Language Independent Abstract Metamodel for Quality Analysis and Improvement of OO Systems

Mircea Trifu and Peter Szulman
FZI Forschungszentrum Informatik
Karlsruhe, Germany
{mtrifu, szulman}@fzi.de

1 Introduction

Existing software systems need to change with time, to adapt to expanding business needs by evolving into state-of-the-art IT solutions. During the long life cycle of a software system, continuous changes often lead to the degradation of its structure. Due to its high complexity, the maintenance of such anomalous structures represents a great challenge nowadays. However this process can be greatly supported by several analysis and transformation techniques providing (semi-automatic) detection and correction of design flaws in existing systems.

Our work within the QBench[1] project aims to make a step forward from classical quality analyses and propose appropriate corrective measures in the form of source-code transformations which, if applied, would lead to a better software system. Simply put, while traditional quality analysis techniques try to find problems in source code, the QBench vision is to try and find solutions for these problems.

Both quality analyses and transformations can be expressed efficiently on models which have a higher level of abstraction than the source code of the legacy system itself. Such models contain only the relevant artifacts and not the entire syntactic details, and they could be constructed from the source code using fact extractors. This paper contains a brief introduction to the structure and the semantics of the QBench System-model [4] as it is defined by its metamodel.

The metamodel features support for four OO languages (C++, Java, Delphi, C#) as well as mixtures of these languages (heterogeneous projects written in more than one language),

The following sections present the main parts of this metamodel and discuss some of the design decisions taken.

2 Structure of the Language Independent Metamodel

Given the two main usage scenarios we have in QBench (quality assessment and quality improvement) as well as to ease the understanding of our metamodel, we have mentally divided it into two parts: the analysis metamodel and the transformation metamodel. The analysis metamodel contains the abstractions and operations needed to express state-of-the-art quality analyses, while the transformation metamodel is an extension of this analysis core, which adds transformation capabilities to the model. The following sections present the two parts in greater detail.

[1]http://www.qbench.de

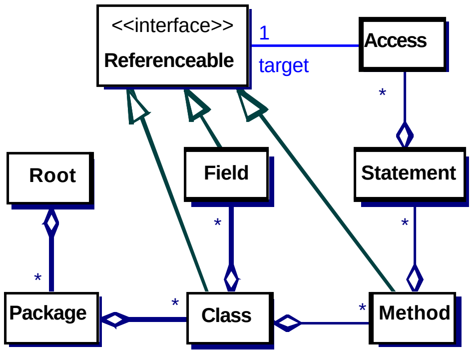

Figure 1: Simplified view of the language independent metamodel

3 The Analysis Metamodel

The goal of the analysis metamodel is to allow the specification of quality analyses in a language independent manner. Our approach for building this language independent abstract metamodel was to take a comprehensive view of all the important features found in the main four OO languages (Java, C++, C# and Delphi) while at the same time unifying similar concepts such as C++ *or* C# *namespaces* and *Java packages*. The result was a more powerful metamodel which also allows the specification of language specific quality analyses (that refer to established language idioms) in addition to quality analyses expressible on a metamodel representing the common core of these languages.

Our metamodel can also accommodate heterogeneous projects (projects written in several languages), which until now could only be analyzed for one language at a time, usually using different tools having not necessarily the same capabilities. Such projects are a reality in today's industrial environment and are increasingly present as technologies such as *CORBA*, *COM+* or more recently *.NET* are already widely used.

When it comes to implementing quality analyses, the language independence of our metamodel pays off again. Since most of these analyses are described in the literature in abstract terms, using metrics, heuristics and quantified violations of well-established OO design principles which apply to all OO languages, it makes sense to have only one implementation for these quality analyses (a *God-Class* [3] in Java is also a *God-Class* when translated to C++, C# or Delphi). As for the language specific quality analyses (violations of language idioms), they can be easily implemented because the abstract OO language described by our metamodel is a superset of each of the individual OO languages.

Figure 1 shows a simplified view of the major abstractions contained in our metamodel. A complete presentation of our metamodel can be found in [4]. The novelty of our metamodel lies in its finer granularity represented by an abstract statements layer. This layer enable us to implement all the metrics and heuristics as dynamic queries on the metamodel as opposed to precomputed values during fact extraction. As we will see in the next section, dynamic metrics are an absolute requirement for the transformation metamodel.

Moreover, our analysis metamodel integrates code duplication information in a natural, intuitive way by mapping code clones to program elements (each clone instance is represented by a list of duplicated abstract statement objects). To our knowledge, our approach to integrate code clones is unique in this respect and has the advantage that clone information may be easily used to refine the other structural analyses as opposed to the separate analysis step typically allowed by text-based code duplication techniques.

4 The Transformation Metamodel

The goal of the transformation metamodel is to allow simulation of common code transformations on the model in order to predict the impact of the real transformations on the internal quality of the software system. As previously mentioned, the aim of *QBench* is to find not only quality related problems in source code, but also appropriate solutions for these problems and produce a list of source-code transformations which, if executed, would lead to better internal quality of the system.

Simulating transformations on a language independent transformation metamodel has the advantage that these transformations can be expressed unitary for all languages in abstract terms, without the huge syntactic overhead that could make the complexity of such an undertaking explode. In most cases, the actual implementation of the above mentioned abstract language independent transformations for the various languages differ only in these language specific syntactic details that have negligible impact on the inner quality of software systems.

In addition, it is a well-known fact that source-to-source transformations on some of the OO languages (e.g. C++, Delphi) are extremely difficult to implement partly due to some syntactic constructs such as *pointers* which make it very difficult and sometimes impossible to guarantee correctness of these transformations. Other OO languages, on the other hand, such as Java or C# benefit from extensive support for source-to-source transformations and refactorings. Our approach makes it possible to offer solutions to improve inner quality even for languages without source-to-source transformation support, where transformations could be carried out by hand.

Simulating transformations means that the abstract transformations are carried out directly on the transformation model. Due to the dynamic nature of the metrics and heuristics implemented on the analysis metamodel the model will always reflect the current structure and quality attributes of the system.

For example, when trying to correct a problem such as *Long method* [2], the obvious solution is to split up the method and move some of the statements to another method (existing or newly created). When simulated on our transformation metamodel, such a transformation would result in automatic updates of the complexity and size measurements of the method in question, as well as the coupling measurements between the class containing the method and the classes referenced in the moved *Statement* objects through the attached *Access* objects. See figure 1 for more details.

5 Conclusion and Future Work

In this work we have defined a metamodel for quality analysis and transformation simulation. Its main advantage is that it lies on a higher abstraction level than an AST, thus hiding unnecessary complexity, while at the same time is fine-grained enough to allow the specification of state-of-the-art quality related analysis as well as to simulate with sufficient accuracy various transformations. The metamodel also makes it possible to propose a list of transformations for languages where automatic source-code transformations are not yet available. It also saves implementation effort when a new language is desired. One only need to implement a language specific mapping and a fact extractor for the given language. We have already implemented fact extractors for Java, C++ and Delphi, while C# is still in the works.

References

[1] CompoBench-Team. Compobench-metamodell (untere schichten). CompoBench Milestone METAMOD, Jul. 2003.

[2] M. Fowler. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[3] A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

[4] M. Trifu, P. Szulman, and V. Kuttruff. Qbench-systemmetamodell. Project Deliverable, Dec. 2004.