# A Declarative Meta-Programming Approach to Framework Documentation

Tom Tourwé & Tom Mens
{tom.tourwe,tom.mens}@vub.ac.be
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2
1050-Brussel-Belgium

**Abstract**

The documentation of software artifacts in general, and object-oriented frameworks in particular, has always been problematic. In this paper, we advocate the use of a declarative meta-programming environment to document software artifacts. In particular, we show how a significant and important part of the design of a framework can be adequately and concisely documented in such an environment, and how this allows us to use this documentation in an active way.

## 1 Introduction

Documentation of software artifacts has been, and appears to remain, a major problem. Documentation is often non-existent, hopelessly out of date and/or inconsistent with the current state of the implementation [3]. Developers, who are responsible for evolving and maintaining those artifacts, are thus heavily discouraged to use the documentation, and are tempted not to read even the most well-crafted documentation [9]. Consequently, this gives rise to serious problems such as code duplication, design inconsistencies, design erosion [11], architectural drift [4], etc..

In particular for framework-based software development, the framework's design is the most important asset to document [6]. It is the design that should be reused by many different applications, and it is the design that should be changed when the framework should evolve. In what follows, we will show how a declarative meta-programming environment can be used to document (part of) the design of a framework, and how it enables us to use this information in an *active way*, rather than passively as is the case now. This active use includes checking the completeness of the documentation, e.g. whether all important parts are included, and its correctness, e.g. whether it is still consistent with the current state of the implementation. As such, we are able to detect whenever this documentation is out of date, and which of its parts are affected.

## 2 Why Declarative Meta Programming?

We conducted our experiments in the SOUL declarative meta-programming environment [12]. We believe such an environment is extremely well suited for documenting a framework and using its documentation actively, for the following reasons:

**The declarative nature** of SOUL, and logic programming languages in general, allows us to represent all sorts of knowledge in a straightforward, accurate and concise way [7]. For our purposes, we will use *logic facts* to describe the appropriate design knowledge.

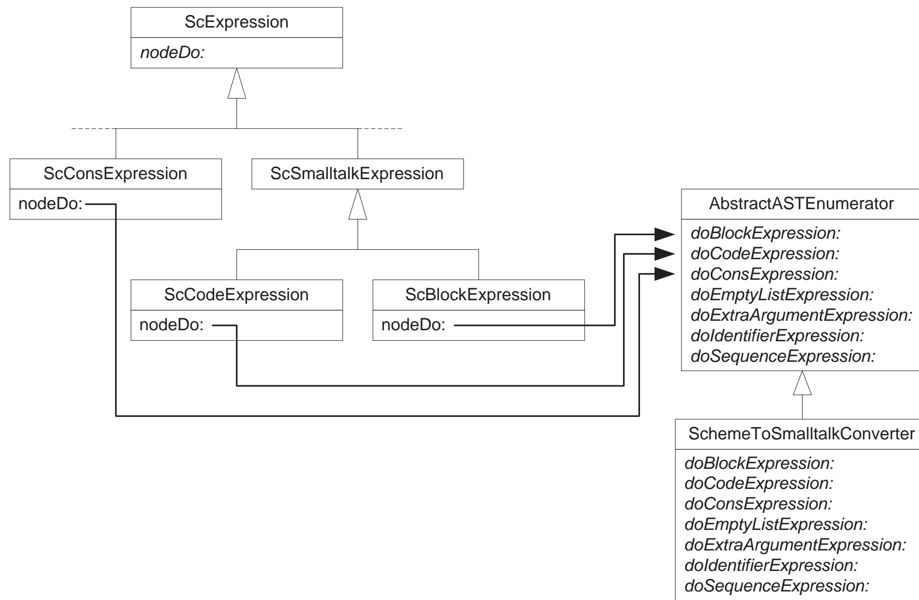


Figure 1: An instance of the *Visitor* design pattern

**The powerful reasoning capabilities** of logic programming languages are also very useful for our purposes. We can use *logic rules* to derive additional information from existing facts, for example, or to reason about the information represented by these facts. Such reasoning will allow us to check the completeness of the documentation.

**The tight integration** of SOUL with the standard Smalltalk development environment allows it to consult and reason about the current implementation of the framework. As such, it enables us to write logic rules that access the implementation and check whether the documentation is still correct.

# 3 Documenting a Framework's Design

Frameworks allow applications to plug in their specific behavior by defining appropriate *hot spots* [8]. Documenting a framework's design thus boils down to documenting the corresponding hot spots, and in particular how these hot spots are expected to be used.

As has already been shown in literature, design patterns are excellent means for documenting the hot spots of a framework [5, 2]. Design patterns define how the hot spots they implement can be used by applications to plug in their specific behavior and expose important information about the particular roles and responsibilities of the classes and methods involved. This is the kind of information that we will explicitly document in our declarative meta-programming environment.

## 3.1 An Example Design Pattern Instance

Consider the example instance of the *Visitor* design pattern depicted in Figure 1. It shows part of an *expression* hierarchy and its associated *visitor* hierarchy, that are used in a framework for building Scheme interpreters [1, 10]. The *Visitor* design pattern is used in the `ScExpression` hierarchy to allow adding new operations on expressions in a straightforward and flexible way. This simply boils down to adding a new subclass of the `AbstractASTEnumerator` class, and does not require us to change the expression classes.

Moreover, the *Visitor* design pattern defines the particular *participants* that should be present in its implementation, and exposes information about the specific *roles* and *responsibilities* of these participants. For example, it defines *abstractElement* and *abstractVisitor* roles, and requires corresponding class participants to provide an implementation for the *acceptMethod* and *visitMethod* roles respectively.

We can document this particular instance of the *Visitor* design pattern in our declarative meta-programming environment by using logic facts as follows:

```
dpRole(astVisitor,abstractElement,ScExpression).
dpRole(astVisitor,concreteElement,ScConsExpression).
dpRole(astVisitor,concreteElement,ScBlockExpression).
dpRole(astVisitor,concreteElement,ScCodeExpression).
...
dpRole(astVisitor,acceptMethod,nodeDo:).
dpRole(astVisitor,abstractVisitor,AbstractASTEnumerator).
dpRole(astVisitor,concreteVisitor,SchemeToSmalltalkConverter).
dpRole(astVisitor,visitMethod,doConsExpression:).
dpRole(astVisitor,visitMethod,doBlockExpression:).
dpRole(astVisitor,visitMethod,doCodeExpression:).
...
dpRelation(astVisitor,<ScConsExpression,doConsExpression:>).
dpRelation(astVisitor,<ScCodeExpression,doCodeExpression:>).
dpRelation(astVisitor,<ScBlockExpression,doBlockExpression:>).
...
```

The *dpRole* predicate maps roles onto participants. Its first argument is used to identify the particular design pattern instance that is being documented, the second argument denotes the role, and the third argument denotes the class, method or variable playing that role. The *dpRelation* predicate is used to document relations between participants. In this case, it is used to reflect the relation between *concreteElement* and *visitMethod* participants (e.g. the *Visitor* design pattern expects each *concreteElement* participant to define an *acceptMethod* participant that calls a specific *visitMethod* participant).

# 4 Actively Using the Documentation

Documenting the design of a framework in a meta-programming environment allows us to check the completeness of this documentation and to verify whether it is still up to date with the current implementation. This will be shown in the following sections.

## 4.1 Checking Completeness

We can easily check whether the documentation of a design pattern is complete, e.g. whether it includes all necessary roles and participants. This requires us to first state which roles a design pattern instance should provide. This is achieved as follows for our *Visitor* design pattern example:

```
requiredRole(visitorDP,abstractElement).
requiredRole(visitorDP,concreteElement).
requiredRole(visitorDP,abstractVisitor).
requiredRole(visitorDP,concreteVisitor).
requiredRole(visitorDP,acceptMethod).
requiredRole(visitorDP,visitMethod).
```

Based on this information, we use a logic rule that consults the documentation of the design pattern instance to see if it effectively includes a description for every required role:

```
checkPatternInstance(?pattern, ?instance, ?absentRoles) if
     findall(?role,
               and(requiredRole(?pattern,?role),
                      not(dpRole(?instance,?role,?))),
               ?absentRoles)
```

As a side effect, this rule returns the list of roles that is not included in the design pattern instance documentation.

Conversely, for some specific participants, we can check whether they are included in the documentation, as they should. The following rule, for instance, checks whether all concrete subclasses of an *abstractVisitor* class participant are registered as *concreteVisitor* participants:

```
checkPatternInstance(?, ?instance, ?absentParticipants) if
     dpRole(?instance,abstractVisitor,?abstractVisitor),
     findall(?role,
               and(hierarchy(?abstractVisitor,?class),
                      concreteClass(?class)
                      not(dpRole(?instance,concreteVisitor,?class))),
               ?absentParticipants)
```

The *hierarchy* predicate returns all (possibly indirect) subclasses of the *abstractVisitor* class participant, while the *concreteClass* predicate checks whether a class is indeed a concrete class.

### 4.2 Checking Consistency

We are also able to check whether the documentation is still consistent with the current implementation of the framework. This is particularly important given the fact that framework evolve over time.

We can achieve such verification thanks to the fact that design patterns impose constraints upon an implementation, and that we can represent these constraints explicitly in SOUL. For instance, one constraint of the *Visitor* design pattern is that it requires each *concreteElement* participant to be a (possibly indirect) subclass of the *abstractElement* participant. We can express this constraint in a logic rule that consults both the documentation and the implementation and checks whether they are consistent, as follows:

```
patternConstraint(visitorDP,?instance,incorrectCE(?violators)) if
[1] dpRole(?instance,abstractElement,?abstractElement),
     findall(?concreteElement,
[2]            and(dpRole(?instance,concreteElement,?concreteElement),
[3]                   not(hierarchy(?abstractElement,?concreteElement))),
               ?violators)
```

The logic predicates at line 1 and 2 consult the documentation of the design pattern instance, whereas the *hierarchy* predicate consults the implementation to see if the correct inheritance relation holds between the two classes. Classes that do not adhere to the above rule are reported as *incorrect concrete element* participants.

The above example only shows one constraint for the *Visitor* design pattern. Others can be defined in a similar way. Furthermore, similar constraints can be implemented in a similar way for other design patterns as well.

## 5 Conclusion

In this paper, we have shown how declarative meta-programming can be used for documenting a framework's design. Thanks to the declarative nature of SOUL, we achieved this in a straightforward, accurate and concise way. Moreover, thanks to SOUL's powerful reasoning and meta-programming capabilities, we were able to use this documentation actively to check whether it is both complete and correct with respect to the current implementation.

We strongly believe this is an important first step towards better and more active documentation for frameworks and software artifacts in general. The ideas presented in this paper actually form part of a more general approach to document and reason about a framework's implementation, its instantiation and evolution at a high-level. We refer the interested reader to [10].

## References

[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[2] Kent Beck and Ralph Johnson. Patterns Generate Architectures. In *Proceedings of the European Conference on Object-Oriented Programming*, 1994.

[3] Greg Butler and Pierre Dénommée. *Documenting Frameworks to Assist Application Developers*, chapter 7. John Wiley and Sons, 1999.

[4] C. B. Jaktman, J. Leaney, and M. Liu. Structural Analysis of the Software Architecture – a Maintenance Assesment Case Study. In *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*. Kluwer Academic, 1999.

[5] Ralph Johnson. Documenting Frameworks Using Patterns. In *Proceedings of the OOPSLA Conference on Object-Oriented Programming, Systems, Languages and Applications*, 1992.

[6] Ralph Johnson and Brian Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1988.

[7] G. F. Luber and W. A. Stubblefield. *Artificial Intelligence, Structures and Strategies for Complex Problem Solving*. Addison-Wesley, 1998.

[8] Wolfgang Pree. Essential Framework Design Patterns. *Object Magazine*, 1997.

[9] Mark Rettig. Nobody Reads Documentation. *Communications of the ACM*, 34(7):19–24, 1991.

[10] Tom Tourwé. *Automated Support for Framework-Based Software Evolution*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2002.

[11] Jilles van Gurp and Jan Bosch. Design Erosion: Problems & Causes. *Journal of Systems & Software*, 61(2):105–119, 2001.

[12] Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 2001.