Towards an Automation of Software Evolution Good Practices

Chouki Tibermacine
LIRMM, CNRS and Montpellier University, France
Chouki.Tibermacine@lirmm.fr

Soraya Sakhraoui
University of Setif, Algeria
Soraya.Sakhraoui@univ-ubs.fr

Vincent Le Gloahec
Alkante SAS and VALORIA, University of South-Brittany, France
v.legloahec@alkante.com

Régis Fleurquin
IRISA, INRIA Rennes, France
fleurquin@irisa.fr

Salah Sadou
VALORIA, University of South Brittany, France
sadou@univ-ubs.fr

Abstract

It is well known that software evolution is an inescapable activity in the software lifecycle. In order to prevent the negative effects of this activity (decreased quality, increased complexity, etc.), some good practices have been recommended in the past. In this paper, we present a method which aims at automating a kind of assistance to software evolution. This assistance makes it possible to guide the developer when applying changes on a given software by making persistent some good practices, which can be considered as some kind of knowledge in the software engineering practice. In this method, a domain metamodel is firstly introduced. A set of constraints formalizing the good practices are then associated to this metamodel. Together, these two elements compose the basis upon which the automatic support for the evolution assistance has been built.

1 Introduction

The study of software evolution, as a first-class phenomenon, started at the end of the sixties. The need for studying software evolution is motivated by the society's increasing dependency on software that implements business processes, which are naturally subject to evolution. This important activity has high costs within a software project budget. These are estimated nowadays for more than 80 % of the global cost [4, 13].

To reduce the evolution cost there have been some proposals for good practices. These"good practices" correspond to an experiment-based methodology which leads a given software evolution to the desired result at a lower cost. For this purpose, Lehman's works [8, 6] consisted in undertaking several empirical studies on software evolution [7, 8]. From these studies he specified laws and good practices for improving the software evolution process. These good practices, which are considered as units of knowledge acquired through a long experience on evolving existing software systems, have been presented in the form of recommendations addressing different contexts. Examples of good practices include : "the necessity to monitor the number of user-generated fault reports per release in order to check if the fault rate is increasing".

It is interesting to specify a catalog of good practices, but leaving the checking of their respect to human may lead to errors. The causes of errors are multiple. This can be simply the non-use of the rules or their bad interpretation. Indeed, rules defined textually in a natural language can lead to this kind of situation. In order to avoid this, it will be necessary to make an automatic checking of these recommendations. For that aim we have to: i) formalize the description of the good practices; ii) provide a tool able to interpret these descriptions in order to automate the checking of the compliance of the evolution process with the good practices.

In this paper we propose to formalize the description of good practices as a set of rules using meta-models (Section 3) and the OCL language [10] (Section 4). To automate the verification of compliance with good practices, we propose a tool designed as an Eclipse plugin (Section 5). Based on the information system of a software project and the set of rules, this tool checks the compliance of the project with the good practices. Before concluding this paper, we present some related work in Section 6.

In the following section we describe the main principles of our approach.

2 Proposed approach

In [14], we proposed to formalize the link between non-functional properties and their corresponding architectural choices, in order to limit the effects of software aging [12]. Using a tool that checks, at every stage of a development, the validity of the architectural choices, it is possible to warn on the non-functional properties potentially affected. By regulating possible actions in response to these warnings, we have built a control system which contributes on the one hand to updating the documentation and on the other hand to checking the non regression of the system. We have shown that this kind of validation significantly increases the chance to reach a well-documented solution, complying with the new requirements, while preserving the quality attributes that should not be tampered with. Thus, we provide a way to induce compliance with the good practice which consists in *"systematically checking, during an evolution process, the consistency of the software's non-functional requirements specification with its existing design"*.

Our aim is to generalize this mode of automatic control to other good practices described in the literature such as those introduced by Lehman's laws or those referenced in models of maturity-level assessing, such as CMMI [3].

2.1 General architecture of the framework

Good practices specify a number of actions or measures that improve the effectiveness of the evolution process. Ideally, starting from a formal expression of good practices, we should be able to automatically generate the code needed to check their respect in the tools used to achieve the evolution.

To concretize this goal, we propose the following means to software developers:

1. a language for documenting good practices they wish to see implemented during an evolution process. This documentation must be in a format that is independent from any software project or an IDE (Integrated Development Environment, just like a PIM (Platform-Independent Model) in MDE (Model-Driving Engineering).

2. a tool able to automatically translate or help in the translation of these good practices in the context of a given project or a tool. This translated documentation corresponds to a PSM (Platform-Specific Model in MDE) of good practices.

3. Finally, a tool able to interpret the PSM good practices and to introspect the current project and the IDE's underlying information system to check if it complies.

2.2 Mapping to a particular case

To experiment the proposed approach, we have taken the following steps:

1. We have defined a metamodel, called SEMM (Software Evolution MetaModel), which identifies the concepts manipulated by the good practices defined by Lehman [8] for the software evolution process. This metamodel will be described in the next section. This is the basis for writing rules of good practices regardless of any tool (PIM format).

2. We have described the rules of good practices as OCL constraints on SEMM.

3. We have produced the metamodel of AlkoWeb[1] development tool which is used by our industrial partner. This metamodel defines the concepts handled by the project information system of AlkoWeb (model, view, version, author, date, etc.).

4. We have translated the constraints written on the basis of SEMM to their equivalents in AlkoWeb's metamodel.

5. We have developed an Eclipse plugin, called SEGPE, which checks the compliance with these rules of the evolution of a design defined with AlkoWeb.

6. Finally, we have used SEGPE on a particular project.

In the following, we present SEMM, examples of constraints corresponding to the rules of good practices written on this metamodel, the tool developed for writing and checking these constraints, and an experimentation made on a concrete project.

3 Domain Metamodel

An important step towards the automation of the good practices is to specify the relations between various concepts in software engineering (software, process, activity, quality, etc.). We do this by defining a metamodel (SEMM) for this domain. As described in Section 2, a set of concepts was constructed. These concepts form the entities of the metamodel and are represented as metaclasses. As the concepts were classified according to the context to which they are addressed, the complete metamodel was subdivided in two parts. Each part covers a particular aspect of the domain.

Figures 1 and 2 represent the different parts (aspects discussed above) of the domain metamodel. Assembled, they represent a complete illustration of SEMM. Each aspect is discussed in the following sub-sections.

[1]This tool is briefly described in section 5.

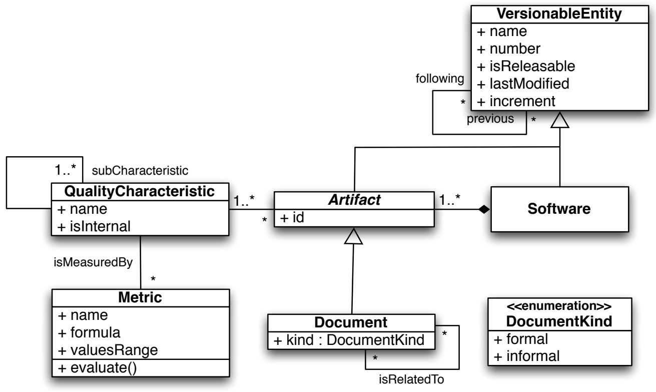

Figure 1. Domain Metamodel -part 1: software product aspect-

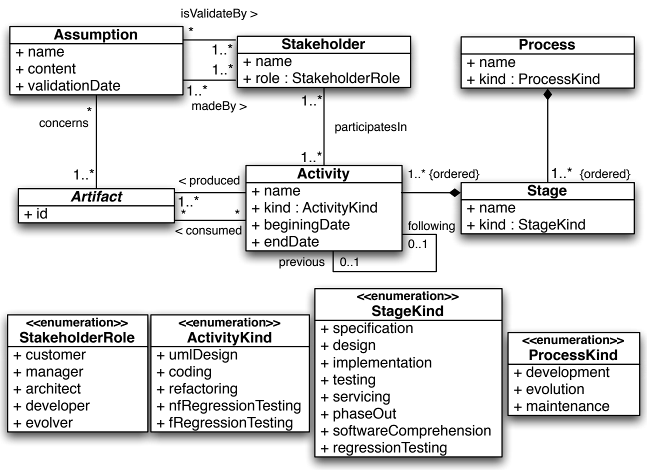

Figure 2. Domain Metamodel -part 2: software process aspect-

3.1 Software Product Aspect

In figure 1, the metamodel represents a software product as a non-empty set of artifacts. A software and an artifact are considered as versionable entities, having thus a name and a number. A versionable entity can be releasable or not. It becomes a releasable entity when some major changes are made throughout several versions. It has a date meta-attribute (lastModified), which corresponds to the date when the software entity has been versioned (the date of a commit in an SVN system, for example). The `increment` represents the modifications made on the software or on one of its artifacts since the last version. Given a versionable entity, we can navigate in this metamodel to the previous or the following versionable entities. In SEMM, `Artifact` is an abstract class. The concrete concept here is `Document`. A document can be formal (formal specification document, architecture design document, source code, ...) or informal (informal non-functional requirements specification, source code documentation, ...).

To a given artifact is associated one or several quality characteristics. These characteristics have the same semantics as in the ISO 9126 standard[2]. Each characteristic (reliability, portability, maintainability, ...) can be externally or internally visible. It has several sub-characteristics. These are measured by the means of metrics. Each named metric is defined by a formula which is used for its evaluation. A values range can be specified for a given metric. It represents the context-free acceptable values for the metric.

[2]Software engineering – Product quality – Part 1: Quality model. The International Organization for Standardization Website: http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22749

3.2 Software Process Aspect

Figure 2 shows the software process part of SEMM. The Artifact metaclass is the common class between the two parts of the metamodel (the process and the product parts). In this part of the metamodel, this concept is seen as a dynamic entity which is managed within some software actions. In SEMM, we abstract three distinct levels of granularity for actions where artifacts are managed. A *Process* is a set of ordered elements of type *Stage* and a Stage is a set of ordered elements of type *Activity*. A process in this metamodel is named and has a kind. It can be a development, a maintenance or an evolution process. A stage (part of a software process) is named and can be of different kinds: specification, design, implementation, testing (for software development process), servicing and phase-out (for evolution process), and software comprehension and regression testing (for maintenance process). An activity is the most atomic action. It represents the low-level operations performed within a software stage (see Figure 2).

Starting from each activity, we can get the eventual previous or following activity. An activity is characterized by a beginning date, where it can consume several artifacts, and an end date, where it produces one or several artifacts. A stakeholder like a project manager, a customer or a developer can participate in one or several activities. During her/his implication in an activity, a stakeholder can make some assumptions. Each assumption is named, has a content and a validation date. Different stakeholders can be responsible for the validation of assumptions made on some artifacts.

4 Expression of Constraints

We associated to SEMM a set of constraints. These constraints correspond to formal expressions of the rules representing the good practices of software evolution, usually expressed in the natural language. First, we briefly introduce the constraint language used in this work. We illustrate then two examples of constraints defined with this language upon the metamodel presented previously.

4.1 Constraint Language: OCL

As stated at the beginning of this paper, the constraint language used for formalizing software evolution good practices is the OMG's Object Constraint Language. The choice of this language is motivated by its simplicity and the presence of reliable support tools. OCL is a first order logic language which allows, in a declarative way, the description of expressions representing constraints for precising the semantics of UML models. In our approach, these constraints have as a context an element in the domain metamodel (SEMM). This means that the constraints will be checked on all the models (software processes, software artifacts, ...) instances of this metamodel.

4.2 Examples of PIM rules

In this subsection, we show how two examples of Lehman's evolution good practices are formalized using OCL. The first good practice states that [8]:

> ”... modifications to the specification, assumptions that accompany them and independent assumptions may be recorded in user or other documentation. ... (assumptions) must be confirmed as not being inconsistent with the specification ... (records of changes to individual specifications record additions and changes to the assumption set that underlie them.).”

This good practice, which is associated to the first evolution law (Continuing Change) is specified in OCL as follows:

```
context Assumption inv:
self.concernedArtifacts->forAll(art |
art.lastModified < self.lastValidated)->isEmpty()
```

In the context of SEMM, this constraint stipulates that each artifact attached to an assumption should have a date of modification prior to the date of last validation of its associated assumption.

We introduce below another example of a good practice defined in Lehman's evolution laws:

> ”... monitor the number of user generated fault reports per release to obtain a display of the fault rate trend with time. A fitted trend line can then indicate whether the rate is increasing ...”

In OCL and in a simplified form, we can describe this rule, which is associated to the seventh law (Declining Quality), as follows:

```
context Software inv:
self.artifact->select(a | a.isReleasable)
->oclAsType(Document)->select(doc |
doc.name='ErrorReport')->asSet()->size() <=
self.artifact.previous->select(a | a.isReleasable)
->oclAsType(Document)->select(doc |
doc.name='ErrorReport')->asSet()->size()
```

In this constraint, we query the collection of error (fault) reports associated to two successive versions of releasable artifacts. The size of these collections should decrease or be constant.

4.3 PIM to PSM translation

Our choice of OCL was also motivated by the possibility offered by this language to express constraints as models that transformation languages, which are MOF QVT-compliant [11], can handle. Thus, the transformation of PIM constraints to their equivalent PSM ones becomes possible by using ATL[3] or Kermeta[4], for example.

We have formalized many rules of good practices, but we noticed that some of them defined at PIM level have no equivalent in PSM one. Examples include the following rule which states that *”any documentation created must be clear and comprehensible”*:

```
context Software inv:
self.artifact-> oclAsType(Document)
->forAll(d|d.qualityCharacteristic
->exists(q|q.name = "Comprehensibility")
and d.qualityCharacteristic
->exists(q|q.name = "Clarity"))
```

As shown above, this constraint is easily expressed with OCL on SEMM. However, it cannot be easily checked on real-world IDEs, because existing tools' information systems do not provide the necessary information for performing this task (quantifying comprehensibility and clarity). In other words, concepts manipulated by this constraint have no equivalent in the PSM metamodel. However, it is important to keep this kind of constraints at PIM level in case of possible tools evolution in order to incorporate concepts needed for their evaluation (possible definition in the future of some metrics for measuring such quality characteristics that are specific to the targeted tools).

[3] ATLAS Transformation Language: http://www.eclipse.org/m2m/atl/
[4] Kermeta, Triskell Metamodeling Kernel: http://www.kermeta.org/

5 SEGPE : a tool to automate Good Practices validation

In order to validate the approach presented in this paper, we have implemented a tool named SEGPE (Software Evolution Good Practices Evaluator). This tool is composed of a set of plugins that are integrated in the Eclipse platform. This platform has been chosen for several reasons : i) the Eclipse community focuses researches concerning software engineering and offers many frameworks for supporting and promoting model-based development; ii) our team has already developed Eclipse plug-ins: AURES, an architecture evolution assistant tool [14], and AlkoWeb-Builder, a tool for component-based Web development [5].

AlkoWeb-Builder, built in a partnership project between VALORIA laboratory (University of South-Brittany) and Alkante[5], has been linked with SEGPE. SEGPE has been built as a plug-in, using two main frameworks provided by the Eclipse platform : EMF (Eclipse Modeling Framework) and MDT OCL (Model Development Tools - OCL).

5.1 SEGPE architecture

The metamodel discussed in this paper has been firstly designed as an EMF Ecore metamodel (Ecore is an implementation of the OMG's MOF (Meta Object Facility)). Then, using this Ecore metamodel, a plugin has been automatically generated, which allows to create and edit instances of our metamodel. The way SEGPE has been designed intends to preserve a low-coupling between the metamodel and the plug-in itself. The metamodel can be extended with no consequences on the plugin.

OCL constraints are defined at the metamodel level and are directly incorporated in the meta-classes (Assumption, Artifact, etc...). At this level, the constraints does only need a syntactic validation, to check if they are grammatically correct. When a developer wants to apply good practices on a real project, the plugin allows to create and edit a model which conforms to SEMM. By creating model objects, which are instances of the meta-classes, the constraints are automatically inherited from the metamodel. At the model level, the context of an OCL constraint is given by the model object the constraint is attached to. Although the constraints are defined at the metamodel level, they are in fact evaluated at the model level. The evaluation of the constraints allows to check compliance of the project's state with the described rules.

The evaluation part of the tool has been designed as another plugin which consists of two views. **Good Practices Contract:** used to view the contract's rules (inherited from the metamodel) in the context of a model object. The view allows to evaluate the contract (all its OCL constraints). It is also possible to launch an evaluation on the whole model, where all model object contracts will be evaluated. **Good Practices Report**: this view is used to display a summary each time a contract is evaluated.

[5]Alkante is a company specialized in consulting and engineering information technology (Web, geographical and territorial information systems).

5.2 Integration with AlkoWeb-Builder

In order to validate SEGPE on a real-world project, the tool has been associated to AlkoWeb-Builder. AlkoWeb-Builder allows to model a component-based Web application by graphically composing hierarchical components. To do so, we have translated the rules written using SEMM into their equivalents in AlkoWeb metamodel. This translation was rather easy since the same concepts were present on both metamodels. However, the translation has shown some deficiencies in AlkoWeb metamodel. For example, there were no links between artifacts and assumptions on which they depend. This is important in order to check some rules, such as the one presented in section 4. Therefore, we believe that it is important to keep the rules written with SEMM and then translate them into the targeted metamodels. We can thus check the consistency of the targeted metamodels towards the good practice rules.

6 Related Work

The good practices in software engineering aim at producing rationally software, by decreasing for example their complexity. Design patterns are a good example of such practices which have as a goal the capitalization of the programming know-how. Following the same philisophy, a *process pattern* suggests a sequence of proven processes that solves a frequently recurring problem [1]. They represent a way to enforce some good practices. This however needs to be automated to prevent human errors. We preferred a solution based on constraints expressing declaratively good practices because, unlike the process patterns, they can express all the space of acceptable solutions.

The reader can see many similarities between our metamodels and OMG's SPEM (Software Process Engineering Metamodel [9]). Indeed, SPEM is intended to metamodel processes of software engineering (like Rational Unified Process), and this is what we partly aimed to do. The main difference between the two kinds of metamodels is that SPEM is a standard for defining processes to allow building standardized tools for managing (authoring and customizing) processes, while the metamodels we presented aim at formalizing good practices of software evolution. The metamodels presented in this paper are voluntarily simplified in order to facilitate the description of constraints. Many abstract concepts present in SPEM are thus not present in these metamodels. This makes easier the navigation in these metamodels for describing constraints. Besides, our metamodels cover some aspects not treated (voluntarily) by SPEM designers. Indeed in SPEM, project management is not metamodeled. Authors of the SPEM specification [9] affirm that this aspect does not meet their concerns, and argue that this will make SPEM more complex, because this standard wants to accommodate a wide range of existing software development processes. In our work, the software project management is an aspect which is widely dealt with in the existing software evolution good practices. Hence, we met the need to integrate it in SEMM.

7 Conclusion and Future Work

In this paper we presented a language, a method and a tool to automate a kind of disciplined software evolution. By disciplined evolution, we mean here a software maintenance that complies with some good practices. The presented approach is built upon two technologies whose maturing is ever increasing, MOF metamodeling and the OCL constraint language. OCL language has already proven its usefulness in software maintenance [2]. We are convinced that the precision granted by the formality of OCL comes at a much lower cost than when using other formal specification constructs. In addition MOF has become a *de facto* metamodeling standard, and more people are expected to master OCL and use it currently. The set of the formalized good practices is not exhaustive but the approach is flexible enough to be applied at all the good practices found in the literature and practice. New specifications could be added to the metamodel describing the new determined concepts. The methodology presented in this paper helps: (a) to define a formal specification of the good practices that were only informally defined as well as expressing them in a checkable form, (b) to ensure that there are no ambiguities on the context under which the good practices are defined, (c) to propose a library of good practices specifications in a format that can be used and validated by different stakeholders.

In the literature and the practice, there is no work addressing the formalization of software evolution good practices towards their automation. We are convinced that this is a promising research field, and much work should be done in order to be able to formalize as much as possible (and thus automate the checking of) developers' knowledge and experience acquired while participating in software evolution tasks. The work presented in this paper is a contribution in this direction based on simple and standard languages. It targets the only known catalog of software evolution recommendations: Lehman's evolution laws.

As a perspective to this work on the conceptual level, we plan to study other kinds of good practices and thus to extend the existing domain metamodel and constraint set. On the tool level, an interesting issue that has not been yet explored is how to make a SEGPE model aware of the underlying information system. For the moment, a model's artifact references a physical data (a Java class for example) by its name, but this reference has to be done manually by the person in charge of editing the model. So, what lacks by now is to make a model's artifact aware of any changes of its referenced data. Even though this issue is not trivial, some features provided by the Eclipse platform could be used to keep a model up-to-date with its associated information system.

References

[1] S. W. Ambler and B. Hanscome. *Process Patterns: Building Large-Scale Systems Using Object Technology*. Cambridge University Press, 1998.

[2] L. C. Briand, Y. Labiche, H. D. Yan, and M. Di Penta. A controlled experiment on the impact of the object constraint language in uml-based maintenance. In *Proc. of ICSM'04*, pages 380–389, Chicago, Illinois, USA, 2004.

[3] M. B. Chrissis, M. Konrad, and S. Shrum. *CMMI: Guidelines for Process Integration and Product Improvement, 2nd edition*. Addison-Wesley Professional, 2006.

[4] L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Professional*, 2(3), 2000.

[5] R. Kadri, C. Tibermacine, and V. Le Gloahec. Building the presentation-tier of rich web applications with hierarchical components. In *Proc. of WISE'07*, pages 123–134, Nancy, France, December 2007. LNCS, Springer-Verlag.

[6] M. Lehman and J. Ramil. Towards a theory of software evolution - and its practical impact. In *Proc. of ISPSE'00*, pages 2–11. IEEE Computer Society, 2000.

[7] B W Chatters, M M Lehman, J F Ramil, and P Wernick Modelling A Software Evolution Process : A Long-term Case Study In *J. of Softw. Proc.: Improvement and Practice*, issue 2-3, pages 95-102, July 2000.

[8] M. M. Lehman and J. F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11(1):15–44, 2001.

[9] OMG. Software process engineering metamodel, version 1.1, document formal/2005-01-06. OMG Website: http://www.omg.org/cgi-bin/doc?formal/2005-01-06

[10] OMG. Ocl language specification, version 2.0, document formal/2006-05-01. OMG Website: http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf

[11] OMG. MOF Query / Views / Transformations OMG Website: http://www.omg.org/cgi-bin/doc?ptc/2007-07-07

[12] D. L. Parnas. Software aging. In *Proc. of ICSE'94*, pages 279–287, Sorrento, Italy, 1994. IEEE CS Press

[13] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. SEI Series in Software Engineering. Pearson Education, 2003.

[14] C. Tibermacine, R. Fleurquin, and S. Sadou. On-demand quality-oriented assistance in component-based software evolution. In *Proc. of CBSE'06*, pages 294–309, Vasteras, Sweden, June 2006. Springer LNCS.