XML to Manage Source Engineering in Object-Oriented Development: an Example.

Daniel Deveaux
Lab. VALORIA (AGLAE)
Université de Bretagne Sud
VANNES — FRANCE
Tel: (33) 297 463 175
daniel.deveaux@univ-ubs.fr

Yves Le Traon
IRISA (Action Triskell)
Université de Rennes 1
RENNES FRANCE
Tel : (33) 299
yletraon@irisa.fr

ABSTRACT
In software engineering, XML to date has mostly been used to support three sub-activities: documentation management, data interchange and lightweight data storage. In this position paper, we give an example of using XML technology as the infrastructure for the integrated management of all core software development information.

For several years now we have been developing the concept of *Design for Testability* based on "Design by Contract Approach" and a self-documented and self-testable class model. Since two years we have proposed a master document type that captures all relevant information for this classes, i.e. documentation, contracts, tests, and so on. This document is defined by an XML DTD and we have tentatively named the resulting markup language OOPML: *Object-Oriented Programming Markup Language*. Currently, we are building a dedicated software development framework for several object-oriented languages based on this markup language.

In this paper we present particularly a *mutation tool* build with this technology. This tool is designed to improve the classes tests and contracts quality. Benefits and drawbacks of use of XML technology are anlysed on this example.

Keywords
component-based software engineering, software development process, testing and test tools, software engineering environments, programming by contract, self-testable class, java, XML, XSLT

1 INTRODUCTION

In this position paper, we propose to use XML technology as the main framework to manage object-oriented software development. The approach, in conformance with the basic philosophy of the literate-programming community [2, 3], relies on the observation that the majority of forward and reverse-engineering software tools are built around an internal representation, that is always a more or less sophisticated abstract syntax tree. The incompatible nature of all these proprietary representations is a major obstacle for tool interoperability. The first consequence is a loss of continuity in the development, evolution and maintenance processes. The downstream process flow continuity and especially the upstream one are made very difficult: no automated way exists for propagating any source modification upstream to the design level and checking whether this modification is consistent with the overall design.

XML appears as a well-adapted solution for the problem of integrating all core software development information in order to manage it. The existing gap between analysis and design tools (UML-based) on one hand, and code engineering tools (source code parsing) on the other hand may be bridged thanks to the XML capacity for manipulating high-level syntax trees. The main benefit of this approach is a real downstream and upstream continuity in the process flow. Theoretically, since a common XML basis is provided, the overall consistency through the several levels of design is ensured after any modification. Moreover, quality requirements can be embedded as a part of the process: information such as abstract tests, required testability, robustness or trustability can be attached and verified during the development. Based on this framework, we briefly present a mutation tool that serves for certifying component trustability by checking the consistency between tests, code and specification. The mutations operators do not depend on the implementation language. They are expressed in a generic object-oriented language called OOPML: the tool architecture has been successfully tested for java source code and can be extended to any classical procedural or object-oriented language, and may even be adapted to certify the trustability of dynamic components such as .NET components.

2 MODEL LEVELS TO MANAGE DEVELOPMENT PROCESS CONTINUITY

Since several years, we propose to define all a software project as a documents management project; this *Docware* approach has been presented in [5]. The contribution on which we focus in this paper is the introduction of OOPML (Object-Oriented Programming Meta-Language) as a basis for the source code engineering facet of a software project.

accepted for publication - to appear in XSE01 workshop at ICSE'2001

The goal of our OOPML[1] project is to propose models that catch all the information needed for source engineering with testability and trustability requirements and tools that facilitate direct and reverse source engineering for several object-oriented languages.

The trend in object-oriented software engineering is to raise the development activities in more abstract levels. Many programmers work now with UML and use generation tools to build source code and executable materials. This approach is attractive but it presents still several drawbacks:

- despite of the introduction of new languages such as the OCL (for expressing requirements/properties/contracts) and ASL (for expressing behavior), UML has not yet a full support for semantic information and synchronization: a full generation process implies that code are written as simple comments without structure nor control;
- run-time errors are always reported referring source code lines and it is very difficult to connect a run-time error with the corresponding fault in high level diagram;
- while the generation path, that consists in information addition through generation scripts, is quite feasible, the reverse-engineering path, that should implement an abstraction mechanism, is not yet efficient.

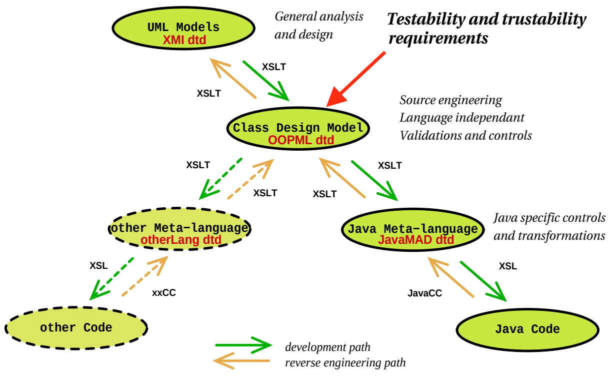

Figure 1: DTD levels in (Java) development process

For these reasons, the way of direct code generation from UML is not easy to implement in real practice and tackles a real problem of process continuity: the more little modification in the application should be made at the model level and the entire generation cycle should be started again. We argue that it is useful to introduce an intermediary *Source Design* level between the *General Analysis and Design* level (that is managed by UML and its interchange language XMI [10]) and the *Source* level. This intermediary level captures, in addition to functionality and semantics of the model, testability and trustability requirements, versioning control and language specific constructions that are usually transversal items to many applications.

[1]URL: http://www.univ-ubs.fr/valoria/oopml/

Figure 1 presents our approach to this problem: the main benefit of this XML-based approach is to integrate the reverse-engineering paths (upstream process flow) and to check the validity of the reversing operations. To preserve the continuity, we propose to share the development process in levels, each one with a corresponding DTD. This continuity process structure is detailed in figure 1 for the application to Java language. Between UML model that is available as XMI representation and the source code, we use two intermediary levels modeled as DTD's:

- JavaMAD (*Java Markup Annoted and Documented*), like JavaML from G.J. Badros [1] captures all Java syntax, but also annotations that can be used to trace elements (in reverse engineering) and more structured documentation (identification of documentation levels, contracts and tests);
- OOPML (*Object-Oriented Programing Meta Language*) that stores a class description without reference to a dedicated programming language: generic languages like OCL or ASL are used to describe contract constraints and algorithms in class methods. The most important points here are the extensibility to other object-oriented languages and the fact that all information relative to the class (technical documentation, contracts, tests and so on) is handled at this level *(Note: OOPML is yet under development)*. In particular, non-functional expected properties can be expressed at this level, for instance testability and trustability requirements. The presented tool manipulates an XML representation of code (java in the first version) and its associated tests to produce a trustability index.

This continuous process makes the transformation scripts simpler and more reusable for the code generation process. Concerning the reverse engineering path, consistency checks may be introduced at each level. The main difficulty of this reverse-transformation consists of defining the abstraction of a current view that should go up in the upper level.

As mentioned in the figure 1 (left branch), the adaptation on a new language is limited; all controls and tools that work on *Class Design Model* level are factorized for any programming languages. A description of OOPML project can be found on the following web site 'http://www.univ-ubs.fr/valoria/oopml/'.

3 THE "DESIGN FOR TRUSTABILITY" APPROACH

Despite the growing interest for component-based systems, few works tackle the question of the trust we can bring into a component. In [8] and [7] we presented a pragmatic approach for linking design and testing of classes, seen as basic unit test components. Each component is enhanced by the ability to invoke its own tests: components are made *self-testable*. While giving to a component the ability to embed its self-test is a good thing for its testability, estimating the quality of the embedded tests becomes crucial for the component trustability.

Software trustability [6], as an abstract software property, is difficult to estimate directly, one can only approach it by analyzing concrete factors that influence this qualitative property. We consider here that the truthfulness in the component test cases is the main indirect factor that brings trust into a component.

It is particularly adapted to a *design-by-contract* approach [9], where the specification is systematically derived into executable assertions (invariant properties, pre/postconditions of methods). A component is seen as an organic set composed of a specification, a given implementation and its embedded test cases: the trust we have is based on the consistency of the component three facets. The methodology is an original adaptation from mutation analysis principles [4]. Faulty programs, called *"mutants"*, are generated by systematic fault injection in the implementation. Our current choice of mutation operators includes selective relational and arithmetic operator replacement, variable perturbation, but also referencing faults (aliasing errors) for declared objects. During the test selection process, a mutant program is said to be killed if at least one test case detects the fault injected into the mutant. Conversely, a mutant is said to be alive if no test cases detect the injected fault. The quality of tests is related to the *Mutation Score*, i.e. the proportion of faulty programs it detects. The trustability approach based on mutation analysis must be applied at a high level of design (`OOPML`) to offer the same basis of comparison between components written in several languages. Even if language-dependent mutation operators can be added, the main interest is the independence of the qualification process from the language target, as it will be detailed in the next section.

4 AN APPLICATION EXAMPLE: THE MUTATION TOOL

In this section, we present a generic mutation tool for building trustable object-oriented components. In the long term, `OOPML` will be the main support for source code engineering: all class information will be stored in `XML` files or `DOM` databases; our `Java` development framework in construction is a prototype of such environment (see `oopml` web site). Another solution consists of using `OOPML` or `JavaMAD` trees as the main data structure for source code related tools. Up to now, we have developed two tools dedicated to this approach:

- support for contracts and self-test handling in `Java` classes (SCoT project),
- `JMutator`, a tool to support mutation test improvements that we present here.

Concerning `JMutator` tool, its execution includes the generation of mutants, the application of test cases against each mutant and the mutation score estimate. The test quality of a component is simply obtained by computing the mutation score for the component unit testing test suite executed with the self-test method. One can choose to use as a decision the difference between the result of the initial implementation and the mutant result. One can choose to use contracts and embedded oracle function (that will be expressed in the `OCL` at the *Class Design Model* level) to make the decision.

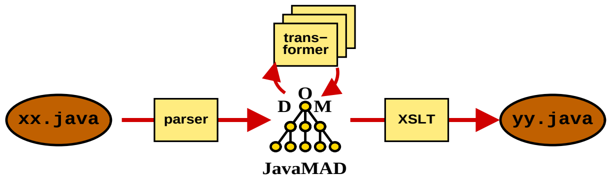

Figure 2: `Java`-to-`Java` transformation tools generic architecture

Figure 2 presents the generic architecture of `JavaMAD` based transformation tools: the base is a pretty printer builded on `DOM` representation of `JavaMAD` document; controls or transformations of `Java` code are made by *plug-ins* that act on `DOM`. A second implementation way is to make explicit `XML` files that is used by `XSLT` engines or specialized transformation tools; this way has been used in the first prototype of `JMutator`. A more complete description of the tool will be presented in an extended version of the paper.

5 CONCLUSION

As a conclusion, a first analysis of the use of `XML` gives a mixed-feeling result since performances of the mutation tool are worse than a language-dependent one (we must generate hundred mutant programs, and execute their tests). A more accurate analysis shows that the actual version of the tool can be easily improved by embedding many `java` parsing parts into the `XML` core. So, even in terms of performances, the approach is promising since we really work at an intermediary level between design and code, by integrating quality requirements such as trustability ones. Moreover we ensure a continuity in the development process flow, upstream and downstream. Further works will concentrate on the testability analysis by pinpointing parts of the `UML` model where potential testing conflicts may occur: constraints rules will be added to avoid undesirable polymorphic and untestable interactions at the code level.

ACKNOWLEDGEMENTS
The *design for testability* approach and self-testable class concept have been developed in collaboration with J-

M. Jézéquel from IRISA's *Triskell Action*[2] and P. Frison from the *AGLAE team*[3]. The `JMutator` tool has been developed by D. Baudoin, F. Brunel, B. Glorot and E. Goasdoue, in a students project of the IFSIC-DIIC[4] engineers school.

REFERENCES

[1] G. J. Badros. Javaml: a markup language for java source code. In *WWW9, Ninth International World Wide Web Conference*, Amsterdam, May 2000.

[2] A. B. Coates. XML and literate programming. 'http://www.ems.uq.edu.au/Seminars/XML_LitProg/', 1998. World-Wide Web document.

[3] R. Cover. Xml and semantic transparency. http://www.oasis-open.org/cover/xmlAndSemantics.html, Nov. 1998. On line article.

[4] R. De Millo, R. Lipton, and F. Sayward. Hints on test data selection : Help for the practicing programmer. *IEEE Computer*, 11:34–41, 1978.

[5] D. Deveaux, G. Saint-Denis, and R. K. Keller. XML support to design for testability. In *Proc. of XOT'2000 workshop at ECOOP'2000*, Cannes (France), June 2000.

[6] W. E. Howden and Y. Huang. Software trustability. In *Proc. of the IEEE Symposium on Adaptive processes- Decision and Control*, number XVII, pages 5.1–5.5, 1970.

[7] J.-M. Jézéquel, D. Deveaux, and Y. L. Traon. Reliable Objects: a Lightweight Approach Applied to Java. *IEEE-Software*, 2001. accepted for publication - to appear.

[8] Y. Le Traon, D. Deveaux, and J.-M. Jézéquel. Self-Testable Components: from Pragmatic Tests to a Design-for-Testability Methodology. In *Proc. of TOOLS-Europe'99*. TOOLS, June 1999.

[9] B. Meyer. Practice to perfect: The quality first model. *IEEE Computer*, 30(5):102–106, May 1997.

[10] OMG. Xml metadata interchange (xmi). ftp://ftp.omg.org/pub/docs/ad/98-10-05.pdf, Oct. 1998. Document ad/98-10-05.

Note: many references on `XML` technologies that have been used in this work are not mentioned here; you can find a larger bibliography on our `XML4SE`[5] web site.

[2]URL: `http://www.irisa.fr/triskell/`
[3]URL: `http://www.univ-ubs.fr/valoria/aglae/`
[4]URL: `http://www.ifsic.univ-rennes1.fr/formation_initiale/diic/`
[5]URL: `http://www.iro.umontreal.ca/labs/gelo/xml4se/`