Towards a generic framework for empirical studies of Model-Driven Engineering

Benoit Vanderose and Naji Habra

PReCISE Research Centre
Faculty of Computer Science
University of Namur
5000 Namur, Belgium
{bva,nha}@info.fundp.ac.be
http://www.fundp.ac.be/precise

Abstract. The goal of this paper is to introduce a work-in-progress approach that intends to formalize and facilitate empirical studies in Software Engineering in general and in Model-Driven Engineering in particular. The main idea is to use a detailed model of software that makes explicit the different intermediate models used at the different levels of abstraction, their different quality characteristics together with their relationships. The expected benefits of using such an explicit modeling is illustrated though five examples for which empirical studies are designed (but not yet conducted) on basis of that approach.

1 Introduction

Though Model-Driven Engineering techniques are very in vogue in academic world, their introduction into industry seems very slow. One of the suggested reasons is the difficulty to convince decision makers of Model-Driven Engineering advantages in terms of qualities and consequently regarding return on investment. Indeed, such argumentation necessitates empirical evidence.

Some major problems in conducting empirical studies in MDE are related to the use of classical quality models. In this paper, we claim that considering software as a single product with a list of quality characteristics (maintainability readability, efficiency...) is too rough to be used as basis for empirical studies in MDE. Instead, we suggest the use of a detailed model of the software products that makes explicit the different intermediate products (the different interrelated models) together with their relationships. The ultimate goal is to elaborate a framework to help design empirical studies in MDE. The remainder of this paper is organized as follows: Section 2 presents some related works our approach relies on and complements as well as the remaining problems we intend to address. Section 3 describes the framework and the approach themselves while Section 4 describes some examples of use.

2 Related work and Issues

The effort described here relies on research linked to both quality measurement and empirical software engineering. The basis of empirical studies in software engineering can be found in [28, 18]. Software measurement has witnessed too many metric proposals to cite them here but also benefits from generic theoretical works like [9, 16] while quality assessment benefits from numerous quality models — notably McCall's [25, 19], Boehm's [3, 2], FURPS [15], ISO [17], Dromey's [8]. Unfortunately those works do not take *explicitly* into account design-related quality. Software design quality still benefits from its own research. For instance, [6] summarizes many object-oriented design measures and studies the relationships between them and software quality. We can find publications, notably [13, 14, 12, 11, 10, 20, 21], that focus on how to estimate the quality of *models*. Finally, a generic approach to evaluate design is also proposed in [7].

Nevertheless we still identify some difficulties and limitations that appear when conducting empirical investigations in Software Engineering in general and in Model-Driven Engineering in particular.

To begin with, software measurement methods in general lack clarity about what they really measure. Theoretically, measurement process consists in quantifying relevant features (attributes) of a product (entity) in order to estimate another feature (quality) that is not directly quantifiable [9]. Practically, the entity supposed to be measured is very frequently not defined in a precise way and roughly called "software". If this view is sufficient for some empirical studies focusing on the quality of the software product as a whole, investigating Model-Driven Engineering necessitates more. In fact, since Model-Driven Engineering copes with different models and handles them as distinguished products, the qualities of the different models have to be clearly distinguished.

Moreover, the attribute supposed to be measured is frequently unclear. Numbers produced by applying a measurement method on a given model are sometimes used to estimate or predict different attributes without any clarification — e.g., a complexity measurement method used as size measurement. This can lead to incongruous use of software metrics and therefore to a completely wrong, or at least biased, quality assessment [16].

Model-Driven Engineering deals with a succession of models, from the more abstract ones to the code. Each model corresponds to a given abstraction level and has its own concerns about quality. But, as any model produced during software development is a part of an overall workflow, the inner quality of a given model is as important as the preservation of this quality through subsequent transformation steps leading to the final product. As a consequence, it can be unclear whether a quality criteria of a given model from a given development step actually improves or worsens the quality of the overall software product.

Also, Model-Driven Engineering is based on model *transformations*. However, research focuses are usually put on the preservation of semantic properties — the correctness of a transformation is defined on that basis. Traditional semantics approaches only encapsulate the "functional" aspects while empirical studies are needed to estimate a larger set of software quality attributes — including various non-functional aspects. Whatever quality model and vocabulary is used quality attributes include not only the functionality but also other attributes which are based on cognitive sub-characteristics — e.g., understandability, modifiability,. . . — related to the syntax. Empirical investigation in Model-Driven Engineering should thus take this specific type of qualities into account.

Finally, software development is mainly a matter of information transmission between people and transformation through different levels of abstraction and viewpoints [7]. Nevertheless, almost every effort relating to quality does not address such aspects — i.e. how easy the transformation process of a given model will be. Model-Driven Engineering also supposes that most of the models should be generated through automated — or at least very systematic — transformations. Investigating the preservation of the quality attributes all along the transformation process trough empirical studies should thus also imply to question the quality of these transmissions and transformations.

3 An approach to improve empirical studies of quality

This section introduces a framework we are elaborating with the intent to address the previously cited issues. The basic idea is to use model of software that makes explicit the various models used in the development as well as their relationships in terms of quality. Section 3.1 deals with the model of the software while Section 3.2 is concerned with the quality characteristics.

3.1 Modeling the software product(s).

Our framework relies on an *explicit* representation of software as a *complex* and *composite* product. In the followings we introduce a generic structure of software (see Figure 1) we claim to be sufficient to illustrate our approach. Our generic model introduces two levels of decomposition. The first level of decomposition focuses on software as the collection of outcomes of a multifaceted process involving many distinct activities. As a matter of fact, software life cycle models intend to organize those activities in a structured and rational manner [26] and the outcome of these activities is mostly composed of models — it is even arguable that this outcome is *exclusively* composed of models if “model” is defined as *just a representation.* In order to be as general as possible and to cover most life cycle models, we use a generic structure which divides the process into three categories of activities: Requirement Engineering, Architectural Design and Implementation where each category includes all the (sub)activities involved in the production of three *global artifacts*: the Requirements, the Design and the Source Code. Even though the process behind a software life cycle is not necessary linear — as shown notably in [4, 5, 24, 22] —, each *global artifact* relies on another previously produced one (except for the Requirements).

The second level of decomposition logically focuses on the internal structure of those composite *global artifacts.* Each of them is in fact made of one or more specialized elements referenced to as *elementary artifacts* in the framework. The

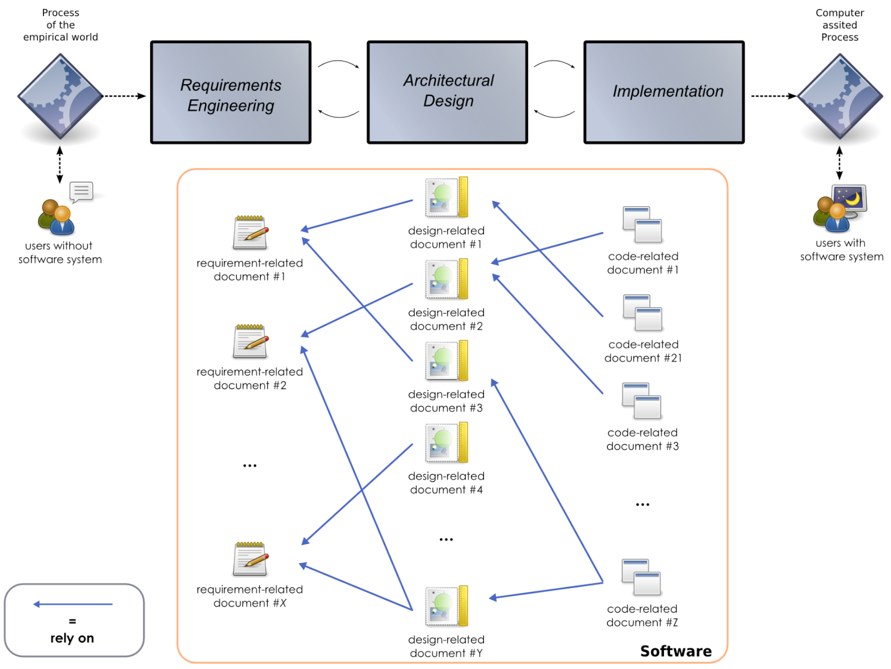
Process of the empirical world
Requirements Engineering
Architectural Design
Implementation
Computer assited Process
users without software system
users with software system
requirement-related document #1
requirement-related document #2
...
requirement-related document #X
design-related document #1
design-related document #2
design-related document #3
design-related document #4
...
design-related document #Y
code-related document #1
code-related document #21
code-related document #3
...
code-related document #Z
Software
= rely on

Fig. 1. Software as a composite and complex artifact

word *elementary artifact* means here any self sufficient piece of information comprised in a *global artifact* (e.g. a diagram, a structured text, a list of items in a text, a file, etc.). Each *elementary artifact* has a type (e.g., static structure diagram, dynamic structure diagram, source file, etc.), is written in a given language (e.g., UML, java, etc.), with a given level of abstraction and can be requirement-, design-, or code-related. Moreover, the granularity of the decomposition is variable so that it is possible to define *elementary artifacts* with more or less important scope. Finally, each element is part of an interconnected network of dependencies partially inherited from the dependencies of the composite *global artifacts*. At this level of our investigation, such a generic model seems to be sufficient to support the approach.

3.2 Modeling the software characteristics.

The main use of this view is to support a refined definition of the different quality characteristics, to relate each of them to the adequate product (elementary artifact or composite one) and to express and study relationships between them. The final aim is to build a quality model that is more flexible and adequate for model-driven approaches than the existing ones. Practically, as the quality of any type of elements can almost certainly be evaluated through a set of proposed "quality characteristics" — as hinted notably in [12, 11, 10]—, a powerful quality model could be built by defining a list of characteristics assigned to each element then by determining the “influence” relationship between attributes. At this point, we can illustrate the “influence relationship” with the following partial example based on our practice, and common sense.

Each entry of the table below has the following structure:

Globalartifact.TypeOfElementaryartifact.Characteristic, where

- **Globalartifact** is either R(equirements), D(esign) or C(ode).
- **TypeOfElementaryartifact** is : NFu stands for non functional requirements, Fu for functional requirements, Struc for structural aspect, Behav for behavioral aspects, Run for aspects involved at runtime or “*” which means that any type of *Elementary Artifact* is involved. The type is used to categorize the elementary artifacts according to the role they play in the development. Not every type is present in every *global artifact*.
- **Characteristic** is either Main(tainability), Usab(ility), Func(tionality), Effi (ciency) [17] or Pres(ence) — since the presence of a given artifact can be considered as quality-related information in this approach as we will see in Section 4. A characteristic could be meaningful for only a subset of a type of elementary artifact but this preliminary attempt intends to be as generic and simple as possible.
- Each cell contains either N(no influence expected) or I (some influence expected)

Table 1. Expected Influence relationships

	...	D.Struc.Main	D.Behav.Main	C.Run.Func	C.Run.Effi	C.*.Main
R.NFu.Usab	...	*N*	*I*	*N*	*I*	*N*
R.Fu.Usab	...	*I*	*N*	*I*	*N*	*N*
R.*.Pres	...	*I*	*I*	*I*	*I*	*I*
D.Struc.Main	...	\	*N*	*I*	*I*	*I*
D.Behav.Main	...	*N*	\	*I*	*I*	*I*
D.*.Pres	...	*I*	*I*	*I*	*I*	*I*
...	...	...	...	...	...	...

The table intends to give a new formulation of the questions to be dealt with through empirical studies. It shows the plausible expectations in terms of influence at a very general level. That is, its content represents a set of plausible hypotheses to explore through empirical investigations, each of them involving going to a lower level (subtype of element artifact, quality sub-characteristic and internal metrics). The table is far from being final at this point and each "path" can be questioned. Also, this current table is only focusing on very generic characteristics so that each user of the framework can expand, propose and study new relationships or refine existing ones with different characteristics or other classifications of artifacts. So it is only through the massive use of the proposed method within the community of Empirical Software Engineering that a consensual and widely accepted table of this kind could be achieved. We believe that such table would be a more helpful basis to conduct empirical studies in model-driven engineering than usual quality models.

4 Some typical scenarios of use

The novelty of the framework introduced above lies in the particular point of view adopted. It is more flexible in the sense that it allows to get *inside* Model-Driven process and investigate the influence of various artifacts on other ones. Then this approach supports more specialized investigation such as the relevance of a particular transformation in comparison with another.

The first step in order to use the framework is to make *explicit* the dependencies between studied artifacts — e.g., *elementary artifact* c1 is produced thanks to d1 and d2. Then the idea is to consider these variables as "typed" variables where each "type" (requirement-, design, code-related) has its own set of meaningful quality characteristics. Finally, the table of influences is used to express the hypotheses about the characteristics involved and their relationships. Besides being an innovative way to support empirical studies in general, the present approach particularly suits Model-Driven Engineering for two main reasons. First, most of elementary artifacts, are supposed to be models. Since elementary artifacts are almost the primary form of expression of our framework and models are the one of MDE, the two approaches should be compatible. Secondly, our approach is designed to address the transformation of information, which is a core concept of any model-driven approach.

The remainder of this Section illustrates the use of the framework on five hypothetical experiments and highlights the benefits of the framework in those situations. The structure of the cases is inspired by the GQM paradigm [1, 27].
Case 1

– **Goal.** Study the impact of the presence of a design pattern P on the maintainability of the produced code.
– **Question.** Let d1, d2 be 2 class diagrams, where d2 satisfies the same set of requirement-related *elementary artifacts* REQ1 than d1, but d1 uses a design pattern P while d2 does not; c1 & c2 are two pieces of java program produced from d1, d2, respectively, through a transformation T. Is c2 more maintainable than c1?

- **Metrics.** To study the maintainability of the code, the experimenter could use the effort needed to complete a maintenance task as a metric. This is consistent with classical quality models which propose to decompose maintainability into sub-characteristics which are mainly measurable through effort. Though the choice of such measurement method is questionable by itself, it could still be used as an approximation.
- **Discussion.** This experiment could be achieved without the support of the framework but would miss some benefits. With our framework, this experiment can be expressed as the investigation of the influence of a structural design-related *elementary artifact* on the maintainability of a code-related *elementary artifact.* This path is present in the table of expected influences (Table 3.2), which means that the experiment seems relevant. Moreover, the use of the framework highlights the fact than design patterns are *part of the design* and not the code, a view that is not always admitted.

Case 2

- **Goal.** Study the impact of one design characteristic on the same characteristic at the code level. For instance the goal could be to investigate whether a focus on the maintainability at the design level does not produce more complex code and therefore impact negatively the global maintainability of the software product.
- **Question.** Let d1, d2 be 2 diagrams at the design level, where d1 is more maintainable than d2, and c1 & c2 be two pieces of java program produced from d1 & d2, respectively, through a transformation T. Is c2 more maintainable than c1?
- **Metrics.** This situation illustrates perfectly the case where the present approach complements and is nurtured by other related works when it comes to selecting the adequate metrics. Indeed, we can use and integrate to the framework the research that has been done regarding the maintainability of UML diagrams and how to evaluate it thanks to internal metrics [12, 11]. For the code maintainability, see Case 1.
- **Discussion.** This case is almost similar to the first one but illustrates how the framework allows us to further investigate software quality. While the first case was about the influence of a technique on a quality characteristic, this case proposes to confront the *quality characteristics* of two entities and see how they are related. Without the framework and its specific point of view, this experiment — the investigation of the influence of the maintainability of a design-related *elementary artifact* on the maintainability of a code-related *elementary artifact* — would need an extra descriptive effort to show that maintainability does not apply to the same entity on both sides of the relationship.

Case 3

- **Goal.** Study the impact of one design characteristic on another characteristic at the code level. For instance the goal could be to investigate whether a

focus on the maintainability all along the design process does not impact negatively the efficiency of the software code.
- **Question.** Let d1, d2 be 2 design models where d1 is more maintainable than d2 and c1 & c2 be two pieces of java program produced from d1, d2, respectively, through a transformation T. Is c2 more efficient than c1?
- **Metrics.** The same principles as in Case 2 apply for d1 and d2. Metrics related to efficiency could be specifically designed for the experiment or taken from ISO standards.
- **Discussion.** This case is set at the same level than Case 2 : the aim is to study internal mechanisms of software quality by questioning the relationship between two quality characteristics. The framework helps give sense to this experiment : though maintainability and efficiency do not seem to be related in any way when applied to the same entity, the framework allows us express the fact that a relation of cause and effect exists between the two *elementary artifacts* and probably between their respective quality characteristics. In this context, the legitimacy of the question is clearer.

Case 4

- **Goal.** Study the benefits of a given transformation methodology (or tool) regarding the preservation of a given quality characteristic. For instance the goal could be to question whether a transformation T preserves, improves or worsens the complexity of the software code.
- **Question.** Let DES be a collection of design-related *elementary artifacts* (e.g., class diagrams & statecharts & sequence diagrams) and let C1, C2,. . . , Cn be different source code — and thus *global artifacts* — produced by the transformations T1, T2,. . . ,Tn, respectively. Is C1 more complex than C2,. . . , Cn?
- **Metrics.** McCabe's number could be used to assess complexity according that some precaution is taken [23].
- **Discussion.** This case illustrates how the framework can support investigations about the quality of the engineering process. In previous cases, the transformation process was fixed and the studied variable was an *elementary artifact*. Here, we fix the design-related *elementary artifacts* and investigate how various transformations provide various level of quality at the code level.

Case 5

- **Goal.** Determine the relevant level of abstraction — requirement-, design- or code-related — and the suitable models involved where it would be meaningful to take a given characteristic into account — e.g., security.
- **Question.** Let $< R1, D1, C1 >$, $< R2, D2, C2 >$ and $< R3, D3, C3 >$ be three software products; the non-functional requirement of security is modeled by an adequate elementary artifact since the requirement level in $< R1, D1, C1 >$, since the design level in $< R2, D2, C2 >$ and in the code in $< R3, D3, C3 >$, which is the most secure product?
- **Metrics.** Security metrics are not proposed here but could be designed specially for the experiment.

– **Discussion.** This case illustrates how the framework can express questions about the "temporal" impact of the introduction of some *elementary artifacts*. It also shows how the particular point of view chosen in this approach allows to make a quality characteristic out of a very simple attribute like the presence of absence of a given artifact.

5 Conclusion and future work

The approach introduced in this paper is a first and currently evolving attempt to address some limitations of software quality assessment. Though relying on a very simple and light mechanism, the main benefit brought by this framework is to force the experimenter to adopt a more accurate and flexible view of software. The examples given in Section 4 are just some of the possible experimentations that could benefit from it. In each case, the use of "typed variables" clarifies the "dimension" involved and allows to avoid ambiguity about what the experimenter expects to address. Section 4 also illustrates how basic attributes like the presence of a given *elementary artifact* become valuable quality criteria when software is considered from this point of view — i.e., as a composite artifact resulting from a network of interconnected pieces of information.

As an early work, the approach naturally lacks experimental data to confirm all the benefits we expect. The next step of the development will be to apply the framework to an actual empirical study in the context of a student development project. The table of expected influences also need experimental confirmation, but should eventually constitute a valuable reference for anyone interested in further transversal investigations about the internal mechanisms of software quality.

References

1. Basili, V.R.: Using Measurement to Build Core Competencies in Software. Seminar sponsored by Data and Analysis Center for Software. (2005)
2. Boehm, B.W., Brown, J. R., Lipow, M.: Quantitative evaluation of software quality. In: International Conference on Software Engineering (1976)
3. Boehm, B.W., Brown, J. R., Kaspar, H., Lipow, M., McLeod, G., and Merritt, M.: Characteristics of Software Quality. North Holland (1978)
4. Boehm, B. W.: Software Engineering Economics. 1st. Prentice Hall PTR. (1981)
5. Boehm, B. W.: A Spiral Model of Software Development and Enhancement. Computer 21, 5,pp. 61-72. (1988)
6. Briand, L.C., Wust, J., Daly, J.W., Porter, D.V.: Exploring the relationships between design measures and software quality. In: object-oriented systems, Journal of Systems and Software, 51, 3, pp. 245-273. (2000)
7. Budgen, D.: Software Design. 2. Addison-Wesley Longman Publishing Co., Inc.(2003)
8. Dromey, R. G.: A model for software product quality. In: IEEE Transactions on Software Engineering, no. 2, pp. 146-163. IEEE Computer Society, Los Alamitos (1995)

9. Fenton, N. E., Pfleeger, S. L. : Software Metrics: a Rigorous and Practical Approach. 2nd. PWS Publishing Co. (1998)
10. Genero, M., Piattini, M., Calero, C.: Empirical Validation of Class Diagram Metrics. In Proceedings of the 2002 international Symposium on Empirical Software Engineering (October 03 - 04, 2002). International Symposium on Empirical Software Engineering. IEEE Computer Society, Washington, DC (2002)
11. Genero, M., Piattini, M., Manso, E., Cantone, G.: Building UML Class Diagram Maintainability Prediction Models Based on Early Metrics. In: Proceedings of the 9th international Symposium on Software Metrics (September 03-05, 2003),METRICS. IEEE Computer Society, Washington, DC (2003)
12. Genero Bocco, M., Moody, D. L., Piattini, M. : Assessing the capability of internal metrics as early indicators of maintenance effort through experimentation :Research Articles. J. Softw. Maint. Evol. 17, 3, pp.225-246 (2005)
13. Genero, M., Piattini, M., Calero, C.: A Survey of Metrics for UML Class Diagrams Journal of Object Technology, 4, 9, 59-92. (2005)
14. Genero, M.: Metrics for Software Conceptual Models. World Scientific Publishing Co., Inc. (2005)
15. Grady, R. B.: Practical Software Metrics for Project Management and Process Improvement. Prentice-Hall, Inc, Upper Saddle River, NJ, USA (1992). Practical Software Metrics for Project Management and Process Improvement. Prentice Hall, p. 32.
16. Habra, N., Abran, A., Lopez, M., and Sellami, A.: A framework for the design and verification of software measurement methods. J. Syst. Softw. 81, 5, pp633-648. (2008)
17. ISO/IEC 9126. Software Product Evaluation–Quality Characteristics and Guidelines for the User, Geneva, International Organization for Standardization. (2001)
18. Juristo, N., Moreno, A.M.: Basics of Software Engineering Experimentation. Kluwer Academic Publishers. (2001)
19. Kitchenham, B., Pfleeger, S. L.: Software quality: the elusive target [special issues section]. IEEE Software, no. 1, pp. 12-21 (1996)
20. Lange, C.F.J.: Empirical Investigations in Software Architecture Completeness. TU Eindhoven; November 12. (2003)
21. Lange, C., Chaudron, M., Muskens, J.: In Practice: UML Software Architecture and Design Description. In: IEEE Software ,vol. 23, no. 2, pp. 40-46. (2006)
22. Larman, C. and Basili, V. R.: Iterative and Incremental Development: A Brief History. Computer 36, 6,pp. 47-56. (2003)
23. Lopez, M., Habra, N., Abran, A.: A Structured Analysis of the McCabe Cyclomatic Complexity Measure. In: Proceedings of the 14th International Workshop on Software Measurement (IWSM2004) Berlin, Germany (2004)
24. Martin, J.: Rapid Application Development. Macmillan Publishing Co., Inc.(1991)
25. McCall, J. A., Richards, P. K., Walters, G. F.: Factors in Software Quality. Nat'l Tech.Information Service, Vol. 1, 2 and 3 (1977)
26. Pressman, R. S.: Software Engineering: a Practitioner's Approach. McGraw-Hill Science/Engineering/Math.(2004)
27. Van Solingen, R.: The Goal/Question/Metric Method. McGraw-Hill Education. (1999)
28. Wohlin, C., Runeson, P., Hst, M., Ohlsson, M.C., Regnell, B., Wessln, A.: Experimentation in software engineering: an introduction. Kluwer Academic Publishers (2000)