Codifying High-Level Software Abstractions as Virtual Classifications

Position paper submitted to the ECOOP'2000 Workshop on objects and classification: a natural convergence

Kim Mens* and Tom Mens

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, B-1050 Brussel, Belgium
E-mail: { kimmens | tommens }@vub.ac.be

Abstract

Current-day software abstractions (architectures, coding conventions, design patterns, interaction protocols) are often not explicitly linked to the code. This lack of traceability causes problems like architectural drift and software erosion. In order to alleviate these problems, we propose to use virtual classifications to codify high-level software abstractions as logic predicates over the implementation. Besides being explicitly linked to the code, these classifications have the additional advantage that they allow us to declare software abstractions in an expressive, concise, readable and intentional way.

1 Introduction

In current-day software development, and object-oriented software development in particular, descending from higher levels of abstraction to lower levels (for example, from design to implementation) is relatively straightforward and sufficiently supported by software engineering methods and tools. Transition in the opposite direction, however, is typically much less supported. As a consequence, current-day software development often follows a top-down approach, starting at high levels of abstractions that are gradually refined to lower level ones. Once the software starts to evolve, however, in the face of time constraints, modifications are often applied directly to the implementation level only, leading to problems like *architectural drift* and *software erosion*: the high-level abstractions will not conform anymore to the modified code. The main reason for these problems is the lack of an explicit link between software abstractions and the corresponding implementation code.

One way to solve the above problem is by following a *logic meta-programming approach* [1] that introduces a logic language at meta-level to reason about software artifacts in the underlying object-oriented base language. In this approach, abstractions over the code are simply expressed as logic predicates at meta-level. This has several advantages. First of all the abstractions are explicitly linked to the code, in a verifiable way. Secondly the declarative

*Research funded by the Brussels' Capital Region (Belgium) and Getronics Belgium.

nature of logic predicates allows us to describe the software abstractions in an intentional and concise way. Finally, there is no limitation to the kind of software abstractions that can be described, as we can resort to the full expressive power of a logic programming language.

Another way to express high-level software abstractions is by using *software classifications* [2]. In its simplest form, software classifications are groups of software artifacts that are qualified with the same tag. Together with the possibility to nest classifications, this approach results in a powerful way to represent high-level software abstractions.

Although these two alternative approaches for representing high-level software abstractions seem completely different, we will show they can be reconciled by using the notion of *virtual classifications*. Instead of explicitly tagging software artifacts, virtual classifications provide an implicit tagging by means of some computational classifier. This is a logical predicate that describes in an intuitive way which artifacts are *intended* to belong to that classification.

This position paper takes a closer look at the advantages of virtual classifications over other kinds of classifications, and motivates the use of virtual classifications for representing high-level software abstractions, using software architecture as a concrete example.

2 Taxonomy of classifications

First we provide a taxonomy of existing classification models. Based on this taxonomy, we discuss the advantages of virtual classifications over other kinds of classification.

2.1 Traditional OO classifications

Classification is a central idea in the object-oriented programming paradigm. Consider for example the Smalltalk language: methods and instance variables are grouped in *classes*, objects are *instances* of a class, classes are instances of a *meta class*, classes belong to inheritance *hierarchies*, methods are grouped in method *protocols*, classes are classified in class *categories*, changes to the Smalltalk image are grouped in *change sets*, and so on. All of these can be considered as a kind of *predefined* classifications. Enhancements of the Smalltalk language, such as the Envy/Developer version management system, extend the classification possibilities even further (e.g., Envy contains a notion of *versions* and *applications*).

2.2 The software classification model

De Hondt [2] reports on positive experiences with recovering design knowledge in terms of simple *software classifications*. All artifacts in such a classification typically share some important characteristic. For example, in a financial application it could be interesting to group all software artifacts dealing with "handling deposits" together in a single classification.

A software artifact can belong to different classifications and a single classification can contain many different kinds of software artifacts. A classification does not necessarily correspond to the traditional classifications that are typically found in the programming language or the development environment, but may be user defined. More precisely, De Hondt [2] distinguishes essentially two kinds of classifications: *manually-defined* classifications and *virtual or computed* classifications.

- A manually-defined classification allows a developer to group a set of software artifacts that are not necessarily (or not explicitly) related in the software. An example of

manually-defined classifications are the so-called *collaboration-contract classifications* where the grouping is based on a user-specified collaboration between software artifacts.

- Both virtual and computed classifications are specified intentionally and 'compute' their elements. Examples are: subclass hierarchies, senders and implementers, and so on. The only difference between virtual classifications and computed ones is that the former are always kept synchronized with the software, whereas the latter are only recomputed on demand.

Computed and virtual classifications are clearly more flexible than manually constructed ones, because they actually *describe* which artifacts are intended to belong to the classification, instead of explicitly enumerating them. Furthermore, when declared in a logic medium, their definitions are often very intuitive and concise, and can be used in multiple ways (e.g., checking, generating, ...).

As opposed to De Hondt [2], we will make no terminological distinction between virtual and computed classifications. Whether or not they are kept synchronized with the software, depends on the support tool and on how the classifications are to be used. Therefore, in this paper, we will uniformly refer to both kinds of classifications as *virtual classifications*.

2.3 Criteria

In this subsection we present a list of criteria that can be used to compare existing classification models.

Intentionality Does the classification enumerate its elements explicitly or through tagging (extensional), or does it provide a description of how to *compute* its elements (intentional)?

Cross-cutting. Is it possible to define cross-cutting classifications that group artifacts that cut across the dominant structure of the software? In other words, is it possible to define overlapping classifications?

User-defined. Can a user define his or her own classifications or is there only a predefined set of classifications available?

Genericity. Is it possible to define generic, or parameterized classifications that depend on the value of their parameters?

Robustness. Is the classification mechanism robust with respect to evolution? When the software evolves, is there a chance that earlier classifications become incorrect?

Nesting. Is it possible to define nested classifications?

Heterogeneity. Is it possible to mix different kinds of artifacts in the same classification (heterogeneous), or can we only put artifacts of the same kind in each classification (homogeneous)?

Constraints. Can we put extra (enforceable, or verifiable) constraints on the artifacts contained in a classification, or even put constraints between different classifications?

2.4 Virtual classifications

Based on the above list of criteria, in this subsection, we argue that virtual classifications provide an extremely powerful classification mechanism.

The main advantage of virtual classifications over explicit enumerations of software artifacts is their intentional character. In natural language, the *intention* of a word is that part of meaning that follows from general principles in semantic memory. The *extension* of a word is the set of all existing things to which the word applies. The intention of 'mammal', for example, is a definition, such as "warm-blooded animal, vertebrate, having hair and secreting milk for nourishing its young"; the extension is the set of all mammals in the world [5]. Similarly, in set or type theory, the extension is the collection of all values belonging to that set or type. The intention is a formal definition of these values in terms of some property they all have in common.

Because of their intentional character, virtual software classifications have many advantages over explicitly enumerated software classifications. First of all, an intentional definition often has a much more concise representation. Secondly, an extensional definition is less intuitive than an intentional one. An extensional definition is also less precise than the intentional one. For example, two classifications can have the same extension, but a different intention. The converse is not true: two classifications that have the same intention, must always have the same extension. Let us illustrate this again with a natural language example taken from [5]. Since 'grandfather' and 'father of parent' have the same intention, they must apply to exactly the same people. On the other hand, 'featherless biped' and 'animal with speech' have the same extension, the set of human beings; but they have different intentions. Finally, intentional definitions are more robust towards change than extensional definitions. This is because intentions are true by definition, whereas extensions can be falsified by changing events: plucking a chicken results in a featherless biped that cannot speak.

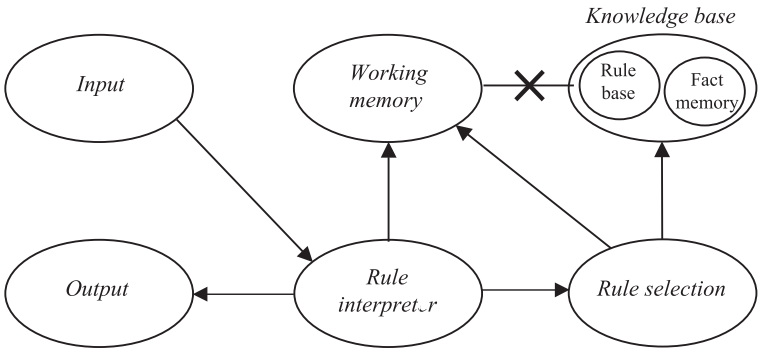
In the next section we will take a look at a concrete case study. In addition to illustrating the intentional nature of virtual classifications, we explain that virtual classifications can also adequately address the criteria of cross-cutting, genericity, nesting, heterogeneity and constraints.

3 Virtual classifications for software architectures

In [3, 4], we explored the expressive power of virtual software classifications to codify and reason about software architectures. In particular, we used virtual classifications and their interrelationships for declaring software architectures as high-level abstractions over implementation artifacts, and proposed and implemented an algorithm for checking conformance of the implementation to this high-level architecture.

As a concrete experiment, we codified and checked conformance of the rule-based architecture of the Smalltalk implementation of SOUL, a logic language with a tight symbiosis with the Smalltalk development environment [6]. The figure below depicts this architecture. The kernel of SOUL is a logic query interpreter which conforms to this architecture. Moreover, this architecture is representative for rule bases in general and is sufficiently challenging to be used as a case study.

The architecture is codified in terms of relationships between virtual classifications. SOUL being implemented in Smalltalk, methods, classes and variables were considered as building blocks. For instance, the Working Memory was simple to define: it contains all classes that



derive from a root class that specifies the generic structure of variable-value bindings. A more challenging example is the rule that specifies how methods are classified as belonging to the Rule Interpreter:

```
methodIsClassifiedAs(Method, 'Rule Interpreter') :-
    classImplements('SOULQuery', 'interpret:repository:', M),
    reaches(M, Method).
```

This rule computes every `Method` that is invoked directly or indirectly by the method with name `interpret:repository:` of the class `SOULQuery`, capturing the intention that this particular method is the one that launches the interpretation process. The predicate `classImplements` retrieves the method with a certain name from a given class, and the predicate `reaches` computes the transitive closure of the invocation relationship.

In addition to illustrating the *intentional* character of virtual classifications, this virtual classification is an illustration of the *cross-cutting* criterion, because the generated classification cross-cuts the static class-hierarchy structure of the SOUL implementation.

As an illustration of the *heterogeneity* as well as the *genericity* criterion, we can provide a generalization of the above predicate so that it does not only return methods belonging to some (parameterized) architectural concept (like `'Rule Interpreter'`) but also all instance variables and classes belonging to that concept:

```
isClassifiedAs(Entity, Concept) :-
    methodIsClassifiedAs(Entity, Concept);
    variableIsClassifiedAs(Entity, Concept);
    classIsClassifiedAs(Entity, Concept).
```

Filling in the `Concept` parameter with a concrete classifier, as in the query `isClassifiedAs(Entity, 'Rule Interpreter')` returns an actual heterogenous classification containing all methods, variables and classes corresponding to the Rule Interpreter concept.

In [4], Rule Interpreter itself is considered as an architecture that is subdivided into an Interpreter and a Substitution concept. Therefore, we could also generalize the logic predicate `isClassifiedAs` so that it does not simply return a list of all artifacts corresponding to Rule Interpreter, but instead subdivides them hierarchically in Interpreter artifacts and Substitution artifacts. The generated result is a nested classification, illustrating the *nesting criterion*.

Finally, it should be clear that logic predicates cannot only be used to specify classifications, but also to express *constraints* or relationships between these classifications. In the figure, these constraints are represented by arrows (representing *uses* and *creates* relationships) between the architectural concepts (the ellipses) defined by the various classifications.

In [4], a declarative definition for the *uses* and *creates* relationships was given for each of the kinds of implementation artifacts. These relationships could then be combined with universal and existential cardinality constraints to define a limited family of architectural relationships; this resulted in a specification of the architecture as a ten-line logic fact. Conformance checking of the SOUL implementation could then easily be implemented by 'applying' the defined relationships over the corresponding classifications, making use of the specified cardinalities.

4 Conclusion

Our experiments with architectural conformance checking [3, 4] illustrate that the notion of virtual software classifications, expressed in a logic meta language, provides a viable formalism to reason about the structure of a software system at a sufficiently high level of abstraction. Virtual classifications proved their worth as suitable high-level abstractions of implementation artifacts. They hide the details of the lower-level artifacts on which they are mapped, yet allow us to reason about their relationships with other classifications independently of the artifacts they actually contain.

Virtual classifications have many advantages over explicitly enumerated classifications, mainly because of their more intentional nature. This, combined with the expressive power gained by adopting a logic meta-programming approach, makes virtual classification an ideal mechanism for codifying high-level software abstractions.

References

[1] T. D'Hondt, K. De Volder, K. Mens, and R. Wuyts. Co-evolution of object-oriented software design and implementation. In *Proceedings of SACT 2000*. Kluwer Academic Publishers, January 2000. International symposium on Software Architectures and Component Technology.

[2] K. D. Hondt. *A Novel Approach to Architectural Recovery in Evolving Object-Oriented Systems*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1998.

[3] K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 2000. In preparation.

[4] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *TOOLS 29 — Technology of Object-Oriented Languages and Systems*, pages 33–45. IEEE Computer Society Press, 1999. Nancy, France, June 7-10.

[5] J. F. Sowa. *Conceptual Structures — Information processing in mind and machine*. The Systems Programming Series. Addison-Wesley, 1984.

[6] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings TOOLS USA'98*, pages 112–124. IEEE Computer Society Press, 1998.