QAOOSE 2006 Proceedings

10th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering

3 July 2006 — Nantes, France

Edited by:

Michele Lanza, Fernando Brito e Abreu, Coral Calero, Yann-Gaël Guéhéneuc, Houari Sahraoui

Organizers

Fernando Brito e Abreu, *Univ. of Lisbon, Portugal*

Coral Calero, *Univ. of Castilla, Spain*

Yann-Gaël Guéhéneuc, *Univ. of Montreal, Canada*

Michele Lanza, *Univ. of Lugano, Switzerland*

Houari Sahraoui, *Univ. of Montreal, Canada*

Outline

QAOOSE 2006 is a direct continuation of nine successful workshops, held during previous editions of ECOOP in Glasgow (2005), Oslo (2004), Darmstadt (2003), Malaga (2002), Budapest (2001), Cannes (2000), Lisbon (1999), Brussels (1998) and Aarhus (1995).

The QAOOSE series of workshops has attracted participants from both academia and industry that are involved/interested in the application of quantitative methods in object-oriented software engineering research and practice. Quantitative approaches in the object-oriented field is a broad and active research area that develops andor evaluates methods, practical guidelines, techniques, and tools to improve the quality of software products and the efficiency and effectiveness of software processes. The workshop is open to other technologies related to object-oriented such as component-based systems, web-based systems, and agent-based systems.

This workshop provides a forum to discuss the current state of the art and the practice in the field of quantitative approaches in the fields related to object-orientation. A blend of researchers and practitioners from industry and academia is expected to share recent advances in the field-success or failure stories, lessons learnedand seek to identify new fundamental problems arising in the field.

Contents

Metrics, Components, Aspects

Visualization, Evolution

Quality Models, Metrics, Detection, Refactoring

Measuring the Complexity of Aspect-Oriented Programs with Multiparadigm Metric*

Norbert Pataki, Ádám Sipos, and Zoltán Porkoláb

Department of Programming Languages and Compilers
Eötvös Loránd University, Faculty of Informatics
Pázmány Péter sétány 1/C H-1117 Budapest, Hungary
patakino@elte.hu, shp@elte.hu, gsd@elte.hu

Abstract. Aspect-oriented programming (AOP) is a promising new software development technique claimed to improve code modularization and therefore reduce complexity of object-oriented programs. However, exact quantitative inspections on the problem details are still under way. In this paper we describe a multiparadigm software metric and its extension to AspectJ. We use the metric to compute structural complexity of all the object-oriented, aspect-related and procedural components of AOP code. We tested our metric on two functionally equal implementations of GoF design patterns made in aspect-oriented way and in pure object-oriented style and compared the results.

1 Introduction

Metrics play an important role in modern software engineering. Testing, bugfixing cover an increasing percentage of the software lifecycle. In software design the most significant part of the cost is spent on the maintenance of the product. The cost of software maintenance highly correlates with the structural complexity of the code. The critical parts of the software can be identified in the early stages of the developement process with the aid of a good complexity-measurement tool. Based on software metrics we can give recommendations and define coding conventions on the development of sound, manageable and hygienic code. Even though general recommendations for specifying sensible metrics exist [23], the concrete measurement tools are typically paradigm-, and language-dependent.

In the software development process *abstractions* play a central role. An abstraction focuses on the essence of a problem and excludes the special details [4]. Abstractions depend on many factors: user requirements, technical environment, and the key design decisions. In software technology a *paradigm* represents the directives in creating abstractions. The paradigm is the principle by which a problem can be comprehended and decomposed into manageable components [1]. In practice, a paradigm directs us in identifying the elements in which a problem will be decomposed and projected. The paradigm sets up the rules and

* Supported by GVOP-3.2.2.-2004-07-0005/3.0

properties, but also offers tools for developing applications. These methods and tools are not independent of their environment in which they occur.

The last 50 years of software design has seen several programming paradigms from *automated programming* and the FORTRAN language in the mid-fifties, to *procedural programming* with structured imperative languages (ALGOL, Pascal), to the *object-oriented* paradigm with languages like Smalltalk, C++ and Java. However, it is important to understand that new paradigms cannot entirely replace the previous ones, but rather form a new structural layer on the top of them. Object-orientation is a new form of expressing relations between data and functions, however, these relations implicitly existed in the procedural paradigm.

The need for new programming paradigms is a result of the ever-growing complexity of software. Object-oriented programming (OOP) is widely used in the software industry for managing large projects, but recently some of the weaknesses emerged. Problems like cross-cutting concerns, multi-dimensional separation of concerns, symmetric extension of a class hierarchy [22] are hard to handle. Modern programming languages have made possible the birth of new programming paradigms like *(C++) template metaprogramming* (TMP) [5], *generic programming* (GP) [24], and *aspect-oriented programming* (AOP) [15].

Software metrics have always been strongly related to the paradigm used in the respective period. The McCabe *Cyclomatic complexity* number [2] was designed for measuring the testing efforts of non-structural FORTRAN programs. Piwowarksi [17], Howatt and Baker [11] extended the cyclomatic complexity with the notion of *nesting level* in order to describe structured programs better. After the object-oriented paradigm became widely accepted and used, both the academic world, and the IT industry focused on metrics based on special object-oriented features, like *number of classes*, *depth of inheritance tree*, *number of children classes*, etc. [3]. Several implementations of such metrics are available for the most popular languages (like Java, C#, C++) and platforms (like Eclipse) [25].

Most programs are written by using more paradigms. Object-oriented programs have large procedural components in implementations of methods. AOP implementations (among which the most widely-used is AspectJ), highly rely on OOP principles. AspectJ essentially integrates tools for modularizing crosscutting concerns into object-oriented programs. Moreover *multiparadigm programs* [4] appear in C++, Java, on the .NET platform, and others.

Metrics applied to different paradigms than the one they were designed for, might report false results [21]. Therefore an adequate measure applied to multiparadigm programs should not be based on special features of only one programming paradigm. A multiparadigm metric has to be based on basic language elements and constuction rules applied to different paradigms. A paradigm-independent software metric is applicable to programs using different paradigms or in a multiparadigm environment. The paradigm-independent metric should be based on general programming language features which are paradigm- and language independent.

Here we give the structure of the paper. In section 2 we define our multiparadigm metric, the *AV-graph*. After that we define the complexity of the class, where class is defined as a set of data (attributes) and control structures (member functions, methods) carrying out operations on the attributes. Afterwards, in section 4 we explain how our metric applies to AOP notions and constructs. Section 5 describes our test results when applying the metric to the OOP and AOP versions of the GoF design patterns. We explain how AOP affects the complexity of these implementations, and in which cases AOP provided a simpler solution.

2 A multiparadigm metric

The well-known measure of McCabe, the cyclomatic complexity [2] is based only on the number of predicates in a program: $V(G) = p + 1$. The inadequacy of the measure becomes clear, if we realize that cyclomatic complexity ignores the nesting level of the predicate nodes. Improvements as weighting the control structure with the nesting level were proposed by Harrison and Magel [10], by Piwowarski [17] and by Howatt and Baker [11]. The *scope* of a predicate node is a set of statements, whose execution depends on the decision made in the predicate node. The nesting level of a statement is defined as the number of predicate nodes whose scope contains the statement.

The nesting level notation reflects procedural programs in an adequate way. At the same time, it does not take into account data handling, which has central role in modern programming languages. We have to take the complexity of the data defined and the complexity of data handling into consideration. Accordingly, the *AV complexity* of a program is a sum of three components:

1. *Control structure of program.* Most programs have the same control statements irrespectively of the paradigm used. The control structure is represented by a graph where nodes are statements and (directed) edges represent the possible flow of control. Nodes with more than one output edge are called predicate nodes. Nesting level is used weighting statement nodes.
2. *Complexity of data types.* It reflects the complexity of data used (like in the case of classes). Data nodes are represented as different type of nodes in the control graph.
3. *Complexity of data access.* Connection between control structure and data is represented by edges between the data nodes and statements that use them. The direction of these edges reflect the data flow (for example in case of reading data the edge is directed from the data node to the statement). Data edges are nested by the nesting level of their statement node.

An important feature of our metric is that it does not count the complexity of data handling based on the place of the declaration. The metric encounters that value exactly at the point of data handling. Of course, the metric also measures the declaration in an implicite way: local variables are used only in the local code context (in the subprogram).

There is an other possible way to get these results. Let us suppose that we have no data nodes and data edges in our graph, but we replace them whit special control nodes: „reader" and/or „writer". These control nodes only send and receive information. They will be inserted just before and after the real control nodes which read and/or write data. The nesting depth and complexity value we get with this model is the 0same as the one calculated with AV graphs.

We can naturally extend our model to object-oriented programs. The central notion of the object-oriented paradigm is the class. Therefore we describe how we measure the complexity of a class first. On the base of the previous sections we can see the class definition as a set of (local) data and a set of methods accessing them.

The complexity of a class is the sum of the complexity of the methods and the data members (attributes). As the control nodes (nodes belonging to the control structure of one of the member graphs) were unique, there is no path from one member graph to another one. However, there could be attributes (data nodes) which are used by more than one member graph. These attributes have data reference edges to different member graphs.

This is a natural model of the class. It reflects the fact that a class is a coherent set of attributes (data) and the methods working on the attributes. Here the *methods* (member functions) are procedures represented by individual AV graphs (the member graphs). Every member graph has its own start node and terminal node, as they are individually callable functions. What makes this set of procedures more than an ordinary library is the common set of attributes used by the member procedures. Here the attributes are not local to one procedure but local to the object, and can be accessed by several procedures.

Let us consider that the definition of the AV graph permits the empty set of control nodes. In that case we get a classical data structure. The complexity of a classical data structure is the sum of the data nodes. The opposite situation is also possible. When a „class" contains disjunct methods – there is no common data shared between them –, we compute the complexity of the class as the sum of the complexities of the disjunct functions. We can identify this construct as an ordinay function library.

These examples also point to the fact that we use paradigm-independent notions, so we can apply our measure to procedural, object-oriented, or even mixed-style programs. This was our goal.

3 Aspect-Oriented Programming

Aspect-oriented programming (AOP) is one of the most promising new software development techniques. AOP aids a better handling of *crosscutting concerns* [12], as compared to object-orientation. Thus AOP is a generative programming paradigm that aims to help in writing more modularized, and more maintainable code. Today's AOP implementations (among which the most widely-used is AspectJ), mostly rely on OOP. AspectJ essentially integrates tools for mod-

ularizing crosscutting concerns into object-oriented programs. AOP defines the following important constructs, aside the OOP notions:

1. *Pointcut definitions* are made up of Pointcut type, and Pointcut signature. The pointcut type describes *what* happens, e.g. `call` stands for function call, `execution` stands for the execution of a function. The signature describes *which* kind of functions are monitored by the pointcut definition. `(void || int f(*))` means all the functions with the name `f`, that have either a `void` or an `int` return type, and receive one parameter of any type.
2. *Pointcuts* are sets of pointcut definitions bound by Pointcut operators (`||`, `&&`). A *named pointcut* is a pointcut that can be referred to by a name, therefore it is not necessary to be defined repeatedly.
   `pointcut p() : call(void || int f(*)) || execution(* g());`
3. *Advice* constructs specify the action to be taken at a certain pointcut (bound to the advice). The `before`, `after` and `around` keywords define *when* the body part of the advice is executed with respect to the pointcut. Otherwise the body of an advice is very similar to the body of a Java method.
4. *Inter-type declarations* allow among others declaring aspect precedences, custom compilation errors or warnings.
5. *Aspects* contain pointcuts, advices, and inter-type declarations. On the other hand, they also have a class-like behavior, as they can have their own attributes and methods.

Nowadays AOP is widely used in both academic, and industrial world. Practice shows that AOP programs are in many cases shorter, have more modular structure and are easier to understand. Numerous publications discuss the advantages of AOP design and implementation. However, we still have not found appropriate metric tools to present quantitative results on the structural complexity of AOP programs.

One possible reason might be the lack of multiparadigm metrics that are valid on both object-oriented and generative paradigm. There are proposals to measure specific features of AOP programs [6, 9], but our approach is that in practice a more suitable metric has to be able to measure soundly in more paradigms at once. The complexity of an AOP program depends on the OOP components and the AOP-specific constructs. Therefore the complexity could be scattered between the AOP-specific parts (in pointcut-definitions, advices, etc.), the object-oriented constructs (classes, inheritance, etc.), and even in the procedural-style implementation of the methods. Hence in our opinion we need to apply a metric that measures well more paradigms at the same time.

Experiences show that AOP provides a better solution for a certain set of problems (e.g. logging, debugging, etc.). In this paper we investigate what is common in these problem groups that renders the AOP solution intuitively easier. Can we find problem sets in which AOP provides a better solution? Why do we see one solution easier understandable than the other if they are implemented using different paradigms? How can we prove that for those aforementioned AOP problems the solutions are not only intuitively but also objectively better? In

order to answer these questions, we aim to analyze the Gang-of-Four (GoF) Design patterns [7] and their implementations in pure Java and an AOP version in AspectJ [14].

4 Extending the metric

Extending the metric requires the identification of AOP-specific program elements, and their mapping to an AV-graph. In section 3 we have enumerated the most important AspectJ constructions, now we examine how our multiparadigm metric applies to them. In order to measure programs, we also needed to extend the measurement tool.

1. *Pointcut definitions, and pointcuts.* Aspect-oriented programming is a kind of metaprogramming. With the help of pointcut definitions we describe notions to control the compilation and weaving process. A pointcut defines a condition which triggers the possible execution of a code defined in the appropriate advice. In that sense a pointcut definition is a metaprogram conditional statement. Therefore we map pointcut definitions to the AV-graph as predicate nodes, and its constitutes (the pointcut type and the signature as input nodes). As in the case of run-time programs, where a predicate node might use complex expressions, a pointcut definition can use pointcut operators to express complex conditions.
   We measure pointcut definitions by summing up the value (1 by default) assigned to the definition's type (`call`, `execution`, etc) and the complexity of the signature. The complexity of pointcuts is the sum of the definitions' complexities. The signature can be expressed as a regular expression, for which metrics already exist [18]. We have decided to add a constant 1 complexity to each token occuring in the expression. A token is a string literal (like: `foo`), a keyword (like: `int`), or a regular expression metacharacter (like: `*`). The rationale behind the definition is the following. It takes the same effort to understand that a signature applies to all functions (in the form: `* *(*)`) or to exactly one (`void foo(int) throws IOException`). However, more complex patterns cause decisions harder to understand, like in `void f*oo(int,*) throws *`.
2. *Named pointcuts.* The complexity of named pointcuts is the sum of the complexity of their names (1 by default) and the pointcut itself. Thus if the programmer defines a certain pointcut, names it, and instead of repeatedly defining it again refers to the pointcut by its name, the complexity of the code can be reduced. In section 2 we have seen, that the usage of functions decreases the complexity, because by making a function call, the added complexity is only the function's name, and its parameters. The usage of named pointcuts is analogous to that procedure.
3. *Advices*, from our metric's point of view are built up from two parts: the function part, and the pointcut part.
   - The method for measuring the pointcut part has already been described in the item 1.

- The purely function part is as follows. An advice's header is like that of a special function's, with the keywords `before`, `after`, or `around` as the name, followed by the regular parameter list. The "name" might be preceeded by a return type. The body of an advice does not seem different for the programmer than the body of a function would. Even in an `around` advice, the keyword `proceed` does not seem different from an ordinary function call. Therefore the function part's complexity is measured the same way as Java methods.

The pointcut decides when a certain advice's body part is executed. This is as if the body part of the advice would be in the *scope* of the predicates defined by the pointcut. Complex pointcut definitions behave like nested predicates. Thus the complexity of an advice is the complexity of advice's body multiplied by the complexity of its pointcut.

4. *Aspects* and classes have a lot in common from the complexity point of view. Both may include data, and member functions. Thus these members of aspects can be measured the same way as if they were in classes, for this the method is described in 2. Aspects may also have members of AOP-specific constructs. We have classified these constructs into two groups.
   - The complexity of *advices*, and *named pointcuts* is taken into consideration when measuring the aspect. These constructs directly affect the way the programmer sees the code. She needs to understand these members to be able to comprehend the complex construct described by the program.
   - As of now inter-type declarations like *declare parents*, *declare errors*, *declare warning*, and others are not taken into account when measuring complexity. We consider these auxiliary constructs in AspectJ which do not directly affect the complexity seen by the programmer, but rather as tools to easier express certain notions.

   These complexity values are summed up with the complexities of the data, and the member functions of the aspects.

We have seen in section 2 that the visibility of classes, and its members does not influence their complexity. For the same reasons we do not take this attribute into consideration in the case of aspects, and its members either.

5 Results

To validate our metric we have chosen the GoF Design Patterns' ([7]) implementations ([14]) for measuring. One of the reasons was that in [14] we find a functionally equivalent implementation of each pattern in AOP and OOP. At the same time, the renowned authors behind the implementations let us assume that the aspect-oriented techniques were handled correctly. We also supposed that DPs are neutral to crosscutting concerns. We did not choose examples that are well-known crosscutting problems (e.g. logging, tracing, etc), but more general ones, that *might* be in this problem set.

Many people think AOP reduces the complexity of the design patterns' implementations because of the patterns' crosscutting approaches. Obviously, the design patterns have been created as solutions for the non-trivial problems in OOP. The approach of AOP can describe these solutions easier by AOP's new language constructs.

The structure of these implementations is as follows. Each pattern has a Java and an AspectJ implementation. The AspectJ implementations also utilize a common library, otherwise independent of the patterns. As of now our measurement tool is able to parse and measure 14 out of 23 design pattern implementations. The table shows the AV-metric values, and the effective lines of code (ELOC) per patterns.

Design Pattern	Implementation	AV Complexity	Effective LOC
`adapter`	java	77	27
	AOP	51	22
`bridge`	java	235	75
	AOP	237	79
`builder`	java	219	55
	AOP	201	66
`decorator`	java	91	34
	AOP	93	25
`factoryMethod`	java	113	54
	AOP	129	67
`flyweight`	java	299	66
	AOP	286	71
`interpreter`	java	567	115
	AOP	567	113
`memento`	java	99	33
	AOP	186	47
`observer`	java	374	93
	AOP	305	87
`prototype`	java	187	53
	AOP	204	56
`state`	java	259	97
	AOP	179	103
`strategy`	java	265	56
	AOP	732	68
`templateMethod`	java	158	43
	AOP	158	45
`visitor`	java	300	83
	AOP	362	85

A comparison between the implementation of the design patterns in the OOP and AOP way can be found in [14, 16]. These papers explain that 17 out of 23 patterns had exhibited some degree of crosscutting. They also declare that implementing the patterns in AOP has many benefits, among them the most important being the ability to localize the code for a given pattern. Many patterns can be implemented as a single aspect, or as 2 closely related aspects. Our metric especially rewards code localization. The OOP versions can not be as well-structured as the AOP versions, where the code is more maintainable, and comprehensible. Another important benefit is the code's obliviousness. This benefit results directly from localization: as the pattern is localized in an aspect, it does not invade its participants. Henneman and Kiczales stated that the AOP versions are more modular by 74%, and more reuseable by 52%. According to [14] some patterns' implementation may disappear into the code because of AOP's constructs (e.g. the decorator pattern). This can also lead to complexity decrease.

In some cases we can see that the AOP implementation was less complex by our metric even if the ELOC number was greater. These are the cases when using AOP was adequate, these are the `adapter`, `builder`, `observer`, and `state` patterns. However, we can see a number of patterns where the AOP implementations were at least as complex as their Java counterparts. Patterns like `memento`, `visitor` and most typically `strategy` belong here. This shows that inadequate use of AOP can even be disadvantageous.

6 Conclusion and future work

In this paper we described a multiparadigm metric which was extended for aspect-oriented programs. The metric can measure the complexity of procedural, object-oriented, and aspect-related parts of programs implemented in AspectJ. We tested our metric on two functionally equal implementations of GoF design patterns: one of them is written in pure Java, the other is based on AspectJ. The metric revealed that aspect-orientation does not necessarily reduce the complexity on its own – the gain highly depends on the actual problem. Future investigations are needed to clarify the details.

References

1. Cardelli, L., Wegner, P.: On Understanding Types, Data Abstraction, and Polymorphism, ACM Computing Surveys 17(4), pp. 471-522, 1985
2. McCabe, T.J., A Complexity Measure, IEEE Trans. Software Engineering, SE-2(4), pp. 308-320, 1976
3. Chidamber S.R., Kemerer, C.F., A metrics suit for object oriented design, IEEE Trans. Software Engeneering, vol.20, pp.476-498, (1994).
4. Coplien, J.O.: Multi-Paradigm Design for C++, Addison-Wesley, 1998
5. Czarnezki K., Eisenecker, U.W.: Generative Programming, Addison-Wesley, 2000

6. Figueiredo, E., Garcia, A., Sant' Anna, C., Kulesza, U., Lucena, C.: Assessing Aspect-Oriented Artifactsd: Towards a Tool-Supported Quantitative Method, QAOOSE Workshop, ECOOP, Glasgow, pp. 58-69, 2005
7. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns – Elements of Reusable Object Oriented Software, Addison-Wesley, 1995
8. Gradecki, J.D., Lesiecki, N.: Mastering AspectJ, Wiley, 2003
9. Jean-Yves Guyomarc'h, Yann-Gael Guéhéneuc: On the Impact of Aspect-Oriented Programming on Object-Oriented Metrics, QAOOSE Workshop, ECOOP, Glasgow, pp. 42-47, 2005
10. Harrison, W.A., Magel, K.I., A Complexity Measure Based on Nesting Level, ACM Sigplan Notices,16(3), pp.6
11. Howatt, J.W., Baker, A.L.: Rigorous Definition and Analysis of Program Complexity Measures: An Example Using Nesting, The Journal of Systems and Software 10, pp.139-150, 1989
12. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, Jean-Marc, Irwin, J.: Aspect-Oriented Programming, ECOOP, Finland, Springer-Verlang LNCS vol. 1241, pp. 220-242, 1997
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ, LNCS vol. 2072, pp. 327-355, 2001
14. Kiczales G., Henneman, J.: Design Pattern Implementation in Java and AspectJ, OOPSLA, pp. 161-173, 2002
15. Kiczales, G.: Aspect-Oriented Programming, AOP Computing surveys 28(es), 154-p, 1996
16. Lesiecki, N.: Enhance design patterns with AspectJ, IBM `http://www.developers.net/external/730`
17. Piwowarski, R.E.: A Nesting Level Complexity Measure, ACM Sigplan Notices, 17(9), pp.44-50, 1982
18. James F. Power, Brian A. Malloy: A metrics suite for grammar-based software. Journal of Software Maintenance 16(6): 405-426 (2004)
19. Porkoláb, Z., Sillye, Á.: Towards a multiparadigm complexity measure, QAOOSE Workshop, ECOOP, Glasgow, pp.134-142, 2005
20. Schmidmeier, A., Hanenberg S., Unland, R.: Implementing Known Concepts in AspectJ, 2003
21. Grégory Seront, Miguel Lopez, Valérie Paulus, Naji Habra: On the Relationship between Cyclomatic Complexity and the Degree of Object Orientation, QAOOSE Workshop, ECOOP, Glasgow, pp. 109-117
22. Wadler, P.: The expression problem, Posted on the Java Genericity mailing list, 1998
23. Weyuker, E.J.: Evaluating software complexity measures, IEEE Trans. Software Engineering, vol.14, pp.1357-1365, 1988
24. Java 1.5, `http://java.sun.com/developer/technicalArticles/releases/j2se15`
25. Eclipse.org formation, `http://www.eclipse.org/org/index.html`

On the Influence of Practitioners' Expertise in Component Based Software Reviews

Miguel Goulão[1], Fernando Brito e Abreu[1]

[1] QUASAR Research Group, Centro de Informática e Tecnologias de Informação
Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Portugal
{miguel.goulao, fba}@di.fct.unl.pt
http://ctp.di.fct.unl.pt/QUASAR/

Abstract. *Objective:* To assess the influence of practitioners' expertise in code inspection of software components. *Method:* Subjects expertise was determined based on their independently assessed academic record. Inspection outcome was represented by the diversity of defects found at two levels of abstraction. *Results:* Statistically significant correlations among expertise and inspection outcomes were found in several cases. *Conclusion:* The effect of expertise is observable in the inspection outcome and thus can be used in for software quality management purposes.

1 Introduction

Software development team members' expertise is an essential factor to the success of a software project. Skilled developers are likely to produce better software than less skilled ones. Good code reviewers are likely to detect more defects than bad ones. In this paper, we assess the impact of practitioners' skills in the context of code reviews performed on component-based (CB) software. The reported results are part of a wider experiment, briefly described in the following section, where we assess practitioner's performances in the development, quality control and integration of software components, and compare them with an independent assessment of their skills.

A code review is a peer review of source code intended to detect defects before the testing phase begins, thus improving overall code quality. There are a number of code review processes being used in industry. Fagan inspections [1, 2] are considered seminal in this area. Other processes have emerged since then, that try to lower the costs involved in code inspections without sacrificing their benefits. Examples include conducting inspections offline, thus skipping the inspection meeting [3], or performing phased inspections, where the inspectors focus on a specific class of defects [4], although the latter technique has been criticized for being more costly than conventional inspections [5].

Understanding what drives inspections' success has been a long time concern in the software community. Based on data collected from over 6000 inspections, Weller

studied the impact of the inspection process on software quality [6]. Among several other remarks, he pointed to the familiarity of the inspection team with the artifact being inspected as a key factor in inspection success. We may regard this as kind of domain expertise. Siy observed that while structural changes were largely ineffective in improving the results of inspections, the inputs for those inspections (the reviewers and code being inspected) were far more influential in the inspection outcome [7]. These findings were further explored in [8], to conclude that better inspection techniques, rather than processes, were the key to improving inspection effectiveness. Biffl and Halling combined reviewers' expertise measures (software development skills, experience and an inspection capability pre-test) with different code inspection techniques [9]. While they could not find significant relationships between development skills and experience and inspectors performance, they found the inspection capability pre-test useful to optimize the inspection outcome by selecting ideal inspection teams. They also identified performance differences related to alternative code reading techniques, a result that is consistent with the findings of Laitenberger and DeBaud, in their systematic review on code inspections reading techniques [10]. Sauer et al. identified individual's task expertise as the primary driver of review performance [11].

In a totally different context (social psychology), Kruger and Dunning observed that the skill of a person in performing a task is closely related to the required skill to assess his own performance in the same task [12]. If we instantiate this insight into code production and code reviewing, we would expect the best programmers to also be the most effective code reviewers.

2 Problem statement

Our global goal is to analyze the outcome of a CB development process, for evaluation purposes, with respect to the impact of practitioner's expertise on defect introduction and detection, from the point of view of a project manager (in this case, the research team), in the context of an academic simulation of a component marketplace.

In this paper we are concerned with the impact of practitioners' expertise in the outcome of CB software code reviews performed at the component level. We are seeking evidence on possible causal relationships between the expertise of practitioners involved in the code inspections and the diversity of defects reported during those inspections (**Fig. 1**). All inspections were carried out by a review team (RT) which included the development team (DT) and a peer team (PT). PTs consisted of developers of a different component, participating as independent code reviewers. The remaining process and product inputs are fairly similar for all code inspections, to minimize possible confounding effects.

We consider four potential causal relationships. The expertise of the development team (1) may have a negative effect on the diversity of defects found. The rationale is that expert developers tend to introduce fewer defects on their code. Conversely, peer reviewers expertise may have a positive effect on the defects diversity (2). A similar rationale leads to the possible causal effect between the expertise of the review teams

as a whole (3) and defect diversity. Finally, we consider the difference of expertise between the developer team and the peer team (4) as a negative effect on defect diversity. If the expertise of the developers is higher than that of their peers, defect diversity is expected to be smaller than when the opposite occurs.

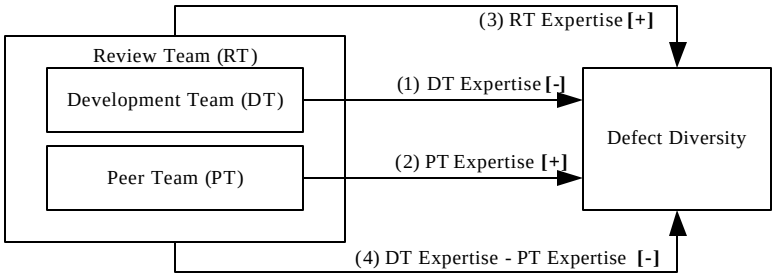


Fig. 1. Exploring the impact of practitioner's expertise in the outcome of a review process.

3 Experiment planning

3.1 Context selection

This experiment occurred in the context of a Software Engineering course held at the Universidade Nova de Lisboa, during the Spring semester of 2005. This course is offered on the 8th semester of their 5-years informatics degree. According to the new harmonized academic curricula adopted in Europe (Bologna model), these are 2nd cycle degree (MSc) graduate students. The course's project consisted in developing a CB elevator system simulator from requirements definition to final product delivery. The programming language was Java, well-known to all subjects in the experiment.

Among other activities, the development process included a Fagan inspection performed on all the developed components. While PT members were knowledgeable in the inspected code basic requirements, they were not developing an alternative implementation of that same component, to avoid biasing their review, besides better reproducing industry practice.

Standard Fagan inspection roles were assigned to the four RT members. The DT members got the moderator and author roles and the PT members the remaining ones (reader and recorder). An extensive checklist of common defects in Java programs was distributed (and its contents explained) to all RTs before code inspections took place. This paper focuses on the analysis of the outcome of these inspections.

3.2 Hypothesis formulation

The observations on the problem statement section lead us to testing four different basic hypotheses, to assess the effect of practitioners' expertise on the outcome of the code inspection, in terms of the inspected defects diversity. We identify the hypothe-

ses as HA, HB, HC, and HD. For each of them, we formulate both a null and an alternative hypothesis (e.g. HA_0 and HA_1).

As we shall see in the next section, we will break down each of these hypotheses into several specialized versions, to try out different expertise assessment metrics.

HA_0:	Developer skill has no effect on the inspected defect diversity.
HA_1:	Developer skill has an effect on the inspected defect diversity.
HB_0:	Peer skill has no effect on the inspected defect diversity.
HB_1:	Peer skill has an effect on the inspected defect diversity.
HC_0:	Reviewer expertise has no effect on the inspected defect diversity.
HC_1:	Reviewer expertise has an effect on the inspected defect diversity.
HD_0:	The gap of expertise between developer and peer has no effect on the inspected defect diversity.
HD_1:	The gap of expertise between developer and peer has an effect on the inspected defect diversity.

3.3 Variables selection

Independent variables. The basic independent variable of this experiment is the subjects' expertise. We use two measures of our subject's expertise: their ***A**verage **G**rade* (***AG***) throughout their academic path, based on the independent evaluation our subjects received in over 30 different courses, and the *number of semesters (**NSem**)* it took them to complete those courses. We assume that there is a higher merit in obtaining a given ***AG*** in the *recommended number of semesters* (***RSem***), than in a higher ***NSem***. The ***S**imple **W**eighted **A**verage **G**rade* (***SWAG***) and the ***C**omplex **W**eighted **A**verage **G**rade* (***CWAG***) expertise metrics, defined below, follow this rationale. Note that ***SWAG*** causes a bigger penalty than ***CWAG***, as ***NSem*** increases.

$$SWAG = AG \times \frac{RSem}{Max(RSem, NSem)} \qquad CWAG = AG \times \sqrt{\frac{RSem}{Max(RSem, NSem)}}$$

For each of the RTs, concerning expertise, we use the best subject, the worst subject, and the average within the team. Finally, we also consider the difference between the expertise of the DT and that of the PT as an independent variable. In summary, we have 3 alternative rating schemes for grades, and 3 ways of combining grades within teams. This implies that we have 9 different ways for quantifying our independent variable (the expertise). These alternatives are used for each hypothesis under test.

Dependent variables. The dependent variables used in this experiment represent the diversity of defects found during code inspection. A defect classification checklist was distributed to all participants. The checklist contained 16 different defect classes, which were then subdivided into a total of 81 different defect codes. In summary, our dependent variables are:

- ***NDDClass*:** the number of different defect classes reported in the inspection;
- ***NDDCode*:** the number of different defect codes reported in the inspection.

At first, *NDDClass* may seem unnecessary, given the usage of a finer grained measure (*NDDCode*). However, if two code inspections report a similar number of different defect codes, but one of them uses a lot less defect classes than the other, it may be the case that this reflects a lower coverage of the kinds of problems to be found during the inspection. We used *NDDClass* to detect this kind of problem, should it occur.

3.4 Selection of subjects

The 87 subjects participating in this experiment are a convenient, but also representative sample of the informatics students which annually graduate from our university (the *numerus clausus* for the informatics degree is 160, and the number of students graduating each year is around 60).

3.5 Experimental design

Regarding the experiment instrumentation, the calculation of subjects' expertise was done upon the data available from the university's academic database. The information concerning code inspections was collected from the normalized inspection reports submitted by subjects after they performed the Fagan inspections. Potential threats to validity [13], and how they were dealt with, are identified throughout the paper.

4 Data Collection

Preparation. The subjects were not aware of the aspects being researched, at the time they participated in the experiment, as this could jeopardize the validity of the results. They were only aware of our intention to use the data collected during the project.

Prior to the implementation of the components that were later inspected, subjects received a Java coding style guide, along with the set of standard public APIs for the components (specified as Java interfaces). Concerning code inspections, besides the referred checklist, subjects received a report template, so that they would perform the inspection and write down the report in a standardized fashion. They also received training on how to perform Fagan inspections, prior to actually starting them.

Execution. The experimental process was not allowed to disturb in any way the subjects' activities in the projects. Subjects performed their normal tasks while developing this project, from requirements specification down to project delivery. Code inspection data was collected from the project's deliverables, which was checked-in in a contents management system made available to students.

Data validation. In the beginning of the semester, there were 93 students enrolled in the course. Five of them dropped out before the experiment started, and one also gave up before turning in the first implementation of his group's component. The remaining 87 students completed the project and are the subjects of this experiment. They were paired into 44 DTs. 43 of those DTs produced components that were inspected. The deliverables of these 43 inspections were used to collect the dependent variables.

5 Data analysis

We summarize the most relevant findings of our hypotheses tests. Further details are available at http://ctp.di.fct.unl.pt/QUASAR/Projects/CBSE/CodeInspections/.

5.1 Data set reduction

Outlier and extreme values can change our view on the relations between dependent and independent variables. For each dependent variable, we conducted a linear regression analysis using the average, best and worst cases of the independent variables. We repeated this analysis for each of our hypotheses and flagged as outliers those cases where the standard residual is greater than 1.5 times the standard deviation. This resulted in the removal of four cases in our analysis, with each of the dependent variables. The outlier removal process is illustrated in **Fig. 2**, where cases 15, 19, 25, and 38 are removed.

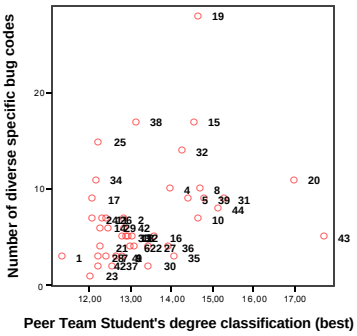


Fig. 2. Number of diverse specific bug codes, by PT expertise, including outliers.

5.2 Normality tests

We used the Kolmogorov-Smirnov (with the Lilliefors correction) to check normality. Using a confidence interval of 99% (test significance = 0.01), we can not reject the normality hypothesis, both for the independent and dependent variables. Therefore, we can use parametric tests, such as the Pearson correlation coefficient.

5.3 Hypothesis testing

We started by performing correlation analysis, using the Pearson coefficient, among each of the independent and the dependent variables, to determine whether or not a relationship exists. These correlations were not significant with the predictors of hypotheses HA and HC. As such, neither the influence of the DT skill, nor the influence of the overall RT skill in the outcome of the reviews, in terms of the diversity of the defect codes and classes, was confirmed.

From **Table 1** we can observe significant correlations between our independent and dependent variables for hypotheses HB and HD. Most of the candidate predictors for HB have a significant positive correlation of above 40%. This relationship is observed both with *NDDCode* and *NDDClass*. The predictors for HD have a significant negative correlation with the dependent variables. These correlations are stronger with *SWAG* and *CWAG* than with *AG*. Again, the same effect is observable both with *NDDCode* and *NDDClass*.

Table 1. Pearson correlations for the variables in hypotheses HB and HD, considering 39 cases (the outlier values referred in section 5.1 were removed, before the correlation analysis).

		HB						HD					
		PT						Diff_DT_PT					
		AG	sig.	SWAG	sig.	CWAG	sig.	AG	sig.	SWAG	sig.	CWAG	sig.
NDD Code	Avg.	.469	.003	.429	.006	.462	.003	-.402	.011	-.454	.004	-.471	.002
	Best	.419	.008	.385	.016	.413	.009	-.409	.010	-.393	.013	-.419	.008
	Worst	.479	.002	.441	.005	.470	.003	-.350	.029	-.470	.003	-.468	.003
NDD Class	Avg.	.416	.009	.378	.018	.407	.010	-.356	.026	-.433	.006	-.443	.005
	Best	.394	.013	.315	.051	.353	.028	-.376	.018	-.370	.020	-.395	.013
	Worst	.392	.013	.413	.009	.430	.006	-.293	.070	-.451	.004	-.451	.004

We further explored the HB and HD hypotheses, to check for significant differences observed in different groups of code reviews, using the ANOVA test. We started by computing the quartile values for the independent variables, and assigned the reviews to the respective quartile group. For each test, we had four groups with a growing expertise of the PT (in hypothesis HB), or with a growing difference between the expertise of the DT and the expertise of the PT (in hypothesis HD). The latter ranges from a negative value (DT has less expertise than PT) to a positive one (DT has a higher expertise than PT). **Table 2** shows an example of this means comparison test, for hypothesis HD.

Concerning HB, we observed a variation among the different review groups that always followed the same pattern. The reviews on the 4th quartile (the ones with the most expert peer teams) were always the ones with the highest *NDDCode* and *NDDClass*. Except when using predictors based on the worse PT element, or the average value of *SWAG*, these differences were statistically significant. *NDDCode* and *NDDClass* showed an increase ranging from 36% to 111%. This trend is not visible in the first three quartiles, for some of the used metrics. The scatterplot presented on **Fig. 2** is an example of a typical distribution of defect diversity *vs.* PT expertise. In summary, we can reject the null hypothesis HB_0. We were able to find several measures of the expertise within the PT which can be used as predictors of the diversity of the reported defects.

With respect to HD, we observe the opposite pattern. With the expertise functions being used, the average number of different reported bug codes and classes decreased between 19% and 49%, when comparing the first with the last quartiles. In other words, the number of diverse defect codes and classes decreases as we move from DTs with lower expertise than their PTs to the opposite case. As such, we can reject the null hypothesis HD_0. We found several measures of the difference between the expertise of the members of DT and PT which can be used as predictors of the diversity of the reported defects.

Table 2. Mean number of diverse defect codes found during code inspections. The difference between average AG of the DTs and PTs metric is used to place the PTs into the respective quartiles. Note that on the 1st quartile, DT has a much lower expertise than PT, while on the 4th, PT has a much lower expertise than DT.

Quartile	Mean Diverse Defects	N	Std. Dev.
1st	7.00	10	2.828
2nd	5.40	10	3.688
3rd	6.10	10	2.558
4th	4.78	9	2.819
Total	5.85	39	3.005

6 Discussion

HA. We expected the best developers to produce components with fewer defects, but this was not confirmed. This result may be explainable in different ways. We did not use any information conceming neither the relative severity of the defects found in this analysis, nor their expected impact on maintenance. Moreover, we used defect code and class diversity, but not the actual number of reported defects in this analysis. Therefore, it may be the case that our dependent variable is too simplistic. It may also be the case that, because PTs were also part of the RTs, their expertise countered the effect of a lower variety of problems with that of a higher efficiency in finding them. A way to circumvent this would be to have several inspections being performed on the same artifacts by different teams , but this was not feasible in our context.

HB. As expected, we observed that the expertise of the PT does have a positive effect on the variety of problems uncovered during code inspections. We also note that the average and higher element expertise within the PT have stronger correlations with the outcome of the review than the expertise of the "weaker" element of the PT. Along with the significant boost of results with the PTs on the best quartile, this increases our confidence on the positive effect of expert peer reviewers in the reviewer team and also points to a small effect of "leadership" within those teams.

HC. The expertise of the whole review team did not show a significant relationship with the outcome of the review. The considerations concerning a possible over-simplification of our dependent variable, combined with the cancellation effect also described with respect to HA may be responsible for this discrepancy between the expected result and the outcome of this experiment.

HD. As expected, when PTs of low expertise analyze the work of DTs with a higher expertise, the outcome of the code review shows a lower variety of defects found. Conversely, more defects are found in inspections where the PTs have a higher expertise than the one of the DTs. A potential leadership effect of a reviewer over the others is not visible from the data analyzed while testing this hypothesis.

With the experiment design of this last hypothesis, we have an alternative perspective on the inspection group dynamics, when compared to hypothesis HC. On HC we had no indication of how the expertise was distributed within the group, thus being

vulnerable to the cancellation effect occurring when (i) having good experts examining their own code and not finding many problems with it, because they were not there, or (ii) weaker programmers examining their own code and not realizing the problems in it. Both situations lead to a cancellation effect that might explain the unexpected results with hypothesis HC.

There is a curious effect in the evolution of the variety of defects found between the second and third quartiles of HD (the second quartile has DTs with a lower expertise than their PTs, while the third inverts this relationship). One could expect the variety of defects to be lower on the third quartile, following the tendency found from the first to the forth quartiles. However, the expertise level is very close, within groups 2 and 3. Therefore, it may be the case that it is the domain level expertise that dominates the outcome of the inspection. With a better knowledge of the deliverables being inspected, allied with a slightly better expertise than their peers, the authors may be responsible for this locally increased benefit of the code review. As the gap of expertise between DT and PT members widens, this effect would be mitigated by the dominating effect of the higher code quality and lower external reviewer expertise.

7 Conclusions

We described an experiment carried out to help understanding the effect of practitioner's expertise in the deliverables produced in the context of CB development.

We focused our attention on the outcome of code inspections, and, in particular, on the variety of problems reported during those inspections. We confirmed the expected positive effect of the expertise of the peer review teams in the outcome of the reviews, observable through the increased variety of defects found when peer experts were available. We also confirmed that having expert peers collaborating in the inspection of components developed by less skilled peers has a positive impact on the outcome of the review. Moreover, there is also a learning effect, not studied here but vastly commented on the literature, when combining experts with non-experts. This is also expected for the opposite case, where non-experts participate on the review of code developed by experts. However, in this case, a lower variety of defects is found, both because the code is likely to have a higher quality, and because the external reviewers have less capacity to detect its problems. Given the main goal of inspections (maximizing defect detection), the results are poorer.

When observed in isolation, the expertise of the DTs did not show a significant relationship with the variety of problems found. The expertise of the RTs was also not shown to be a good indicator of the outcome of the inspection. Further research is required to determine whether these were the results of cancellation effects of expertise, or if more sophisticated review outcome metrics should have been used here.

As future work, we expect to expand on this experiment by exploring this interpretation of why two of our hypotheses were not confirmed. The deliverables of the project that served as a basis for this experiment include some details that were not explored in this paper, such as code complexity metrics, and the practitioners' assessment of the potential impact of the problems reported. We plan to further explore these data to

strengthen the conclusions reported here and to explore other related hypotheses on the effect of expertise throughout the development process.

Acknowledgments

This work is sponsored by the FCT STACOS project (POSI/CHS/48875/2002).

References

1. Fagan, M.E., *Design and Code Inspections to Reduce Errors in Program Development*. IBM Systems Journal, 1976. 15(3): p. 182-211.
2. Fagan, M.E., *Advances in Software Inspections.* IEEE Transactions on Software Engineering., 1986. 12(7): p. 744-753.
3. Parnas, D.L. and Weiss, D.M., *Active Design Reviews: Principles and Practices.* Journal of Systems and Software, 1987. 4(7): p. 259-265.
4. Knight, J.C. and Myers, E.A., *An Improved Inspection Technique*. Communications of the ACM, 1993. 36(11): p. 51-61.
5. Porter, A. and Votta, L., *What Makes Inspections Work?* IEEE Software, 1997. 14(6): p. 99-102.
6. Weller, E.F., *Lessons from Three Years of Inspection Data*. IEEE Software, 1993. 10(5): p. 38-45.
7. Siy, H., *Identifying the Mechanisms to Improve Code Inspection Costs and Benefits*, PhD Thesis, University of Maryland, USA. 1996.
8. Porter, A., Siy, H., Mockus, A., and Votta, L., *Understanding the sources of variation in software inspections.* ACM Transactions on Software Engineering and Methodology, 1998. 7(1): p. 41-79.
9. Biffl, S. and Halling, M. *Investigating the Influence of Inspector Capability Factors with Four Inspection Techniques on Inspection Performance*. in *Eighth IEEE International Symposium on Software Metrics (Metrics'02)*. 2002.
10. Laitenberger, O. and DeBaud, J.-M., *An Encompassing Life-Cycle Centric Survey on Software Inspection.* Journal for Systems and Software, 2000. 50(1): p. 5-31.
11. Sauer, C., Jeffery, R., Land, L., and Yetton, P., *The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research.* IEEE Transactions on Software Engineering, 2000. 26(1): p. 1-14.
12. Kruger, J. and Dunning, D., *Unskilled and Unaware of It: How Difficulties in Recognizing One's Own Incompetence Lead to Inflated Self-Assessments.* Journal of Personality and Social Psycology, 1999. 77(6): p. 1121-1134.
13. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., and Wesslén, A., *Experimentation in Software Engineering: An Introduction*. Vol. 6. 1999, Boston, EUA: Kluwer Academic Publishers. 224 pages.

A Substitution Model for Software Components

Bart George, Régis Fleurquin, and Salah Sadou

VALORIA Lab., University of South Brittany, France
{Bart.George,Regis.Fleurquin,Salah.Sadou}@univ-ubs.fr

Abstract. One of Software Engineering's main goals is to build complex applications in a simple way. For that, software components must be described by its functional and non-functional properties. Then, the problem is to know which component satisfies a specific need in a specific composition context, during software conception or maintenance. We state that this is a substitution problem in any of the two cases. From this statement, we propose a need-aware substitution model that takes into account functional and non-functional properties.

1 Introduction

Component-oriented programming should allow us to build a software like a puzzle whose parts would be units "subjects to composition by a third party" [13]. Examples of such units are COTS (*Components-Off-The-Shelf*), which are commercial products from several constructors and origins. When one develops and maintains a component-based software, some problems occur, and we will notice two main ones: how to select, during conception of such a software, the most suitable component in order to satisfy an identified need ? And during a maintenance, if this need evolves, will the chosen component remain suitable, or shall we replace it ?

We think that these problems are related to a substitution problem. In fact, when one conceives or maintains an application, some needs appear. And to describe them, the designer or the maintainer can imagine *ideal components*. These are virtual components representing the best ones satisfying these specific needs. Then the problem is to find the concrete components which are the closest to the ideal ones. In other words, trying to compose or maintain components means trying to make concrete components substitute ideal ones.

However, composition doesn't concern only the functional aspect. Most components are "black boxes" which must describe not only functional, but also non-functional properties. As every software needs a certain quality, one can't think about composing components whose non-functional properties are unknown, and at the same time hope having its quality requirements satisfied anyway. This is why substitution must take functional and non-functional properties into account.

So, how to substitute ? Some may say we just have to use subtyping, as some object-oriented languages made it a general way of substitution. However, an ideal component describes more than general needs: it describes the application's

context, a notion that is absent from objects. Let us explain what we mean by "context". If we take a need, modeled by an ideal component, we will try to find a concrete one to substitute it. Now, let us suppose that we already found a suitable component. We may need to check if there isn't another one better than the first one. However, trying to substitute the old candidate by a new one would be a mistake, because the key notion isn't the candidate, but the need it is supposed to satisfy. Plus, if this need changes, a former candidate may no longer remain suitable. So substitution of an ideal component by a concrete one is performed only into the context of the need modeled by the ideal component. This is why a candidate component can replace another one without any subtyping relation between them, as every candidate is compared only to the ideal component.

In this paper, we consider a generic component model and a quality model, and into this framework we define (section 2) a component-oriented substitution model, including a distance from a candidate component to an ideal one. In order to illustrate the possibilities of such a model, we describe the different substitution cases during the life cycle using a short application example (section 3). Then, before concluding, we describe some related works (section 4).

2 Our substitution model

Definitions given in this paper are placed in the following framework: one component model, holding a type system such as Java for EJB, and one quality model such as ISO 9126 standard [11]. In this framework, we suppose the existence of metrics to measure non-functional properties, so that our contribution will focus only on the substitution model definition.

Here, we will present only the basic concepts of this model. A more detailed description is available in [7]. Note that in this version of our work, we perform substitution at the individual component level.

2.1 Component and quality generic models

Our goal is not to give yet another definition of what a component is, or what non-functional properties are. It is to define a component-oriented substitution that we can apply on many existing component and quality models. That is why we prefer to give generic models, on which we can apply our substitution concepts.

The generic component model includes component **artifacts**, representing the component's architectural elements, which are common to most existing component models, and which have non-functional properties. As shown in figure 1, we chose to keep three kinds of component artifacts: components themselves, interfaces, and operations. A component contains provided and required interfaces, and interfaces contain operations. In the remaining of the paper, we refer to **candidate component** and **substitutable component** when the first one tries to substitute the second one. Their elements are called respectively **candidate elements** and **substitutable elements**. When we find the best candidate for the

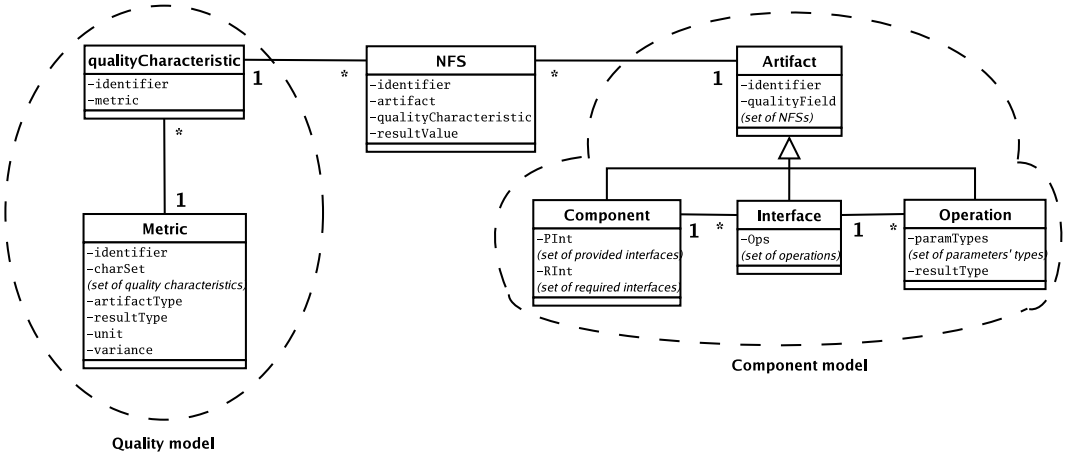


Fig. 1. Our generic model

substitution, we say the substitutable component or element can be **replaced** by this candidate.

Beside the component model, we define a generic quality model, on which the quality properties of the component model's elements are based. Elements of the quality model are quality characteristics (such as those from ISO 9126 [11]), and metrics. We use existing metrics to evaluate and compare non-functional properties (see [8] for a survey). But why metrics ? In the literature, several methods for defining and evaluating non-functional properties already exist (see [1] for a survey). But such methods usually focus on one specific property, or family of properties, for example quality of service, which is only a part of the whole software quality. Metrics may be applied to many families of properties, and allow comparisons. This is why we think that in our case, metrics represent the best method for comparing different non-functional properties.

2.2 Non-functional specifications

Elements of the component model are linked to elements of the quality model using a **non-functional specification** (noted **NFS**). An artifact may be related to several quality elements, so several NFSs belong to only one artifact. An NFS describes the effect of a quality characteristic on the artifact it belongs to, and uses the metric applied on the latter. Several NFSs of a same component artifact may share the same metric, but not the same characteristic. The set of an artifact's NFSs is called a **quality field**.

In Figure 1, the *resultValue* attribute of an NFS is given by the metric's measurement on the artifact. In the case of an ideal component, this attribute value is given by the application's designer.

2.3 Comparability of elements

We can try to compare two NFSs only if we can compare the artifacts they belong to. And we can try to compare artifacts of the same kind only. Two NFSs of comparable artifacts are comparable only if they measure the same characteristic (which means they use the same metric too, as one characteristic is measured by only one metric). Two NFSs are equal if they are comparable and their *resultValue* attributes are equal.

Two operations are comparable if their signatures are comparable. Two operations are equal if their signatures are equal modulo the renaming of the type names, and if their quality fields are equal.

A candidate provided interface PI_1 is comparable to a substitutable provided interface PI_0 if for each operation of PI_0 there exists a comparable operation in PI_1. A candidate required interface RI_1 is comparable to a substitutable required interface RI_0 if for each operation of RI_1, there exists a comparable operation in RI_0. Two interfaces (provided or required) are equal if their quality fields are equal and if, for each operation of one interface, there exists an equal operation in the other interface, and *vice versa*.

A candidate component C_1 is comparable to a substitutable component C_0 if for each provided interface of C_0 there exists a comparable provided interface of C_1, and for each required interface of C_1, there exists a comparable required interface of C_0. If C_1 is not comparable to C_0, it can not pretend to substitute C_0.

2.4 Weights and penalties

For each NFS, the ideal component's designer attaches a **weight** (or comparison weight) and a **penalty**. These two (absolute) values define the NFS's importance for the artifact it belongs to, the importance that the designer gives to it. The higher these two values are, the more important this NFS is, in the whole substitutable component. If a substitutable artifact owns an NFS and a candidate artifact owns a comparable one with a superior value, the candidate's chances increase proportionally with the comparison weight. Else, the penalty will be used to sanction this lack. A candidate component may also bring his own new NFSs that the substitutable component doesn't have. These new elements will be evaluated by the ideal component designer, who will choose a value for each one of them.

2.5 Substitution distance

The **substitution distance**, or distance, is defined using these weights, penalties, and NFS' *resultValue*s. This distance will inform on the substitutability of an NFS or an artifact. The best candidate for substitution is the one with the lowest distance. If the distance is negative, the candidate element can be considered as "better" (in terms of quality) than the substitutable one, according to the current context. If the distance is positive, then the candidate is worse. If

the distance equals to 0, then the two compared elements are "equivalent" each to the other, but it doesn't mean that they are equal.

Substitution distance between components is obtained with the sum of all the distances of each one of their comparable sub-elements. A more precise description of formulas is available in [7].

For each component, there is a **maximal distance** for substitution, fixed by its designer. Let us consider a component C_1, a candidate for the substitution of another component C_0. If the substitution distance between C_1 and C_0 is bigger than the maximal distance associated to C_0, then C_1 will be rejected.

3 Substitution in practice

Now let us take the example of an application that requires a Digital Video ("DV") camera component, with an interface for video stream and another one for camera control. It must also conform to the DV standard. This video camera example is taken from [3].

3.1 Modeling an ideal component

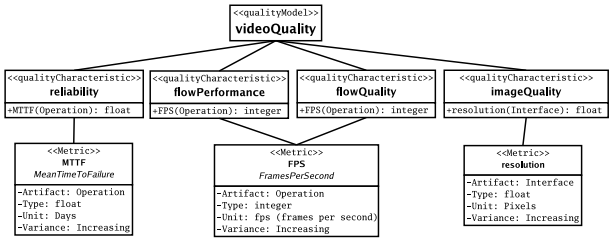


Fig. 2. Example of quality model.

The above requirements could be expressed by an ideal component called *videoCamera*. The latter contains a provided interface *videoStream* (with an operation $outputVideoFlow$), a provided interface *cameraControl* (with basic operations such as *on*, *record* and *eject*[1]), and a required interface $DVFormat$ (with an operation $inputDVFlow$ that asks for a DV tape).

The needs are not just about functional part, but also about non-functional properties and their respective importance. For example, we suppose that a high level of reliability for *record* and *eject* operations is required (so that the camera does not crash while recording, nor refuse to eject a video tape). We also assume that a high image quality, such as a 1 million pixels (1 MPixels) screen resolution,

[1] For simplicity and brevity reasons, we limit this provided interface to only three operations.

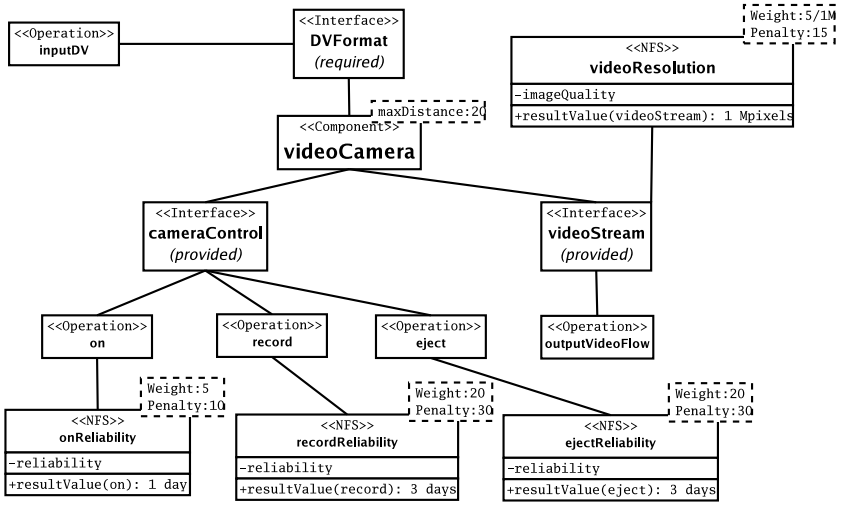


Fig. 3. Example of ideal component: *videoCamera*.

is required for *videoStream* interface. According to the quality model of Figure 2, we use the following characteristics: *reliability* and *imageQuality*. Their respective metrics are: $MeanTimeToFailure$ ($MTTF$) and *screenResolution*. Then we attach to the ideal component several NFSs. To each operation of the *cameraControl* interface, we attach an NFS using *reliability* characteristic (*onReliability* for *on* operation, *recordReliability* for *record* operation, and *ejectReliability* for *eject* operation). To *videoStream* interface, we attach the NFS *cameraResolution*, using the characteristic *imageQuality*.

Finally, the designer fixes expected $resultValue$s, weights and penalties for each NFS, and also fixes a maximal distance for the ideal component *videoCamera*. On Figure 3, we see that the expected value for *cameraResolution* is 1 million pixels, and the expected values for NFSs using *reliability* characteristic vary from operation to operation. The values required for *recordReliability* and *ejectReliability* are higher than those for *onReliability*. The penalties attached to *cameraResolution*, *recordReliability* and *ejectReliability* are very high in order to enforce candidate components to contain these NFSs. *cameraResolution* has a low comparison weight, which means that a big difference on the image quality is not very important. However, *recordReliability* and *ejectReliability* have higher weights, which means that a big difference on the reliability measurements of *record* and *eject* is very important. The maximal distance is fixed at a low level, so that the lack of one of these three NFSs in a candidate component will hardly be accepted.

3.2 Component lifecycle and substitution cases

Now that our ideal component is modeled, we can look for the best concrete candidate one to substitute it. Here are the different substitution cases:

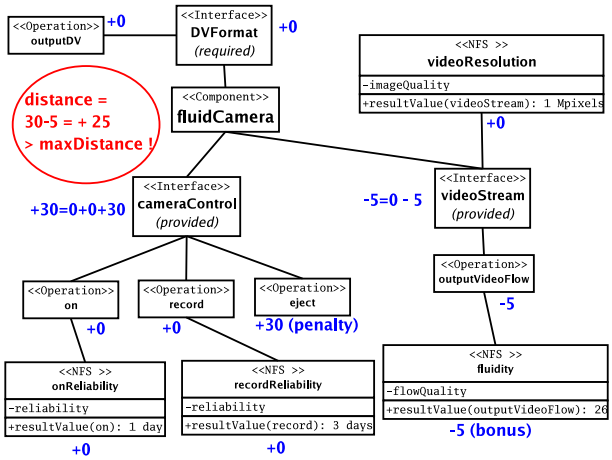


Fig. 4. Example of rejected candidate: $fluidCamera$.

First composition. Trying to plug a component into an application (in order to satisfy a given need) means trying to make this concrete component substitute the ideal one (corresponding to this need). Let us take the video camera example. Now that we modeled an ideal camera component, we have to check which concrete camera is the best candidate to substitute it.

First, according to our substitution model, a candidate must meet all the functional requirements, i.e. it must have all the ideal component's provided services (interfaces and operations), and must not bring more required ones. Else, it will be rejected even if it has a higher quality. For example, let us consider a $VHSCamera$ component meeting all functional requirements, but one (it requires VHS tapes instead of DV ones). No matter its quality, we need a camera that requires only DV tapes, and this candidate adds a required interface, so it is rejected.

Then, a candidate, like the $fluidCamera$ component on Figure 4, may add new NFSs unanticipated by the ideal component designer. For example video flow's number of frames per second. That corresponds to the metric FPS (for Frames Per Second), which measures $flowPerformance$ and $flowQuality$ characteristics (all of them are shown in Figure 2). It may be interesting to have a new NFS using $flowQuality$ characteristic on the $outputVideoFlow$ operation, but the candidate ($fluidCamera$) lacks an important NFS. The penalty is so high that it is rejected.

We can also have candidates providing at the same time some lower qualities, and other higher ones, than ideal component. In this case, a candidate component would rather have good "scores" in the most important NFSs. For example, let us take a candidate $goodImageCamera$ which has an excellent image quality (2 million pixels instead of 1 million) and an average reliability (2.5 days instead of 3 for operations $record$ and $eject$), while candidate $reliableCamera$ shown in

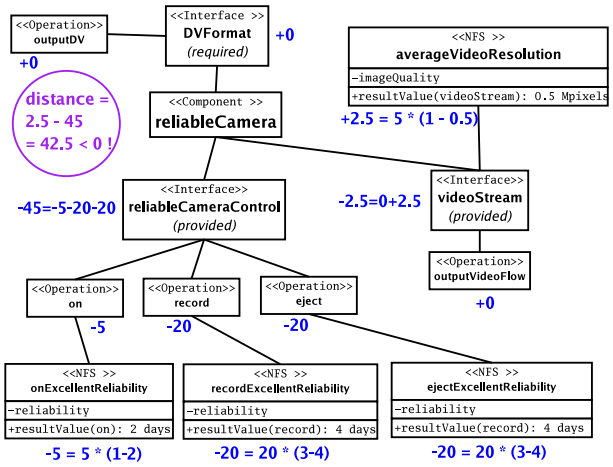


Fig. 5. Example of accepted : *reliableCamera*.

Figure 5 has an average image quality and an excellent reliability. We are not directly comparing them to find which one is "better" than the other. We are comparing each one of them, separately, with the ideal component, in order to find if it is an acceptable candidate. If we consider this ideal component, and the distance obtained for each one of the candidates, we can say that both are acceptable (distance with candidate *goodImageCamera* would equal to +15), but the *reliableCamera* is the best one.

Maintenance. The application now has its camera component, but it could have a "better" one. If the needs are the same, the ideal component that models them is exactly the same, but we can have new candidates. So we have to compare each one of them to this ideal component, ignoring the previous candidate. Otherwise, if the needs change, so does the ideal component. So this time, we must compare each candidate (including previous accepted one) with the new ideal component, providing "evolving needs" may mean several different things. For example, it can be the need for a new service, whether it is functional (a new artifact) or non-functional (a new NFS). Or it can simply be the need for re-evaluated qualities, which means a modification of the ideal component's expected values for its NFSs and/or a modification of its weights and penalties. In any case, we are brought back to the first composition scheme.

4 Related work

We said in introduction that substitutability was a well-known problem in object-oriented languages which include typing [5] and subtyping [12]. It is also an industrial problem, as referred in [14], who asks how to make sure that changes

on a component won't affect existing applications of a component, and try to answer by setting rules based on subtyping. It was tempting for us, in order to substitute components, to base our work on subtyping too. But as it was criticized [13] and accused of being too rigid and restrictive for componentware, and unable to deal with context, we preferred to try a more flexible approach.

Premysl Brada explored the notion of contextual substitutability [4], which consists in comparing a candidate component with a sub-component containing the "old" component's used part of its services (provided and required services that are bound to other components). Brada's substituability is "architecture-aware" and his context depends on its deployment in global architecture, whereas our approach is rather "need-aware", and our context considers an ideal component (modeling a need) and a concrete one which could substitute it.

Our substitution model was inspired by Zaremski and Wing's specification and signature matching for library components [16, 17]. We went further, by taking context and non-functional properties into account, and applying our substitution rules on generic component models. Also, our notion of weights can be compared to Scott Henninger's approach [9], that creates library "components" from keywords and places them into a valued network. However, our approach is at a different level, because we search and select candidates from components' structure instead of keywords. It can be used in such retrieval mechanisms in order to refine component search, and create more trustable libraries.

For our quality generic model, we were inspired by quality standards like ISO-9126 [11] and metrics standards like IEEE-1061 [10]. Example of existing metrics that could be used with our model can be found in [8, 2, 15]. But the quality part of our model can also be used with quality of service contracts languages, such as Jan Aagedal's CQML language [1]. In particular, our concern about substituting non-functional properties can be compared to CQML's substitutability of QoS "profiles". However, contrary to CQML, which, like most QoS languages, doesn't take functional aspects into account, our model combines functional and non-functional ones. And while Aagedal separates primitive component substitutability and composite component one, we deal with contextual substitutability of two components, no matter their internal structure.

5 Conclusion and future work

We proposed a substitution model including several elements: i) a generic quality model, able to use existing quality metrics in order to specify non-functional properties. ii) a generic component model, able to use existing research and industrial approaches. iii) a substitution distance, able measure the substitutability of a candidate component. We also introduced the notion of ideal component, that models functional and non-functional conceptual needs and takes the context of these needs into account. Right now, our substitution model is at the individual component level. A possible area of research is to bring it at architecture level.

In our current framework, we chose to consider one component model using existing quality characteristics and metrics from one quality model, because in the actual research and industrial schemes, composition concerns mainly components that come from a same component model. Other choices we made might change in the future. For example, the ideal component's current designer fixes and redistributes all the values, weights and penalties, which can lead to arbitrary decisions. This is why we are working right now on a normalization of metrics comparisons (considering values and units) instead of letting the designer assume everything in a risky way.

Right now, we have a tool [6] that allows us to check if a component can substitute another one according to our substitution distance measurement. This tool aims to help designers to find the best candidates for their needs.

References

1. J. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.
2. M. Bertoa and A. Vallecillo. Quality attributes for cots components. In *Proceedings of the ECOOP Workshop on QaOOSE*, June 2002.
3. G. Blair and J.-B. Stefani. *Open Distributed Processing and Multimedia*. Addison-Wesley, 1997.
4. P. Brada. *Specification-Based Component Substituability and Revision Identification*. PhD thesis, Charles University in Pragues, 2003.
5. L. Cardelli. Type systems. In A. B. Tucker, editor, *The Computer Science and Engineering Handbook*, chapter 97. CRC Press, 2004.
6. B. George. Substitute tool. http://www-valoria.univ-ubs.fr/SE/Substitute/, 2006.
7. B. George, R. Fleurquin, and S. Sadou. A component-oriented substitution model. In *To appear in : 9th International Conference on Software Reuse*, June 2006.
8. M. Goulao and F. B. e Abreu. Software components evaluation : an overview. In *CAPSI 2004*, November 2004.
9. S. Henninger. Constructing effective software reuse repositories. In *ACM TOSEM 1997*, 1997.
10. IEEE. *IEEE Std. 1061-1998 : IEEE Standard for a Software Quality Metrics Methodology*, ieee computer society press edition, 1998.
11. ISO Int. Standards Organisation, Geneva, Switzerland. *ISO/IEC 9126-1:2001 Software Engineering - Product Quality - Part I : Quality model*, 2001.
12. B. Liskov and J. Wing. A behavioral notion of subtyping. In *ACM Transactions on Programming Languages and Systems 1994*, 1994.
13. C. Szyperski. *Component Software : Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, second edition, 2002.
14. R. Van Ommering. Software reuse in product populations. *IEEE Transactions on Software Engineering*, 31 (7):537–550, july 2005.
15. H. Washizaki, H. Yamamoto, and Y. Fukazawa. A metrics suite for measuring reusability of software components. In *Metrics 2003*, 2003.
16. A. Zaremski and J. Wing. Signature matching : a tool for using software libraries. In *ACM TOSEM 1995*, 1995.
17. A. M. Zaremski and J. Wing. Specification matching of software components. In *ACM TOSEM 1997*, 1997.

Towards Task-Oriented Modeling using UML

Christian F.J. Lange, Martijn A.M. Wijns, and Michel R.V. Chaudron

Department of Mathematics and Computer Science, Technische Universiteit Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands,
C.F.J.Lange@tue.nl, MartijnWijns@gmail.com, M.R.V.Chaudron@tue.nl

Abstract. The Unified Modeling Language (UML) is a collection of 13 diagram notations to describe different views of a software system. The existing diagram types display UML model elements and their relations. This information is sufficient for the description of software systems. However, software engineering is becoming more and more model-centric, such that software engineers start using UML models for more tasks than just describing the system. Tasks such as analysis or prediction of system properties require additional information such as metrics of the UML model or from external sources, e.g. a version control system. In this position paper we identify tasks of model-centric software engineering and information that is required to fulfill these tasks. We present views to visualize the information to support fulfilling the tasks. This paper reports on industrial case studies and a light-weight user experiment to validate the usefulness of the proposed views that are implemented in our MetricView Evolution tool.

1 Introduction

The Unified Modeling Language (UML) is the de facto standard for modeling object-oriented systems. The building blocks of UML models are model elements, which are specified in the UML meta model [10]. These elements represent concepts of software programs or relations between them. Classes, methods, objects and messages are examples of UML model elements. The UML has 13 diagram types to visualize UML models. Each diagram type views a projection of a UML model from a certain perspective. Besides the model itself, today considerable amounts of related data are often available such as metrics [3], evolution data, documentation and problem reports.

Similar to the position taken in [9] we argue that in model-centric software engineering, views on the available data must be aligned with the tasks in which the views are used. The purpose of this paper is to identify available data, typical tasks and by proposing new views and visualization techniques, to improve the use of UML models for practitioners with respect to fulfilling these tasks. Furthermore, we report on industrial case studies and a light-weight user experiment to validate the usefulness of the proposed views that are implemented in our MetricView Evolution tool.

2 Task Oriented Modeling Framework

In this section the three underlying concepts of our proposed Task Oriented Modeling (TOM) framework and their relations are described. These concepts are: Properties, Views and Tasks and their relations are illustrated in figure 1. As a first step we create an initial overview of model element properties. Properties are in this case defined as characteristics of UML model elements. Then we define views and visualization techniques for the identified properties. After that a collection of tasks common to model-centric software engineering is presented. We don't claim that the examples given for each concept are complete, we rather present them to illustrate our approach and provoke discussion about the topic.

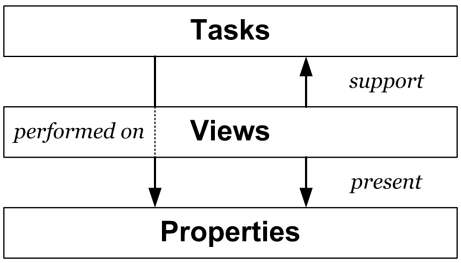


Fig. 1. The three underlying concepts and their relations

2.1 Properties

The UML meta model defines the model elements which UML models are constructed of. For each model element type, such as class, association, classifier instantiation, etc. a number of properties of the element are defined. We identify three different types of properties for model elements:
Direct Internal. Those properties of an element that are solely and directly based on information that is present in the model. General examples of this kind of property are the name of an element or the owner. Example properties for a class are its operations, its attributes and its relations to other classes.
Indirect Internal. Besides the information that is directly present within the model, we identify properties (or information) associated with model elements that can be derived based on the model. General examples of this kind of property are metrics and history data. For a class specific examples are the number of methods, the number of instantiations of the class or the complexity of the class (for example based on an associated state diagram).
External. A third type of property is based on information from outside the model. General examples of this kind of property include documentation, code facts, evolution data (e.g. obtained from a version control system like CVS ,

problem reports, change requests, and (empirical) data about characteristics of an element like its reliability. Specific examples for a class include: the number of lines of code of the implemented class, the number of times the class has been changed, or the name of the engineer who designed the class.

2.2 Views

Typically, UML models are visually represented in diagrams. The UML meta model defines a variety of diagram types as views on a model from a certain perspective. This specification is only concerned with internal properties and not even all of these properties are viewable in UML diagrams. The relations between model elements in different diagram types are often not intuitively presented by UML. It is, for example, often difficult and tedious to find out on which places a class is instantiated in the model, because this relation is not explicitly present in the views. We argue that in the design of the UML the choice of which properties can be viewed in UML diagrams and the visualization techniques used to represent them are not optimal for common tasks in software engineering.

Views offer visual representations of a model by by creating a mapping from properties of the model to visual attributes. Examples of these visual attributes are: Position (Layout), Size (Width, Height, Depth), Color (Hue, Saturation, Luminance), Shape and Orientation. In [8] several mappings from properties to visual attributes, called *polymetric* views, are explored. The main difference to our work is that polymetric views are general software visualizations targeted at reverse-engineering, while our work consists of UML model visualizations targeted at various model-centric software engineering tasks.

2.3 Tasks

Out of the set of UML model related tasks performed during the software engineering process, we have chosen the following six to form the basis for the implementation of the proposed views in our tooling and validation thereof:
Program understanding. Examples of activities related to this task are: identifying key classes, which classes implement which functionality, related classes and identifying complex interactions.
Model development. Creation of models is often an incremental and iterative process including many changes. Examples of activities employed in this type of task are: adding, changing, removing elements or diagrams.
Testing. An often used technique to improve the quality of a software system is testing. A testing task common in model-centric software development is the (automatic) generation of test cases from sequence diagrams.
Model maintenance. Changes made to a model to match changes in the requirements are called model maintenance. Some activities in which UML models are involved include: extension of a system, bug fixing, handling change requests and performing impact analysis before making a change.
Quality Evaluation. Another type of task in which UML models can be of help is quality evaluation. The evaluation of the quality of a model can take place

at several abstraction levels, e.g. separate elements, diagrams or the system as a whole. Besides evaluating a single version of a model one can also investigate multiple versions at once, to detect trends.
Completeness / Maturity Evaluation. Related to quality evaluation is the evaluation of the completeness of a model. In general this task consists of applying specific metrics to measure these properties and analyzing the results.

3 Proposed Views

In this section ideas for visualizing the aforementioned properties of UML model elements are listed. Some of these visualizations have existing UML diagrams as a basis, others are totally independent. The initial set of ideas was stated in [5]. Since then, it has been extended during the MetricView and MetricView Evolution [11] projects. Most of the ideas are implemented in our tool [1].

3.1 Context View

The context of a model element consists of all model elements it relates to. The model elements of a model are typically scattered over several diagrams. UML diagrams are projections of the entire model, they typically do not contain all model elements. Accordingly, it often occurs that only a limited context of model elements is viewed in one diagram. To fully understand a model element it might be necessary to know its entire context. Therefore we propose the *context view* comprising a single diagram (Figure 2). The model element whose context is viewed is centered in the diagram. All model elements that are directly related to the particular model element are viewed on a circle around this model element. The example in the figure is a class where the metric 'number of children' is 28 (for explanations of common object oriented system metrics see [2]). It would be tedious to analyze this outlier by browsing through all diagrams where inheritance relations of the class are viewed.

3.2 Quality Tree

Quality models provide a structure to relate metrics to quality properties such as maintainability. The most common approach to create such models is used in so called decompositional quality models. At the lowest level, at the leafs of the tree, are metrics. These metrics can be applied to a UML model and the results be used to calculate values for each of the attributes in the model. For this calculation a definition is needed for the nature of the relation between attributes. These definitions are functions, and although not currently implemented, would allow the rule based learning quality model as suggested in [4]. The quality tree offers a framework in which a reference model can be tailored to represent several quality models, such as our proposed quality model for UML [6]. This tailoring is possible in several ways: by changing the structure of the tree, by changing its connecting functions and by changing the metrics in the leafs.

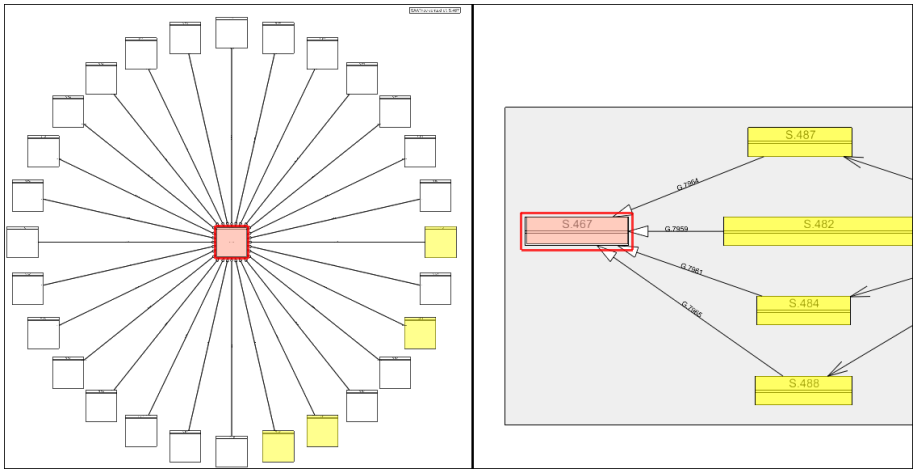


Fig. 2. Context Diagram (left) showing all the children of a single class, compared with regular class diagram (right)

3.3 MetaDiagram

Figure 3 shows a *metadiagram* in which inter-diagram relations are visualized. The purpose of this is as follows.

The main problem that exists in conventional views is that each of the diagrams is shown separately, obscuring the relation between different diagrams and model elements. Our proposed solution to these problems is the *metadiagram*. Its purpose is to give an overview of the diagrams that describe the model and makes it possible to show the relations between (elements at) different diagrams. This last feature allows tracing through the different abstraction levels that the different types of diagrams offer.

Figure 3 shows the four types of elements that take part in this example. A use case, an object that occurs in the sequence diagram describing the use case, the object's class, and the state diagram describing the class.

The *metadiagram* can be applied in program understanding and maintenance tasks. Browsing through a model for instance is a program understanding task that is actively supported by this view. Another example is impact prediction for which the visualization of inter-diagram relations can be useful.

3.4 MetricView

Figure 4 shows an example of the proposed *metricview* visualization on the right in which three different metrics are visualized on a regular class diagram. This is the basis of the *metricview* idea, combining the visualization of metrics and UML models using a set of techniques adopted from visualizing geographical information systems (GIS) [5]. Applying metrics to a UML model can generate

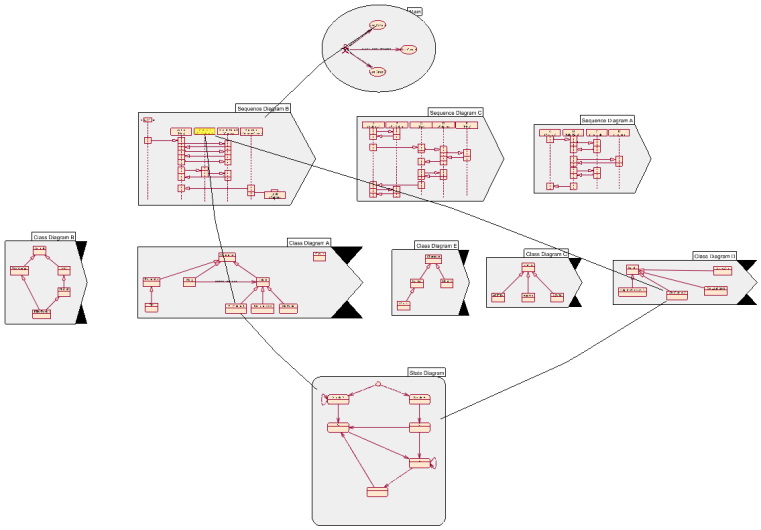


Fig. 3. MetaDiagram: Tracing through different diagrams

an overwhelming amount of data, even for small models. Traditional methods to process and visualize this data like statistical analysis and various kind of charts work well to summarize data and to find outliers but lack the direct connection to the model the metrics are calculated over. This makes it harder for the creator or maintainer of a model to relate the results of this kind of metric analysis to the artifact. *Metricview* helps to solve this problem by integrating the model and metric visualization. Visualization techniques include but are not limited to color, size and/or shape. The tasks supported by this view are: program understanding, quality evaluation and maturity/completeness evaluation. Metrics such as coupling, complexity, or the number of changes can be visualized, such that the reader can intuitively identify clusters of classes in the model.

3.5 UML-City View

The left part of figure 4 shows an example *UML-city view*. This view combines the concepts of the *metadiagram* and *metricview*. As metric visualization the '3D-heightbar' is used, this visualization shows a box on top of the model element where the height and the color of the box indicate the value of the metric. Low metric values are depicted by flat green boxes while high values are depicted by tall red boxes.

3.6 Search and Highlight

In any UML model, but especially in large ones, it can be hard to find a specific piece of information. This problem is caused by the large amount of informa-

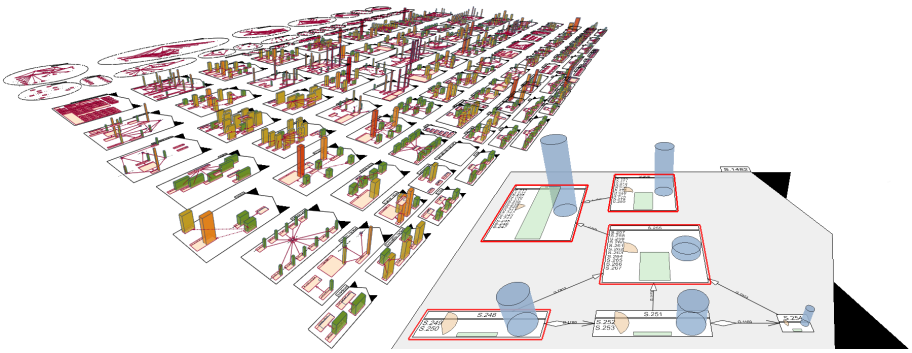


Fig. 4. UML-City View and MetricView: Combining UML and Metrics Visualization

tion and this information being spread over multiple diagrams. Our solution for tackling these causes uses the *metadiagram*, which gives an overview of all diagrams. By adding string search functionality and highlighting the results in the *metadiagram* the user can quickly identify the diagrams in which relevant information is present. Combined with the navigation capabilities that are present in the *metadiagram* this supports fast searching. When highlighting all elements that are related to a specific keyword the tool also shows implicit (not modelled) relations between these elements themselves.

3.7 Evolution View

Figure 5 shows the *evolution view*, in which the two familiar concepts graph and calendar are combined to identify trends. The reason for using a graph is that it is an effective way to visualize the evolution of metric data. The purpose of the *evolution view* is to enable users to spot trends in the values of quality attributes and/or metrics at multiple abstraction levels. At system level such a graph can be used to give an overview on changes in aggregated data. By combining it with the concept of a calendar, i.e. mapping time on the horizontal axis and values of the vertical axis, and adding color to indicate whether a given value is considered good or bad it becomes a compact and intuitive way to enable the evaluation of the evolution of quality data. The same technique can be applied at diagram and element level to allow for different analysis granularity.

4 Validation

4.1 Case Studies

Characteristics. Our five case studies stem from different industrial application domains. The size of the models ranges from 36 to 606 classes, the size of the view from 16 to 606 diagrams.

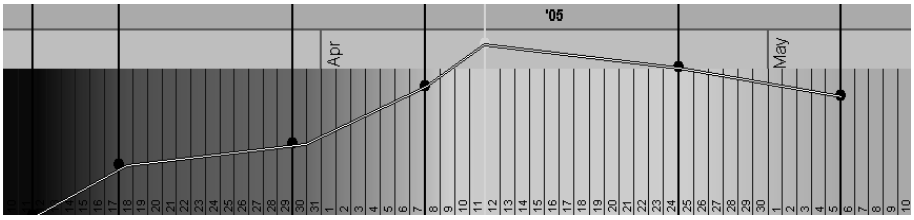


Fig. 5. Evolution View: Combining Calendar and Graph

Approach. First, an analysis is performed of the model using MetricView Evolution and tools that provide external data. The tool is then used to demonstrate a visualization of a selection of the results of the analysis. This happens at the partner site from which the model originates. During and after the demonstration there is discussion about the findings in the model and the visualization.

Findings. The visualizations received positive feedback during the discussions and were regarded as a promising direction for future research. The first large case used to validate MetricView Evolution revealed some scalability issues. For large cases with many diagrams there is little space available for each individual diagram. Space efficient layouting helps to some extent to reduce the effects of this problem. Additionally, we implemented zooming functionality in the tool to allow users to have both a bird's eye view and a closer view. It also turned out to be hard to find information about a specific model element if only (part of) the name was known but not the diagrams it occurs in. To assist with this task the search and highlight functionality proved to be very helpful.

Furthermore, we found that human expertise remains necessary. This follows the position taken in [7]. The tool alone should not make a judgement about the quality of a UML model. Instead, it helps the user to apply his expertise to come to a good judgement by providing him with an appropriate view on the properties of the model.

4.2 User Experiment

Design. During the experiment 13 subjects have evaluated the tool. The subjects were researchers and PhD students in the area of software engineering or visualization and, hence, they had relevant experience to evaluate the tool. To make the participants familiar with the tool, first a demonstration was given of its features and intended usage. Then the participants were asked to perform specific tasks. These tasks were put in the form of answering a number of questions, such as "Which class plays the major role in the implementation of the 'Initialization' Use Case?" The idea behind these questions is to let the participants explore the tool. During the evaluation the 'speak-aloud-protocol' was followed, meaning that the subjects were encouraged to ask questions and make remarks about what they are trying to do and how they feel about the

tool. Additionally, the subjects had to fill in a questionnaire concerning their evaluative findings about the tool and their background.

Results. The background questionnaire shows that the participants have sufficient knowledge in the areas relevant to the experiment.

An interesting suggestion we received is related to the *evolution view*. We received the remark that this view shows great potential and should be the starting point from which to explore the quality of a model. For this to work better it should be possible to show trends of multiple metrics or quality attributes together in different colors. Also showing the actual values in the *evolution view* together with the graph is a suggestion we got from multiple users.

The highest rated features with respect to usefulness both for correctly and efficiently performing tasks are: *metadiagram*, *search and highlight* and *context view*. The histograms in figure 6 show an overview of the results: for each task type what percentage of the users found a particular feature useful. The histograms show the cumulated answers to yes/no questions (e.g. "Is view X useful to perform task Y correctly?")

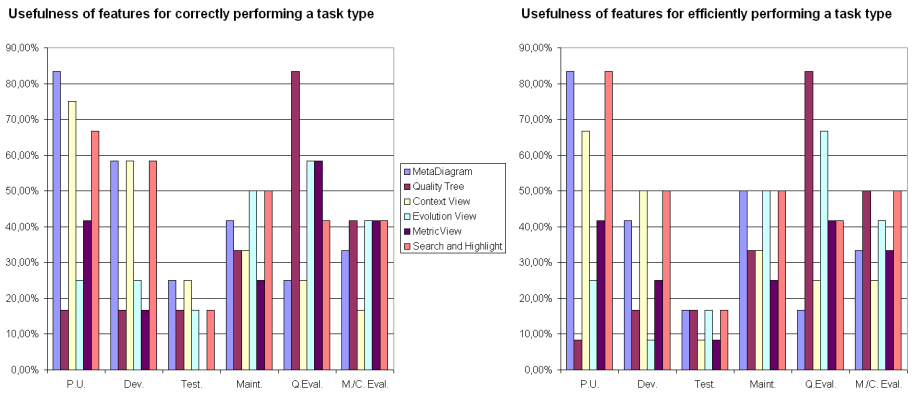


Fig. 6. 'P.U.': Program Understanding, 'Dev.': Development, 'Test.': Testing, 'Maint.': Maintenance, 'Q. Eval.': Quality Evaluation, 'M./C. Eval.': Maturity/Completeness Evaluation

The evaluation of the understandability of the tasks shows that the *quality tree* was the easiest to understand. Of the four main views, the *context view* was hardest to understand. Observations made during the experiment reveal that most users had problems activating the *context view*, because it involves actions in multiple windows that have to be performed in the right order. These observations also made clear that once the *context view* was activated participants had little problems using it.

5 Conclusions

In this paper we state the problem that for some common model-centric software engineering tasks representations of UML models are not sufficient. We propose an initial Task Oriented Modeling framework consisting of UML model elements, properties of these elements from various sources, and software engineering tasks, that form a basis to develop new views that are aligned with the tasks. Based on this framework we propose seven views to support different tasks. The views are implemented in our MetricView Evolution tool. In industrial case studies and a light-weight user experiment the proposed views were evaluated and we received positive feedback from the users.

In future work software engineering tasks should be analyzed in more detail to refine our proposed framework. We expect that this will lead to the development of more specific views. Our proposed views and future views should be validated empirically. Another point is the integration of the tool in the daily-build system at the site of an industrial partner. Doing this would allow us to automatically gather the data needed to study the evolution of quality of UML models. Another item of future work is adding a filtering mechanism to the views, which should make it easier to locate and analyze specific information.

References

1. MetricView and MetricView Evolution. Available on: `http://www.win.tue.nl/empanada/metricview/`.
2. S. R. Chidamber and C. F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.
3. Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics, A Rigorous and Practical Approach*. Thomson Computer Press, second edition, 1996.
4. Khashayar Khosravi and Yann-Gaël Guéhéneuc. Open issues with quality models. In *proceedings of the* 9^{th} *QAOOSE workshop(ECOOP)*, July 2005.
5. Christian F. J. Lange and Michel R. V. Chaudron. Combining metrics data and the structure of UML models using GIS visualization approaches. In *Proceedings of the ITCC 2005*, April 2005.
6. Christian F. J. Lange and Michel R. V. Chaudron. Managing model quality in UML-based software development. In *Proceedings of STEP05*, 2005.
7. Guillaume Langelier, Houari Sahraoui, and Pierre Poulin. Visualization-based analysis of quality for large-scale software systems. In *Proc. of ASE '05*, 2005.
8. Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003.
9. Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Čubranić. The emergent structure of development tasks. In *ECOOP 2005*, July 2005.
10. Object Management Group. *Unified Modeling Language, Adopted Final Specification, Version 2.0*, ptc/03-09-15 edition, December 2003.
11. Martijn A. M. Wijns. MetricView Evolution: Monitoring Architectural Quality. MSc thesis, Technische Universiteit Eindhoven, The Netherlands, March 2006.

Animation Coherence in Representing Software Evolution

Guillaume Langelier, Houari A. Sahraoui, and Pierre Poulin

{langelig, sahraouh, poulin}@iro.umontreal.ca

Abstract. Software evolution study is crucial to the understanding of software in general and software quality in particular. However, the study of software evolution requires the analysis of large amounts of data, i.e., source code information for every single version of a program. Considering the size, manual analysis of this information is virtually impossible. Automatic analysis is quick, but requires strong assumptions on data which is hard to establish in our domain. We propose a semi-automatic approach based on visualization to represent software versions. We use animation to represent the transitions between versions. We exploit the coherence between two successive versions and we transform it into visual coherence that can be perceived by the user. Our solution is interesting because the movement in the graphical representation can be aligned naturally with the code modifications. It also reduces the required space in 3D by using a fourth dimension which is time.

Note: Most figures of this paper should be viewed in color to better understand their perceptual values. Figures and animated sequences about the proposed techniques, can be found in http://www.iro.umontreal.ca/~labgelo/qaoose06.

1 Introduction

Nowadays, programs are becoming more and more large and complex, which makes the maintenance tasks costly and time-consuming. Moreover, while in the past, small teams were building programs that meet functional needs with the concern of optimizing computer resources, the focus has shifted today to the optimization of financial resources. In this context, it is important to understand the factors that influence software quality for anticipating potential problems. Much work has been done to predict maintainability factors using software internal attribute metrics [4]. However, many problems related to quality cannot be easily understood by analyzing a single version of a program. Most of the time, it is possible to see the symptoms but not necessarily the causes. Evolution study is essential to understand the design decision sequences that can result in a problem.

With the advent of Internet and open source programs, there are now large data samples for studying software evolution. However, fully automatic approaches require strong assumptions on data and often generate many false

positives [12]. From the other hand, the size of these samples makes it very difficult to perform analysis tasks efficiently by human. In order to fully exploit the analytic capabilities of humans, it is required to preprocess these data. A good way to do so is to use visualization. Visualization is a semi-automatic approach that combines the automatic data preprocessing and presentation with human visual system capabilities.

In this position paper, we show how it is possible to use visualization and animation to view the different versions of a program throughout its lifetime. We explain that the simple solution that consists in showing in a single picture a sequence of several versions is not efficient because the user loose track of individual elements and has to re-understand the structure of the program each time. We animate the transitions with in-between frames to help users follow entities. We also highlight the fact that the visual coherence is the key element that helps detecting changes from one version to another.

The rest of this paper is organized as follows. Section 2 introduces the principles of single version visualization in our approach. Section 3 describes why coherence is a major point in evolution visualization. Different visualization techniques that exploit the coherence are presented and compared in Section 4. Their possible applications are briefly described in section Section 5. Section 6 gives a brief overview of exiting work in evolution visualization and layout algorithms. Finally, Section 7 gives a short conclusion and presents the future work.

2 Single Version Visualization

2.1 Class Visualization

In order to simplify the representation of programs and to make the analysis easier, we decided to use a vector of metrics as an underlying model for classes. This model has proven to be efficient for quality analysis, i.e., statistical transformations are easily performed on the metric values. Since we target programs in Java, we use object oriented metrics (see for example [1]). We use metrics that capture important OO attributes: coupling (such as CBO, Coupling Between Object), size/complexity (such as WMC, Weighted Methods per Class), cohesion (such as LCOM5, Lack of Cohesion in Methods) and inheritance (such as DIT, Depth in Inheritance Tree). Metrics are extracted from Java programs using a home made tool called POM [5]. POM generates an Xml file which can be read by our environment. This makes this later independent from the static analysis tools and to a certain extent from the OO programming languages.

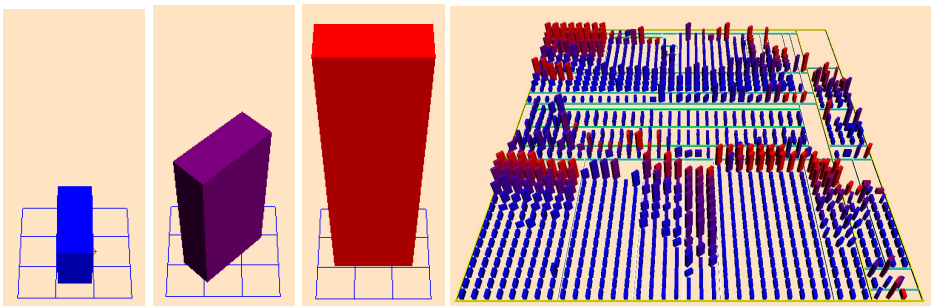
Classes like any code artifact do not have any natural representation [8]. Their intend is to be understood by human and machines and they have no concrete reality outside these purposes. Therefore, we have to represent classes with arbitrary figures. We chose to represent classes as 3D boxes. A box is simple and has several non-interfering graphical attributes (See [6] for a discussion on biases and interfering graphical attributes) such as the height, the color and the twist. Moreover, it can be easily rendered by graphical cards and easily processed by the human brain.

A linear mapping binds a metric and a graphical attribute, with clamped metric values above a maximum M_{max} or below a minimum M_{min}. Formally, the mapping is defined as follows:

$$M'_v = \begin{cases} M_{min} & M_v < M_{min} \\ M_v & M_{min} \leq M_v \leq M_{max} \\ M_{max} & M_v > M_{max} \end{cases}$$

$$G_v = G_{min} + (G_{max} - G_{min}) \left(\frac{M'_v - M_{min}}{M_{max} - M_{min}} \right)$$

where M_v is the metric value, G_v is the resulting graphical value, G_{min} and G_{max} are respectively the practical minimum and maximum values of the graphical characteristics, and M_{min} and M_{max} respectively the minimum and maximum practical value for the metric. The user can easily choose any mapping that suits him. Figure 1(a) shows three representations of classes.



(a) (b)

Fig. 1. (a)Three examples of class representation using the mapping between metrics CBO, LCOM5, and WMC, and graphical attributes color, twist, and height, respectively. (b)Representation of the *PCGEN* application (1129 classes) using our adapted *Treemap* algorithm.

2.2 Program Visualization

We use the layout of classes to express additional information on the architecture of a program. This layout also contributes to the comprehension of programs as entities instead of groups of elements. The layout technique used is highly inspired by the *Treemap* algorithm [7] which is useful to represent file system hierarchies. Since Java programs use classes and packages that can be included into other packages, this representation is suitable for our needs. The idea is to use the rectangle of the screen as the root element in the hierarchy. After that, we simply split the screen vertically and give a portion to each child which is proportional to its size. The process is then repeated recursively for children

alternating the splitting direction between horizontal and vertical at each level. However, this algorithm cannot be used "as is" because it only works for continuous values. Indeed, our representation uses discrete values (each class occupies the same space on the plane). We simply let the packages take as much space as they need and readjust their parent size afterward. Because of the 3D and the navigation possibilities, we are not constrained with the screen space anymore. We use color separators to better differentiate the levels in the *Treemap*. Using this layout, it is possible to see common characteristics in a package and to study packages as entities. Figure 1(b) gives an example of how programs are displayed in our environment.

In addition to program layout, we offer a navigation system to let the user zoom in on more important parts and move around the program. The view angle can be changed so it is possible to view hidden classes. The camera can move around the semi-sphere over the plane and move in all four directions. Filters are also available to either give information on the statistical distribution of metric values or give structural information on each class. In this second case, a filter is a dynamic way of displaying UML-like relationships between classes. More details on single version visualization can be found in [9].

3 Software Coherence

Building software is an incremental process which takes place over time. Therefore, modifications from version v_n to v_{n+1} are highly influenced by v_n. A class is created or modified because there was something missing in the previous version or because the previous version was not ideally implemented. Most of the time, the difference between two successive versions is small and targeted. Therefore, there exists some coherence intrinsic to software and we think that a good visualization system should exploit this coherence.

The study of evolution should take into account the chronological order of events in order to capture essential information. Users are interested not only in global information, but also on what happens between two versions, and how it is possible to improve this transition. It is important to know what was modified and how it was modified but also what was kept intact for long time. In this paper, we concentrate on transferring the coherence of software into visual coherence.

4 Approach

The main idea of all the techniques described below is to use animation to represent transitions from one version to another. To do so, we use a principle called the "time slider" to give the user the opportunity to control time. The idea is to consider time like any other dimension and the user can go back and forth the same way he does for the Cartesian coordinates. This way, specific transitions can be replayed or passed over quickly depending on the user needs. As stated by Rilling and Mudur [15], the feeling of immersion is important for

any 3D visualization systems. This should also be true for the time dimension. All algorithm variations below use the same principle but use different way to display transitions and the versions themselves. For comparison purpose, we first present a non-animated technique.

4.1 One Image; Many Versions

This solution is used by many of the current research projects on software evolution. It allows seeing all the necessary information at a glance. It shows all the information on one image; therefore a lot of space is required to represent only few versions. On the other hand, data can be summarized to save space. However, in our opinion, this is a source of many analysis errors mainly because of the many constraints that we have to take into account when combining measures.

Using this technique, both coherence and metric modifications are difficult to understand because we cannot follow elements through time. When represented explicitly, the architecture is hard to understand because many different ones are present at the same time.

4.2 Fixed Position

In this technique, classes remain at the same position in the layout for each version displayed in the visualization. In order to assign each class a specific position, we create a virtual architecture tree containing all classes of all the versions of the program and simply apply the *Treemap* algorithm on this tree. We then only display classes present in a particular version whenever the user choose this version using the time slider. Since classes are not moving, the only animations required are the deletion and the adding of classes and the modification of class characteristics. Indeed, the modification of code usually leads to the modification of the metric vector which in turn leads to the modification of graphics. This implies that for a given version v_n some space in the visualization is left empty. The first reason is that we must keep some space for classes that will be created in versions subsequent to v_n and the second reason is that classes deleted from the system are not replaced by other classes.

Since objects are not moving, this technique gives good results in terms of temporal coherence. Characteristics modifications are also very easy to understand because you can concentrate on them. However, this technique demonstrates a bad use of space, especially for earlier versions of a program. The space lost is negligible for the latest versions because only a few classes disappeared in general. The architecture understanding requires some effort because the user must not take into account the empty packages and the empty space in packages. Figure 2(A) shows a few frames of this algorithm applied on the *Freemind* application.

4.3 Moving Classes

For this technique, the layouts are computed normally for all versions. During transitions, classes are slowly translated to their next position. This way, indi-

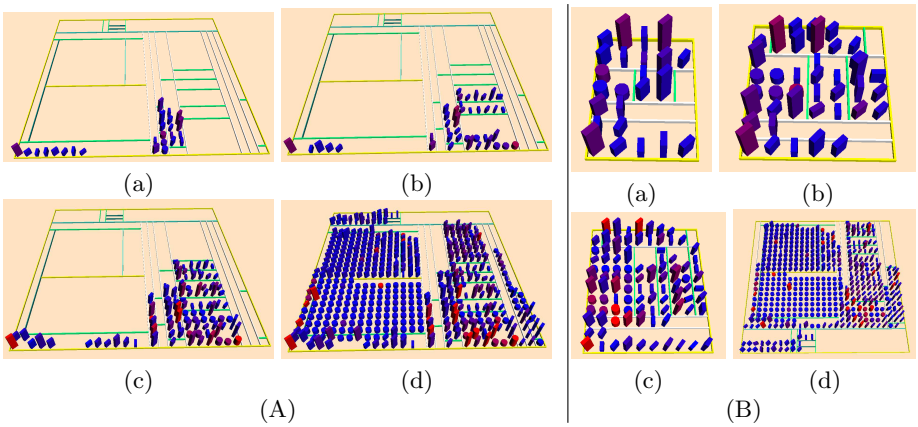


Fig. 2. (A) Algorithm with pre-calculated tree where classes have a stable position throughout de visualization. (B) Algorithm with moving classes representing the same software. To better perceive the differences between the these two techniques, please consult the web site mentioned earlier to see the animated sequences corresponding to each technique.

vidual layouts are preserved and space is used optimally. However, as mentioned in [3] animating both the movement and the characteristics modification (height, color, twist) at the same time overwhelms the human perceptual system. The solution is simply to animate the transitions in two phases: the movement and then the characteristics modifications. This leaves time for the user to first understand the layout modifications and then observe the characteristics modifications. Note that the two-step transition can be applied also to the fixed position technique.

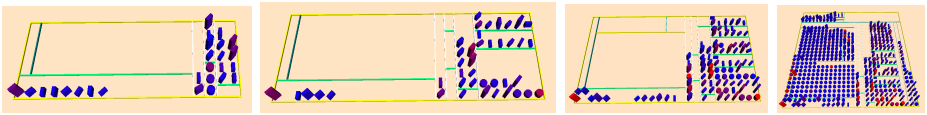
Unfortunately, the algorithm can introduce unnecessary movements during the visualization. For example, classes added in the middle of a package will automatically displace some existing classes in the program. We circumvent the problem by forcing the classes to stay at the same place whenever it is possible. When doing this, movement within a same package is reduced and the user can concentrate on real differences in the classes' layout.

Another property of this technique is the optimal use of the space. Although some holes can be present because of the *Treemap* limitation, the layout of each version is computed individually regardless the other versions. In opposition, the biggest shortcomings of this technique are in the time coherence. It is still possible to follow the classes throughout the versions but this requires more attention and effort from the user even if the fact that we force classes to keep the same position whenever it is possible helps reducing this effort. Moreover, since the movement and the modification of class metrics are done in two different steps. Users can concentrate only on movement at first which helps them even more. Similarly, the process in two steps helps to better perceive and interpret the metric modifications as it was the case for the previous technique. Figure 2(B) presents some screenshots of this technique.

4.4 Hybrid

This technique combines the two approaches described previously. Classes are placed according to a virtual tree containing all classes that ever existed. However, instead of displaying the empty space, separators are placed such that some empty packages are not represented. This way, classes have a fixed position but the space loss in the early versions where a few classes are present is diminished.

This technique has similar advantages and disadvantages than the fixed position approach. It is slightly better for space optimization because many empty packages are removed. The architecture comprehension stays a bit confuse because unnecessary packages are still present. Figure 3 shows an example of this approach.



(a) (b) (c) (d)

Fig. 3. Example of the Hybrid approach. A corresponding animated sequence is in the web site.

4.5 Summary

To better highlight strengths and weaknesses of each discussed techniques, we present a summary of their qualitative evaluation. The evaluation is done according to four characteristics: space optimization, temporal coherence, comprehension of characteristics modification, and architecture comprehension. Table 1 summarizes the observations on each technique.

Algorithm	**Space**	**Temporal coherence**	**Characteristics**	**Architecture**
side by side	bad	bad	very bad	average
fixed position	average	very good	very good	average
moving classes	very good	bad	good	very good
Hybrid	good	good	very good	average

Table 1. Comparative table of evolution visualization techniques

5 Possible Applications

Our environment can be used by developers or quality analysts to see whether or not the new code respects the quality general principle rules. For example, classes

may have grown out of proportion or the coupling or cohesion may have been affected by the new modifications. Developers are also able to detect at what point things started to degenerate or if a problem was caused by a single commit or is the result of a long process. Similarly, users can detect class renaming. When a class is renamed, the box associated with it disappears and reappears at another position. In general, a class rarely changes dramatically between two successive versions. Its characteristics then remain unchanged. Consequently, as the shape of its associated box almost does not change, the disappearance and reappearance are perceived as a movement.

Researchers can also use our tool. It is interesting to study if a recurrent problem is caused by the same sequence of events. By doing this, it is possible to classify some patterns of software evolution or study their impact on the quality. We have already started to investigate this direction. Our tool can also be used to confirm or to accelerate the verification of phenomena found with automatic algorithms.

6 Previous Works

6.1 Evolution Visualization

Lanza and Ducasse [10] presented a tool which takes the form of an evolution matrix. They use 2D boxes in order to represent two metrics at time. One is associated with the height and the other is associated to the width. Those boxes are then placed in a matrix in which the columns represent versions in chronological order and the rows represent each class. They also described a classification of software elements according to the representation of their lifecycle. This classification is based on an astronomy metaphor. Mesnage and Lanza [11] chose to utilize a standard scatter plot representation for versions visualization, however they use 3D boxes instead of points. Their tool, *White Coasts*, is useful to interpret information extracted from version control systems. They use two main views: the author view and the evolution matrix view. Metrics are represented by several graphical attributes from the 3D box such as the color, the position (X, Y, Z) and the size (width, height, depth). Collberg *et al.* [2] use traditional graphs to represent information extracted from CVS repositories as well as inheritance graphs and call graphs of a program. They use multiple frames to represent every modification in a given time frame. A technique of weighted nodes and edges insures smooth modifications of graphs during the evolution.

6.2 Coherence Between Frames and Layouts

Nguyen and Huang [13] propose a technique to animate layout in a standard tree representation(arcs and nodes). If the user choose to inspect a node, their algorithm execute a transition to go from a larger view to a view where the selected node is the root; thus the center of attraction. The chosen node slowly goes up in the hierarchy taking the place of its predecessors while its children follow it

taking the space it was occupying in the previous step. Fekete and Plaisant [3] briefly discussed the animation of Treemaps. They use the fact that objects contain other objects to accelerate the animation and do linear transformation in two phases. They change the object position if they have to and then change their size. They do not introduce any new elements either. They transform their representation into a more complex one or they modify the metric observed, which in turn, modifies the size of squares. North [14] proposes heuristics to create an incremental layout of nodes in a directed graph. Nodes and arcs are either marked modifiable or not and he uses backtracking whenever necessary. The objective is that each node keeps a static position, however the movement is inevitable in some cases. He measures the node stability in order to validate his results.

7 Conclusion and Future Work

In this paper, we have shown that it is possible to use coherence through visualization for a better comprehension of software evolution. We have described how individual versions are represented before going to the representation of several versions. To circumvent the problem of cognitive discontinuities, we have used animation in different ways. All the proposed techniques were compared in order to identify the advantages and disadvantages of each one. All the techniques have shown strengths and weaknesses in terms of space utilization, temporal coherence, comprehension of characteristics modifications, and architecture comprehension. Some techniques can be better for certain analysis tasks while being worst for others. Techniques without class movement offer a better comprehension of characteristics changes but lacks in terms of architecture comprehension. However, techniques with moving classes tend to better use the space and reveal more information about architecture modifications. The presented techniques are novel because very little work is dedicated to the use of animation to represent software evolution. From our experience, the animation contributes to reduce cognitive discontinuities caused by the versions switching.

Many extensions of the proposed approach are possible. First, we can give more control to the user via the time slider principle. The comprehension would be improved if the user was able to stop in the middle of transition or choose the speed of the transition. Moreover, it should be possible to skip versions in order to visualize the modifications between more sparse versions if necessary. The animation of separators would also give the user a better feeling of how packages are reorganized.

We are currently developing a new layout based on relaxation to better exploit the coherence between versions. When we animate the evolution, classes can be introduced anywhere without major perturbations to the rest of the software.

From the evaluation perspective, an empirical comparative study with subjects will be conducted in order to confirm the qualitative results presented in this paper. The environment will also be compared to existing approaches. More-

over, many visual patterns must be described in order to prove that our approach is efficient in solving concrete evolution problems.

References

1. S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.
2. C. Collberg, S. Kobourov, J. Nagra, J. Pitts, and K. Wampler. A system for graph-based visualization of the evolution of software. In *SoftVis '03: Proceedings of 2003 ACM symposium on Software visualization*, pages 77–86, New York, NY, USA, 2003. ACM Press.
3. J.-D. Fekete and C. Plaisant. Interactive information visualization of a million items. In *INFOVIS '02: Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, page 117, Washington, DC, USA, 2002. IEEE Computer Society.
4. N. E. Fenton and S. L. Pfleeger. *Software Metrics : A Rigorous and Practical Approach*. Course Technology, 1998.
5. Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In E. Stroulia and A. de Lucia, editors, *proceedings of* 11^{th} *Working Conference on Reverse Engineering*, pages 172–181. IEEE Computer Society Press, November 2004.
6. C. G. Healey and J. T. Enns. Large datasets at a glance: Combining textures and colors in scientific visualization. *IEEE Transactions on Visualization and Computer Graphics*, 5(2):145–167, 1999.
7. B. Johnson and B. Shneiderman. Treemaps: A space-filling approach to the visualization of hierarchical information structures. In *IEEE Visualization Conference*, October 1991.
8. C. Knight and M. Munro. Virtual but visible software. In *IV '00: Proceedings of International Conference on Information Visualisation*, pages 198–205, July 2000.
9. G. Langelier, H. A. Sahraoui, and P. Poulin. Visualization-based analysis of quality for large-scale software systems. In *ASE'05: IEEE/ACM International Conference on Automated Software Engineering*, pages 214–223, November 2005.
10. M. Lanza and S. Ducasse. Understanding software evolution using a combination of software visualization and software metrics. In *LMO'02: Proceedings of Langages et Modèles à Objets*, pages 135–149, 2002.
11. C. Mesnage and M. Lanza. White coats: Web-visualization of evolving software in 3d. In *VISSOFT'05: Proceedings of 3rd International Workshop on Visualizing Software for Understanding and Analysis*, pages 40–45, 2005.
12. P. F. Mihancea and R. Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In *CSMR'05: Proceedings of 9th European Conference on Software Maintenance and Reengineering*, pages 92–101, 2005.
13. Q. V. Nguyen and M. L. Huang. A space-optimized tree visualization. In *INFOVIS '02: Proceedings of the IEEE Symposium on Information Visualization (InfoVis'02)*, page 85, Washington, DC, USA, 2002. IEEE Computer Society.
14. S. C. North. Incremental layout in dynadag. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 409–418, London, UK, 1996. Springer-Verlag.
15. J. Rilling and S. Mudur. 3d visualization techniques to support slicing-based program comprehension. *Computers & Graphics*, 29(3):311–329, 2005.

Computing Ripple Effect for Object Oriented Software

Haider Bilal[1] and Sue Black[1]

[1] Centre for Systems and Software Engineering
Department of Software Development and Computer Networking
Faculty of Business, Computing and Information Management
London South Bank University
{bilalhz, blackse}@lsbu.ac.uk

Abstract. Software metrics can provide us with information regarding the quality of software. The ripple effect metric shows what impact changes to software will have on the rest of the system. It can be used during software maintenance to keep the system at a high level of quality. The computation of ripple effect is based on the effect that a change to a single variable will have on the rest of a program; it provides a measure of the program's complexity. The original algorithm used to compute ripple effect has been reformulated to provide clarity in the operations involved and the measurement of ripple effect for procedural software. This paper describes the ripple effect metric and considers its applicability as a software complexity measure for object oriented software. Extensions are proposed to the computation of ripple effect to accommodate different aspects of the object oriented paradigm.

Keywords: Software Metrics, Change Impact Analysis, Ripple Effect, OOP.

1 Introduction

Software plays an important role in our lives. Products that affect people's lives must have quality attributes. Therefore, good quality software is required and in order to determine the quality of software we need metrics to measure it. A key point here is that the quality of a product may change over time and software is no exception. In the early days of computing, software costs represented a small percentage of the overall cost of a computer-based system. Hence, a sizable error in estimates of software cost had relatively little impact. Today software is the most expensive element in many computer-based systems. Therefore steps taken to reduce the cost of software can make the difference between the profit and loss of a company. So by determining the quality attributes of software, more precise, predictable and repeatable control over the software development process and product will be achieved.

Software is supposed to change. So, why does the software community struggle with the problems of software maintenance and the software's requisite change? Much of the concern has more to do with the complexity and sheer size of current applications than it has to do with change. As we develop large software systems (now in the 10s of millions of lines of code) incorporating more features and newer technology, the need for new Change Impact Analysis (CIA) technology has emerged

[9]. Changing requirements are endemic to software [11]; many researchers have written about software changes and their consequences [6]. Final requirements seldom exist for software systems since they are continually being augmented to accommodate changes in user expectations, operational environment, and the like [2]. Therefore, many software systems are never really complete until their function in the organization becomes obsolete.

Basic software change activities can be summarized as: understanding software with respect to the change, implementing the change within the existing system, and retesting the newly modified system. Each of these activities has some element of impact determination. To understand the software with respect to the change, we must ascertain parts of the system that will be affected by the change and examine them for possible further impacts. While implementing the change within the existing system, we need to be aware of ripple effects caused by the change and record them so that nothing is overlooked. Once the change has been designed and implemented, we need to find test cases that may need to be re-examined for redesign based on new requirements [8].

2 Software Measurement

To improve the quality of the software during its development, we need models of the development process, and within the process we need to select and deploy specific methods and approaches and employ proper tools and technologies. We need measures of the characteristics and quality parameters of the software development process and its stages. We need metrics and quality models to help ensure that the development process is under control to meet the quality objective of the product. What is measured is improved. Data and measurements are the most basic prerequisites for the improvement and maturity of any scientific or engineering discipline. Yet, in the discipline of software engineering, this area is perhaps one that has many critical problems and one that needs concerted effort for improvement.

Measurements for software projects should be well thought out before being used. Each metric used should be subjected to an examination of the basic principles of measurement scale, the operational definition, and validity and reliability issues should be well thought out [15]. As the software industry has matured, resources have shifted from being devoted to developing new software systems to making modifications to evolving software systems: software maintenance. A major problem for developers in a changing environment is that small changes can ripple through software to cause major unintended impacts elsewhere. Therefore, software developers need mechanisms to understand how a change to a software system will impact the rest of the system. This process is called CIA. Making software changes without understanding their effects can lead to unreliable software products. CIA can be used to reduce the amount of maintenance required; thereby increasing the reliability of the software, since fewer errors would have been introduced.

3 Software Maintenance

Over the years, several software maintenance models have been proposed, often to emphasize particular aspects of software maintenance. Among these models, there are common activities. The following is a brief summary of software maintenance models reported in the literature.

Boehm's model [7] consists of three major phases: understanding the software, modifying the software and revalidating the software. The Martin-McClure model is similar [21], consisting of program understanding, program modification, and program revalidation. Parikh [24] has formulated a description of maintenance that emphasizes the identification of objectives before understanding the software, modifying the code, and validating the modified program. Sharpley's model [26] has a different focus, it highlights the corrective maintenance activities through problem verification, problem diagnosis, reprogramming, and baseline reverification. Osborne's model of software maintenance [23] concentrates on managing the maintenance activities and determining appropriate measurements applied for visibility, but not into impacts of changes. The Yau and Patkow models are useful in evaluating the effects of change on the system to be maintained. Yau [29] focuses on software stability through analysis of the ripple effect of software changes. A distinctive feature of this model is the post-change impact analysis provided by the evaluation of ripple effect. This model of software maintenance involves: 1) determining the maintenance objective, 2) understanding the program, 3) generating a maintenance change proposal, 4) accounting for the ripple effect, and 5) regression testing the program [29].

4 Ripple Effect

CIA information can be used for planning changes, making changes and tracing through the effects of changes. Research into CIA has been concerned mostly with procedural software: function-based programs not class-based. However, this work will be concerned with object oriented software, since current software development projects most commonly use object oriented programming.

Ripple effect is just one of many types of CIA techniques. It can make the potential effects of changes visible before their implementation, making it easier to perform maintenance changes more accurately. The term 'ripple effect' was first introduced 1972 by Haney, who used a technique called 'module connection analysis' which was a measure of probability. Myers [22] used matrices to quantify matrix independence. Soong [27] used the joint probability of connection of all elements within a system to produce a program stability measure. In 1978, Yau and Collofello introduced their version of ripple effect which uses ideas from Haney, Myers and Soong's work. It is proposed as a measure of complexity as opposed to probability, which could amongst other things be used during software maintenance to evaluate and compare various program modifications to source code [30]. Computation of ripple effect involved using error flow analysis where all program variable definitions involved in an initial modification represented primary error sources from which inconsistency could

propagate to other program areas. Propagation continued until no new error sources were created.

The computation of ripple effect was reformulated in 2001 to make the calculation more explicit [4]. The reformulation revealed how the algorithm's structure can be broken down into separate parts thus providing clarity and enhancing the understanding of its structure. To facilitate the software implementation of the new algorithm an approximation was made, greatly simplifying the calculation that is the basis of automatic ripple effect computation.

The current computation of ripple effect is based on the effect that a change to a single variable will have on the rest of the program, it is used to determine the scope of the change and to provide a measure of the program's complexity. The effect of the change may not necessarily be local to the modification, but may also propagate to other parts of the program. There are two types of change propagation that are used to calculate ripple effect values [4]:

- **Intramodule change propagation**: Propagation from one variable to another within a module, (Fig. 1), e.g. propagation between *y, x* and *z* in Module1.
- **Intermodule change propagation**: propagation from one module to another, (Fig. 1), e.g. propagation from Module1 to Module2.

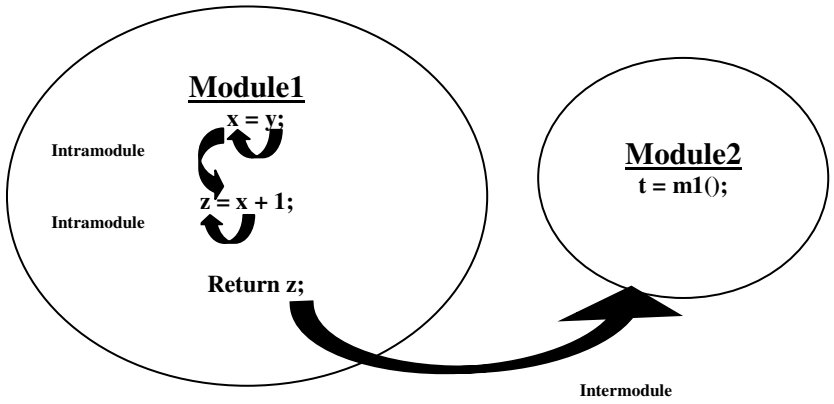


Fig. 1. Intramodule & Intermodule Change Propagation for Procedural Software

Intramodule change propagation is used to identify all variables which are affected by ripple effect as a consequence of a modification to a specific variable within the module. Intramodule change propagation uses information from *assignment*, i.e. change propagation from the right-hand side of an assignment to the left-hand side; and *definition-use*, i.e. change propagation from the definition of a variable to subsequent use of that variable, (Fig. 2).

The combination of information from assignment and definition-use pairings supply the required information for calculating intramodule change propagation. However, Propagation from one module to another is called intermodule change

propagation. Where, a change to a variable can propagate to other variables via: global variables, output parameters and variables used as input parameters to called modules.

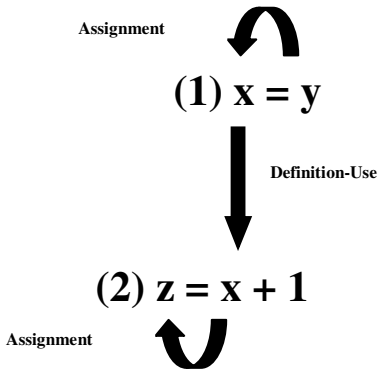


Fig. 2. Assignment and Definition-Use Pairings

5 Object Oriented Ripple Effect

Object Oriented (OO) programming is now the focus of the software engineering community. The use of OO software development techniques has added new elements to software complexity in the software development process and in the final product. Understanding the OO paradigm is the first step towards the definition of metrics for that paradigm. Terminologies vary among OO programming languages. However, all OO languages share some concepts. Some of the characteristics of the OO paradigm that will be considered in this research include: objects, class, overriding, inheritance, polymorphism and implicit parameters [17].

Elish and Rine [12] present an algorithm for computing ripple effect for object oriented programs at the design level, i.e. at a more coarse grained level than the ripple effect presented in this paper. Chauman et al [10] study the impact of changes across an object oriented system written in C++ by making one change at a time and studying the resulting impact. Li and Offut [19] carry out CIA for object oriented programs with the aim of highlighting modules that need to be re-tested.

Thus far, research built on the work of Black [4] has focused on the automatic computation of ripple effect measures within a practical timescale for procedural software, using the C programming language. A tool, REST, produced which uses an approximation algorithm to compute ripple effect for the C programming language [4]. The research proposed here focuses on measuring ripple effect for C++ object oriented software and possibly software written using other object oriented programming languages, for example Java.

6 Implementation Model

The proposed research focuses on the implementation and possible reformulation of the ripple effect algorithm to produce the automatic computation of ripple effect measures for object oriented software and possibly the subsequent evaluation of its potential benefits.

The following software analysis tools are being used to assist in collecting the required information for the measurement of ripple effect for object oriented software systems:

- **REST** [4]: A software tool that was developed at the Centre for Systems and Software Engineering to automate the production of ripple effect measures for C code. A C++ parser for REST is currently being developed to allow computation of ripple effect for object oriented software.
- **CodeSurfer** [28]: A C/C++ source code analysis and navigation tool. Codesurfer is a code browser produced by GammaTech. It can be used for program understanding, maintenance, CIA, re-engineering and reuse.

Using the above software analysis tools, different versions of the ripple effect algorithm for object oriented software will be implemented and compared for validation, (Fig. 3):

1. Concatenating all code within a class, omitting calls to local methods. Calculating ripple effect between this class and other classes, (i.e. Ripple effect calculation at the *class* granularity).
2. Looking at ripples across methods and classes, ignoring all propagations within methods, (i.e. Ripple effect calculation at the *class* granularity, taking methods into account).
3. Looking at ripples across methods and classes, calculating all propagations within methods, between methods and between classes, (i.e. Ripple effect calculation at both *method* and *class* granularity).
4. Looking at ripples across methods within each class, ignoring all propagations between classes, (i.e. Ripple effect calculation at the *method* granularity).

An example application of computing ripple effect for a small *C++* program has already shown that it is applicable and useful [5]. However, to compute ripple effect for the object oriented program using the current REST parser, the C++ code had to be first converted to C. This involved removing all classes from the code and converting all member-functions and member data into regular C functions and global variables respectively. A future version of the REST tool will parse C++ code to compute the ripple effect directly without the need of conversion.

Building on the work of Black [4], ripple effect will be computed for object oriented software, keeping in mind the following characteristics of object oriented software which have been identified as being important:

1. **Implicit Parameters:** When a call is made to a different non-parent class, a C++ member-function (or Java method) parameters are augmented by an *implicit variable*, which is a pointer to the target object itself. If the function changes the state of the target object (i.e. *mutator*) then the value of the implicit variable changes. Therefore, implicit parameters must be considered when considering starting points for the intramodule change propagation [5].

2. **Polymorphic Function/Method Calls:** In most compiled procedural languages, it can be determined before run-time which piece of code will be entered after the invocation of a function. However, a characteristic of object oriented languages is that the binding of some calls to particular function code only takes place at runtime. This is of course crucial to the intermodule change propagation calculations. Therefore, special parsing of subclasses of the target class or implemented classes of the target interface will need to be carried out to determine whether multiple potential target methods could be called [5].
3. **Class Relationships & Links:** A class can be defined as a group of variables and a group of methods. A change can be applied to a class, to a variable or to a method. Different types of relationships and links between a changed class and its impacted classes will be looked into and taken into consideration for the calculation of ripple effect. These relationships are: *Inheritance*, *Association*, *Invocation*, *Aggregation* and *Friendship* [14].

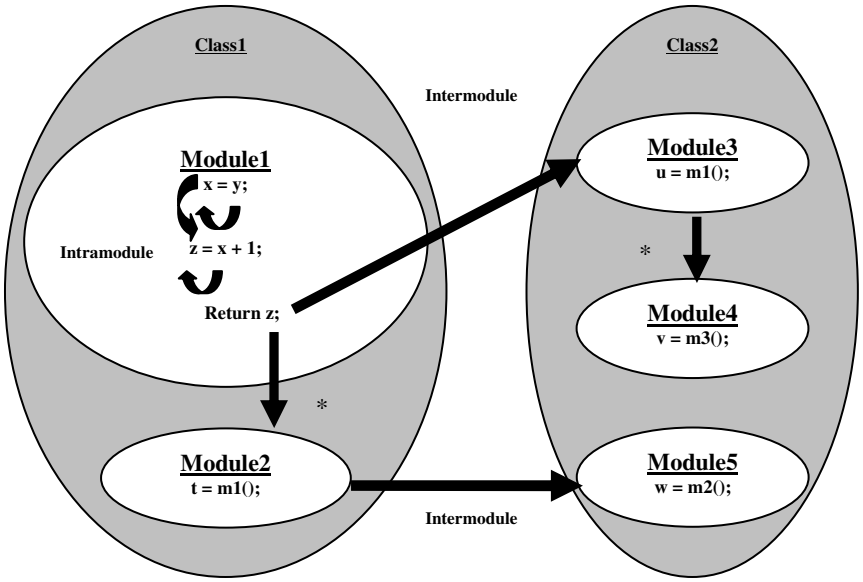


Fig. 3. Intramodule & Intermodule Change Propagation for Object Oriented Software (* *intra-module/intermodule change propagation depending on the version of the ripple effect algorithm used*)

So far, there appear to be no major obstacles to computing ripple effect for object oriented software. The most important issues that need further investigation and resolution are the treatment of implicit variables in all object calls and intermodule change propagation computation for polymorphic calls.

7 Research Implications

Software maintenance has been recognized as the most costly phase in the software life cycle [18]. Over the life of a software system, software maintenance effort has been estimated to be frequently more than 50% of the total life cycle cost [25]. This work has the potential to improve maintenance of object oriented software, thereby reducing its cost. Measurement of object oriented software using ripple effect computation will help in:

- Understanding the nature of the software.
- Estimating the cost, the schedule and the effort devoted to a project.
- Determining the quality of the software.
- Predicting the maintainability of the software.
- Validating best practices for software development.
- Providing optimal maintenance solutions.

By identifying potential impacts before making a change, the risks associated with embarking on a costly change can be reduced, because the cost of unexpected problems generally increases with the lateness of their discovery. The more a particular change causes other changes, the higher the cost is. Carrying out ripple effect computation will allow an assessment of the cost of the change and help management to choose between alternative changes. It will also allow managers and engineers to evaluate the appropriateness of a proposed modification. If a proposed change has the possibility of impacting large, disjoint sections of a program, the change will need to be re-examined to determine whether a safer change is possible [16]. This proposed research offers the potential to improve the stability and efficiency of object oriented software and cut the cost of software maintenance.

8 Conclusion and Future Work

Because software now plays a very important role in our lives we need to ensure that our software products are of good quality. Using CIA and specifically ripple effect as part of a software measurement program can give useful feedback which can then be used to improve future iterations of the product. Previous work has concentrated on measuring ripple effect for procedural software. This research will focus on implementing ripple effect measurement for object oriented software, for example C++, to ensure that the quality of the software is enhanced and maintained. A brief description of CIA, software maintenance, software measurement, object oriented paradigm and ripple effect have been given. Explanation of the two fundamental features of ripple effect computation: intramodule and intermodule change propagation have been presented. Also, object oriented constructs relevant to computing ripple effect have been discussed.

The ideas presented in this paper will be taken further by drawing up a much more detailed framework of ripple effect measurement for object oriented software. For this work to be useful, guidelines for the practical implementation of the ideas presented are being drawn up and will be utilized. This work will enable and show the automatic computation of ripple effect for object oriented software.

Work will be evaluated as it progresses by comparing the output of the implementation of all 4 different versions of the ripple effect algorithm and the difference in the ripple effect measure between object oriented and procedural software systems. To the benefit of the first author, who has come to this work with almost five years of industrial experience in software maintenance, the ideas proposed and future results can be thoroughly debated and self-criticized.

References

1. Anderson, P., Reps, T., Teitelbaum, T., Zarins, M.: Tool Support for Fine-Grained Software Inspection. IEEE Software 20(4): 2003, 42-50
2. Arnold, R.S., Bohner, S.A.: Impact Analysis – Towards A Framework for Comparison. Proc. of the Conf. on Software Maintenance, Pages 292-301, September 1993
3. Bilal, H.Z., Black, S.E.: Using the Ripple Effect to Measure Software Quality. SQM 2005, Cheltenham, Gloucestershire, UK. 21st-23rd March 2005
4. Black, S.E.: Computation of Ripple Effect Measures for Software. Ph.D. thesis, London South Bank University, London, UK, 2001
5. Black, S.E., Rosner, P.E.: Measuring Ripple Effect for the Object Oriented Paradigm. IASTED International Conference on Software Engineering, 15th-17th Innsbruck, Austria, February 2005
6. Boehm, B.: Improving Software Productivity. IEEE Computer, September 1987, Pages 43-57
7. Boehm, B.: Software Engineering. IEEE Trans. On Computers, No. 25, Vol. 12, December 1976, Pages 1226-1242
8. Bohner, S.A.: Impact Analysis in the Software Change Process: A Year 2000 Perspective. In Proceedings International Conference on Software Maintenance ICSM'96, pages 42-51. IEEE Computer Society Press, November 1996
9. Bohner, S.A., Arnold, R.S.: Software Change Impact Analysis. IEEE Computer Society Tutorial, IEEE Computer Society Press, 1996
10. Chaumun, M.A., Kabaili, H., Keller, R.K., Lustman, F.A.: Change Impact Model for Changeability Assessment in Object-Oriented Software Systems. Science of Computer Programming, 45(2-3), 2002, 155-174
11. Davis, A.: Software Requirements: Analysis and Specification. Prentice-Hall, New Jersey, 1989
12. Elish, M.O., Rine, D.: Investigation of Metrics for Object Oriented Design Logical Stability. In Proceedings of the Seventh European Conference on Software Maintenance and Reengineering. 26-28 March 2003, 193-200
13. Haney, F.M.: Module Connection Analysis - a Tool for Scheduling of Software Debugging Activities. Proceedings Fall Joint Computer Conference, 1972, 173-179
14. Kabaili, H., Keller, R.K., Lustman, R.A.: Change Impact Model Encompassing Ripple Effect and Regression Testing. In Proceedings of the Fifth International Workshop on Quantitative Approaches in Object-Oriented Software Engineering, Budapest, Hungary, 2001, 25-33
15. Kan, S.H., Basili, V.R., Shapiro, L.N.: Software Quality: An Overview from the Perspective of Total Quality Management. IBM Systems Journal, VOL 33, No. 1, 1994
16. Lee, M.L.: Change Impact Analysis of Object-Oriented Software. Technical Report ISE-TR-99-06, George Mason University, 1998

17. Lewis, J., Shields, M., Meijer, H.J.M.: Implicit Parameters: Dynamic Scoping with Static Types. In Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (pp. 108-118). Boston, Massachusetts, USA, 2000
18. Li, W., Henry, S.: An Empirical Study of Maintenance Activities in Two Object-oriented Systems. Journal of Software Maintenance, Research and Practice, Volume 7, No. 2 March-April 1995, Pages 131-147
19. Li, L., Offutt, A.J.: Algorithmic Analysis of the Impact of Changes to Object-Oriented Software. In Proceedings of the International Conference on Software Maintenance, IEEE, Monterey, CA, USA, November 1996, 171-184
20. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. Communications of the ACM 21(6) 1978, 466–471
21. Martin, J., McClure, C.: Software Maintenance: The Problem and its Solutions. Prentice-Hall, London, 1983
22. Myers, G.J.: A Model of Program Stability. Van Nostrand Reinhold Company, 135 West 50th Street, NY 10020, Chapter 10, 1980, 137-155
23. Osborne, W.M.: Building and Sustaining Software Maintainability. Proceedings of Conference on Software Maintenance, October 1987, Pages 13-23
24. Parikh, G.: Some Tips, Techniques and Guidelines for Program and System Maintenance. Winthrup Publishers, Cambridge, Mass., 1982, Pages 65-70
25. ReiBing, R.: Towards a Model for Object-Oriented Design Measurement. Proceedings of the 5th International ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering, pp. 71-84, 2001
26. Sharpley, W.K.: Software Maintenance Planning for Embedded Computer Systems. Proceedings of the IEEE COMPSAC, November 1977, Pages 520-526
27. Soong, N.L.: A Program Stability Measure. Proceedings 1977 Annual ACM conference, Boulder, Colorado, 1977, 163-173
28. Teitelbaum, T., Reps, T. CodeSurfer. GrammaTech Inc., http://www.grammatech.com/products/codesurfer/overview.html, last accessed 9th April 2006
29. Yau, S.S., Collofello, J.S.: Some Stability Measures for Software Maintenance. IEEE Trans. On Software Engineering, Vol. SE-6, No. 6, November 1980, 545-552
30. Yau, S.S., Collofello, J.S., McGregor, T.M.: Ripple Effect Analysis of Software Maintenance. Proceedings COMPSAC '78 (1978), 60-65
31. 2nd Analysis, Slicing & Transformation Network Workshop, ASTReNet, London, UK, June 2005, http://www.dcs.kcl.ac.uk/staff/zheng/astrenet/index.html, last accessed 9th April 2006

Using Coupling Metrics for Change Impact Analysis in Object-Oriented Systems

M.K ABDI[1], H. LOUNIS[2], and H. SAHRAOUI[1]

[1] Department of Computer Science and Operations Research, Université de Montréal, Canada
{abdimust, sahraouh}@iro.umontreal.ca
[2] Department of Computer Science, Université du Québec à Montréal, Canada
lounis.hakim@uqam.ca

Abstract. The development of software products consumes a lot of time and resources. On the other hand, these development costs are lower than maintenance costs, which represent a major concern, specially, for systems designed with recent technologies. Systems modification should be taken rigorously, and change effects must be considered. In this paper, we propose an approach, both analytical and experimental; its objective is to analyze and predict changes impacts in Object-Oriented (OO) systems. The method we follow consists first, to choose an existing impact model, and adapt it afterward. An impact calculation technique based on a meta-model is developed. To evaluate our approach, an empirical study was led on a real system in which a correlation hypothesis between coupling and change impact was advanced. A concrete change was done in the target system and coupling metrics were extracted from it. The hypothesis was verified with machine-learning (ML) techniques and results are presented and commented.

1 Introduction

Maintenance is the last phase of the software life cycle. It is defined as the process of modification of software in operation to allow it to always satisfy current and future specifications [25]. According to Pfleeger [26], maintenance cost depends in large part (40 %) on the modification of software architecture, interactions between components, procedures/methods, and, variables. Systems modification should be taken seriously; changes effects must be considered. A small change can have considerable and unexpected effects on the system. Risks incurred during the modification are related to the consequence of the impact of a given change. When modularity is adequately used, it limits the effects relating to changes. Nevertheless, change impacts are subtle and difficult to discover; designers and maintainers need mechanisms to analyze changes and to know how they are propagated in the whole system.

The main motivation of our work is to improve the maintenance of object-oriented systems, and to intervene more specifically on change impact analysis. We mainly aim at the reduction of effort as well as maintenance costs. Effort reduction can be carried out with time reduction between a change proposition, its implementation and finally, its realization, while ensuring the quality of the system. Effort can be also reduced if one can predict system behaviour in front of possible changes. Our work is much more in this research orientation; the more analysis and change impact prediction are systematic, the more effort reduction is optimal. Good decisions can be taken before introducing changes; by identifying the potential impact of a modification, one reduces the risk to deal with expensive and unpredictable changes. This change impact analysis will allow the maintenance responsible to know change consequences. The more a change affects classes, the more its realization cost is high. Thus, change impact analysis allows to estimate change cost and to make a compromise between various suggested changes.

The present paper is organised as follows: section 2 presents various works done in the topic of change impact analysis. Our approach is presented in the third section; we explain the fundamental points on which it is

based, as well as the general stages of the adopted method. Then, we present the chosen change impact model and its adaptation to the Java language. The way we calculate change impact expressions, based on a meta-model approach, will finish this section. Section 4 is concerned with the empirical study, and the discussion of obtained results. Finally, our work perspectives are discussed in the conclusion.

2 Related works

Several studies were conducted to validate metrics and to relate them to some maintainability properties. Li and Henry [11] took five metrics of Chidamber and Kemerer [4], added three of their own, to show that there is a strong relationship between these metrics and maintenance effort, expressed in number of changed lines code. In [16], the authors showed that the choice of architectures, in early stages of software systems design, has an important impact on a number of quality factors, for instance, maintainability, efficiency, and reusability. Lounis & al [19] proposed a succession of 24 code metrics to generate predictive models related to the fault- proneness. Finally, in [17], the authors also studied relationships between most of the coupling metrics, cohesion, and, inheritance ones in one side, and classes fault-proneness in the other side.

Less works have been conducted on change impact. Han [7] developed an approach for computing change impact on design and implementation documents. His approach does not consider the invocation dependencies. Furthermore, impacts are not defined in a formal way. In [1], the authors predicted evolving object-oriented systems size starting from the analysis of the classes impacted by a change request. They predicted changes size in terms of added/modified lines of code. On the other hand, Kung and al [9], interested by regression testing, developed a change impact model based on three links: inheritance, association, and, aggregation. They also defined formal algorithms to calculate all the impacted classes including ripple effects. Li and Offutt examined in [12], the effects of encapsulation, inheritance, and, polymorphism on change impact; they also proposed algorithms for calculating the complete impact of changes made in a given class. However, some changes, implying for instance inheritance and aggregation, were not completely covered by their algorithms.

Lastly, Briand and al., in [2], tried to see if coupling measures, capturing all kinds of collaboration between classes, can help to analyze change impact. Strategy adopted in this study is different from other strategies since it is purely empirical. This study, (i) showed that some coupling metrics, related to aggregation and invocation, are connected to ripple effect, and, (ii), it allows performing dependence analysis and reducing impact analysis effort. In [3] and [8], a change impact model was defined at an abstract level, to study the changeability of object-oriented systems. The adopted approach uses characteristic properties of object-oriented systems design, measured by metrics, to predict changeability.

In short, studies made in [2] and [15] are examples of purely empirical approaches. Works in [9], [12], [18], and, [10] exploit approaches based mainly on models like, dependence graphs, possibly enriched by some formalisms. Studies in [3] and [8] propose a different approach; we will speak about it afterward. Moreover, we noticed through this synthesis, that there are more works based on dependence graphs than works based on other abstractions or purely empirical works. On the other hand, generally, impact is not calculated in a systematic way and most of done experiments were on small systems; it doesn't allow the generalization of obtained results (rules, relationships, laws, etc.). In the next section, we present in detail our approach.

3 The approach

From the beginning, we have decided that our approach will not be purely empirical; indeed, we want to obtain general results e.g., rules, relationships between causes and effects, etc., we can apply to wide application domains. However, we do not reduce the importance and the necessity of empirical studies; our approach is both analytical and empirical. In our opinion, the study of changes and their impacts analysis must be done, at first, at a high level of abstraction. Results found at such a level must be necessarily verified afterward, by empirical studies. On the other hand, in our domain literature review, we have noticed that very few works propose a more or less complete definition of change impact, i.e., model taking into account main links that one can find in an object-oriented design (namely, association, aggregation, invocation and inheritance) [1] and [7]. In our study,

we took the impact model defined in the SPOOL[1] project [14]; [3] and [8] noticed that this model is one of the most general, and it allows impact calculation in a systematic way. It is an important factor concerning effort and maintenance cost reduction. The SPOOL project had as main objective the understanding of industrial systems design properties and their influences on maintenance and evolution.

In our work, we use this model to lead our experiments, which are directed by hypotheses statements. By analogy with the other works made in the domain [19] and [17], hypotheses express generally relationships between some system design characteristics (or architectural properties) and change impact, in our case. These design characteristics are measured by metrics, and, the choice of these metrics is part of our empirical study orientation. Verifying these hypotheses can be made by different techniques, e.g., statistical models [8]. In our work, we choose artificial intelligence techniques, more specifically, machine-learning ones, for two main reasons: (i) these techniques were not yet used in previous works on change impact analysis, and, (ii) models produced by these techniques are knowledge pieces, represented according to a given formalism, which can be integrated into a knowledge-based decision system. Finally, in this work, we aim at analyzing software systems coded with the Java language; an adaptation of the model (defined at abstract level) to this language is necessary for impact calculation of any possible atomic change in Java. We need also tools, which allow analyzing the code of the system under test. The implementation of the change impact calculation expressions, deduced at an abstract level, by the model, depends first, on considered changes, and secondly, on the analysis tool. The calculation of selected metrics is also a task, which can be realized within the used tool framework, as it can be totally independent. Finally, the choice of machine-learning algorithms techniques for hypotheses verification depends on their applicability and performances.

3.1 Change impact model

When a change is considered, it is necessary to identify system components that will be impacted; it will ensure that the system will still run correctly after change implementation. Our concern is then focussed on how the system reacts to a change (in general). It is generally accepted that a system absorbs easily a change if the number of impacted components is small. A system is seen as a set of classes connected by different links; a class is defined as a group of methods, which serve as public interface or for internal operations, and a section of variables, which define the state of classes' instances. A component refers to a class, a method, or a variable. As examples of changes, one can have the deletion of a variable, the change in a method's scope from "public" to "protected" or the removal of the relationship between a class and its parent. Table 1 consigns main changes in object-oriented systems, at the design level; they are categorized according to the component they affect and a total of 13 changes are identified.

Once a given component is subject to change, a specific part may be affected, in case it is related to the changed component via a link. These links are among the four following types: S (association: a class makes reference to variables of another class); G (aggregation: the definition of a class implies objects of another class); H (inheritance: a class inherits the characteristics defined in another class (parent)); I (invocation: the methods of a class call upon methods defined in another class). We also consider a special notation commonly used in Boolean algebra: the absence of operator between 2 links means an *intersection*. The "+" operator means a *union*. The "~" before a link means the *negation*. For instance, ~G means the set of classes that are not linked to the specified class by an aggregation. Moreover, links are independent from each other, and, we can expect to find any number and any type of links between two classes. A change of a class can also have an impact in the same class. Pseudo-link L (local) is introduced to denote this.

[1] SPOOL: "Spreading desirable Properties into the design of Object-Oriented Large-scale software systems". This project was organized by CSER (Consortium for Software Engineering Research), and subsidized by BELL Canada, NSERC (Natural Sciences and Research Council of Canada) and NRC (National Research Council Canada).

Table 1. Main changes at the abstract level

Component	***Change description***
Variable	Type change
	Variable scope change
	Addition
	Deletion
Method	Return type change
	Implementation change
	Signature change
	Method scope change
	Addition
	Deletion
Class	Inheritance structure change
	Addition
	Deletion

We call change impact the set of classes that require a correction after this change. It depends on two factors. First, the change category; for example, changing the type of a variable has an impact on all classes referencing this variable, whereas the addition of a variable has no impact on these classes. Given a change category, the type of link between classes is the second factor that influences impact. Consider a change in the scope of a method from "public" to "protected"; classes invoking this method will be impacted, excepted for those, which are derived from the changed class. However, let us notice that several links, between a changed class and an impacted one, can be involved in the calculation. Thus, for a given change ch_i in class cl_j, the set of impacted classes is expressed as a Boolean expression in which the variables stand for the links. For instance, the impact formula for such a hypothetical change may be given by:

$$\text{Impact } (cl_j, ch_i) = S\sim H+G \ .$$

This expression means that classes in association (**S**) with cl_j, and not derived (**~H**) from the changed class cl_j, or classes that are in an aggregation link (**G**) with cl_j, are impacted. Table 2 gives examples of change impact expressions for each type of constituent.

Table 2. Changes examples and their expressions

Component	***Change description***	***Impact expression Impact (cl_j, ch_i)***
Variable	Type Change	S+L
Method	Scope change from "public" to " protected	I~H
Class	Deletion	H+G+S+I

Let us note, that this impact model allows predicting which classes would be impacted if a change were really made. In our work, we are interested only in changes that have a syntactic impact; a given change is characterized by a code transformation somewhere in the system. If the system is successfully re-compiled, then there is no impact; otherwise, we have an impact, i.e., code modifications must be done elsewhere in the system to obtain a syntactically correct code that will re-compile. Since our focus is only on syntactic impact, appropriate measures we have to apply are based on impacts that are dependent on the static nature of the source code. Thus, the impact, which can occur during the execution, because of polymorphism, is not considered.

3.2 Model adaptation to the Java Language

We have indicated that this model, defined at the abstract level, was already adapted to the C++ language, in the SPOOL project [3] [8] [14]; it was a requirement of the industrial partner. In the present study, we target software systems coded in Java; an adaptation operation of this model to this language turns out to be necessary. Some changes are common to the two languages, e.g., variable type change, method signature change, class inheritance structure change, etc. On the other hand, some other changes are specific to C++, e.g., those related

to "virtual" (virtual method or virtual class) and "friendship" (friendly class) concepts. The model adaptation to Java is presented in table A of the appendix. The final list contains a total of 52 changes, including 12 changes for variables, 25 for methods, and, 15 for classes. Finally, for our experiments, we opted for the PTIDEJ[2] tool. In [6], Guéhéneuc proposes and describes models and algorithms to ensure the traceability of design motives[3] between implementation and retro design phase's programs. The PTIDEJ tool is an implementation in Java of these models and algorithms. It is integrated into the Eclipse development environment [13].

4 Empirical study and Results

In our study, we focalise on inter-classes dependencies; they are supposed to have an impact on ripple effects. We study the relationship between coupling, an architectural property, and change impact. We propose to verify the following hypothesis:

"Coupling influences somehow change impact in object-oriented systems".

We quoted in section 2 several works on this architectural property, but our objective in this study is to see which types of coupling influences more change impact. We chose a program analysis toolbox system, called BOAP, and, developed at the computer science research center of Montreal (CRIM) [5]. It is a set of integrated software tools, which allow an expert to evaluate some software qualities, e.g., conceptual or structural weaknesses, too complex instructions, etc. We considered the BOAP system in its version 1.1.0; it is written in Java and contains 394 classes. The change we chose is the variable type one. We have determined a class that presents an important number of associations with other classes; the goal is to have a rather considerable impact on the rest of the system, according to the envisaged change. Then, we selected a variable and carried our change. We obtain:

Considered class: *dbClass* (of the DBLMR package)
Variable chosen: *sizeInBytes*
Change: from *"long"* type to type *"integer"*
The impact expression of this change is: S + L (see table A, in Appendix).

This expression means that there is change impact locally at first (in the changed class itself) and also in all the classes of the system that are in association with the changed class "*dbClass*". The calculation technique of change impact returns a total of 42 classes; there are thus 42 impacted classes after this change. We proceeded the same way for all others system classes. Let us note that it can involve other variable types changes, as it is completely possible that a change does not create any impact (null impact). On the other hand, we extracted from the BOAP target system, a set of metrics related to coupling. They are presented in table 3. We calculated them by the tool developed in [21].

As already indicated, we used in this study, machine-learning techniques. We have exploited the Weka environment (Waikato Environment for Knowledge Analysis) [20]; it is a set of tools implementing most of machine-learning algorithms. It is written in Java and is open source. We wanted in this experimentation, to use several machine-learning algorithms, in order to find various relationships between coupling metrics and change impact. The choice of these algorithms was based on three criteria: (i) interpretability of the produced models, (ii) models complementarity, and, (iii) precision (accuracy) of the results. Our learning data set gather 11 variables (10 independent variables + the dependent variable). The independent variables represent coupling metrics while the dependent variable represents change impact. All the independent variables are numeric. On the other hand, the dependent variable is discrete; it was initially numeric as it resulted from our calculation technique with the impact expression (S+L). However, it was necessary to transform it into a discrete variable to use effectively the 3 chosen ML algorithms, i.e., J48, PART, and, NBTree.

[2] Ptidej: Pattern Trace Identification, Detection, and Enhancement in Java.
[3] A design motive is the solution of a pattern design.

Table 3. The selected coupling metrics

Metrics	Definition
RFC	Response For a Class: number of methods called upon in response to a message.
MPC	Message Passing Coupling: number of messages sent by a class in direction of the other classes of the system.
CBOU	CBO Using: refers to the classes used by the target class.
CBOIUB	CBO Is Used By: refers to the classes using the target class.
CBO	Coupling Between Object: number of classes with which a class is coupled.
CBONA	CBO No Ancestors: CBO without considering classes ancestors.
AMMIC	Ancestors Method–Method Import Coupling: number of parents' classes with which a class has an interaction of the method-method type and a coupling of the type IC.
OMMIC	Others Method–Method Import Coupling: number of classes (others that super classes and subclasses) with which a class has an interaction of the method-method type and a coupling of the type IC.
DMMEC	Descendants Method–Method Export Coupling: number of subclasses with which a class has an interaction of the method-method type and a coupling of the type EC.
OMMEC	Others Method–Method Export Coupling: number of classes (others that super classes and subclasses) with which a class has an interaction of the method-method type and a coupling of the type EC.

J48 is an implementation of the well-known C4.5 algorithm [22]. It is a supervised learning algorithm that induces a classification model represented by a decision tree or rules. PART [23] allows the induction of rules by the iterative generation of partial decision trees; its main idea is to build a partial decision tree instead of an entirely explored one. It provides results as precise as those of J48 algorithm. Finally, with NBTree (Naïve-Bayes decision-Tree), Kohavi [24] proposes a hybrid approach combining naive Bayesian classifiers and classifiers based on decision trees. This hybrid approach frequently obtains a very high precision compared to naive Bayesian classifiers or decision trees classifiers. It exploits a tree structure to divide the instances space into subspaces and to generate a naive Bayesian classifier for each subspace. In a conventional decision tree, each leaf is marked with only one class and the algorithm predicts this class for the instances that reach the leaf, whereas a naive Bayesian tree uses a local naive Bayesian classifier to predict the classes of these instances.

During the use of these 3 ML algorithms, the computation of models accuracy is done thanks to a cross-validation procedure. It is helpful when the amount of data for training and testing is limited, which is our case; we try a fixed number of approximately equal partitions of the data, and each in turn is used for testing while the remainder is used for training. At the end, every instance has been used exactly once for testing.

J48 obtains an accuracy of 73.85 %; it is pretty high and interesting. This rate expresses that on 394 instances, 291 were correctly classified. On the other hand, we find that the generated decision tree is rather large (size=67). Consequently, it is difficult to extract causality rules from this tree; to obtain a more compact decision tree, we chose a data pre-processing: we keep a reduced set of attributes (or independent variables), the most relevant, instead of considering the set of all attributes. Weka offers such a simple filtering algorithm that arranges subsets of attributes according to a correlation based on a heuristics evaluation function; some attributes should be ignored because they will have a low correlation with the variable to be predicted. The attributes which were selected to participate (to the learning process) are: MPC, CBOU, CBONA, AMMIC, and OMMIC (see table 3). We have run again J48 on this new data set, and, the obtained accuracy was very close to the previous (73.30 %). However, the induced decision tree is well reduced (size=31) and compact. It contains 16 leaves; every path from the root to a given leaf is a causality rule. We have obtained then a set of 16 rules. Figure 1 presents some of these rules.

Rule 1:
MPC ≤ 21
OMMIC ≤ 4
AMMIC = 0
→ impact: weak (119.0/51.0)

Rule 2:
MPC ≤ 21
OMMIC ≤ 4
AMMIC > 3
→ impact: weak (32.0)

Rule 11:
MPC ≤ 21
OMMIC > 4
CBOU ≤ 7
→ impact: weak (68.0/12.0)

Rule 12:
MPC ≤ 21
OMMIC > 4
CBOU > 7
→ impact: average (9.0/1.0)

Rule 15:
MPC > 36
CBOU > *14*
AMMIC ≤ 5
→ impact: average (4.0/1.0)

Rule 16:
MPC > 36
CBOU > *14*
AMMIC > 5
→ impact: very-strong (2.0)

Fig. 1. Causality rules induced by J48

The first remark to be made on this set of rules is that there are 14 rules on 16, where import coupling metrics are implied. It illustrates the influence of this particular coupling property on change impact. By observing well this subset of 14 rules, one can still distinguish 3 particulars subsets; the first subset, formed of 10 rules, evokes two import coupling metrics, i.e., OMMIC and AMMIC; the second one is formed of 2 rules, where appears only the OMMIC metric; finally, the third subset is also formed of 2 rules, where appears only the AMMIC metric. On the other hand, in the first subset, we notice that in most cases, impact is weak or very-weak for classes which present a weak Method-Method-Others import coupling (OMMIC ≤ 4, average is 9.26) and a weak/medium Method-Method-Ancestors import coupling (AMMIC between 0 and 3, average is 2.15). For the second subset, the OMMIC metric is not deciding but it represents an important element to consider in impact prediction; it is weak or medium according to whether the number of classes used by the target class is medium or great (CBOU ≤ 7 or > 7, average is 3.57). That is valid for classes, which have not a very large number of static invocation methods (MPC ≤ 21, average is 11.34) and a Method-Method-Others import coupling not too small (OMMIC > 4, average is 9.26). For the third subset, rules express that for classes with a great number of static invocation methods (MPC > 36) and a great number of classes used by the target class (CBOU > 14), the Method-Method-Ancestors import coupling measure is determining in the sense that impact for these classes, will be medium or very strong.

On the other hand, the PART algorithm obtains an accuracy of 65.48 %, it is weaker than the score obtained by J48; the induced knowledge is represented by a set of 25 rules. The first observation to make on this rule-set is that 16 rules involve import coupling metrics. That confirms the remark made before, concerning the influence of this particular coupling property on change impact. In addition, several rules are similar with those found by J48. For instance, rules 3 and 4 of PART are close to rules 2 and 11 of J48, and for the two algorithms, rules 15 are identical. That partially confirms the important result found by J48. Figure 2 shows some rules chosen among the set of rules generated by PART.

Finally, by running the NBTree algorithm on our data set, we obtain an accuracy of 66.75%. The induced decision tree is compact (size=17). It contains 9 leaves, containing each one a Bayesian classifier. Each path from the root to a given leaf is a probabilistic causality rule; we thus have a set of 9 rules. Figure 3 presents some of them. Let us note that in the conclusion part of each rule, we find the identifier of the naive Bayesian classifier (it is NB3 for rule 1), followed by the class to be predicted, which is in fact, the class with the highest probability. This probability is given between brackets. The results of this algorithm affirm that in addition of import coupling, coupling measured by CBONA and CBOU metrics also influences impact. Rules 1 and 9 (see figure 3) illustrate well that the impact is very weak or strong according to values' of these metrics (small or large). On the other hand, rules 2 and 3 express that the impact becomes increasingly weak when the Method-Method-Ancestors import coupling (AMMIC) increases; it confirms a result already found by J48.

Rule 3:
MPC <= 13
AMMIC > 3
→ impact: weak (33.0/1.0)

Rule 4:
MPC <= 13
OMMIC > 4
CBOU <= 1
→ impact: weak (6.0)

Rule 15:
MPC > 36
CBOU > 14
AMMIC <= 5
→ impact: average (4.0/1.0)

Figure 2. Causality rules induced by PART

Rule 1:
CBONA $\leq$ *3.5*
CBOU $\leq$*0.5*
→ NB3: impact very-weak (0.46)

Rule 2:
CBONA $\leq$ *3.5*
CBOU $\in$ *]0.5,1.5]*
AMMIC $\leq$*0.5*
→ NB5: impact weak (0.54)

Rule 3:
CBONA $\leq$ *3.5*
CBOU $\in$ *]0.5,1.5]*
AMMIC > *0.5*
→ NB6: impact very-weak (0.76)

Rule 9:
CBONA > *3.5*
CBOU > *36.5*
→ NB16: impact strong (0.48)

Figure 3. Causality rules induced by NBTree

5 Conclusion

We proposed in this article an approach of change impact analysis and prediction for object-oriented systems. We chose an existing impact model and we adapted it to the Java language. Then, we proposed a calculation technique of change impact expressions using a meta-model approach. To verify our approach, we developed an empirical study in which we stated a hypothesis between coupling and change impact. Some experiments were carried out on a target Java system; a change was concretely made on this system (variable type change) and its impact was deduced by our model then calculated by a calculation technique. On the other hand, a set of metrics related to coupling was extracted from the target system. Finally, we exploited 3 ML algorithms to verify our hypothesis.

The accuracies obtained by the 3 algorithms seem rather interesting. The results of J48 then confirmed by PART, express that import coupling influences much more change impact than other types of coupling, since in most cases, the impact is mainly related to this type of coupling. On the other hand, it turns out that for the classes for which the number of static methods invocations, as well as the number of classes used with the target class, is large, import coupling (measured by the AMMIC metric) determines change impact. This result was found by J48 then partially confirmed by PART. Finally, NBTree results added more details to the results already found by J48 and PART, and showed that coupling measured by CBONA and CBOU metrics also influences the impact.

We are now working on other experiments on other systems in order to confirm these results. We are also interested by other coupling measures, and, others types of architectural properties, that could be related to mechanisms explaining ripple effect in object-oriented systems. It would be interesting, in our opinion, to compare change impact through different systems and then find results applicable to a wide category of them.

References

1. G. Antoniol, G. Canfora and A. De Lucia. Estimating the size of changes for evolving object-oriented systems: a Case Study. In Proceedings of the 6th International Software Metrics Symposium, pages 250-258, Boca Raton, Florida, Nov 1999.
2. L. C. Briand, J. Wüst, and H. Lounis, "Using Coupling Measurement for Impact Analysis in Object-Oriented Systems" in proceedings of the International Conference on Software Maintenance ICSM'99, Oxford, England, August 30 – September 3, 1999.
3. M. A. Chaumun. "Change Impact Analysis in Object-Oriented Systems: Conceptual Model and Application to C++". Master's thesis, Université de Montréal, Canada, November 1998.
4. S. R. Chidamber and C. F. Kemerer. "A Metrics Suite for Object Oriented Design" in IEEE Transactions on Software Engineering, Vol. 20, No. 6, pages 476-493, June 1994.
5. E.H Alikacem, H. Snoussi, "BOAP 1.1.0, Manuel d'utilisation", CRIM, Janvier 2002.
6. Y-G Guéhéneuc, Un cadre pour la traçabilité des motifs de conception. Thèse de doctorat de l'université de Nantes, École Nationale Supérieure des Techniques Industrielles et des Mines de Nantes, juin 2003
7. J. Han. " Supporting Impact Analysis and Change Propagation in Software Engineering Environments" in Proceedings of the STEP97, London, England, pages 172-182, July 1997.
8. H. Kabaili, "Changeabilité des logiciels orientés objet propriétés architecturales et indicateurs de qualité", PhD thesis, Université de Montréal, Canada, Janvier, 2002
9. D. C. Kung, J. Gao, P. Hsia, J. Lin and Y. Toyoshima. "Class firewall, test order, and regression testing of object-oriented programs" in Journal of Object-Oriented Programming, Vol. 8, No. 2, pages 51-65, May 1995.
10. M. L. Lee, "Change Impact Analysis for Object-Oriented Software". PhD thesis, George Mason University, Virginia, USA, 1998.
11. W. Li and S. Henry, "Object-Oriented Metrics that Predict Maintainability" in Journal of Systems and Software, Vol. 23, pages 111-122, 1993.
12. L. Li and A. J. Offutt, "Algorithmic Analysis of the Impact of Changes to Object-Oriented Software" in ICSM96, pages 171-184, 1996.
13. Object Technology International, Inc. / IBM. Eclipse platform – A universal tool platform, July 2001.
14. R. Schauer, R. K. Keller, B. Laguë, G. Knapen, S. Robitaille, and G. Saint-Denis. The SPOOL Design Repository: Architecture, Schema, and Mechanisms. In Hakan Erdogmus and Oryal Tanir editors, Advances in Software Engineering. Topics in Evolution, Comprehension, and Evaluation. Springer-Verlag, 2001.
15. F. G. Wilkie, B. A. Kitchenham, "Coupling Measures and Change Ripples in C++ Application Software", published in the proceedings of EASE'99, University of Keele, UK, 1998.
16. L. C. Briand, S. J. Carrière, R. Kazman, J. Wüst, "A Comprehensive Framework for Architecture Evaluation", International Software Engineering Research Network Report ISERN-98-28.
17. L.C. Briand, J. Wust, H. Lounis. "Replicated Case Studies for Investigating Quality Factors in Object-Oriented Designs".In Empirical Software Engineering, an International Journal, 6(1):11-58, March 2001, Kluwer Academic Publishers.
18. R. Cantave "Abstractions via un modèle générique d'application orientée objet", Master's thesis, Université Laval, Canada, Avril 2001
19. H. Lounis, H.A. Sahraoui, and W.L. Melo, "Defining, Measuring and Using Coupling metrics in Object-Oriented Environment" in SIGPLAN OOPSLA'97 Workshop on Object-Oriented Product Metrics, 1997, Atlanta, Georgia, USA, 1997.
20. I. H. Witten and E. Frank, "Data Mining: Practical Machine Learning Tools and Techniques with Java Implementation", © 2000 Morgan Kaufmann Publishers
21. L. Cheïkhi, "Estimation de l'impact du changement dans les programmes à Objets", Master's thesis, Université de Montréal, Canada, November 2004.
22. J.R Quinlan, "C4.5: Programs for Machine Learning". Morgan Kaufmann Publishers, Sao Mateo, CA, 1993.
23. E. FRANK, I.H. WITTEN, "*Generating Accurate Rule Sets Without Global* Optimization" in Proceedings of the Fifteenth International Conference, Morgan Kaufmann Publishers, San Francisco, CA, 1998.
24. R. KOHAVI, "*Scaling up the accuracy of naive-Bayes classifiers: a decision tree hybrid*" in Proceedings of the Second International Conference on Knowledge Discovery and Data Mining, (1996).
25. Computer Society Press, "Standards Collection Software Engineering", The Institute of Electrical and Electronics Engineers, Inc., 1993.
26. S. L. PFLEEGER, "A Framework for Software Maintenance Metrics" in IEEE Transactions on Software Engineering, pages 320-327, May 1990.

Appendix

Table A. Results of change impact for Java

Change Id	Change Description	Impact Expression
v.1.1	Variable value change	-
v.1.2	Variable type change	S+L
v.1.3	Variable addition	-
v.1.4	Variable deletion	S+L
v.1.5	Variable scope change	
v.1.5.1	Public → Private	S
v.1.5.2	Public → Protected	S~H
v.1.5.3	Protected → Private	SH
v.1.5.4	Protected → Public	-
v.1.5.5	Private → Public	-
v.1.5.6	Private → Protected	-
v.1.6	Variable change (Static/Non-static)	
v.1.6.1	Static → Non-static	S+L
v.1.6.2	Non-static → Static	-
m.2.1	Method change (Static/Non-static)	
m.2.1.1	Static → Non-static	I+L
m.2.1.2	Non-static → Static	L
m.2.2	Method change (Abstract/Non-abstract)	
m.2.2.1	Abstract → Non-abstract	H+ie(3.1.2)+L
m.2.2.2	Non-abstract → Abstract	H+ie(3.1.1)+L
m.2.3	Method Return type change	
m.2.3.1	Non-abstract method	H+ie(3.1.2)+L
m.2.3.2	Abstract method	H+L
m.2.4	Method implementation change	L
m.2.5	Method signature change	
m.2.5.1	Non-abstract method	I+ie(3.1.2)+L
m.2.5.2	Abstract method	H+L
m.2.6	Method Scope change	
m.2.6.1	Public → Private	
m.2.6.1.1	Non-abstract method	I
m.2.6.1.2	Abstract method	-
m.2.6.2	Public → Protected	
m.2.6.2.1	Non-abstract method	~H I
m.2.6.2.2	Abstract method	-
m.2.6.3	Protected → Private	
m.2.6.3.1	Non-abstract method	H I
m.2.6.3.2	Abstract method	-
m.2.6.4	Protected → Public	
m.2.6.4.1	Non-abstract method	-
m.2.6.4.2	Abstract method	-
m.2.6.5	Private → Public	
m.2.6.5.1	Non-abstract method	-
m.2.6.5.2	Abstract method	
m.2.6.6	Private → Protected	
m.2.6.6.1	Non-abstract method	-
m.2.6.6.2	Abstract method	-
m.2.7	Method addition	
m.2.7.1	Abstract method	ie(3.1.1)
m.2.7.2	Non-abstract method	I+ie(3.1.2)+L
m.2.8	Method deletion	
m.2.8.1	Abstract method	ie(3.1.2)
m.2.8.2	Non-abstract method	I + ie(3.1.1)+ L
c.3.1	Classe change (Abstract/Non-abstract)	
c.3.1.1	Non-abstract → Abstract	G+H+I+L
c.3.1.2	Abstract → Non-abstract	H+L
c.3.2	Classe deletion	
c.3.2.1	Non-abstract class	S+G+H+I
c.3.2.2	Abstract class	S+H+I
c.3.4	Class inheritance derivation	
c.3.4.1	Public → Private	S+I
c.3.4.2	Public → Protected	~H(S+I)
c.3.4.3	Protected → Private	H(S+~SG+~S I)
c.3.4.4	Protected → Public	-
c.3.4.5	Private → Public	-
c.3.4.6	Private → Protected	-
c.3.5	Class addition	-
c.3.6	Class inheritance structure	
c.3.6.1	Abstract class addition	S+G+H+I+ ie(3.1.1)+ L
c.3.6.2	Non abstract class addition	H+L
c.3.6.3	Abstract class deletion	H+ie(3.1.2)+ L
c.3.6.4	Non abstract class deletion	H+L

A maintainability analysis of the code produced by an EJBs automatic generator

Ignacio García-Rodríguez de Guzmán, Macario Polo, Mario Piattini

ALARCOS Research Group
Information Systems and Technologies Departament
UCLM-Soluziona Research and Development Institute
University of Castilla-La Mancha
Paseo de la Universidad, 4 – 13071 Ciudad Real, Spain
{Ignacio.GRodriguez, Macario.Polo, Mario.Piattini}@uclm.es

Abstract. Design and development of Web applications is an increasinglydemanded topic. However, successive changes to their code and databases result in a progressive decreasing of its quality and maintainability. Because of that, we have built a tool for the automatic generation of multilayer web components-based applications to manage databases. The source code of these applications is automatically generated, being this one optimized, corrected and already pre-tested and standardized according to a set of code templates. This paper makes an overview of the code generation process and, then, shows some quantitative analysis related to the obtained code, that are useful to evaluate its maintainability. This study is important for developers since they will probably require to implement some changes for its adaptation to the final requirements.

1. Introduction

Reengineering is one of the most powerful tools offered by software engineering to maintain legacy systems (**Fig. 1**). According to [1], reengineering is composed in turn by other two techniques, the "forward" and the "reverse engineering"

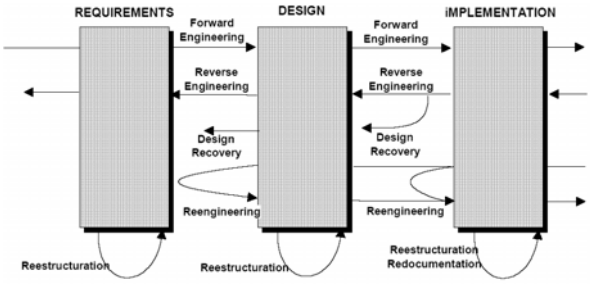


Fig. 1. Simplified reengineering model

Reverse engineering is the process of building abstract formal specifications from source code of a legacy system that can be later used to build new versions of the system, but now, using forward engineering [1].

In this context, we have developed a tool which generates Web applications from a relational database applying complete reengineering process. These Web applications are generated automatically, and support the management of a relational database. According to [2-5], the most usual practice when reverse engineering is applied to databases is to obtain an entity-relation scheme, although, other proposals get an object oriented representation from the database, usually as a class diagram [6, 7]. The use of class diagrams instead of ER schemas provides, from a reengineering point of view, the possibility of taking advantage of the object oriented paradigm constructions for the later steps.

Because of their nature, web applications have a complex development process, especially when a *middleware* must support the management of the database, and security of transaction constraints must be taken into account. *Enterprise JavaBeans* is a technology specifically designed for dealing whit this problems, but these characteristics (such as indirect relationship among classes and interfaces, that are managed by component containers) make difficult its development and maintenance.

Our proposal is based on a tool which automatically generates distributed component-based applications (specifically EJB components and Web Services, both written in Java), using some principles of software engineering inside them. Some of these principles are the use of design patterns (which provide great consistency, extensibility and understandability to the application). As a result, applications can be easily extended adding new features which implement new. Furthermore, some technical documentation is generated when the web application is generated. This documentation helps us in the afore-mentioned maintenance process, making easier the modification of the source code. In addition, automatic development of these kind of web applications lets to the development team to save a lot of time. In order to analyze the maintainability of the generated code, in this paper we make a quantitative analysis of the generated source code by means of the use of some object-oriented metrics. A quantitative way, an overview of such easy is to maintain these applications.

This paper is organized as follows: Section 2 contains an overview of some related technologies and metrics; in Section 3, some metrics are applied to an example web application obtained from a relational database by our tool. Result are shown and commented in the same section; finally, we draw our conclusions and future lines of work in Section 4.

2. Web Application technologies and Metrics for source code evaluation

From a relational database, our tool generates a multilayer application [8] based on EJB components and JSP pages. **Fig. 2** shows the general architecture of the automatically web applications.

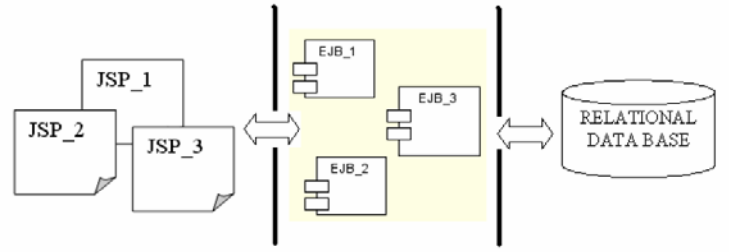


Fig. 2. Basic architecture of the automatic generated applications

In **Fig. 2** we can distinguish a layer made up of JSP pages whose goal is to offer a friendly interface to the user in order to manage the database. The middle layer is a middleware composed by EJB components which implements the logic to perform the management of the database. The third layer is made up of the relational database and, maybe, some additional classes. We do not provide an analysis of neither the tool nor of the generated web application from since this points are out of the scope of this paper.

The most important elements of the generated web applications are the EJB (*Enterprise Java Bean*) components, which are components written in Java language. An EJB component has a couple of interfaces, a class which implements the methods (of the interfaces and others) and a set of additional classes which gives support for some features that could be necessary implement. Actually exits three types of EJB components (*Entity Beans*, *Message Driven Bean* and *Session Bean*. For our proposal, the most interesting EJB type is the *Entity Bean*, because this one referents a persistent entity existing in the relational database.

As we said in the beginning, these applications carry a substantial complexity, because there are some technologies involved in the development process in order to implement all the features, and also, to delegate the database management to the component-based *middleware* requires an additional effort. This is due to the fact that we have to program the necessary logic to orchestrate all the components in such way that the database integrity be respected. That corresponds to define the choreography among the EJB components.

After studying the problem, we notice that the development process of such applications could be performed in an automatically way, because the generated source code could be predicted. The preliminary analysis let us to generate free-error code, and as far as possible, this code is already optimized by means of the use of design patterns. In this manner, we obtain the basic number of classes, with the basic number of methods per class for each component, being written both classes and methods in a clearly and concise. This allows the possibility of realize task of adaptive and perfective maintenance in the future, when new features and requirements have to be added to the web applications in order extend the offered services.

To check these assertions, we will use some well-known software metrics to verify the quality of the source code of the generated applications. The used metrics are the following:

- LOC (*Lines Of Code*): This metric is the sum of lines of the source code of the
- class.
- WMC (*Weighted Methods Class*): This metric is the sum of the complexities of methods of a class, this is, the sum of the ciclomatic complexities.
- CBO (*Coupling Between Objects Classes*): This metric measures coupling among classes.

According to several studies, high coupling is the best predictor of the fault proneness of classes [9]. When the coupling or complexity understandability and testability of the system decreases, and any attempt of change something in maintenance task will be hard and difficult. So, these metrics are good predictors of the quality of our generated Web applications.

A database example (see **Fig. 3**) has been designed to illustrate the results of applying these metrics to the obtained source code. The database schema is very simple but enough for our illustrative goal.

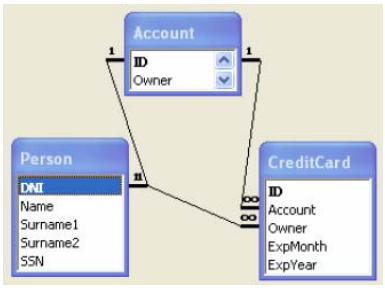


Fig. 3. A simple database

Once we have the database schema, the last step is generating the source code of the Web application. The following sections deal with the measure of this generated source code.

3. Source code quality measure

With out tool, an EJB is generated for each table. **Fig. 4** shows the signature of the operations generated for the *CreditCard* table (**Fig. 3**).

Next sections concrete present the calculus of the values of these metrics for these EJBs; Section 3.4 includes the description of the equations for predicting their values from the database schema.

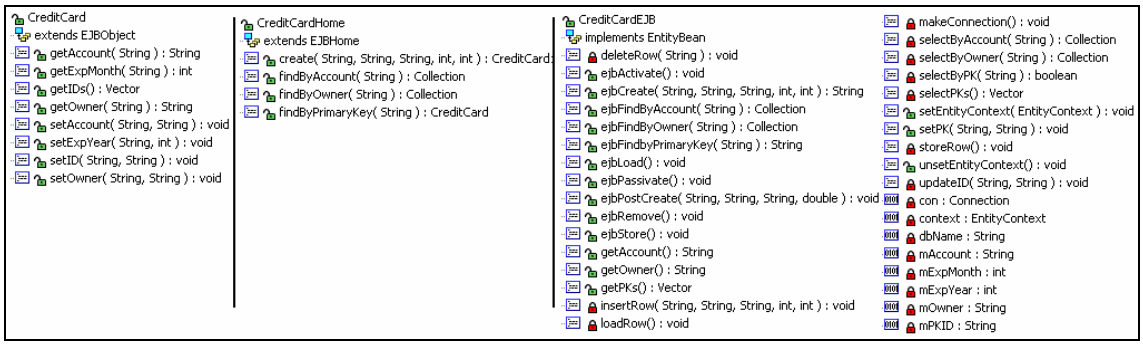


Fig. 4. Classes and interfaces automatically generated from the table CreditCard

3.1. Lines of Code

This metric measures the total number of lines ended with a semicolon in classes and interfaces. Below, we show the results for each element of each *Enterprise Java Bean* generated from the original data base:

Account (LOC)		
Account.java	AccountHome.java	AccountEJB.java
12	7	114
CreditCard (LOC)		
CreditCard.java	CreditCardHome.java	CreditCardEJB.java
12	8	151
Person (LOC)		
Person.java	PersonHome.java	PersonEJB.java
12	7	125

The number of lines of code generated depends on the schema of the database, the number of columns and tables, foreign keys, indexes and stored procedures. Also the number of LOC source code generated is very predictable, because lines of code generated from a table are directly proportional to the elements related with it.

3.2. Coupling between objects classes

The high coupling is a non-desirable characteristic in an OO system that can be measured using the *Coupling Between Object Classes* metric (CBO). CBO is a count of the number of classes a class is coupled to. It is measured by counting the number of related class hierarchies on which a class depends [10].

Inside the source code generated by our tool, coupling depends directly on the scheme of the database too. So coupling is directly proportional to the number of foreign keys existing among tables. For example, if there is a table in the database with three foreign keys to other tables, the EJB which represents this table will be

related with the other three EJBs epresenting the tables whose primary keys are foreign keys in the first table.

For this reason, the coupling measured here will be the existing coupling among components, not among classes, because coupling among classes automatically generated will be a constant. Other thing is the coupling caused by an external developer that modifies the source code in order to add some functionality or new features to the generated application. Because this relation among components depends on the number of foreign keys in the tables of the database, the level of coupling of the system will be also predictable.

```
package mypackage2;

import java.util.Collection;
import java.rmi.RemoteException;
import javax.ejb.*;

public interface CreditCardHome extends EJBHome
{

  public CreditCard create(String ID, String Account, String Owner, int ExpMonth, int Year)
    throws RemoteException, CreateException;

  public CreditCard findByPrimaryKey(String ID)
    throws FinderException, RemoteException;

  public Collection findByAccount(String Account)
    throws FinderException, RemoteException;

  public Collection findByOwner(String Owner)
    throws FinderException, RemoteException;

}
```

Fig. 5. CreditCardHome Interface with its 8 lines of code

Continuing with our example, the coupling from the component point of view is represented in **Fig. 6**. As we can see, the *CreditCard EJB* depends of the *Account* and the *Person* EJBs. This figure can be compared with **Fig. 3**, where we can clearly see the foreign keys.

According to [10], coupling between objects should not be greater than 5 since higher CBO decreases system understandability, avoids the reuse of components and makes more costly maintenance. Our tool keeps the coupling between classes and components at the minimum level.

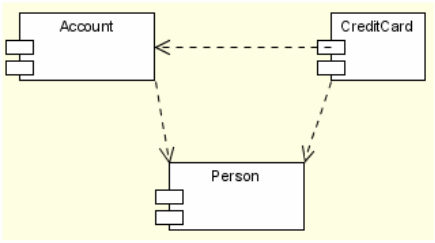


Fig. 6. Coupling between EJBs

3.3. Weighted Methods per Class

The last metric applied to the generated source code by our tool from a relational database is the *Weighted Methods per Class* (WMC). This metric is very similar to the *McCabe Ciclomatic Complexity* [11].

As Ciclomatic Complexity, [11] WMC gives the minimum number of test cases for a given system, supposing each decision condition as a different decision node; when the complexity is greater than 10, the probability of find faults in code grows, and so, we should raise again the architecture of the module which obtains this punctuation.

According to [10], WMC, must be lower than 100, so a class must have at most twenty methods per class and the ciclomatic complexity per method must be lees than 5. WMC is given by the following expression:

$$\sum_{i \in classes} \sum_{j \in Methods_i} Ciclomatic\ Complexity(j)$$

In our small example, WMC for each class and for all the components are the following:

Account (WMC)		
Account.java	AccountHome.java	AccountEJB.java
8	3	41
TOTAL: 52		
CreditCard (WMC)		
CreditCard.java	CreditCardHome.java	CreditCardEJB.java
8	4	44
TOTAL: 56		
Person (WMC)		
Person.java	PersonHome.java	PersonEJB.java
8	3	40
TOTAL: 51		

As we can notice see, none of the EJB in the example overcomes the limit imposed by [10]. In case, the code generated is fault-free.

In the case that other developers add some code generated by themselves, complexity of web applications could be increased depending on the ability of these developers, although to follow the code and design styles our tool adds some technical documentation in addition to the generated code, and so, developers can notice the design styles and follow them.

```
private void loadRow() throws SQLException
{
  String selectStatement="SELECT Account, Owner, ExpMonth, ExpYear "+
                         "FROM CreditCard WHERE ID = ?;";
  PreparedStatement prepStmt=con.prepareStatement(selectStatement);

  prepStmt.setString(1, mPKID);
  ResultSet r=prepStmt.executeQuery();

  if (r.next())
  {
    mAccount = r.getString(1);
    mOwner = r.getString(2);
    mExpMonth = r.getInt(3);
    mExpYear = r.getInt(4);
    prepStmt.close();
  } else {
      prepStmt.close();
      throw new NoSuchEntityException ("Row with ID:" + mPKID + " not found in database");
 }
}
```

Fig. 7. loadRow method with a 2 ciclomatic complexity level

3.4. Equations to predict metrics (and its maintainability)

Finally, in sight of the result afore-obtained and the source code generated, we have derived some equations. These equations allow to predict some characteristics of the web applications generated from a database.

To predict the *Number of Lines of Code* (LOC) for the EJB components, we can apply the following equation:

$$LOC = K_{LOC} + 13 * N^{o} OfIndexes + 8 * N^{o} Col + 3 * N^{o} ColFK \quad (1)$$

Where K_{LOC} is a constant representing the minimum lines to be always generated and its value is 90; *NºOfIndexes* is the number of indexes in the table associated to the EJB; *NºCol* is the number of columns in the table and *NºColFK* is the number of columns of the table which are foreign keys.

Coupling between objects (CBO), for a given EJB, can be predicted from the table by means of the following equation:

$$CBO_{EJB} = \sum_{i=0}^{FKs} N^{o} Cols(FK_i) \quad (2)$$

Where *FKs* is the set of foreign keys of the table represented by the EJB, *FK* is the foreign key that is being examined, *NºCols()* is a function that obtains the number of columns that targets to different tables inside de same foreign key. Note that a consequence to take in account when we realize this operation is that if columns belonging to the current foreign key are targeting to the same table, functions returns one.

To estimate the *Weighted Methods per Class*, we have obtained other equation:

$$WMC = K_{WMC} + 4 * N^{o} Col + 2 * N^{o} ColFK \quad (3)$$

Where K_{WMC} is a constant representing the minimum ciclomatic complexity to be always generated and its value is 20, *NºCol* is the number of columns of the table associated to the EJB, and *NºColFK* is the number of foreign key columns.

Also, if there is stored procedures in the database, an additional EJB is generated containing methods to call them. In this case, this EJB is not an *Entity Bean* but a *Session Bean*. As well as an *Entity Bean* materializes a record from a table, a *Session Bean* only interacts with the client. For our purpose, the *Session Bean* will allow us to invoke the stored procedures of the database. In order to estimate the effect caused to the calculated metrics, we derive two very simple expressions which give us a measure of LOC and WMC (coupling is not affected). The estimated metrics for the *Session Bean* representing the stored procedures are:

$$LOC = K_{LOC} + 12 * N^{o} StorProc \quad (4)$$

$$WMC = K_{WMC} + 3 * N^{o} StorProc \quad (5)$$

In the *LOC* equation, K_{LOC} is the minimum number of lines always included in the bean, and *NºStorProc* is the number of stored procedures the database In the *WMC* equation, K_{WMC} is a constant which value is 7, and *NºStorProc* is the number of stored procedures in the database. For the stored procedures owned by the system, the tool does not generate code.

As it is seen, the design of the database has a strong influence on the quality of the application that manages it. Using the thresholds proposed by NASA [10] together to equations 1-6 (as predictors of the quality of the application), it is possible to determine, before the application development, that a change in database design is required in order to keep adequate values of maintainability and fault proneness in the application.

4. Conclusions and future work

Development of component-based web applications constitutes a complex process which involve some technologies. For this reason, a tool has been developed in order to automate this process. The fact of generating correct web applications is so important that writing of optimized, easily understandable and documented source code.

The tool presented, give us a very simple method to develop web applications to support the management of a relational database. This management is realized by means of a set of EJB components which constitutes the middleware that implements

all the necessary logic. As the generated application must probably be modified to adapt it to the actual requirements, we have studied the quality of the generated source code from the maintainability point of view. Thus, we have analysed some features of the code as predictors of maintainability. As our prediction method has demonstrated, the developed tool generates code which is easily to maintain and understand.

Other lines of work could consist in develop other techniques which optimize more the source code obtained, reducing the number of EJB components in the systems. Some of these techniques could be the implementation of any heuristic to optimize the number of tables represented by an EJB, or the choreography defined to coordinate the operations of the EJB during the management of the relational database.

5. Acknowledgements

This work is partially supported by the MÁS project (Mantenimiento Ágil del Software), Ministerio de Ciencia y Tecnología/FEDER, TIC2003-02737-C02-02, and the ENIGMAS project, Plan Regional de Investigación Científica, Desarrollo Tecnológico e Innovación, Junta de Comunidades de Castilla La Mancha, PBI-05-058

References

1. Arnold, R.S., *Software Reengineering*, ed. 0-8186-3272-0. 1992: IEEE Press. pp. 675.
2. Andersson, M. *Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering.* in *13th International Conference on Entity-Relationalship Approach*. 1994. Berlin: Loucopolous.
3. Pedro de Jesus, L. and P. Sousa. *Selection of Reverse Engineering Methods for Relational Dabases.* in *Proceedings of the Third European Conference on Software Maintenance*. 1998. Los Alamitos, California: Nesi, Verhoef.
4. Chiang, R., T. Barron, and V.C. Storey, *Reverse engineering of relational databases: extracting of an EER model from a relational database.* Journal of Data and Knowledge Engineering, 1994. **12**((2)): p. pp. 107-142.
5. Hainaut, J.-L., et al. *Database Design Recovery*. in *Eighth Conferences on Advance Information Systems Engineering*. 1996. Berlin.
6. Polo, M., et al., *Generating three-tier applications from relational databases: a formal and practical approach.* Information & Software Technology, 2002. **44**(15): p. pp. 923-941.
7. García-Rodríguez de Guzmán, I., M. Polo, and M. Piattini. *An Integrated Environment for Reengineering*. in *Proceedings of the 21st International Conference on Software Maintenance (ICSM 2005)*. 2005. Hungary, Budapest: IEEE Computer Society.
8. Larman, C., *Applying UML and Patterns*. 1998, New York: Prentice Hall, Upper Saddle River.
9. Briand, L., J. Wuest, and H. Lounis. *Using Coupling Measurement for Impact Analysis in Object-Oriented System*. in *IEEE International Conference on Software Maintenance (ICSM´99)*. 1999. Oxford.
10. Rosenberg, L., R. Stapko, and A. Gallo, *Applying Object Oriented Metrics*. 1999, NASA.
11. Piattini, M.G., et al., *Análisis y diseño de Aplicaciones Informáticas de Gestión: Una perspectiva de Ingeniería del Software*. 2004, Madrid: RA-MA. 710.

Validation of a Standard- and Metric-Based Software Quality Model

Rüdiger Lincke and Welf Löwe

School of Mathematics and Systems Engineering,
Växjö University, 351 95 Växjö, Sweden
{rudiger.lincke|welf.lowe}@msi.vxu.se

Abstract. This paper describes the layout of a project[1] that we want to discuss with the scientific community. In the project, we will validate the automated assessment of the internal quality of software according to the ISO 9126 quality model. In selected real world projects, automatically derived quality metric values shall be compared with expert opinions and information from bug and test databases. As a side effect, we create a knowledge base containing precise and reusable definitions of automatically assessable metrics and their mapping to factors and criteria of the ISO quality model.

1 Introduction

Different studies show that currently more than half of the total costs in ownership of a software system are maintenance costs [7, 15]. Hence, it is important to control software qualities like maintainability, re-usability, and portability directly during the development of a software system.

Software quality is defined in the ISO/IEC 9126 standard [10], which describes internal and external software qualities and their connection to attributes of software in a so-called Quality Model, cf. ISO/IEC 9126-1:2001. The Quality Model follows the Factor-Criteria-Metric model [17] and defines six quality characteristics (or factors), which are refined into sub-characteristics (or criteria). Sub-characteristics in turn are assessed by metrics; they measure the design and development process and the software itself. ISO/IEC 9126-3:2003 provides a set of internal metrics for measuring attributes of the six defined sub-characteristics.

These measurements are currently intended to be performed manually as they require human insights, e.g., with code and document reviews. Manual approaches, however, have a series of drawbacks:

1. They are error-prone since they highly depend on the subjects performing the measurement. Hence, they are not measurements in the mathematical sense, which are required to be objective and repeatable. Humans might oversee or even deliberately ignore certain problems.

[1] The KK foundation, Sweden: Project "Validation of metric-based quality control", 2005/0218.

2. They are time consuming. When taking, e.g., code reviews seriously, people have to read and understand codes that they haven't created in the first place.
3. They might cause tensions in the organizations. There is a conflict of interest when, e.g., the project/quality manager requests reports from a developer, which at the same time is used to evaluate the performance of that developer.

These drawbacks are getting more severe considering current trends in software development, like outsourcing of development and integration of open source components into proprietary systems. For a reliable software production, it is essential to guarantee not only the functional correctness of external components but also the internal qualities of external components. Manual quality measurement is not an option in these settings.

Finally, many customers of software systems, especially governmental organizations or those operating in security and safety critical areas, demand certified process maturity from their vendors, e.g., as specified in the ISO 9000 series [11–13] or the Capability Maturity Model Integration (CMMI) [4]. They require quantitative reasonable statistical control over product quality as a basis for continuous quality improvement in the software and the software development process. This is, for the aforementioned reasons, hard to establish with manual quality control.

Replacing the ISO/IEC 9126 metrics manually assessing internal qualities with metrics allowing for automatic measurement resolves the above problems. The remaining problem is, however, to ensure that this automated approach is appropriate, i.e. provides a reliable assessment for at least some (sub-) characteristics. Our project aims at this validation.

In the long run, we expect to have a set of tools and methods allowing the automated assessment of software systems. Possible usage scenarios could be:

- monitoring of software quality goals during the development process resulting in early corrective actions;
- assessment of maintenance effort for change or redeveloping decisions;
- control if subcontractors or outsourced software developers meet the agreed quality goals;
- foundation to choose between different software products offered by competing companies. Is the cheaper product really the better choice in the long run?

Yet in all these activities, the tools and methods are expected to be indicators of bad quality, making reviews more efficient and directed. In any case, manual reviews are needed to validate an issue or identify false positives. We do not expect to create a fully automated assessment of software quality which can make decisions completely independent of human insight.

The structure of this paper is as follows. Section 2 explains our research goal and the expected results. Section 3 defines the scientific approaches. Section 4 gives an overview of the knowledge repository. Section 5 provides an overview about the participating companies and projects. Section 6 concludes the discussion.

2 Goal of our Research

We replace ISO/IEC 9126 metrics manually assessing internal qualities with metrics allowing for automatic measurement. This defines an adapted Quality Model. We validate the significance of this Quality Model with experiments in four selected software development projects of four Swedish companies[2] ranging from a small software company to a large company from the SAAB group.

The project will deliver both research insights and practical methods and tool support for participating companies.

On the *research side*, we expect two major contributions:

1. We define an adapted Quality Model, assessing internal quality (sub-) characteristics as defined by an industry standard with well-established metric analyses as proposed by the research community. This quality model is published as a compendium of software quality standards and metrics [16].
2. We validate the significance of that novel Quality Model, i.e. we support or disprove the hypothesis that static metric analyses allow for an assessment of (some) internal qualities of software systems.

Together, (1) and (2) provide the theoretical basis for quality management assessing industrially standardized software qualities in an effective way, since they are significant and objective, and in an efficient way, since they are automated.

On the *practical side*, we produce tools and methods supporting the quality assessment of software under development having a thorough theoretical basis. By implementing them in the participating partner companies, we gain understanding of their practicality and usefulness.

3. We get insights on how our theory, tools and methods integrate with different quality management processes existing in industry. This includes insights on initial efforts for integration and possible/necessary adaptations of these processes.
4. We understand the speed-up in performance of assessing internal quality automatically vs. manually, since we implement both approaches: the manual standard approach for validating the significance of the new automated approach.

Regarding the practical contributions we can expect that the participating companies already have quantitative information available, which can be understood as manual metrics. Of course, the quality of the available information can not be expected to be sufficient for our evaluation, and modifications will be necessary. Therefore, both approaches will be implemented and adjusted in parallel.

As a side effect, we expect a higher awareness of internal quality issues in participating companies as a first result. We even expect improvements of software

[2] Respecting our Intellectual Property Rights agreement we do not name the companies.

quality in the companies as well as improvements in their development process. These effects, however, will be evaluated qualitatively and subjectively in the first place and not validated statistically.

3 Scientific Approaches

This section describes the scientific approach for validating the significance of the metric-based Quality Model. First, we informally summarize the general idea for the validation. Thereby, we also give some examples for the metrics we use. Second, we define the precise goal and questions for the validation. Finally, we provide the background for the statistical analysis.

3.1 Validation Idea

The idea for the validation is to use static and dynamic (metric) analyses applied on the version history of particular software systems, and additional information sources like bug databases, and even human insights.

To avoid confusion, we distinguish model metrics from validation metrics. The former are automated metrics in the new Quality Model mapped to sub-characteristics. The latter are metrics assessing the (sub-) characteristics directly, but with much higher effort, i.e. with dynamic analyses or human involvement, or a posteriori, i.e. by looking backward in the project history.

For instance, a model metric for the sub-characteristic "Testability" might assess the number of independent paths in a method like the McCabe Cyclomatic Complexity metric. A corresponding validation metric of this sub-characteristic might count the actual coverage of test cases for that method. The former can be assessed easily by static metrics; the latter requires dynamic analyses.

A model metric of the sub-characteristic "Changeability" might assess the change dependency of client classes in a system triggered by changes in their server classes like the Change Dependency Between Classes [8] metric. A corresponding validation metric of this sub-characteristic might observe the actual change costs when looking backwards in the project history. Again, the former can be assessed easily by static metrics; the latter requires version analyses and human effort in documenting programming cost for changes.

The model metric of the characteristic "Maintainability" is some weighted sum of the aggregated values of the metrics assigned to the sub-characteristics of "Maintainability" (to be defined precisely in the software quality model). The validation metric of "Maintainability" could compare change requests due to bug reports, bug-fixing changes, and new bug reports, which again requires backwards analyses and human annotations.

For each single version in the version history of particular software systems, source and binary codes are available, which are input to our model metrics. Additional information about bugs reported, test reports informing about failed/passed test cases, and costs of work spent on the system for maintenance or development is available too and can be associated with a certain version.

This information is input to our validation metrics. Based on this, we can, on the one hand, determine the quality of each version according to our Quality Model, and on the other hand determine the quality based on the additional information. We assume that a higher quality according to our model correlates to fewer bugs, fewer failed test cases, and lower maintenance and development costs. Opposed to this, a low quality, according to our model, would correlate to many reported bugs, failed test cases, and higher costs for maintenance and development, etc.

Our validation succeeded if software systems having high quality according to our model metrics have also a high quality according to the validation metrics and vice versa.

3.2 Validation Goals and Questions

The project goal is a Quality Model allowing for automated metrics-based quality assessment with validated significance.

For validating the significance, we apply the Goal-Question-Metric (GQM) approach [1]. The GQM approach suggests defining the experiment goals, to specify questions on how to achieve the goals, and to collect a set of metrics, answering the questions in a quantitative way.

The goal is to validate the significance of our Quality Model based on the model metrics. Questions and sub-questions are derived from the ISO/IEC 9126 directly:

Q1: Can one significantly assess re-usability with the model metrics proposed in the Quality Model?

Q1.1 - Q1.4: Can model metrics significantly assess understandability, learnability, operability, and attractiveness, respectively, in a reuse context?

Q2: Can one significantly assess efficiency with the model metrics proposed in the Quality Model?

Q2.1 - Q2.2: Can model metrics significantly assess time behavior and resource utilization?

Q3: Can one significantly assess maintainability with the model metrics proposed?

Q3.1 - Q3.4: Can model metrics significantly assess analyzability, changeability, stability, and testability?

Q4: Can one significantly assess portability with the model metrics proposed?

Q4.1 - Q4.2: Can model metrics significantly assess adaptability, and replaceability?

For answering each sub-question, we need both a number of model metrics, which are defined in our Quality Model, and validating metrics, which are defined in our experimental setup, cf. examples above.

3.3 Background for Statistical Analysis

The basis for the statistical analysis of an experiment is hypothesis testing [18]. A negative hypothesis is defined formally. Then the data collected during the experiment is used to reject the hypothesis, if possible. If the hypothesis can be rejected, then intended positive conclusions could be drawn.

Our negative null hypothesis H_0 states that correlations of model and validation metrics are only coincidental. This hypothesis must be rejected with as high significance as possible. We start for all our analyses with the standard borderline significance level of 0.05, i.e. observations are not coincidental but significant with at most a 5% error possibility. The alternative hypothesis H_1 is the one that we can assume in case H_0 is rejected.

To define the hypothesis, we classify the measured values as high, average, and low. For this classification we use a self-reference in the software systems under development: systems are naturally divided in sub-systems, e.g., packages, modules, classes etc. More precisely, for each (sub-) characteristic c and each sub-system s:

1. We perform measurements of model and validation metrics.
2. The weighted sum as defined in the Quality Model defines aggregated values $VM(c, s)$ from values measured with model metrics. We abstract even further from these values and classify them instead with abstract values $AM(c, s)$. It is:
   - $AM(c, s)$ = high iff $VM(c, s)$ is among the 25% highest values of all sub-systems,
   - $AM(c, s)$ = low iff $VM(c, s)$ is among the 25% lowest values of all sub-systems, and
   - $AM(c, s)$ = average, otherwise.
3. The validation metrics provide values $VV(c, s)$ for direct assessment of (sub-) characteristics.

Abstraction and normalization from the precise metric values VM to the abstract values AM is necessary, since VM delivers values in different ranges and scales. For instance the Line Of Code metric has positive (in theory infinite) integer values, whereas the Tight Class Cohesion metric delivers rationale values between 0.0 and 1.0. However, they all have in common that there is a range in which values are acceptable and outlier ranges which indicate issues.

Selecting 25% as boundary values seems to be a suitable first assumption. We expect that the majority of the values vary around a (ideal) median, whereas the outliers are clearly distinguished. The values will be adjusted if first analysis results suggest other boundaries.

Our statistical evaluation studies the effect of changes in $AM(c, s)$ (independent variables) on $VV(c, s)$ (dependent variables). The hypotheses H_0 and H_1 have the following form:

H_0 : There is no correlation between $AM(c, s)$ and $VV(c, s)$.
H_1 : There is a correlation between $AM(c, s)$ and $VV(c, s)$.

In order to find out which dependent variables were affected by changes in the independent variables, we may use, e.g., the Univariate General Linear Model [20], as part of the SPSS system [19], provided the obtained data is checked for test condition suitability.

4 The Compendium

Currently, we build a knowledge base [16] mapping standard qualities to well-established software metrics and vice versa. This compendium formalizes our hypotheses.

The goal of the compendium is to provide an information resource precisely defining our interpretation of the software quality standards and the software metrics and their variants. Moreover, we propose connections between them. These connections are the hypotheses to be validated.

Currently, the compendium describes only

- 37 software quality properties (attributes, criteria),
- 14 software quality metrics.

The 37 quality properties are taken from the ISO 9126-1 standard. For a description, we refer to the ISO standard [10]. The 14 software quality metrics are taken from different well know metrics suites like Chidamber and Kemerer [5], Li and Henry [14], Bieman and Kang [2], or Hitz and Montazeri [9, 8] and contain among others Weighted Method Count (WMC), Tight Class Cohesion (TCC), Lack of Cohesion in Methods (LCOM), McCabe Cyclomatic Complexity (CC), Lines Of Code (LOC), Number Of Children (NOC), Depth of Inheritance Tree (DIT), Data Abstraction Coupling (DAC), and Change Dependency Between Classes (CDBC), etc. The non-object-oriented metrics are mainly size and complexity metrics. The object-oriented metrics focus on cohesion, coupling, and inheritance structure.

We chose metrics that have been discussed, are accepted, validated in case studies, and commented, e.g., by the FAMOOS project [3]. The quality properties and metrics are linked to each other over a double index, allowing us to determine the relevance between the metrics and criteria from each point of view. However, this compendium is meant to be a live document going beyond the normal experience sharing in conferences and workshops. We wish to create a compendium in the spirit of "A compendium of NP optimization problems" edited by Pierluigi Crescenzi and Viggo Kann [6]. The difference is that we propose a double index. The community is welcome to contribute with new metrics or comments, corrections, and add-ons to already defined ones. References to validating experiments or industrial experiences are especially appreciated. The contributions proposed by the community in the form of web forms or emails will be edited, added to the compendium, and used to create new references in the double index, or to change/remove references proofed invalid.

5 The Participating Companies and Projects

Each software system assessed in our project implies other constraints, and the emphasis on the parts of the applied Quality Model varies. This is because the participating companies have distinct expectations and priorities on the quality factors and criteria. Additionally, individual requirements for the application of the quality model result from the architecture and design, programming languages, and development environments.

All selected projects allow for quantitative and qualitative analysis. It is possible to look back in time by accessing their software repositories and to observe their long term evolution during the three years the project shall go on. During this time, the metrics (cf. Sect. 4) can be validated empirically for their usefulness, their limits, and their applicability in new areas like web based applications.

The first company is developing Web applications with C# as implementation language, running on .NET Framework (2.0, Windows) and Mono (Linux pendant). Because one and the same application is required to run on both systems, portability is a particularly important quality factor. We plan to assess in particular the adaptability and replaceability characteristics according to our quality model described in the compendium (cf. Sect. 4). Relevant metrics are, among others, WMC, TCC, LCOM, and LOC. Additionally, we need to define appropriate new metrics, assessing peculiarities with C# code written for .NET and Mono, since not all code running on .NET runs without problems on Mono, unless some special rules are followed. The details for these metrics still need to be discussed and formalized with the company.

The second company is developing Web applications with Java 1.4 and JSP for JBoss using Eclipse and Borland Together J. The product is currently in the maintenance phase. The main interest is on assessing the maintainability and on preventing decay in architecture and design. As the maintainability factor is of highest interest, its characteristics of analyzability, changeability, stability and testability are assessed. The compendium connects them in particular, with LOC, WMC, and CC assessing the complexity, and TCC and LCOM assessing at the cohesion. A particular challenge is that Java Applets, JSP and HTML are part of the design which need to be taken into account when assessing the maintainability of the software system. Other metrics might be included to adapt the Quality Model to the product specific needs.

The third company is developing Web applications with Java. They are currently interested in software quality in general, how it can be automatically assessed, and what it can do for them. They do not have a particular expectation or need for an emphasis on a certain quality factor; therefore, the complete Quality Model, as described in the compendium, will be applied, covering quality factors like (re-)usability, maintainability, and portability, but also reliability and efficiency. Once some higher awareness about what the internal quality has been achieved, the quality model will be adjusted.

The last company is developing embedded control software with C. Their main concern is the quality of design and maintainability. This means the fo-

cus lies on the maintainability factor with an emphasis on the architecture and design. Suitable metrics assessing inheritance, coupling, and cohesion are NOC, DIT, DAC, CDBC, LCOM, and TCC, as described in the compendium. Complementing these metrics, design patterns and anti-patterns might become interesting as well.

We do currently not expect particular problems with collecting data for the model metrics, since the source code in all projects is available in version management systems. Collecting data for the validation metrics is expected to work without problems as well, but might involve more human effort, since the data is not always available in an easily processable way.

6 Conclusions

This paper defines the layout of an experiment in quality assessment. In contrast to other experiments of this kind, it addresses two concerns, which are usually on the common wish list of experiments in this area: comparing automated metrics collection vs. manual metrics collection, and the involvement of industry partners. It should answer the question: Is it possible - in general or to some degree - to automatically assess quality of software as defined by the ISO 9126 standards using appropriate metrics?

We are aware of threats to our approach that are hard to control, like the influence of the projects used for validation (their size, programming language, duration, maturity of company and programmers, etc.) and the validity of the version history and additional information sources. Other problems like a precise definition of standards and metrics (and their information bases) appear only to be resolved as a community activity.

Altogether, the paper aims at entering the discussions on usefulness of such a validation in quality assessment, threads, and possible common efforts.

References

1. Basili, V.R., Rombach, H.D.: The TAME project: Towards improvement-oriented software environments. Trans. Software Engineering 14, 6 (June), IEEE, 1988, pp. 758-773.
2. Bieman, J.M., Kang, B.K.: Cohesion and Reuse in an Object-Oriented System. Proceedings of the ACM Symposium on Software Reusability, April 1995.
3. Bär, H., Bauer, M., Ciupke, O., Demeyer, S., Ducasse, St., Lanza, M., Marinescu, R., Nebbe, R., Nierstrasz, O., Przybilski, M., Richner, T., Rieger, M., Riva, C., Sassen, A., Schulz, B., Steyaert, P., Tichelaar, S., Weisbrod, J.: The FAMOOS Object-Oriented Reengineering Handbook. `http://www.iam.unibe.ch/~famoos/handbook/`, October 15, 1999.
4. Capability Maturity Model Integration (CMMI). `http://www.sei.cmu.edu/cmmi/`, 2006.
5. Chidamber, S. R., Kemerer, C. F.: A Metrics Suite for Object Oriented Design. IEEE Transactions on Software Engineering, vol. 20, no. 6, pages 476-493, June 1994.

6. Crescenzi, P., Kann, V., Karpinski, M., Woeginger, G.: A compendium of NP optimization problems. http://www.nada.kth.se/~viggo/wwwcompendium/, 2006.
7. Erlikh, L.: Leveraging legacy system dollars for E-business. IT Pro, IEEE, May/June 2000, pp. 17-23.
8. Hitz, M., Montazeri, B.: Measure Coupling and Cohesion in Object-Oriented Systems. Proceedings of International Symposium on Applied Corporate Computing (ISAAC'95), pages 24, 25, 274, 279, October 1995.
9. Hitz, M., Montazeri, B.: Chidamber and Kemerer's Metrics Suite; A Measurement Theory Perspective. IEEE Transactions on Software Engineering, vol. 22, no. 4, pages 267-271, April 1996.
10. ISO/IEC 9126-1 Software engineering - Product Quality - Part 1: Quality model, 2001.
11. ISO 9000:2005 Quality management systems Fundamentals and vocabulary, 2005.
12. ISO 9001:2000 Quality management systems Requirements, 2001.
13. ISO 9004:2000 Quality management systems Guidelines for performance improvement, 2000.
14. Li, W., Henry, S.: Maintenance Metrics for the Object Oriented Paradigm. IEEE Proceedings of the First International Software Metrics Symposium, pages 52-60, May 1993.
15. Lientz, B.P.,Swanson, E.: Problems in application software maintenance. Communications of the ACM 24 (11), ACM, 1981, pp. 763-769.
16. Lincke, R., Löwe, W.: Compendium of Software Quality Standards and Metrics. http://www.arisa.se/compendium/, 2006.
17. McCall, J. A., Richards, P.G., Walters, G.F.: Factors in Software Quality. Volume I. NTIS AD/A-049 014, NTIS Springfield, VA, 1977.
18. Spiegel, M., Schiller, J., Srinivasan, A.: Probability and statistics. New York: McGraw-Hill, 2001.
19. SPSS. http://www.spss.com, 2005.
20. Walpole, R.E.: Probability and Statistics for Engineers and Scientists. Prentice Hall, NJ, 2002.

A Proposal of a Probabilistic Framework for Web-Based Applications Quality

Ghazwa Malak[1], Houari Sahraoui[1], Linda Badri[2] & Mourad Badri[2]

[(1)] Department of Computer Science and Operational Research
University of Montreal, Montreal, Qc, Canada, H3T 1J4
{rifighaz@iro.umontreal.ca; sahraouh@iro.umontreal.ca}

[(2)] Department of Mathematics and Computer Science
University of Quebec at Trois-Rivières
Trois-Rivières, Qc, Canada, G9A 5H7
{Linda.Badri@uqtr.ca; Mourad.Badri@uqtr.ca}

Abstract. Many studies on quantitative evaluation of Web-based applications quality have proposed metrics, tools and models. Most of these studies do not address some key issues inherent to this field such as causality, uncertainty and subjectivity. In this paper, we propose a framework for assessing Web-based applications quality by using a probabilistic approach. The approach uses a model including most factors related to the evaluation of Web-based applications quality. A methodology regrouping these factors, integrating and extending various existing works in this field is proposed. A tool supporting our assessment methodology is developed. Some preliminary results are reported to demonstrate the effectiveness of our model.

1 Introduction

Web-based applications are nowadays widely used: reservation systems, e-commerce sites, multi-media applications, stock exchange transactions, etc. They allow the user to create, publish, handle, and store data. Quality assurance of these applications is now difficult to circumvent, and quality expectations are very high [23]. Nevertheless, Web-based applications are often developed in an ad hoc manner resulting in poor quality systems [12].

Many authors proposed guidelines, checklists, metrics and tools, methodologies and models [6, 10, 16, 17] to assess the quality of Web sites or pages. However, several quality factors are subjective and the quality assessment is based, at least partially, on human inspection and judgment [3]. Moreover, defining metrics for these applications is still incomplete and confusing. Metrics are sometimes not rigorously defined, nor empirically or theoretically validated [4]. Furthermore, many approaches [3, 17] proposed hierarchical quality models, which are subjective [22].

To make things worst, Web-based applications are evolving systems. Therefore, they often yield uncertain and incomplete measurements [1]. Structuring quality factors comprises uncertainty. Measuring these factors involves inaccuracy and

subjectivity. In addition, clearly identifying the relationships that may exist between some factors is complex.

Our objective, in this work, is to propose a framework that allows to asses more objectively Web-based applications quality. The aim is to solve some complexity, uncertainty and subjectivity problems when representing, structuring and measuring factors. A well-known solution to problems involving uncertainty is Bayesian Networks [15]. We believe that using a probabilistic approach may helps addressing some key aspects not well considered in the existing studies. It helps supporting more effectively causality, uncertainty and subjectivity problems inherent to the web field.

The rest of this position paper is organized as follows: Section 2 discusses the motivation behind the use of a probabilistic approach to model Web-based applications quality. Section 3 illustrates the application of the proposed approach to the evaluation of the Navigability design criterion. Section 4 concludes the paper and gives some future work directions.

2 A probabilistic approach to model Web applications quality

A quality model is essentially a set of criteria that are used to determine if a website reaches certain levels of quality [3]. Many work on Web applications quality assessment [6, 17] was based on the description of the quality characteristics suggested by the ISO/IEC 9126 standard [8] and adapted to the Web.

2.1 Critical analysis of existing studies

Nowadays, an abundance of guidelines and criteria affecting the quality of Web-based applications can be found in the literature. Conversely, little consistency exists between them making it difficult to know which guidelines to use [6]. However, when looking in depth into different works many limitations can be reported:

1- Some criteria are subjective [16, 17].
2- Optimal values, as mentioned by Ivory [6], are often contradictory for many criteria. Therefore, there is uncertainty in the determination of threshold values for several criteria.
3- Given the various application domains in the Web, the importance of balancing criteria emerges [13]. However, attributing different weights to sub-criteria adds subjectivity to the evaluation.
4- Sub-Criteria and criteria have causal relationships, but can be regrouped in different ways. Thus, it is uncertain if the retained grouping is the relevant.
5- The same criterion can affect simultaneously several criteria, other sub-characteristics or characteristics [13]. These interdependencies are difficult to represent in a hierarchical way.

Consequently, we aim in this proposal to develop a framework taking into account: the subjectivity in criteria evaluation, the difficulty in balancing criteria, the uncertainty in the determination of the threshold values, the uncertainty when regrouping criteria and, the interdependencies between criteria. Thus, in building a

quality model, reasoning with probabilities dealt with weighting criteria and uncertainty problems. Using graphical representation provides a naturally interesting interface by which we can model interacting sets of criteria.

2.2 Objectives through the adoption of Bayesian Networks (BNs)

A BN is a directed acyclic graph, whose nodes are the uncertain variables and edges are the causal or influential links between variables. A conditional probability functions model the uncertain relationship between each node and its parents [15]. In our context, and to develop a quality model for Web applications, BNs seem offering an interesting framework. With BNs it is possible to:

- Represent the interrelations between criteria in an intuitive and explicit way by connecting causes to effects. Such a graph, as explained in [14], facilitates the comprehension of the model, its validation, its evolution and its use.
- Incorporate current existing criteria gathered in our previous work [13].
- Resolve the problems of subjectivity of certain criteria and the uncertainty when structuring and weighting criteria by the use of probabilities.
- Use this model to perform predictions about the application quality.
- Exploit expert judgments in the quality predisction.

The Bayesian approach considers the probability as being a dynamic entity that is updated as more data arrive [1]. Therefore, a BN model can be used to evaluate, predict, diagnose or optimize decisions when assessing Web applications quality.

Building a BN for a particular quality model can be done in two stages: build the graph structure and define the probability tables for each node of the graph. To build the graph structure, criteria are considered as random variables and represent the nodes of the BN. Criteria affecting the same criterion should be independent variables. On the other hand, the basis for conditional probabilities in a Bayesian Network can have a different epistemological status, ranging from well-founded theory over frequencies in a database to subjective estimates [9]. Both cases can be used for the same network.

In the following sections, for simplification matter, we illustrate our methodology for a BN fragment corresponding to the navigability design criterion. The Hugin tool software [5] is used to construct and execute our BN.

3 Application of the methodology to the evaluation of the "Navigability Design" criterion

In recent years, navigability design has become the pivot of website design and one of the trickiest areas of website development [25]. Several works recognize the navigability design as an important quality criterion for Web applications [6, 11, 17]. For some authors, the navigability design is a criterion of functionality [17], for others it characterizes usability [6, 11].

According to many definitions, navigability design in a Web application can be determined by: "the facility, for a given user, to recognize his position in the application [16], to locate and link [11, 25] within a suitable time [21] required information, via the effective use of hyper links towards the destination pages [11]". However, this criterion can be also assessed at the page level. In fact, many design elements may be included in a Web page to improve the navigability design.

3.1 Regrouping and classifying criteria

In the existing work about Web navigation quality [11, 16, 17], authors propose many design elements, directives and guidelines to ensure the quality of navigability design. The first step in our methodology [13] consisted in gathering all the suggested criteria that influence the quality of navigability design in a Web page.

Starting from the existing work, we attempt to collect the criteria and guidelines proposed by different authors [11, 17, 21]. The presence of some design elements (e.g. menus, site map) and the respect of the suggested directives (e.g. color change of visited links) are considered as navigability design criteria.

Then, we try to figure out a kind of sub criteria that would be likely to characterize the retained criteria, and allow a better evaluation of the latter as well. For example, it is more accurate to estimate the quality of "Links" criterion by examining the information that may influence it. The result is presented in Table 1.

Table 1. Navigability design criteria.

1 Navigability Design
1.1 Links
1.1.1 Link Number
1.1.2 Link Colors
1.1.3 Link Text
1.1.4 Link Title
1.1.5 Link to Home
1.2 N. Elements
1.2.1 Menus, Bars
1.2.2 Site Map
1.3 Others
1.3.1 Back Button
1.3.2 Search M.
1.3.3 Page Size
1.4 Feedback
1.4.1 Breadcrumb
1.4.1 Current P.
1.4.2 URLs relatives

Table 2. Navigability design criteria refined using the GQM paradigm.

Criteria	Metrics
1. Navigability Design	
1.1 Localize the page	Subjective
1.1.1 Presence of a site map	Y/N
1.1.2 Presence of a current position label	Y/N
1.1.3 Presence of breadcrumbs	Y/N
1.1.4 Relative URLs	Y/N
1.2 Localize the information	Subjective
1.2.1 Presence of navigation elements	Y/N
1.2.2 Presence of a search mechanism	Y/N
1.2.3 Presence of a site map	Y/N
1.2.4 Link text significant	Measure
1.2.5 Presence of a link title	Y/N
1.2.6 Change of color of the visited link	Y/N
1.3 Access or link to the information	Subjective
1.3.1 Hypertext links	Subjective
1.3.1.1 Number of links per page	Measure
1.3.1.2 Presence of breadcrumbs	Y/N
1.3.1.3 Presence of navigation elements	YN
1.3.2 Presence of a site map	Y/N
1.3.3 Back button always active	Y/N
1.3.4 Presence of a link to home	Y/N
1.4 Revisit the page	Subjective
1.4.1 Back button always active	Y/N
1.4.2 Page download time	Measure

3.2 Refinement using the GQM paradigm

The GQM (Goals, Questions, Metrics) paradigm is based on the idea that measurement should be goal-oriented [2]. It allows us to reorganize, extend, improve and validate our quality model and to determine metrics for the retained criteria. Results from applying the GQM paradigm are summarized in Table 2. By comparing Tables 1 and 2, we notice that, after the refinement by GQM, the selected criteria characterize better the criterion "Navigability Design". This new regrouping rises directly from the definition of the navigability design.

We remark that a same sub criterion characterizes different super criteria at the same time. In addition, the evaluation of some criteria is subjective (e.g. Locate, Access). This affects the precision of measurements and thereafter the evaluation of the navigability design quality. The majority of sub criteria can be assessed by their occurrences as 'Yes' or 'No' (e.g. site map, link to home, etc). Some other criteria can be directly measured, for a given Web page, by our evaluation tool [13] (e.g. number of links per page, page download time). However, this classification represents one of the possible ways to assign sub criteria to the super criterion.

3.3 BN construction applied to the navigability design criterion

To build the BN fragment [15] for the navigability design criterion, we construct initially the appropriate graph. According to our proposed definition of Navigability Design, this criterion at the level of a Web page can be determined by the presence of some design elements and mechanisms that allows the user to:

- locate himself and recognize easily the page where he is,
- find within the page required information,
- have the possibility to access this information directly via hyper links,
- have the possibility to return easily to this page, with a suitable time.

For a selected Web page, we suppose that:

- NavigabilityDesignP: is the variable representing the quality of the navigability design criterion at a Web page level.
- Locate: is the variable representing the facility, for a given user, to know exactly in which page of the application he is and localize required information within the page.
- Access: is the variable representing the facility, for a given user, to access to the required information in the destination page from the selected page.
- Revisit: is the variable representing the possibility, for a given user, to return to the selected page with a suitable time.

Thus, NavigabilityDesignP, Locate, Access and Revisit (Figure 1) can be considered as four variables and represented by four nodes. As there is a definition relation between these variables, we can apply the idiom 'definitional / synthesis' [15]. The node NavigabilityDesignP is defined in terms of the three other nodes. The direction of the edges indicates the direction in which a sub criterion defines a criterion, in combination with the other sub criteria.

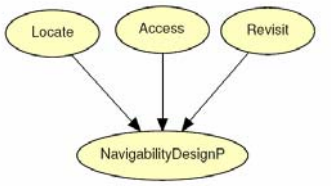


Fig. 1. Instantiation of definitional / synthesis idiom (NavigabilityDesignP).

Furthermore, the NavigabilityDesignP node can not be specified as deterministic function. The definition used is one of different possibilities of defining the navigability design. There is inevitably uncertainty in the relation between the concepts. Thus, we would need to use probabilistic functions to state the degree to which some combination of parent nodes combine to define some child node [15].

The same process is followed to construct the sub networks for "Locate", "Access" and "Revisit" nodes. Then, all fragments are put it together to obtain the BN of the Navigability design at a page level (Fig. 2).

As we can see, the graphical representation is obvious and illustrates causality and interdependencies. The relationships between criteria, even at different levels, are represented clearly. In the following, we exploit probability notions to deal with the uncertainty of the structuration and the subjectivity of certain measures.

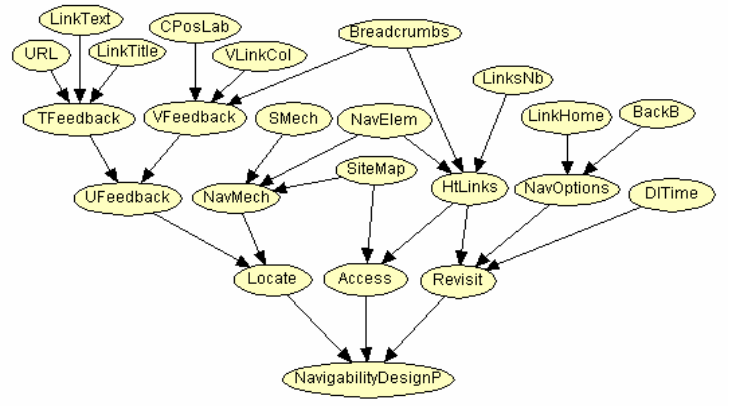


Fig. 2. Final Navigability design BN at Web page level.

3.4 Defining the probability tables for Navigability Design nodes

We build the NPTs (Nodes Probability Tables) using a mixture of empirical data and expert judgments. However, entry nodes of the BN are criteria considered as measurable variables that do not have parents. Intermediate nodes are synthetic nodes defined by their parents and not directly measurable. For these later, we assign conditional probabilities "a priori". Yet, the attribution of probabilities is done differently according to whether the variable is an entry node or an intermediate one.

Intermediate nodes are the criteria affected by their sub-criteria and thus have parents such as nodes TFeedback, Locate or NavigabilityDesignP. These nodes are not directly measurable and their probability distribution is determined by expert judgments. We need to take into account, according to experts, the importance of each one of these variables in the quality assurance of the navigability design for the page. The problem is to find P(TFeedback | LinkText, LinkTitle, URL), and so on for the other nodes.

For entry nodes, the majority of measurements rest on the presence or not of the considered criterion. According to various studies [11, 16, 21], the presence of these sub criteria is recommended, beneficial and contributes to improve the quality of the navigability design. Therefore, a good quality of navigability design, for a Web page, supposes that these criteria are present. So, a probability of 99% is attributed when the criterion is present and a probability of 1% to the contrary case [15].

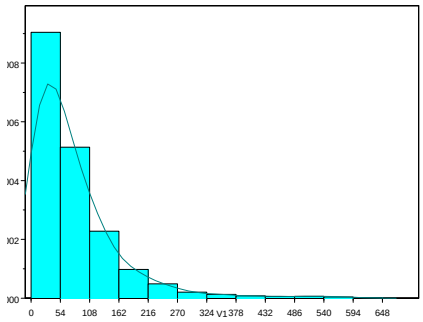


Fig. 3. Frequency histogram of the criterion “Link number” .

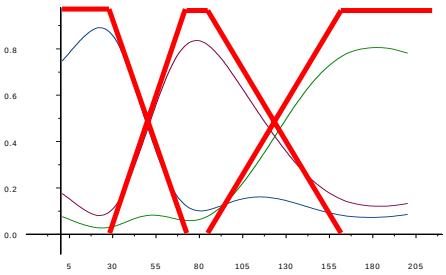


Fig. 4. Fuzzy clusters of the criterion “Link number".

The other entry variables have measurable numerical values. In fact, threshold values for many numerical criteria are contradictory in the literature. For instance, considering the criterion ‘Link number’, we don’t known which value of this criterion is optimal for a Web page in general or what are the limits. Thus, according to [18] a solution consists of using fuzzy logic to turn the variables and the threshold values into fuzzy ones. The fuzzy logic provides an effective conceptual framework for dealing with the problem of knowledge representation in an environment of uncertainty and imprecision [24]. The process used for these nodes is as follows:

1- Measure the criterion value for a large number of Web pages (it is done for over 1700 pages with our automated evaluation tool).
2- Draw a frequency histogram of the measured data (Fig.3). When looking at the obtained frequency histogram, it is difficult to see more than one cluster.
3- Proceed a fuzzy clustering of these data using statistical software (S-Plus) [7]. The result, shown in Fig. 4, reveals the existence of three clusters.
4- Identify the clusters and assign them to fuzzy labels. Starting from these clusters, we can define three fuzzy labels for the Link number criterion.
5- Define the cluster boundaries using approximation method (Fig. 4) (by drawing intersecting lines segments tangent to the curves). Each value of

Link number may then be mapped to three membership values, one for each label (Low, Medium, High).

6- S-Plus computes directly degrees of membership of each measurement. These membership degrees are identified as probabilities. Indeed, Thomas [19] indicates that the Bayesian updating procedure $p(x \mid y) = P(y \mid x).P(x) / P(y)$ can be reinterpreted in terms of fuzzy observations. A given measured value of "Link number" can be reported in the graph of Fig. 4, and probability values are derived and used in the NPT of this criterion.

3.5 Application example of the methodology to the evaluation of a Web Page

The following example reveals the feasibility of our approach using the BN for NavigabilityDesignP. Values for entry nodes are directly measured for a chosen page (http://channel.nationalgeographic.com/channel/programs/) with our evaluation tool.

These values propagate through the BN via the influential links, resulting in updated probabilities for other criteria. Figure 5 shows an application example for the considered page. Although the quality of NavigabilityDesignP will never be known with certainty, it has 88.15% of probability to be good. Many scenarios "What if" can predict the improvement when some design elements are added [15]. Having entered new "evidences" (LinkTitle, VlinkCol and Breadcrumbs), the probability distributions are updated and we get 95.41 of goodness. This scenario demonstrates how much NavigabilityDesignP may be enhanced when some criteria are improved or added.

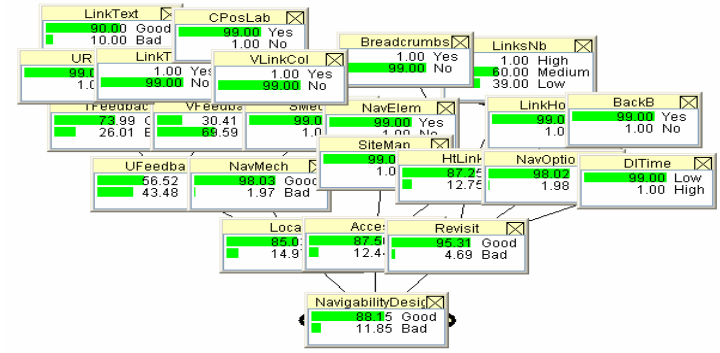


Fig. 5. State of BN probabilities showing the Navigability design quality for the page.

3.6 Rapid validation

As a first experimentation of our approach, we selected some web pages recognized for their good quality (for example starting from *Top 100 of Webby Awards*). Other web pages of poor quality (from *Worst of the Web and Webby Worthy*) were also evaluated for better refining the range of suggested values. We obtained (table 4) a good score for the pages recognized among the *Webby Awards*, and the others from The *Worst of the Web or Webby worthy* achieve a lower score. The results of the

evaluations are significant, which show that the selected and evaluated criteria seem to be relevant.

Table 4. Partial results for the evaluation of "the probability of a good Navigability design quality" at the page level for some selected Web pages.

Web applications	**Navigability Design Quality at page level**	**Navigability Design Quality at page level if High Speed Connection**
Winner of *Webby Awards*	85.44 %	85.44 %
	85.09 %	90.68 %
	80.96 %	89.40 %
	84.83 %	92.81 %
The *Worst of the Web or The Webby Worthy*	59.24 %	67.84 %
	51.22 %	58.26 %
	74.33 %	74.33 %
	57.88 %	57.88 %

We can use the ability of BNs to generate "What if "scenarios. For instance, when we consider high speed internet connection, Download time criterion is improved, and the quality of the Navigability Design criterion is significantly superior.

4 Conclusions and future work

Many studies on quantitative evaluation of Web-based applications quality do not consider causality, uncertainty, inaccuracy and subjectivity problems when dealing with criteria evaluation or regrouping. In this paper, we proposed a general framework supporting the assessment of Web applications quality. The adoption of Bayesian Networks helped us to deal with these weaknesses.

Starting from several studies, we gathered the proposed criteria for Navigability Design quality assessment. Then, criteria and sub criteria were restructured, extended and validated using the GQM paradigm. A BN graph was constructed for the considered criterion. Many experiences, involving different Web pages, were conducted using this BN. A rapid validation of the proposed approach demonstrated its relevancy.

Although if, the model described in this paper was built for only Navigability Design criterion, the proposed framework is extensible and adaptable. It can be used for specific cases to assess a particular criterion, a super criterion, a sub characteristic, a characteristic or the whole quality. This can be done for one page, for specific pages or for all the Web application.

We believe that the present work constitutes an interesting starting point in this field and represents a step up in Web Applications quality evaluation. As future work, we plan to: (1) extend the defined network to cover all quality characteristics, (2) complete a global evaluation for Web applications usability, and (3) validate empirically the proposed model.

References

1. Baldi, P., Frasconi, P., Smyth, P.: Modeling the Internet and the Web; Probabilistic Methods and Algorithms. Wiley (2003)
2. Basili, V.R., Caldiera, G., Rombach, H.D. : The Goal Question Metric Approach. (1996)
3. Brajnik, G.: Towards Valid Quality Models for Websites. Proceedings of the 7th Conference on Human Factors and the Web (2001)
4. Calero, C.; Ruiz, J.; Piattini, M.: A Web Metrics Survey Using WQM. Proceedings of the International Conference on Web Engineering (2004)
5. Hugin Experts at http://www.hugin.com/
6. Ivory, M.: An Empirical Foundation for Automated Web Interface Evaluation. Doctoral Thesis (2001)
7. Insightful: Statistical Analysis Software http://www.insightful.com/products/splus/
8. ISO/IEC (2001) ISO/IEC 9126: Quality characteristics and Guidelines for their use (2001)
9. Jensen, F.V.: Bayesian Networks and Decisions Graphs. Springer-Verlag Inc. (2001)
10. Kirakowski J., Cierlik B.: Measuring the usability of Website. HFES Annual Conference, Chicago (1998) http://www.ucc.ie/hfrg/questionnaires/wammi/research.html
11. Koyani, S. J., Bailey, R. W., and Nall, J. R.: Research-Based Web Design & Usability Guidelines. National Institutes of Health (2003)
12. Lee, C., Suh, W., Lee, H.: Implementing a community web site: a scenario-based methodology. Information & Software Technology, Vol. 46(1). (2004) 17-33
13. Malak G., Badri L., Badri M., Sahraoui H.: Towards a Multidimensional Model for Web-Based Applications Quality Assessment. Proc. of the fifth I. C. E-Commerce and Web Technologies (EC-Web'04), Spain, LNCS Vol. 3182. Springer-Verlag, (2004) 316-327
14. Naïm, P., Wuillemin, P.H., Leray, P., Pourret, O., Becker, A.: Réseaux Bayésiens. (2004)
15. Neil, M., Fenton, N.E., Nielsen, L.,: Building large-scale Bayesian Networks. The Knowledge Engineering Review, 15(3). (2000) 257-284
16. Nielsen, J.: 1996-2006, The Alertbox. Available on-line at www.useit.com/alertbox/
17. Olsina, L. Rossi, G.: Measuring Web Application Quality with WebQEM. IEEE MultiMedia, Vol. 9, No. 4 (2002)
18. Sahraoui, H., Boukadoum, M., Chawiche, H. M., Mai, G. Serhani, M. A, A fuzzy logic framework to improve the performance and interpretation of rule-based quality prediction models for object-oriented software. In the proc. of the 26th Computer Software and Applications Conference (COMPSAC'02). Oxford (2002)
19. Thomas, S.F.: Possibilistic uncertainty and statistical inference. ORSA/TIMS Meeting. Houston, Texas (1981)
20. Vanderdonckt, J., Beirekdar, A.: Automated Web Evaluation By Guideline Review. Journal of Web Engineering, Vol. 4, No.2. (2005) 102-117
21. W3C Recommendation 5-May-1999, Web Content Accessibility Guidelines 1.0.
22. Wikle, C.K.: Hierarchical Models in Environmental Science. International Statistical Review Vol. 71, No.2, (2003) 181–199
23. Wu, Y., Offutt, J.: Modeling and Testing Web-based Applications. GMU ISE Technical ISE-TR-02-08 (2002)
24. Zadeh, L.A.: Knowledge Representation in Fuzzy Logic, IEEE Transaction on Knowledge and Data Engineering, vol. 1, no. 1 (1989) 89-100
25. Zhang, Y., Zhu, H., Greenwood, S.: Website Complexity Metrics for Measuring Navigability. Proceedings of the Fourth International Conference on Quality Software (QSIC'04) (2004)

Investigating Refactoring Impact through a Wider View of Software

Miguel Lopez[1], Naji Habra[2]

[1] Faculty of Computer Science
University of Namur – FUNDP
Namur, Belgium
+32 (0)81 72 49 95
mlo@ info.fundp.ac.be

[2] Faculty of Computer Science
University of Namur – FUNDP
Namur, Belgium
+32 (0)81 72 49 95
nha@info.fundp.ac.be

Abstract

The activity of refactoring —transforming the source-code of a program without changing its external behavior— is now practiced by many software developers. If applied well, refactoring should improve the maintainability of software. To investigate this assumption, we propose a wider view of the software, which includes the different well-known artifacts (requirements, design, source code, tests) and their relationships.
This wider view helps analyzing the impact of a given refactoring on software quality. In this study, we analyze the impact of the refactoring "Replace Conditional with Polymorphsm" by using this wider view of software. And, at the light of this global view of software, it is more difficult to accept that the analyzed refactoring "Replace Conditional with Polymorphsm" improves well the maintainability of software.

Keywords: Refactoring, Maintainability, Dependencies Graph, Measures, Software Model.

1. Introduction

Studying the impact of software refactoring is usually achieved by examining the properties of the software system at only one level. Typically, either the design product or the code product is examined to observe and measure internal properties like coupling, and complexity. The measurement results are then used to assess other external qualities like maintainability.
However any change of a software system at one product level (e.g. the code) would very probably influence other artifacts (e.g. design, test sets …). Classically, refactoring works require, at least, the maintaining of the consistency between these artifacts.
The question is to know if the observations of some properties like "complexity" at one product level is sufficient to capture the effect of the refactoring on the maintainability.
The hypothesis we examine in this paper is the possibility and the pertinence to consider a wider view of the software. The idea is that considering a software system as a composite product including different artifacts (code, design, requirements, tests…) would allow us to observe and to measure other kind of internal properties (e.g. a "complexity" in a wider sense) which would be more relevant to maintainability.
The paper is not an empirical study based on data observation. It is a first step of thought which tries just to investigate the above hypothesis and to clarify the related questions behind it.

2. Refactoring

Refactoring can be defined as "*the process of changing an object-oriented software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure*" (Obdyke, 1992).

Following the previous definition, the main goal of refactoring is to improve the internal structure. Such an improvement could affect external qualities like maintainability, performance, and reliability…

According to (Bois, 2003), (Demeyer, 2003), in most cases, refactoring aims at improving the maintainability of software. Indeed, restructuring a source code should ease the capacity of the software to be modified. Replacing a given structure in the source code by one of the well-known design patterns (Gamma, 1995) is widely admitted as a refactoring which would improve the modifiability.

For instance, implementing the design pattern *Strategy* that allows selecting the suitable implementation of a method in regard with the context would reduce the effort for adding a new behavior. In this sense, the *Strategy* design pattern increases the modifiability of software (Gamma, 1995).

Even if the research on refactoring is mostly focused on source code and design (precisely in the object-oriented paradigm), restructuring activities are also applied on other software artifacts. Restructuring the requirements document with or without changes of the related design or the source code is a recurrent activity in real software projects. Indeed, such phenomenon is known as the instability of requirements, that is, the change requests.

Those other modifications of the software artifacts should be more investigated such as the modifications of the test sets of given software without changing the requirements.

So, though the usual definition of refactoring (Obdyke, 1992) mentions the general term of "software system", the definition of restructuring in general does not impose a restriction to a particular artifact. Indeed, the changes are often performed in the same "level of abstraction". According to the taxonomy of Chikofsky and Cross (Chikofsky, 1990) restructuring is defined as *the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)*.

According to (Mens, 2004), refactoring is often described as a four steps process:

- Identifying where to apply which refactorings (Kataoka, 2001), (Balazinska, 2000), (Simon, 2001)
- Guaranteeing that refactoring preserves software behavior by testing the software or verifying the formal semantics of the program.
- Assessing the effect of refactoring on quality (Tahvildari, 2002).
- Maintaining consistency of refactored software (Bottoni, 2002), (Van Der Straeten , 2003), (Rajlich , 1997)

In the current position paper, we focus on the two last points, that is, the effect on quality and the maintenance of the consistency between artifacts. The developed idea is the importance of considering these two points in conjunction.

Our starting point is the following observation: the positive effect in quality change of a given refactoring can be easily accepted and proved when only one local level of abstraction is considered (that is ignoring the other levels as well as the relationships between the different artifacts). For instance, one can study how the implementation of the *Strategy* design pattern improves the modifiability of the design artifact (or the source code artifact)

Nevertheless, such a restructuring can also affect the relationships with other artifacts, i.e., requirements, test sets.

Therefore, once the effect on the quality is to be assessed, it is safer to study also the impact on the other artifacts. On the one hand, we should check whether the consistency with other artifacts is preserved.

And on the other hand, as the consequences of a given refactoring can seldom affect the dependencies between artifacts, an overhead of complexity can be generated with a possible impact on maintainability.

This overhead can be identified when all the artifacts and their dependencies are considered together. That is when the product "software system" is modeled in a more global and more integrated way than usual.

In order to investigate the above idea, the current work aims at highlighting the possible overhead of "complexity "which introduced by a refactoring.

Here we consider the term "complexity as a general property that corresponds to our intuitive view, such a property needs of course to be defined precisely but it is undoubtedly more general than what is usually

measured on one artifact like Cyclomatic Complexity suggested in (McCabe, 1976).

To do so, we will study an example taken from the refactoring catalogue proposed by Martin Fowler in (Fowler, 1999), and (Refactoring, 2006), that is, *Replace Conditional with Polymorphism*. Precisely, we will analyze the consequences of this refactoring on the dependencies between artifacts in order to determine the impact on maintainability.

3. Running Example

In this Section we describe an example that will be used all along the paper to develop some ideas related to refactoring. This example illustrates a trivial refactoring of object-oriented design, that is, *Replace Conditional with Polymorphism*.

Maintainability Definitions
Before describing the refactoring, we propose some definitions related to maintainability and taken from the ISO/IEC 9126 standard.

Software maintainability is defined as *a set of attributes that bear on the effort needed to make specified modifications* (ISO/IEC 9126, 2001). We only consider three subcharacteristics suggested for the maintainability characteristic in the ISO/IEC 9126 standard: Analyzability, Changeability, and testability.
Analyzability is defined as *Attributes of software that bear on the effort needed for diagnosis of deficiencies or causes of failures, or for identification of parts to be modified* (ISO/IEC 9126, 2001).
Changeability is defined as *Attributes of software that bear on the effort needed for modification, fault removal or for environmental change* (ISO/IEC 9126, 2001).
Testability is defined as *Attributes of software that bear on the effort needed for validating the modified software* (ISO/IEC 9126, 2001).

Replace Conditional with Polymorphism

The initial design depicted in Figure 1 represents class *Document* with a *Print()* method. This method takes one argument *format*, which allows specifying the format of printing (PDF, RTF, HTML). A switch statement implements the Print method.

```
Class Document {
        Print(format) {
            Switch(format) {
                Case 'PDF':
                    PrintPDF();

                Case 'RTF':
                    PrintRTF();

                Case 'HTML':
                   PrintHTML();
        }
      }
}
```

Figure 1 – Code 1

Figure 2 shows the corresponding design of Code 1.

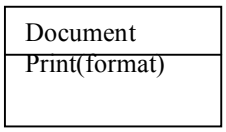


Figure 2 - Design 1

Figure 3 shows the result of the refactoring *Replace Conditional with Polymorphism* on Design 1.

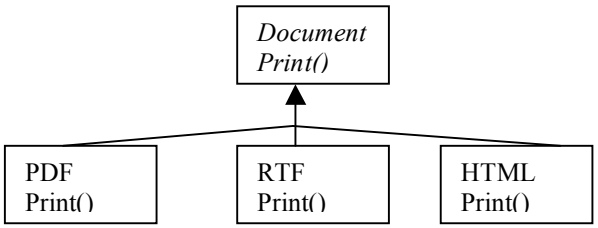


Figure 3 - Design 2

The refactoring adds an abstract class called *Document* within an abstract method called *Print*. The *switch* statement is therefore replaced by three subclasses, which correctly implement according to the format the abstract method *Print*.
In design 2, the parent class *Document* contains an implementation of the part of the behavior (methods) which is shared by the three subclasses.
This type of design, which benefits from the polymorphism of the *Print* method, is known as easier to maintain than the previous design behind the code depicted in Figure 1. For instance, it seems to be easier to add a new feature to print new format (like PostScript). Indeed, we must create a new

subclass PS which implements the *Print* method according to the PostScript format. To test the new behavior, only the new class PS must be tested, since no modification has been performed on the rest of the design.
So, one can reasonably affirm that design 2 is more maintainable, more modifiable, and more understandable than the design behind Figure 1 and Figure 1.

Dependencies Graph

Let us now look at the software as a whole product integrating different artifacts and let us try to highlight the relationships (dependencies) between those artifacts.
To model this global product, we introduce the concept of *dependencies graph*.

Roughly, the dependencies graph is a model of the software that allows navigating through the different artifacts related to a given software by traversing the different dependency relations between these artifacts.
This dependencies graph eases the analysis of the impact generated by a specific refactoring, since it represents all the dependencies existing between artifacts.

More formally, we introduce the concept of dependencies graph as a graph whose nodes are the artifacts or part of artifacts, and the edges are the dependencies between them.

The dependency relationship (represented by an edge in the dependencies graph) is actually a generic relation between two artifacts. (This generic relation can be better specified in further works but at this level limiting this work to the UML notion of dependency is sufficient to highlight the impact of a given refactoring on the dependencies graph.) So, following the UML literature a dependency is defined as *a relationship between parts of an UML model drawn as a dashed line with an arrowhead that indicates that if one thing changes then the change may affect the other thing* (UML, 2001). The dependency is represented by a line without arrowheads when both elements of the relation are mutually dependent, that is, when one element changes the other changes, and vice-versa.

Nodes represent artifacts or part of artifacts. In the current paper, the artifacts considered are the requirements (a textual description of what the software must hold), the classes which are parts of the software design, and the set of unit tests.
Different levels of artifacts are mixed within the same graph: classes with set of unit tests, requirements with classes…
The issue related to the granularity of the artifacts modeled in the dependencies graph is not investigated in the current paper. Therefore, a working assumption concerned in the granularity issue is stated in the paragraph *Working Assumption.*

Working Assumptions

In order to develop the purpose of the current work, two working assumptions must be stated.
Firstly, the "maintainability" attribute (quality) studied in this work can be described as the difficulty a software engineer meets when he/she navigates through the dependencies graph in order to understand, modify and test the software as a whole. In this representation of the maintenance phenomenon, the structure of the dependencies graph described above can affect this difficulty of navigating through it. And in this sense, the more it is difficult to traverse the dependencies graph, the more the software will be hard to maintain, which means that the different maintainability subcharacteristics (understandability, modifiability, and testability) will be negatively affected.
So, the correlation between properties of the dependencies graph and the maintainability subcharacteristics is actually a working assumption that is not tested in the current work. In fact, this assumption represents the base of the current purpose.

Secondly, the granularity of the artifacts used in this work is considered as the unit level. For instance, the requirements described in this paper are seen as the elementary part of a whole set of requirements.
This hypothetic granularity of artifacts helps specifying the dependencies between them. Nevertheless, it is obvious that such assumption must be strongly investigated. In the literature, this issue is known as the *coarse granularity of traceable entities* (Gotel, 1994), (Ramamoorthy , 1986). The traceable entities are the artifacts (requirements, class, code) of the current work.

Structural Indicators of Dependencies Graph & Maintainability

According to the working assumptions previously stated, the more the structure of the dependencies graph is complex, the more the software is difficult to maintain.
By complex, we mean the number of elements and the relations between these elements is very high. In the current case, the structure is the dependencies graph, the elements are the nodes within this graph, and the relationships are the edges between the nodes.

So, the working assumption can be rephrased as follows: the greater the connectivity and the size parameters of the dependencies graph, the more the navigation through this graph is difficult, and the more the software as a whole is difficult to understand, modify and test, that is, to maintain.

According to the rephrasing of the working assumption, the measures candidates to be used are:

- Amount of nodes
- Amount of edges
- Degree

In the next paragraph, we will apply these measures to a dependencies graph before and after a refactoring in order to have a first assessment of the impact on the maintainability of this refactoring according to our assumption.

Dependencies Graph of the Replace Conditional with Polymorphism Refactoring

Figure 4 shows the dependencies graph between three types of artifact (Requirements, Design and Unit Test Set) before applying the refactoring *Replace Conditional with Polymorphism*. In a real case, other artifacts exist (functional tests, integration tests, defects, different design models…). However, in order to be as clear as possible and assure the understandability of the current paper, we reduce the number of dependent artifacts.

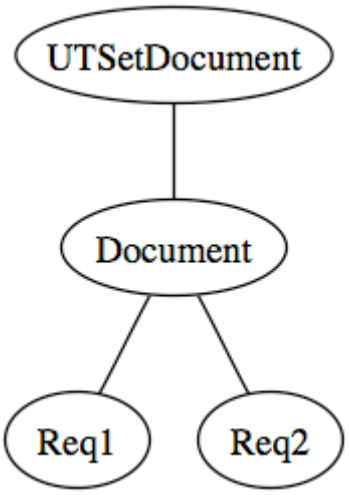


Figure 4 - Before Refactoring (Design 1)

The nodes *Req1* and *Req2* in Figure 4 represent the requirements of the software. *Req1* corresponds with the capacity of editing a document and *Req2* is the capacity of printing a document. Such level of description is certainly insufficient for developing the software. However, it is enough for analyzing the impact of refactoring on the maintanibality.
The *Document* node is the document class. And, the *UTSetDocument* node is the set of unit tests related to the document class. Now, this dependencies graph can be read as follows: *Document* implements the requirements *Req1* and *Req2*. The unit test set *UTSetDocument* tests the design element *Document*. Moreover, the change of the class *Document* could affect the unit tests set *UTSetDocument*, the requirements *Req1*, and *Req2*.

Figure 5 shows the corresponding dependencies graph after refactoring with the same types of artifact.

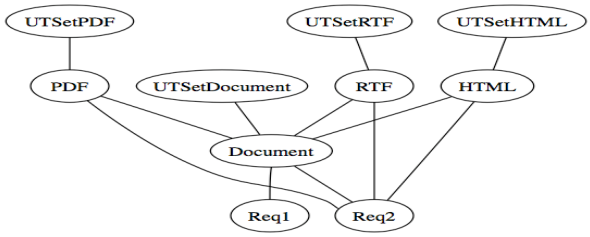


Figure 5 - After Refactoring (Design 2)

Note in Figure 5 that three new design elements (*PDF*, *RTF*, and *HTML*) with the corresponding set of unit tests have been added.

The structural complexity of the dependencies graph seems to be increased by the refactoring. To highlight

this complexity increasing, we compute some volumetric measure of the graph, previously described.
Table A shows the amount of nodes and edges for each design.

	Design 1	Design 2
Nodes	4	10
Edges	3	12

Table A

According to Table A, the amounts of edges and nodes have increased from Design 1 to Design 2. Particularly, the amount of edges of Design 2 is 4 times larger than the amount of edges of Design 1.

Table B shows the measurement of the dependencies graph before refactoring. For each node, the degree is given. The degree of a node in an undirected graph is the number of edges incident to the node.

NODE	DEGREE
REQ1	1
REQ2	1
Document	3
UTSetDocument	1

Table B

Table C shows the same measures after refactoring.

NODE	DEGREE
REQ1	1
REQ2	4
Document	6
PDF	3
RTF	3
HTML	3
UTSetDocument	1
UTSetPDF	1
UTSetRTF	1
UTSetHTML	1

Table C

According to Tables B and C, the degrees of nodes *Req2* and *Document* have increased, respectively from 1 to 4, and from 3 to 6. The degrees of the nodes *Req1* and *UTSetDocument* are constant.
Moreover, six new nodes have appeared in the dependencies graph after refactoring.

So, the amount of elements (artifacts and dependencies, or nodes and edges) has increased after refactoring. In other words, the amount of elements to handle during and after refactoring has increased. Under these new conditions, it is not obvious whether or not the refactoring has really reduced the difficulty to maintain the software (in its global sense).

4. Conclusion

Firstly, it has not been proved neither empirically, nor theoretically (Basili, 1996), (Zuse, 1999), (Kitchenham, 1995) that the dependencies graph indicators used in the previous section are positively correlated with maintainability measures. In a further work, we must answer the following question: Increasing each dependencies graph indicators does it mean that the maintainability decreases?
Nevertheless, even if the measures selected in this first work are not valid, it should be very useful to investigate the properties of such dependencies graph in order to gain understanding on software development

Secondly, the refactoring investigated under the assumptions of this paper with this running example can be considered as having a different effect than one expected. The effect is no more a "reduction" of the complexity but (at least partially) a highlighting of a hidden complexity located in the software or a displacing of this complexity from one artifact to many others.
To observe this highlighting, the software is represented by a dependencies graph. The whole dependencies graph is a global model of the software. And, the refactoring is therefore the operation that reveals some structural complexity of the software dependencies graph.
At the light of this simple example, the structure of the dependencies graph appears to be more complete, and in that sense restructuring can ease the understanding, the modification, and the test. However, the complexity of the software dependencies graph is not reduced, but only clarified. Any clarification can positively affect some maintainability characteristics. But, this assertion is still an assumption that must be verified.

Thirdly, the different software artifacts (requirements, design, code, tests, defects) are often considered as different entities on which some operations can be applied in order to transform one entity into another. For instance, the design is considered as software entity which can be transformed into source code if some transformation rules are correctly applied. According to (Kleppe, 2003), this literature is currently very prolific.

In the current work we propose another view or model of software. Indeed, the different artifacts (requirements, design…) are not distinct entities, but different parts of a more global entity that we call software.
This global entity is made up elementary parts (pieces of requirements, pieces of design…) that interact with each other.
Moreover, this global entity is represented by a dependencies graph whose nodes are the pieces of products (i.e. a class, a requirement…), and whose edges are the dependencies between the pieces of products.
We suggest that the practitioners' view of the "complexity" is close to the complexity of such global entity or dependencies graph.
So, it can be useful and interesting to gain understanding of such dependencies graph, if this represents a suitable and correct global model of software.

5. References

1. (Basili, 1996) V.R. Basili, L.C. Briand and W.L. Melo, `A Validation of Object-Oriented Design Metrics as Quality Indicators', IEEE Transactions on Software Engineering, Vol. 22, No. 10, October 1996, pp. 751-761
2. (Balazinska , 2000) M. Balazinska, E. Merlo, M. Dagenais, and B. Lagüe, and K. Kontogiannis, "Advanced Clone-Analysis to Support Object-Oriented System Refactoring," Proc. Working Conf. Reverse Eng., pp. 98-107, 2000
3. (Bois, 2003)B. Bois and T. Mens," Describing the Impact of Refactoring on Internal Program Quality",http://citeseer.ist.psu.edu/bois03describing.html, 2003
4. (Bottoni, 2002) P. Bottoni, F. Parisi-Presicce, and G. Taentzer, "Coordinated Distributed Diagram Transformation for Software Evolution," Electronic Notes in Theoretical Computer Science, vol. 72, no. 4, 2002.
5. (Chikofsky, 1990) E.J. Chikofsky and J.H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," IEEE Software, vol. 7, no. 1, pp. 13-17, 1990.
6. (Demeyer, 2003) S., Demeyer, "Maintainability versus Performance: What's the Effect of Introducing Polymorphism ?", ICSE 2003, September, 2003
7. (Fowler, 1999) M. Fowler, Refactoring: Improving the Design of Existing Programs. Addison-Wesley, 1999.
8. (Gamma, 1995), Gamma, E., *et al*, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional; 1st edition, January 15, 1995
9. (Gill, 1991) Gill, G., and Kemerer, C., "Cyclomatic Complexity Density and Software Maintenance Productivity," IEEE Transactions on Software Engineering, December 1991.
10. (Gotel, 1994) O. C. Z. Gotel and A. C. W. Finkelstein, "An Analysis of the Requirements Traceability Problem", http://citeseer.ist.psu.edu/gotel94analysis.html, 1994
11. (ISO/IEC 9126, 2001) ISO/IEC 9126-1:2001 "Software engineering -- Product quality -- Part 1: Quality model", ISO, 2001
12. (Kafura , 1987) Kafura, D., and Reddy, G., "The Use of Software Complexity Metrics in Software Maintenance," IEEE Transactions on Software Engineering, March 1987.
13. (Kataoka , 2001) Y. Kataoka, M.D. Ernst, W.G. Griswold, and D. Notkin, "Automated Support for Program Refactoring Using Invariants," Proc. Int'l Conf. Software Maintenance, pp. 736-743, 2001.
14. (Kitchenham, 1995) B. Kitchenham, S. L. Pfleeger, N. Fenton, "Towards a Framework for Software Measurement Validation", IEEE Transactions on Software Engineering, vol.21, No. 12, pp. 929943, December 1995
15. (Kleppe, 2003) Anneke Kleppe and Jos Warmer and Wim Bast, "MDA Explained. The Model Driven Architecture: Practice and Promise", Addison-Wesley, 2003
16. (McCabe, 1976) Thomas J. McCabe, "A measure of complexity", IEEE transaction on software engineering, Vol SE-2, No 4, December 1976
17. (Mens, 2004) Tom Mens, and Tom Tourwe, "A Survey of Software Refactoring", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 30, NO. 2, FEBRUARY 2004
18. (Obdyke, 1992) W.F. Opdyke, "Refactoring: A

Program Restructuring Aid in Designing Object-Oriented Application Frameworks," PhD thesis, Univ. of Illinois at Urbana-Champaign, 1992

19. (Rajlich , 1997) V. Rajlich, "A Model for Change Propagation Based on Graph Rewriting," Proc. Int'l Conf. Software Maintenance, pp. 84-91, 1997
20. (Ramamoorthy , 1986) Ramamoorthy, C.V., Garg, V. & Prakash, A. "Programming in the Large", IEEE Transactions on Software Engineering, Vol. 12, No. 7, pp. 769-783, 1986
21. (Refactoring, 2006) http://www.refactoring.com
22. (Simon, 2001) F. Simon, F. Steinbruckner, and C. Lewerentz, "Metrics Based Refactoring," Proc. European Conf. Software Maintenance and Reeng., pp. 30-38, 2001.
23. (Tahvildari, 2002) L. Tahvildari and K. Kontogiannis, "A Methodology for Developing Transformations Using the Maintainability Soft-Goal Graph," Proc. Working Conf. Reverse Eng., pp. 77-86, Oct. 2002.
24. (UML, 2001) UML specification (version 1.4.2, OMG document: formal/05-04-01), ISO/IEC 19501, 2001
25. (Van Der Straeten , 2003) R. Van Der Straeten, J. Simmonds, T. Mens, and V. Jonckers, "Using Description Logic to Maintain Consistency between UML Models," Proc. Unified Modeling Language Conf. 2003, 2003.
26. (Zuse, 1999) H. Zuse, "Validation of Measures and Prediction Models", 9 International Workshop on Software Measurement, Lac Supérieur, Canada, September, 1999

Relative Thresholds: Case Study to Incorporate Metrics in the Detection of Bad Smells

Yania Crespo[1], Carlos López[2], and Raúl Marticorena[2]

[1] University of Valladolid
Department of Computer Science, Valladolid (Spain)
yania@infor.uva.es
[2] University of Burgos
Area of Languages and Computer Systems, Burgos (Spain)
{clopezno, rmartico}@ubu.es

Abstract. To detect flaws, bad smells, etc, we often use quantitative methods: metrics or measures. It is common in practice to use thresholds to set the correctness of the measures. Most of the current tools use generic values. Nevertheless, there is a certain concern about the effects of threshold applications on obtained values.
Current research is working on case studies of thresholds for several products and different versions. However, product domain and size could also affect the results, so we deal with the question of using generic vs. relative thresholds, looking at what effects this could have in bad smell detection.

Key Words: thresholds, metrics, flaw design, bad smells, refactoring

1 Initial Context

In previous works [3, 11], we tackled the use of frameworks in order to give complete support in the refactoring process: to parse the code to a metamodel, to collect metrics from a metamodel, to detect bad smells from metrics, to support refactorings and to regenerate the transformed code.

However, in this process, there are many points not yet covered. In particular, relations between metrics and bad smells were defined on the basis of generic thresholds in [3]. Although thresholds could be customized, it would be a task for the product manager. Furthermore, the use of thresholds raises certain questions: Which values should we use? Are they correct? Are they suitable for the current product?

This paper proposes a case study of several products of medium size, also focusing on the evolution of some of them. The results obtained should make clear the need to support the relative product values to be applied, in our case, with the aim of a future integration with refactorings.

The remainder of this work is organized as follows: Section 2 shows the current state of the art, Section 3 develops the case study focusing on several products/projects in several versions. Section 4 proposes the application of the

results to bad smells detection. Section 5 concludes by showing some conclusions of the proposed solution.

2 Related Works

Most of the current environments include metric collection. They also include the possibility to fix thresholds on these metrics. It is the programmer who establishes these values, using the development guides of his/her company. These filters are, however, fixed for all products and results obtained do not always suggest catalogued flaws.

There are works in the detection of design flaws. In [9,10] Marinescu proposes the concept of design strategies. Strategies are defined on the basis of metrics and are applied to the information collected in a metamodel. The metamodel contains code information, but it is designed and implemented to allow queries (all operations are translated to SQL sentences), but this does not work for a complete refactoring process. In this work, the use of generic and relative thresholds is discussed, but their suitability is not mentioned.

In [7, 8], we find a catalogue of flaws named bad smells. In order to link them to a metric suite, this assignment suggests selecting generic thresholds in most cases. They also point out the problem of leaving these decisions to the subjective human intuition.

On the other hand, in [12], Tourwé and Mens propose the detection of refactoring opportunities using queries on a logic meta-programming environment. They define queries to suggest the corrective actions to accomplish. Following that work Muñoz, in [1], uses a set of logic queries that compute object-oriented metrics to detect these bad smells with generic thresholds.

Thresholds have also been tackled in [5]. French proposes a system based on statistical methods to determine thresholds for different products, without applying them to bad smell inference or trying to see the effects on several versions.

From these previous works, we want to provide answers to certain issues:

- the correctness of using generic vs. relative product thresholds to detect bad smells in code.
- the influence of the kind and size of software products (frameworks vs. libraries).
- the suitability of the solution on different versions of the same product.

3 Case Study

3.1 First Phase: Compararison between Products

We make a comparative study of six products. We take different products, most of them stable versions, used over a long period of time, with medium or large size. We prefer these samples instead of using "toy" samples of small/tiny size, with low functionality.

The selected products are:

– jfreechart-1.0.0.pre2 (629 classes)
– jhotdraw-6.0b1 (496 classes)
– struts-1.2.8 (273 classes)
– jcoverage-1.0.5 (90 classes)
– easymock-1.0.5 (47 classes)
– junit-3.8.1 (46 classes)

All these software products are written in an object-oriented language, since extracted results will be applied to previous works on this paradigm. In the study, we use Eclipse 3.1 and Metrics 1.3.6 plug-in as the metric collection tool. This issue limits examples to programs written in Java, although, in our opinion, the process is usable in other object-oriented languages[3].

The selected metrics work on classes, choosing metrics related to size, complexity, cohesion, inheritance and specialization [2, 6]:

– NOF number of fields
– NOM number of methods
– WMC ciclomatic complexity
– LCOM lack of cohesion of methods
– DIT depth in the inheritance tree
– NSC number of children
– SIX specialization index
– NORM number of overridden methods

For each one of them, we obtain the values for several descriptive statistics: mean, bounded mean (removing 15% of the extreme values), standard deviation, lower quartile (Q1), median (Q2), upper quartile (Q3), minimum and maximum.

3.2 Partial Conclusions

From the previous results, see Fig. 1, we can say that:

– distributions are not symmetrical, differences between mean and median, and proximity of the median to Q3 quartile prove this. Most cases show distributions with positive asymmetry (distribution tail to the right side). These measures follow this kind of distribution as expected.
– differences between minimum and maximum values are large, and they are also very different in each product. This suggests dispersed data, with different thresholds.
– product size (number of classes) is slightly correlated with some metrics. Size could affect the thresholds. This is more noticeable in metrics such as NOF, NOM and WMC, with a high correlation among them. On the contrary, metrics such as LCOM and DIT show low variations between different products.

[3] Number of classes is conditioned to the use of Metrics 1.3.6 plug-in, which does not count the number of inner classes

	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean JFreeChart 1.0.0-pre2	2,40	10,08	22,98	0,21	2,55	0,36	0,16	0,69
Bounded mean (15%)	1,41	7,45	15,87	0,17	2,47	0,04	0,08	0,46
Q3	3,00	11,00	25,00	0,50	3,00	0,00	0,14	1,00
Median	1,00	5,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	3,00	6,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	5,05	15,01	38,82	0,32	1,14	1,48	0,37	1,23
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	48,00	166,00	490,00	1,00	7,00	16,00	3,20	9,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Junit-3.8,1	2,17	8,13	15,70	0,21	2,70	0,28	0,18	0,35
Bounded mean (15%)	1,50	6,53	12,33	0,18	2,58	0,15	0,09	0,28
Q3	2,00	9,75	15,75	0,50	3,75	0,00	0,12	1,00
Median	1,00	4,50	8,00	0,00	2,00	0,00	0,00	0,00
Q1	0,00	2,00	4,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	3,59	10,35	20,42	0,33	1,84	0,72	0,45	0,60
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	18,00	62,00	106,00	0,91	6,00	3,00	2,00	3,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Jcoverage-1.0.5	1,49	4,35	9,56	0,24	1,78	0,39	0,81	0,28
Bounded mean (15%)	1,23	3,70	8,17	0,20	1,62	0,19	0,10	0,21
Q3	2,00	5,00	14,00	0,50	2,00	0,00	0,00	0,00
Median	1,00	3,00	5,00	0,00	1,00	0,00	0,00	0,00
Q1	0,00	2,00	3,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	1,87	4,46	9,59	0,34	1,05	0,96	0,37	0,52
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	7,00	25,00	46,00	1,00	5,00	4,00	1,67	2,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean easymock-2.0	1,41	5,83	12,54	0,15	1,24	0,09	0,12	0,33
Bounded mean (15%)	1,24	4,13	8,32	0,11	1,08	0,00	0,02	0,16
Q3	2,00	5,00	13,50	0,33	1,00	0,00	0,00	0,00
Median	1,00	3,00	3,50	0,00	1,00	0,00	0,00	0,00
Q1	1,00	3,00	3,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	1,34	7,51	19,25	0,24	0,67	0,46	0,41	0,73
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	6,00	38,00	105,00	0,85	4,00	3,00	2,00	3,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean struts-1.2.8	2,91	8,60	18,84	0,28	2,59	0,46	0,51	0,96
Bounded mean (15%)	2,09	6,66	13,21	0,25	2,45	0,24	0,33	0,67
Q3	4,00	11,00	22,00	0,67	4,00	1,00	0,60	1,00
Median	2,00	4,00	8,00	0,00	2,00	0,00	0,00	0,00
Q1	0,00	2,00	3,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	4,56	11,02	29,13	0,36	1,48	1,13	0,95	2,04
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	40,00	82,00	260,00	0,98	7,00	10,00	5,00	28,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean JHotDraw60b1	1,40	9,51	13,36	0,16	2,84	0,57	0,31	0,73
Bounded mean (15%)	1,09	7,72	10,31	0,11	2,68	0,07	0,16	0,38
Q3	2,00	11,00	14,00	0,00	4,00	0,00	0,32	1,00
Median	1,00	7,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	4,00	5,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	1,86	10,40	16,76	0,30	1,49	3,84	0,74	1,70
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	19,00	90,00	158,00	1,50	9,00	71,00	8,00	19,00

Fig. 1. Overall results

In previous works, we established the possibility to fix metric thresholds with the aim of detecting flaws (in design, not in functionality). A first approximation was to establish these thresholds on the basis of generic values. However, the results show that values should be fitted to the concrete product.

Another factor that could influence results is the kind of product. Similar products such as testing frameworks (junit & easymock), development frameworks (struts & jhotdraw) and libraries (jcoverage & jfreechart), present large differences between minimum and maximum values.

From these results, the hypothesis appears that the absence of thresholds may generate a large change of metric measures (while products increase their size, metrics could increase or decrease over the recommended values). To verify this hypothesis, we carry out a second case study with different versions of some products.

3.3 Second Phase: Version Evolutions

We take different versions of three products: JFreechart, JHotDraw and JUnit. We show the versions and number of classes of each version. These versions have evolved over a medium period of time: *jfreechart-0.9.4* (326 classes, 2002-10-18), *jfreechart-0.9.7* (492 classes, 2003-04-17), *jfreechart-0.9.21* (570 classes, 2004-09-10), *jfreechart-1.0.0-pre2* (629 classes, 2005-03-10) and *jfreechart-1.0.1* (691 classes, 2006-01-27).

	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jfreechart-0.9.4	2,69	7,77	18,00	0,26	3,02	0,40	0,27	0,61
Bounded mean (15%)	1,71	6,13	13,51	0,22	2,85	0,08	0,11	0,32
Q3	3,00	11,00	24,00	0,62	4,00	0,00	0,16	1,00
Median	1,00	4,00	8,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	1,00	3,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	4,87	9,70	25,11	0,35	1,95	1,49	0,70	1,34
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	39,00	60,00	195,00	1,00	7,00	16,00	6,00	8,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jfreechart-0.9.7	2,14	7,03	15,63	0,20	3,21	0,31	0,17	0,49
Bounded mean (15%)	1,22	5,06	11,08	0,15	3,07	0,04	0,07	0,28
Q3	2,00	9,00	19,00	0,50	4,00	0,00	0,09	1,00
Median	0,00	3,00	6,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	1,00	3,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	4,55	10,65	24,45	0,32	1,97	1,34	0,44	1,02
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	39,00	87,00	203,00	1,00	7,00	15,00	3,00	7,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jfreechart-0.9.21	2,38	9,99	22,47	0,21	2,52	0,36	0,16	0,66
Bounded mean (15%)	1,41	7,33	15,44	0,17	2,45	0,05	0,08	0,44
Q3	2,00	12,75	26,00	0,50	3,00	0,00	0,16	1,00
Median	1,00	5,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	3,00	6,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	4,93	15,20	38,66	0,32	1,12	1,47	0,37	1,20
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	47,00	155,00	473,00	0,96	7,00	16,00	3,00	8,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean JFreeChart-1.0.0-pre2	2,40	10,08	22,98	0,21	2,55	0,36	0,16	0,69
Bounded mean (15%)	1,41	7,45	15,87	0,17	2,47	0,04	0,08	0,46
Q3	3,00	11,00	25,00	0,50	3,00	0,00	0,14	1,00
Median	1,00	5,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	3,00	6,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	5,05	15,01	38,82	0,32	1,14	1,48	0,37	1,23
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	48,00	166,00	490,00	1,00	7,00	16,00	3,20	9,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jfreechart-1.0.1	2,22	9,94	22,42	0,19	2,53	0,33	0,16	0,69
Bounded mean (15%)	1,27	7,27	15,25	0,15	2,46	0,03	0,09	0,48
Q3	2,00	11,00	23,00	0,40	3,00	0,00	0,17	1,00
Median	1,00	5,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	4,00	7,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	4,86	15,18	39,39	0,31	1,12	1,41	0,35	1,18
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	46,00	173,00	513,00	1,00	7,00	14,00	3,33	8,00

Fig. 2. JFreeChart evolution results

In the case of JHotDraw, version, number of classes and dates are: *jhotdraw-5.2* (149 classes, 2001-02-18), *jhotdraw-5.3* (208 classes, 2002-01-20), *jhotdraw-5.4b1* (478 classes, 2004-01-31) and *jhotdraw-6.0b1* (497 classes, 2004-02-01).

In the case of JUnit, versions and number of classes are[4]: *junit-2.1* (19 classes), *junit-3.8.1* (47 classes) and *junit-3.2* (32 classes).

For each of these products, we collected previously mentioned metrics, obtaining mean, bounded mean, standard deviation, Q1, median (Q2), Q3, minimum

[4] Product release dates are not available

	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jhotdraw52	1,83	8,30	13,53	0,26	2,81	0,68	0,56	1,28
Bounded mean (15%)	1,52	6,37	10,25	0,23	2,60	0,24	0,48	1,06
Q3	3,00	10,00	15,25	0,60	3,00	0,00	1,00	2,00
Median	1,00	5,00	8,00	0,00	2,00	0,00	0,28	1,00
Q1	0,00	3,00	4,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	2,19	9,82	16,84	0,33	1,70	1,91	0,66	1,69
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	14,00	61,00	108,00	1,50	8,00	12,00	3,11	12,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jhotdraw53	1,83	9,12	15,51	0,27	2,65	0,86	0,51	1,21
Bounded mean (15%)	1,46	7,07	11,60	0,23	2,43	0,20	0,41	0,97
Q3	3,00	10,00	18,00	0,63	3,00	0,00	0,75	2,00
Median	1,50	6,50	12,50	0,00	1,00	0,00	0,00	0,00
Q1	0,00	3,00	4,75	0,00	2,00	0,00	0,00	0,00
Standard Deviation	2,38	10,87	20,54	0,35	1,66	3,53	0,66	1,66
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	17,00	72,00	146,00	1,50	8,00	40,00	3,00	12,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean jhotdraw54b1	1,41	9,67	13,89	0,16	2,90	0,58	0,32	0,73
Bounded mean (15%)	1,12	7,85	10,80	0,11	2,75	0,08	0,17	0,39
Q3	2,00	11,00	15,00	0,00	4,00	0,00	0,33	1,00
Median	1,00	7,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	1,00	4,00	6,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	1,81	10,33	16,88	0,30	1,48	3,89	0,75	1,71
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	16,00	88,00	148,00	1,50	9,00	71,00	8,00	19,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean JHotDraw60b1	1,40	9,51	13,36	0,16	2,84	0,57	0,31	0,73
Bounded mean (15%)	1,09	7,72	10,31	0,11	2,68	0,07	0,16	0,38
Q3	2,00	11,00	14,00	0,00	4,00	0,00	0,32	1,00
Median	1,00	7,00	9,00	0,00	3,00	0,00	0,00	0,00
Q1	0,00	4,00	5,00	0,00	2,00	0,00	0,00	0,00
Standard Deviation	1,86	10,40	16,76	0,30	1,49	3,84	0,74	1,70
Minimum	0,00	0,00	0,00	0,00	1,00	0,00	0,00	0,00
Maximum	19,00	90,00	158,00	1,50	9,00	71,00	8,00	19,00

Fig. 3. JHotDraw evolution results

and maximum, as can be seen in Fig. (2, 3, 4). In Fig. 5, the JFreeChart evolution example (using mean value) shows that the five versions have similar values over four years. This suggests that stable products maintain their thresholds, even after increasing their size (duplicating the size in all cases).

3.4 Conclusions of the Study

These are the conclusions extracted from the study on several versions:

- Thresholds should be relative to the product.
- Thresholds could be maintained between different stable versions.
- The kind of product (framework / library) does not determine how we should fix its thresholds.

From these conclusions, we settle new issues. To develop new products we need to tune new initial thresholds. As a first option, we can estimate values from similar products (same domain, similar functionality and similar size) by taking into account our previous results. If several product versions are available, we can collect values from them to calculate an initial estimation. In both cases, we probably need to fix the values.

	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Junit 2.1	2,16	8,11	14,05	0,22	2,53	0,32	0,31	0,58
Bounded mean (15%)	1,35	7,18	12,41	0,19	2,41	0,18	0,17	0,47
Q3	2,00	8,50	18,00	0,50	3,00	0,00	0,26	1,00
Median	1,00	4,00	6,00	0,00	2,00	0,00	0,00	0,00
Q1	0,00	4,00	4,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	4,06	8,52	15,30	0,31	1,61	0,82	0,70	0,84
Minimum	0,00	1,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	18,00	31,00	55,00	0,89	6,00	3,00	3,00	3,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Junit 3.2	2,72	7,94	14,75	0,25	2,56	0,19	0,13	0,34
Bounded mean (15%)	1,68	5,96	11,29	0,22	2,43	0,04	0,09	0,25
Q3	3,00	11,00	18,50	0,50	3,50	0,00	0,19	1,00
Median	1,00	3,50	5,50	0,00	2,00	0,00	0,00	0,00
Q1	0,00	2,00	2,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	4,85	11,65	21,05	0,34	1,93	0,64	0,24	0,65
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	20,00	60,00	103,00	0,92	6,00	3,00	0,75	3,00
	NOF	NOM	WMC	LCOM	DIT	NSC	SIX	NORM
Mean Junit 3.8.1	2,17	8,13	15,70	0,21	2,70	0,28	0,18	0,35
Bounded mean (15%)	1,50	6,53	12,33	0,18	2,58	0,15	0,09	0,28
Q3	2,00	9,75	15,75	0,50	3,75	0,00	0,12	1,00
Median	1,00	4,50	8,00	0,00	2,00	0,00	0,00	0,00
Q1	0,00	2,00	4,00	0,00	1,00	0,00	0,00	0,00
Standard Deviation	3,59	10,35	20,42	0,33	1,84	0,72	0,45	0,60
Minimum	0,00	0,00	1,00	0,00	1,00	0,00	0,00	0,00
Maximum	18,00	62,00	106,00	0,91	6,00	3,00	2,00	3,00

Fig. 4. JUnit evolution results

4 Applying Relative Thresholds

In previous works, we tackled the usefulness of using metrics as symptoms of bad smells. This term is restricted to refactoring, although it could be generalized to software flaws. We posed the use of thresholds to suggest them. We defined a framework which supports all these concepts. Nevertheless, threshold definition is an open question [5]. In Fig. 6, we have the box plot diagrams of two data distributions: ideal distribution (positive gamma distribution without outlaiers) and actual distribution of WMC metric in JFreeChart 1.0.1. We consider as low and high values those below and above Q1 and Q3 respectively. More concretely, in these subsets, the outliers are the main candidates to point to problematic components. In an ideal process, outliers should be removed by applying detection and correction of bad smells.

We formerly concluded that the application of the same values to different products does not seem to be adequate. Tuning values should be helped. Next, we show a review of previous works, applying the results obtained so far.

4.1 Detection of "Lazy Classes"

Reviewing again the definition [4], there are classes that *"are not doing enough to pay by themselves should be eliminated"*. The established criteria are a low number of methods and fields, low complexity and a high level (low value) in

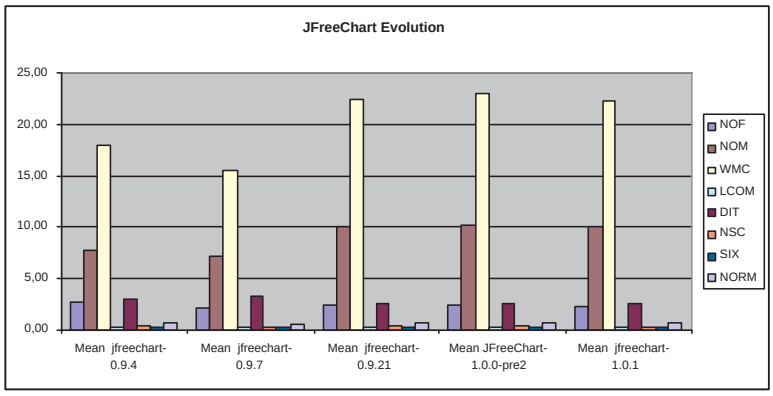


Fig. 5. JFreeChart evolution

the inheritance hierarchy. Furthermore, an additional criterion could be added, such as a low number of children.

In this approach, the problem is: What is considered low in this system? As pointed out in previous works [8, 10], certain systems are particular and general conclusions may not be correct. In our case, we work with quartiles to associate low values with classes below the Q1 quartile. We have the three Q1 quartiles of NOF, NOM and WMC as limits.

As can be seen, there are certain disagreements in the filters to be used (NOF<=Q1 AND NOM<=Q1 AND WMC<=Q1). For instance, what happens if we apply the JUnit collected values to JFreeChart? In this case, we mark as suspect 63 classes, whereas, using its own values, 97 classes are selected. The difference in numbers explains why we should not apply the same criteria to the two products.

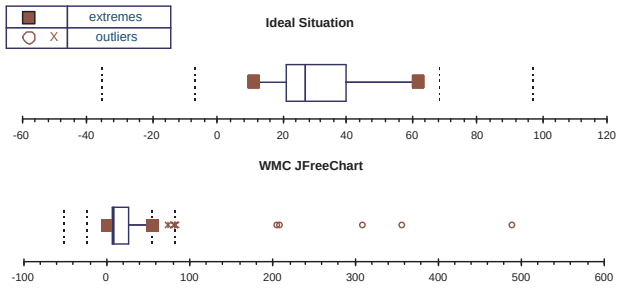


Fig. 6. Box Plot diagrams

4.2 Detection of "Large Classes"

At first, it seemed to be the easiest bad smell to detect. However, as we saw above, each system has its own particularities. More concretely, if we establish the same threshold for all of them, it is possible to fix a high value that does not return any result. On the other hand, if we choose a low value, too many classes are selected in other sets.

Using the knowledge about data distribution, we look for extreme values. We work with quartiles to associate high values with classes above the Q3 quartile. We establish the Q3 quartile as the threshold value for each product, combining NOF/NOM/WMC metrics.

From the study of the values, we infer the problem of applying the filter values to JFreeChart vs. jcoverage, or vice versa. Those metric values in the righthand tail of the distribution will verify the filter (NOF>=Q3 AND NOM>=Q3 AND WMC>=Q3). It should be possible to fit values to find outliers. If we repeat the process, for example the JUnit filter to JFreeChart, we find 148 suspect classes. However, applying the filter to JFreeChart we have just 108 classes.

Results show large differences between applying one or other threshold. The final accuracy, however, depends on manual tuning, taking into account the number of false positives and true negatives.

5 Conclusions and Future Works

Current work fixes, from a case study, the suitability of using generic vs. relative product thresholds. The former solution, generic thresholds, has not been completely abandoned. We continue to give support with metric profiles, although each product usually has its own limits.

Highly different results among software products lead us to assume it is not completely correct to use them as discriminants in the detection of bad smells. Product size, in many cases, limits the values of the metrics. Nevertheless, other metrics seem to be less sensitive to these effects.

In this work, we do not pretend to obtain new thresholds, or new methods to define them. We want to check out, empirically, the suitability of their definition for each kind of product. This detection process should be repeated until stable distributions are achieved, so as to reduce the number of outliers. These results could also help in the systematic software maintenance, as long as we have previous stable versions. Obviously, further analysis on larger samples should be completed to confirm these results.

Finally, we propose to include the current solution in bad smell detection, alongside our metric collection framework. Final tools should be able to aid the users to establish their own criteria in an objective way.

Obviously, there are many lines of work still open:

- We need to validate results, increasing the number of products under study.
- We should check the language influence on the results.
- Experience, knowledge and culture of the programmer could influence the software evolution.

References

1. Francisca Muñoz Bravo. *A Logic Meta-Programming Framework for Supporting the Refactoring Process*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2003.
2. Shyam R. Chimdaber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions On Software Engineering*, 20:476–493, 1994.
3. Yania Crespo, Carlos López, Raul Marticorena, and Esperanza Manso. Language independent metrics support towards refactoring inference. In *9th ECOOP Workshop on QAOOSE 05 (Quantitative Approaches in Object-Oriented Software Engineering). Glasgow, UK. ISBN: 2-89522-065-4*, pages 18–29, jul 2005.
4. Martin Fowler. *Refactoring. Improving the Design of Existing Code*. Number 0-201-48567-2. Addison-Wesley, 2000.
5. V.A. French. Establishing software metric thresholds. *9th International Workshop on Software Measurement*, 1999.
6. Mark Lorenz and Jeff Kidd. *Object-oriented software metrics: a practical guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.
7. Mika Mäntylä. *Bad Smells in Software - a Taxonomy and an Empirical Study*. PhD thesis, Helsinki University of Technology, 2003.
8. Mika Mäntylä, Jari Vanhanen, and Casper Lassenius. Bad smells - humans as code critics. In *20th IEEE International Conference on Software Maintenance*, 2004.
9. Radu Marinescu. Detecting design flaws via metrics in object-oriented systems. In *Proceedings of the TOOLS*, USA 39, Santa Barbara, USA, 2001.
10. Radu Marinescu. *Measurement and Quality in Object-Oriented Design*. PhD thesis, Faculty of Automatics and Computer Science, october, 2002.
11. Raul Marticorena. Analysis and definition of a language independent refactoring catalog. In *17th Conference on Advanced Information Systems Engineering (CAiSE 05). Doctoral Consortium, Porto, Portugal.*, page 8, jun 2005. http://gnomo.fe.up.pt/ caise/.
12. Tom Tourwé and Tom Mens. Identifying Refactoring Opportunities Using Logic Meta Programming. In *Proc. 7th European Conf. on Software Maintenance and Reengineering*, pages 91 – 100, Benvento, Italy, 2003. IEEE Computer Society.