM, setC, c_s)$ refactoring on superclass c_s.

type	$\Delta T(c_i)$
M	-1
A	0
P	$-\#P(T(m_i))$
L	$-\#L(T(m_i))$
E	$-\#E(T(m_i))$

ϵ	$\Delta ET(c_i,\epsilon)_{int}$	$\Delta ET(c_i,\epsilon)_{out}$	$\Delta\overline{ET(c_i,\epsilon)}$
t	0	$-\#(T(m_i) \xrightarrow{t} \overline{T(c_i)})$	$\#(T(m_i) \xrightarrow{t} \overline{T(c_i)})$
c	0	$-\#(T(m_i) \xrightarrow{c} \overline{T(c_i)})$	$\#(T(m_i) \xrightarrow{c} \overline{T(c_i)})$
a	$-\#ET(m_i,a)_{int}$	$-\#(T(m_i) \xrightarrow{a} \overline{T(c_i)})$	$\#(T(m_i) \xrightarrow{a} \overline{T(c_i)})$
u	$-\#ET(m_i,u)_{int}$	$-\#(T(m_i) \xrightarrow{u} \overline{T(c_i)})$	$\#(T(m_i) \xrightarrow{u} \overline{T(c_i)})$

Table 5. Impact of *PullUpMethod(setM, setC, c_s:C)* refactoring on any subclass c_i.

5 Analysing the impact on object-oriented program quality metrics

We will now illustrate how we can use our previous results to analyse the impact of refactorings on object-oriented software metrics. This is crucial to assess the impact of refactorings on program quality, since software metrics are typically used as internal quality factors [8].

The general idea is quite simple: we can formally specify object-oriented program quality metrics in terms of the extended AST of the program structure presented earlier. As such, the impact of a refactoring on the program structure, as denoted in the impact tables provided in the previous section, can be directly translated into the impact of a refactoring on the object-oriented program quality metrics. This formalism for defining metrics is analogue to the one provided in [14], where a graph-based formalisation of object-oriented software metrics is introduced.

5.1 Selected metrics

As the list of object-oriented program quality metrics is virtually endless (i.e. [26] alone describes more than 200 complexity metrics), and the page limit for this paper is not, we will focus on those program metrics which are most commonly used, being Number of Methods, Cyclomatic Complexity, Number of Children, Coupling Between Objects, Response For a Class and Lack of Cohesion among Methods. It can be argued that some of these metrics are not so much measures of program quality but of program size. Yet, as previous work from the context of formalizing object-oriented program quality metrics [2] uses similar primitives to calculate design quality metrics, we are confident that the current set provides a sound sample for the specific purpose of demonstrating the feasibility of using our formalism to investigate the quality drift caused by the application of refactorings. Moreover, [3] validated the Number of Children, Coupling Between Objects and Response For a Class metrics as quality indicators by investigating the relationship with fault probability.

Definitions for these metrics are:

Number of Methods calculates the number of methods of a class. It is an indicator of the functional size of a class.
Cyclomatic Complexity counts the number of possible paths through an algorithm. It is an indicator of the logical complexity of a program, based on the number of flow graph edges and nodes [7].
Number of Children measures the immediate descendants of a class [5]. It is an indicator of the generality of the class.
Coupling Between Objects is a measure for the number of collaborations for a class [18]. It is an indicator of the complexity of the conceptual functionality implemented in the class.
Response For a Class is the number of both defined and inherited methods of a class, including methods of other classes called by these methods [5]. It is an indicator of the vulnerability to change propagations of the class.

Lack of Cohesion among Methods is an inverse cohesion measure (high value means low cohesion). Of the many variants of LCOM, we use LCOM1 as defined by Henderson-Sellers [10] as the number of pairs of methods in a class having no common attribute references. It is an indicator of how well the methods of the class fit together.

Table 6 formalises these metrics on top of our source representation. This will allow us to project the impact of refactorings on program structure - as described in the previous section - in the area of software quality.

Metric	**Formula**	**Metric($Machine$)**
NOM	$\#M(T(c))$	2
CC	insufficient model information	/
NOC	$\#ET(class, i)_{inc}$	2
CBO	$target(ET(class, t)_{out} \cup ET(class, [au][tm])_{out} \cup ET(class, cm)_{out})$	4
RFC	Assume $setM = M(target(ET(class, i*)_{out}) \cup \{class\})$ then $RFC = \#(setM \cup \{m_2 \mid \exists m_1 \in setM \wedge m_2 \in target(ET(m_1, c)_{out})\})$	3
LCOM	$\#\{\{m_1, m_2\} \mid m_1, m_2 \in M(T(c)) \wedge m_1 \neq m_2 \wedge target(T(m_1) \xrightarrow{[au]} T(c)) \cap target(T(m_2) \xrightarrow{[au]} T(c)) = \emptyset\}$	0

Table 6. Formalization of selected metrics on top of our source representation, calculated for the $Machine$ class from the example of section 2.2.

The formalizations provided in Table 6 are defined in terms of our extended tree representation, which is a formal description of program structure. This allows an analysis of the impact of refactorings on internal program quality metrics, by translating the structural changes to the program structure, as described in the impact tables of section 4, into changes on the various metrics.

5.2 Analysing the impact of refactorings

For the purpose of clarifying whether the internal quality (represented by the metric) increases or decreases, we need to analyse in which direction this drift could occur. Therefore, we categorise the effects on a metric value in the following three categories (analogue to the work presented in [23]):

Impact	**Symbol**	**Range of effect on metric value**
nil	0	[0,0]
positive	+	[0,+∞[
negative	-	]-∞,0]

A *nil* impact represents a structural change which *will never* affect the value of the internal program quality metric. A *positive* impact represents a structural change which *might increase* the value of the internal program quality metric or leave it unchanged, yet can never decrease it. Lastly, a *negative* impact represents a structural change which *might decrease* the value of the internal program quality metric or leave it unchanged, yet can never increase it.

In order to illustrate our technique of analysing the impact of refactoring on internal program quality metrics, we elaborate on the most interesting metrics.

As the metric formalizations, denoted in Table 6, are constructed out of a number of different terms, we can analyse the impact of a refactoring on the metric value by analysing its impact on these various terms. To do this, we split out the different terms, and use the impact tables provided in section 4 to identify the impact category (nil, positive or negative).

Table 7 analyses the impact of the refactorings on the Coupling Between Objects metric value by clarifying the potential influence of each refactoring on the different terms of the metric formalization (analogue tables are provided for Response For a Class and Lack of Cohesion among Methods in tables 8 and 9). When the impact of a refactoring is positive for at least one term, and negative for none (nil impacts allowed), the total impact of the refactoring on the metric value is a positive impact (potentially cause the metric value to increase). Conversily, when the impact of a refactoring is negative for at least one term, and positive for none (nil impacts allowed), the total impact of the refactoring on the metric value is a negative impact

Refactoring	$\Delta target(ET(c,t)_{out})$	$\Delta ET(c,cm)_{out}$	$\Delta ET(c,[au][tm])_{out}$	CBO impact
ExtractMethod	+	0	0	+
EncapsulateField	0	0	0	0
PullUpMethod-Superclass	+	+	+	+
PullUpMethod-Subclass	-	-	-	-

Table 7. Analysis of factors which could cause drift of the CBO metric value.

Refactoring	$M(T(c))$	$\Delta target(ET(c,i*)_{out})$	$\Delta target(ET(c,c)_{out})$	RFC impact
ExtractMethod	+	0	0	+
EncapsulateField	+	0	0	+
PullUpMethod-Superclass	+	0	+	+
PullUpMethod-Subclass	-	0	-	-

Table 8. Analysis of factors which could cause drift of the RFC metric value.

Refactoring	$M(T(c))$	$\Delta target(ET(c,[au])_{int})$	RFC impact
ExtractMethod	+	+	+
EncapsulateField	+	0	+
PullUpMethod-Superclass	+	+	+
PullUpMethod-Subclass	-	-	-

Table 9. Analysis of factors which could cause drift of the LCOM metric value.

(potentially causes the metric value to decrease). Two exceptions arise in the reasoning about the impact of a refactoring on a metric value.

First, the selection of the target-nodes of a set of edges inside the metric formalization makes the analysis more complex. It requires semantical reasoning about whether the removal of an edge from a set of edges setE also reduces target(setE). This is an important issue as most of our metric formalizations explicitly depend on the target of a set of edges. I.e. while EncapsulateField increases the number of type-edges departing from the class subtree, it does not affect the target of this set of edges (the classes of which an instance was referenced). This semantic information is lacking from the impact tables as they provide a quantititative description of the change to the cardinality of the entities of the extended tree representation. In the next section, we will describe how to make the analysis of impacts in these situations more easy.

Second, when a refactoring has a different impact on the various terms of a metric value (positive for some, negative for others), a deeper semantical analysis is required, possibly even up to the level of inspection of the specific source code context. This limitation is also discussed in detail in the next section.

The result of this impact analysis is summarised in Table 10. We verified the impact catalogue by applying the refactorings on the LAN example and comparing post- and pre-refactoring measurements, as done in [11]. The drift noticed in these comparisons confirmed our formal analysis.

This impact catalogue can be used as an a-priori feedback on the efficiency of applying specific refactorings, from the perspective of various internal program quality metrics. In example, the table clarifies that applying the Pull Up Method refactoring has an impact which is opposite for the superclass and the subclass. While it potentially decreases the metric values for the internal program quality metrics Number of Method, Coupling Between Objects, Response For a Class and Lack of Cohesion among Methods of the subclass, it potentially increases these metric values of the superclass. This is a detailed description for the fact that the quality drift on the superclass, caused by moving a method up the inheritance hierarchy, is the inverse of the quality drift on the subclass. This allows us to envision that when we want to improve the quality of the subclass, this could possibly cause a deterioration of the superclass quality.

The next section discusses the current limitations of using this technique to tackle the question of quality drift caused by the application of refactoring, and elaborates on a their solutions.

Refactoring	NOM	NOC	CBO	RFC	LCOM
EncapsulateField	+	0	0	+	+
PullUpMethod subclass	-	0	-	-	-
PullUpMethod superclass	+	0	+	+	+
ExtractMethod	+	0	0	+	+

Table 10. Refactoring impact table indicating the impact of a particular refactoring on a particular class quality metric

5.3 Limitations and solutions

Our technique for analysing the impact of refactorings on internal program quality metrics allows the clarification of the drift of specific internal program quality metrics, as caused by the application of particular refactorings. While some early results were presented which demonstrated the feasibility of applying this technique for a number of refactorings and a number of internal program quality metrics, it is clear that the applicability of the technique has a number of limitations.

First, our representation for program structure is a limiting factor, as the impact analysis can only use the information contained in this program representation. We found an example of an internal program quality metric on which the impact of refactorings could not be analysed due to lacking model information (no control flow information in our program structure representation). A solution to this problem could be to simply extend our model, yet this will inevitably make our model more complex. Moreover, the metamodels used in related research on the formalization of metrics demonstrates that most of the measures of the current metric suites can be operationally defined on a program structure representation similar to ours [1, 4, 19, 13]. A detailed investigation of how this limitation reduces the number of internal program quality metrics on which the impact can be analysed requires a deeper study on the operational definitions of currently known program quality metrics.

Second, our formal descriptions of the structural changes on the program structure, expressed in terms of an extended tree representation, lacks semantical information about the sources and targets of the cross-reference edges which are added or removed during the refactoring. This information is currently implicitly contained in the informal description of the refactorings. Therefore, one of the lessons we learned is that a complete formal description of the structural changes caused by the application of a refactoring is required in order to automate the impact analysis process, i.e. using logic engines such as Prolog. While necessary for the next step of analysis of a more extended set of refactorings, the scale of our current work, serving the purpose of a proof-of-concept, did not require automated analysis.

Summarizing, we identified solutions for the two major limitations of our technique, which will simplify the analysis of the impact of refactorings on internal program quality metrics, making it possible to automate the impact analysis process. Such an automation is essential to cope with the massive amount of combinations between refactorings and internal program quality metrics.

6 Related work

Formalisations of software metrics have been provided from the mathematical perspective [1, 4, 19, 13] and the formal specifications perspective, in example Z [15] and OCL [2]. Our formalism is analogue to the mathematical approach of [1], yet they do not provide a formal metamodel specification but rely directly on the cardinality of informally described model features. We feel that the formal metamodel specification helps us in reasoning about program transformations. None of these metamodels incorporated information not contained in the model for program structure presented in this work (except information about modifiers such as abstract, final, public, protected, private).

In previous work, we introduced the existing research field of refactoring, and proposed an extensive list of directions for future research [12]. Most recently, an extension to the UML 1.4 metamodel for the purpose of facilitating refactoring at the UML level while remaining consistent with the source-code was proposed by members of our research group [25].

An experience report on metric collection during a refactoring phase is provided in [22]. A formalization of program transformations is introduced in [13], which formed the basis of this work. The same graph-rewriting foundation for describing refactorings is used in [24], which introduces a hierarchical representation for visualizing program structure.

The work which lies most closely to ours is provided in [23], in which the impact of meta-pattern transformations on an object-oriented metrics suite is provided. Our work is similar in that they are also interested in a-priori feedback on the impact of source-code transformation, and therefore also provide an impact catalogue of source code transformations on object-oriented metrics. Our work is different in that we focus on the impact analysis technique itself and therefore formalise the process of analysing the impact of catalogued refactorings provided by Fowler on internal program quality metrics, while they focus on the reengineering strategy of resolving design flaws through the application of meta-patterns.

A quantitative evaluation method to measure the maintainability enhancement effect of program refactoring is presented in [11]. They analysed three phases in the process of program refactoring, of which their contribution is towards the phase of validation of the refactoring effect. They analyse the effect of a number of refactorings on coupling metrics by pre- and post-refactoring measurements.

Detection of refactoring-candidates using visualisation techniques is introduced in [21]. Automatic detection of transformations is described in [20], in which rules for candidate selection are defined in terms of metric thresholds.

7 Conclusion and Future Work

Our technique for analysing the impact of refactorings on internal program quality metrics allows the clarification of the drift or improvement of specific internal program quality metrics, as caused by the application of particular refactorings. The results presented in this work demonstrated the feasibility of applying this technique for a number of refactorings and a number of internal program quality metrics. The limitations of our current approach were identified and solutions were discussed to resolve them.

In this paper, we presented both a formalism for describing and a technique for analysing the impact of refactorings on internal program quality metrics as indicators of quality factors. As a case study, the technique was applied to a number of representative refactorings from the refactoring categories Composing Methods, Organizing Data and Dealing with Generalization [9], and a number of commonly used internal program quality metrics (Number of Methods, Number of Children, Coupling Between Objects, Response For a Class, Lack of Cohesion among Methods).

The resulting classification of the impact in positive or negative contributions to internal quality metrics delivers a-priori feedback to software maintainers, enabling them to predict the quality drift caused by the application of (a series of) refactorings.

Our technique, improved by the suggestions to counter the limitations, remains to be applied to a more extended set of refactor operations and object-oriented program quality metrics, to form a catalogue of the impact of refactorings on internal quality metrics. Guided by this impact-catalogue on internal quality metrics, we plan experiments to gather empirical data about the impact of refactoring on external program quality metrics (performance, mean time between repair,...).

8 Acknowledgements

This research is funded by the FWO Project G.0452.03 “A formal foundation for software refactoring”. We thank Pieter Van Gorp, Hans Stenten, Serge Demeyer and Andy Zaidman for their useful comments on drafts of this paper.

References

[1] F. B. Abreu and R. Carapuca. Object-oriented software engineering: Measuring and controlling the development process. In *Proc. 4th Int’l Conf. Software Quality*, October 1994.

[2] A. L. Baroni. Formal definition of object-oriented design metrics. Master’s thesis, Vrije Universiteit Brussel and Ecole des Mines de Nantes, Belgium, 2002.

[3] V. R. Basili and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Engineering*, 22(10):751–761, October 1996.

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

[5] S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Software Engineering*, 20(6):476–493, June 1994.

[6] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *IEEE Computer*, pages 44–49, August 1994.

[7] J. C. Coppick and T. J. Cheatham. Software metrics for object-oriented systems. In *Proceedings of the 1992 ACM annual conference on Communications*, pages 317–322. ACM Press, 1992.

[8] N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.
[9] M. Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
[10] B. Henderson-Sellers. *Object-Oriented Metrics: Measures of Complexity*. Prentice-Hall, 1996.
[11] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Proc. Int'l Conf. Software Maintenance*, pages 576–585. IEEE Computer Society Press, 2002.
[12] T. Mens, S. Demeyer, B. Du Bois, H. Stenten, and P. Van Gorp. Refactoring: Current research and future trends. *Language Descriptions, Tools and Applications (LDTA)*, 2002.
[13] T. Mens, S. Demeyer, and D. Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002. Proceedings First International Conference ICGT 2002, Barcelona, Spain.
[14] T. Mens and M. Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.
[15] I. Moore. Automatic inheritance hierarchy restructuring and method refactoring. In *Proceedings Int'l Conf. OOPSLA '96*, ACM SIGPLAN Notices, pages 235–250. ACM Press, 1996.
[16] W. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
[17] W. Opdyke and R. Johnson. Creating abstract superclasses by refactoring. In *Proc. ACM Computer Science Conference*, pages 66–73. ACM Press, 1993.
[18] R. Pressman. *Software Engineering A Practitioner's Approach*. McGraw-Hill, 2001.
[19] R. Reissing. Towards a model for object-oriented design measurement. In F. B. e Abreu, editor, *Proc. 5th Int. ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 71–84, 2001.
[20] H. A. Sahraoui, R. Godin, and T. Miceli. 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, october 2000.
[21] F. Simon, F. Steinbrückner, and C. Lewerentz. Metrics based refactoring. In *Proc. European Conf. Software Maintenance and Reengineering*, pages 30–38. IEEE Computer Society Press, 2001.
[22] E. Stroulia and R. V. Kapoor. Metrics of refactoring-based development: An experience report. In *Proc. of the 7th International Conference on Object-Oriented Information System*, pages 113–122. Springer Verlag, 2001.
[23] L. Tahvildari and K. Kontogiannis. A metric-based approach to enhance design quality through meta-pattern transformations. In *Proc. European Conference on Software Maintenance and Reengineering*, pages 183–192. IEEE Computer Society Press, 2003.
[24] N. Van Eetvelde and D. Janssens. A hierarchical program representation for refactoring. In *Proc. of UniGra'03 Workshop*, 2003.
[25] P. Van Gorp, H. Stenten, T. Mens, and S. Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of UML 2003 – The Unified Modeling Language*. Springer-Verlag, 2003.
[26] H. Zuse. *Software Complexity*. Walter de Gruyter, Berlin, 1991.