Using Software Classifications To Drive Code Generation

Tom Tourwé [*] & Kris De Volder
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium
E-mail: {Tom.Tourwe,kdvolder}@vub.ac.be
WWW: http://prog.vub.ac.be/

March 24, 2000

Abstract

Software classifications are mainly considered as a means for helping the programmer in navigating and exploring the code. This paper illustrates by means of examples that software classification is also useful for code generation. This requires a highly flexible and expressive classification mechanism. We propose to use a logic programming language for this purpose.

1 Introduction

Today, software classification is mainly used for helping a programmer to understand, explore and navigate the source code of a complex software system. Tools that support classification [DH99] typically include different browsers and inspectors which provide the user with appropriate views on (parts of) the software system. As such, we can safely say that classifications currently only serve to inspect the code.

We hold the position that this view should be extended and that code generation on classifications should be possible. The general idea behind it is that a classification groups together several related entities which share some characteristics. Often, this is reflected by these entities having in common some state and behavior. Since these entities need not in any way be related through inheritance, the state and behaviour is spread out and duplicated. When using code generation on classifications, this problem can easily be alleviated. Instead of manually duplicating the code over the different entities, we define behavior and state on the classification and let the code generator take care of all the work.

To achieve this goal, we will provide a definition of a classification in terms of a logic program. Then, we will provide a simple example that explains how this setup can be used to classify software entities and generate code on this classification. Afterwards, a more elaborate example will be presented, explaining how a classification based on relationships between different entities can be defined and used to generate code.

[*]Author financed with a doctoral grant from the Flemish Institute for the Improvement of the Scientific-Technological Research in Industry (IWT).

2 Classifications as logic programs

We propose to define software classifications in terms of logic programs. A classification is defined as the result of a query on the logic database, since the result of a query is typically a set of elements. As such, the elements in a classification are determined by the facts and rules in the database. The most straightforward way of identifying the elements of a classification is by explicitly enumerating them one by one by asserting facts. A less laborious way is using rules, which describe the properties the elements of the classification should provide. We thus rely on the logic evaluator to compute the actual elements. Computed classifications have the advantage that, if new artifacts are introduced into the system that satisfy the appropriate conditions, they are automatically included in the classification.

The environment we use for defining the classifications and generating code is QSOUL, which is an extension of SOUL [Wuy98] with some features that enable code-generation [DV98, DV99]. It is a PROLOG-like, declarative rule-based language, implemented on top of Smalltalk, which provides a way of reasoning about the structure of Smalltalk programs. It incorporates two special terms, called SmalltalkBlocks and QuotedCodeBlocks. The former are delimited by square braces and are used to provide an escape to the underlying Smalltalk environment. The latter contain strings that represent code and are delimited by curly braces. These strings can possibly contain logic variables that can be instantiated by subsequent queries.

The fact that QSOUL allows reasoning about the structure of Smalltalk program allows to define rules that describe a classification and restrict the elements in that classification to those that obey some structural properties.

3 A Simple Example

Many classes define `at:` and `at:ifAbsent:` methods, to retrieve an element from some composite structure. Most of the time, the `at:` method calls the `at:ifAbsent:` method with a default parameter that provides an error message. Instead of defining this method each and every time again, we can group all classes that implement the `at:ifAbsent:` method into a classification by defining the following rule [1]:

```
Rule indexable(?class) if
    class(?class),
    classImplements(?class, [#at:ifAbsent:])
```

This predicate states that a class `?class` is included in the classification only if it implements that `at:ifAbsent:` method. The `classImplements` predicate makes use of the reflective features of QSOUL to check whether a given class effectively implements the given selector.

We can now use this definition to state that all classes present in the classification should implement an `at:` method that calls the `at:ifAbsent:` method with a default parameter as follows:

```
Rule addInstanceMethod(?class, {
    at: anObject
```

[1]Note that logic variables in QSOUL start with a questionmark instead of with a capital as in Prolog.

```
        self at: anObject
             ifAbsent: [ self error: 'Element not present' ]
}) if
    indexable(?class).
```

The addInstanceMethod predicate is a hook into our code generator, which looks for all occurrences of this predicate and actually compiles the code accompanying it into the image. Other predicates exist which provide hooks for adding class methods, removing instance methods and so on.

4 An Extended Example

The example of the previous section is reasonably straightforward. It just uses a single argument predicate to characterize a set of classes that belong to the same classification. The usefulness of this sort of classifications for code generation is rather limited, because it implies that all of the code inserted into each of the classes in the classification must be completely identical. Often however, this is not the case. The Visitor pattern [GHJV95], for example, contains a lot of duplicated code that is not exactly identical, but is more like a filled-in 'template'.

```
AbstractASTVisitor>>visitIfStatementNode: anIfStatementNode
    self subclassResponsibility
AbstractASTVisitor>>visitReturnStatementNode: aReturnStatementNode
    self subclassResponsibility
...
```

In this code, there is clearly some duplication: it differs only in using the name of a different visited class to fill in the method 'template'. The code duplication here is due to the implicit relationship that exists between the visitor class and a number of different classes that it visits. To overcome this kind of code duplication, we can define a classification which holds tuples rather then single elements and thus is able to express a relationship. This is easily accomplished using multi-argumented predicates and queries. For example, if our Visitor classification is characterized by a query visitor(?visitorClass, ?visitedClass), we can use it to generate the duplicated code as follows:

```
Rule addInstanceMethod(?visitorClass, {
    visit?visitedClass: a?visitedClass
        self subclassResponsibility
}) if
    visitor(?visitorClass, ?visitedClass)
```

We can also use this classification to automatically generate the accept: method on all the visited classes:

```
Rule addInstanceMethod(?visitedClass, {
    accept: aVisitor
        ^aVisitor visit?visitedClass: self
}) if
    visitor(?visitorClass, ?visitedClass)
```

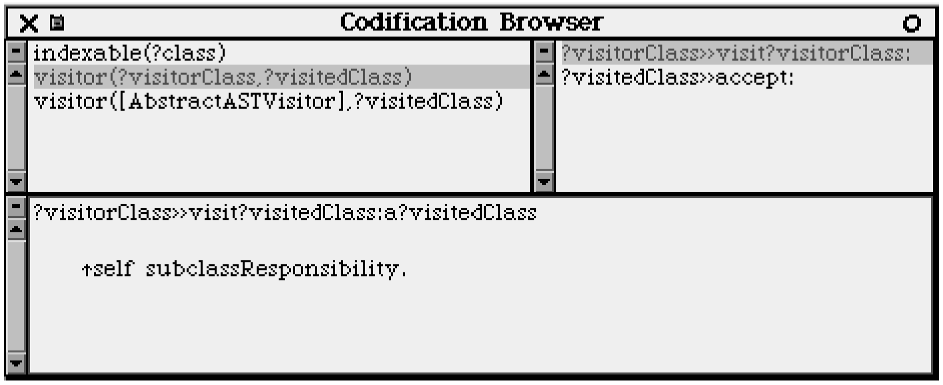
The elements of the classification can again be defined in a number of different ways: by enumerating all possible relationships between a visitor and a visited class one by one using facts, or by writing rules, or by using a combination of computed and explicitly named participants. The latter possibility is shown below:

```
Rule visitor([AbstractASTVisitor], ?visitedClass) if
    class(?visitedClass),
    hierarchy([AbstractASTNode], ?visitedClass),
    isConcreteClass(?visitedClass)
```

This rule expresses first of all that `AbstractASTVisitor` is a visitor. Second, it relates the visitor to its visited classes: all the non-abstract subclasses of the `AbstractASTNode` class.

5 The Codification Browser

Although up till now, all examples were elaborated completely in the syntax of the logic language, we can implement a browser which provides a much more intuitive and user-friendly interface. Below is a screenshot of such a browser.



```
indexable(?class)
visitor(?visitorClass,?visitedClass)
visitor([AbstractASTVisitor],?visitedClass)
```

```
?visitorClass>>visit?visitorClass:
?visitedClass>>accept:
```

```
?visitorClass>>visit?visitedClass:a?visitedClass

    ↑self subclassResponsibility.
```

Figure 1: The Codification Browser

In the upper left pane, the queries representing the existing classifications are listed. When choosing a particular classification, the upper right pane shows a list of the headings of the specified method templates. Upon selecting one of these, the bottom pane shows the entire template. This browser looks a lot more convenient but in fact it is just a view of all rules of the form:

```
Rule addInstanceMethod(?class, { ?template }) if
    ?classificationquery
```

6 Conclusion

In this paper, we argumented that code generation on software classification would be a very interesting extension to using classifications as a means for exploring and navigating code. We defined a software classification as the result of a query on a logic database. This allowed to define the elements of a classification in terms of facts and/or rules. As it turns out, this provides an elegant mechanism for defining classifications that hold tuples rather then single elements, thus expressing an explicit relationship between several entities. This is particularly interesting for code generation, as it allows to use code templates that are instantiated dependent on the relationships between several classes, instead of just one single class.

References

[DH99] Koen De Hondt. *A Novel Approach To Architectural Recovery in Evolving Object-Oriented System.* PhD thesis, Vrije Universiteit Brussel, 1999.

[DV98] Kris De Volder. *Type-Oriented Logic Meta Programming.* PhD thesis, Vrije Universiteit Brussel, Programming Technology Laboratory, June 1998.

[DV99] Kris De Volder. Aspect-oriented logic meta programming. In Pierre Cointe, editor, *Meta-Level Architectures and Reflection, Second International Conference, Reflection'99*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1999.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series, 1995.

[Wuy98] Roel Wuyts. Declarative Reasoning about the Structure of Object-Oriented Systems. In *Technology of Object-Oriented Languages and Systems*, 1998.