

# Using Declarative Metaprogramming To Detect Possible Refactorings

Tom Tourwé Johan Brichau\* Tom Mens  
{tom.tourwe,johan.brichau,tom.mens}@vub.ac.be  
Programming Technology Lab  
Vrije Universiteit Brussel  
Pleinlaan 2  
1050-Brussel-Belgium

July 18, 2002

## Abstract

In this paper, we advocate the use of declarative metaprogramming to detect violations of important (object-oriented) design guidelines and best practices. This is particularly useful for detecting when a design should be refactored, and which refactorings in particular should be applied. As we will show, a declarative environment incorporating metaprogramming capabilities is very well suited for detecting such violations and providing information for possible refactorings.

## 1 Introduction

Many design guidelines and best practices have been proposed over the years, with the specific intent of promoting good object-oriented design principles [?, ?]. At the same time, much research has been devoted to identifying refactorings, e.g. high-level transformations, that can help in transforming an inflexible, lowquality design into a more flexible one [?, ?]. Although some primitive tools exist [?], recognizing when a design guideline is violated, and when refactorings may thus be necessary, remains a manual process, as is identifying which refactorings in particular could be used to remedy the situation.

In this paper, we advocate the use of declarative metaprogramming (DMP) [?, ?] to fill in this gap. The declarative nature of DMP allows us to accurately express design guidelines in a straightforward and intuitive way. Moreover, explicit metaprogramming enables us to reason about the source code of an application so as to actually verify whether it does not violate any of those guidelines.

In what follows, we will present an example of how declarative metaprogramming can be used to detect defective designs. We will achieve this by using the SOUL declarative metaprogramming environment [?], both as a medium to describe the conditions for defective designs, and as a test case for these conditions.

---

\*Research assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

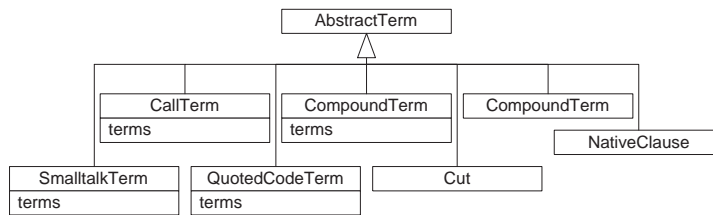


Figure 1: An example of an inappropriate interface

## 2 Why Declarative metaprogramming

Design guidelines are actually rules that the implementation of an application should adhere to. Most of these rules are quite simple and can often be expressed in a straightforward way in natural language. **Misschien hier een paar voorbeelden geven? ”Steeds de juiste supermessage sturen”, ”geen cancellation van methods”, het LSP, etc.**) In order to actively verify these rules, they have to be specified programmatically. This is quite cumbersome to achieve in current-day standard programming languages, such as C++ and Java. First of all, these languages are not particularly well suited to express *rules* per se. Second, most of the current-day standard programming languages do not have adequate meta-programming capabilities (with Smalltalk being the exception that proves the rule).

SOUL on the other hand, is a logic programming language that is tightly integrated with the standard (Smalltalk) development environment. Logic languages naturally allow to express rules, by mere definition. Furthermore, the tight integration allows SOUL to reason about and manipulate programs written in Smalltalk, and does make SOUL perfectly well suited for metaprogramming purposes. Moreover, it also means that the SOUL environment is always synchronized with the development environment. Another advantage of using SOUL is the declarative nature of the logic paradigm. It has already been shown that logic programming languages are particularly well suited for metaprogramming, because they allow meta programs to be specified in an intuitive way [?].

The SOUL programming language is actually a Prolog-dialect with some extensions to allow Smalltalk expressions to be evaluated as part of a logic program. These extensions allow to reify and represent all information from the Smalltalk image as logic facts. SOUL comes with an extensive library of logic programs that reason about this information to conclude more high-level information, such as the existence of design patterns.

## 3 Detecting Design Guideline Violations

### 3.1 Inappropriate Interfaces

Good interfaces are extremely important when designing flexible and reusable object-oriented systems. Any situation in which the interface of a class is inappropriate, incomplete or unclear should thus be avoided at all costs.

As a concrete example, consider the `AbstractTerm` hierarchy depicted in

Figure ?? shows part of the implementation of the SOUL environment. As can be observed, the `CallTerm`, `CompoundTerm`, `SmalltalkTerm` and `QuotedCodeTerm` classes each provide an implementation for the `terms` method, whereas all other classes (including `AbstractTerm`) do not. This situation creates a problem when we want to extend the `AbstractTerm` hierarchy with a new class. It is not directly clear from the design which subclasses of `AbstractTerm` should provide an implementation for the `terms` method, and which subclasses should not. A developer confronted with this situation should thus know exactly what he is doing.

To correct the design, two different solutions are possible. Either an intermediate superclass is inserted between the original superclass and all subclasses that share the interface. This newly introduced superclass should then provide the shared interface. Another option is to extend the interface of the original superclass with the interface shared by the subclasses. This also exposes the interface to subclasses that did not originally provide it, however, which may not be desired.

### 3.2 Problem Statement

The above mentioned problem occurs whenever some (but not all) of the subclasses of a class share an interface, that is not provided by that class itself. Detecting such situation manually is not as straightforward as it may seem, however. Standard browsers included in current-day programming environments only offer a local and narrow view of the source code. Developers thus often lack a more general overview, that would allow them to identify such problems. Appropriate tool support is thus clearly indispensable, and this is where declarative metaprogramming comes in.

### 3.3 Detecting The Problem

Using SOUL, we can easily detect this situation by implementing the following logic rules.

```
implementingSubclasses(?superclass,?selector,?subclasses) if
  subclassImplements(?superclass,?selector, ?),
  not(classImplements(?superclass,?selector)),
  findall(?subclass,
          subclassImplements(?superclass,?selector,?subclass),
          ?subclasses).
```

The *implementingSubclasses/3* predicate calculates all subclasses of a given superclass that implement a particular selector which is not implemented by the superclass itself. The rule is implemented in terms of two auxiliary predicates, *classImplements/2* and *subclassImplements/3*. The latter predicate is implemented as follows:

```
subclassImplements(?superclass,?selector,?subclass) if
  subclass(?subclass,?superclass),
  classImplements(?subclass,?selector)
```

It uses the *subclass/2* and *classImplements/2* predicates that are part of the library of logic rules in SOUL that consult the implementation to retrieve the requested information. The *subclass/2* predicate checks whether there exists

a direct inheritance relation between two classes, while the *classImplements/2* predicate checks whether a class implements a particular selector.

What remains is verifying whether the set of subclasses that is calculated by the *implementingSubclasses/3* predicate does not contain all subclasses of the given class. This simply boils down to comparing sets for equality:

```
inappropriateInterface(?superclass,?selector,?subclasses) if
    implementingSubclasses(?superclass,?selector,?subclasses),
    not(allSubclasses(?superclass,?subclasses))
```

We can now use SOUL to detect inappropriate interfaces in our implementation. Therefore we invoke the following query, which will return the design violation we mentioned:

```
if inappropriateInterface([SoulAbstractTerm],?selector,?subclasses)
```

### 3.4 Discussion

The above discours clearly shows that declarative metaprogramming is very well suited to express the problem of inappropriate interfaces. Moreover, the rules presented above not only detect the situation of inappropriate interfaces, but also convey information about the interface that is shared by the subclasses, and those subclasses themselves. This can prove valuable when a particular refactoring has to be applied.

Using the above rules, we were able to identify several interface conflicts in the implementation of the SOUL environment. All reported conflicts were effectively real conflicts that needed to be solved in order to end up with a better and more suitable design. The information gathered by the logic rules was instrumental in applying the necessary refactorings.

We envision a programming environment where several of these 'design guidelines violations' can be detected using declarative metaprogramming. Depending on the detected violations, a range of refactorings can be proposed to the developer, who can choose the appropriate one to be applied. This linking of violations to the correct refactorings remains to be investigated, but once again, the declarative metaprogramming environment could prove to be ideal to express such information.

## 4 Conclusion

In this paper, we have shown the usefulness of a declarative meta-programming approach for detecting violations of important design guidelines and best practices. We demonstrated that the declarative nature of such an approach allows us to define the conditions under which such violations occur in a straightforward and intuitive way. Moreover, we illustrated that explicit metaprogramming capabilities are absolutely essential for such an approach.

While we have only shown one, rather simple, example of a design guideline violation, we believe the approach is general enough to detect all sorts of other violations as well, even on a much complexer scale. Further experiments in this direction are mandatory, however. It is our firm believe that such an approach could be a first step towards tool support for detecting not only when a design should be refactored, but also which particular refactorings it should undergo.

## References

- [1] Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.
- [2] James R. Cordy and Medha Shukla. Practical metaprogramming. Technical report, Software Technology Laboratory, Queen's University, 1992.
- [3] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Departement Informatica, Vrije Universiteit Brussel, 1998.
- [4] Martin Fowler. *Refactoring: Improving the design of existing code*. Addison Wesley Longman, 1999.
- [5] Y. Kataoka, M. D. Ernst, W. G. Griswold, and D. Notkin. Automated support for program refactoring using invariants. In *Proc. Int'l Conf. Software Maintenance*, pages 736–743. IEEE Computer Society Press, 2001.
- [6] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.
- [7] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Publishing Company, April 1996.
- [8] 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.