Supporting Software Development through Declaratively Codified Programming Patterns

Kim Mens, Isabel Michiels
{ kimmens | Isabel.Michiels }@vub.ac.be
Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel, Belgium

Roel Wuyts
Roel.Wuyts@iam.unibe.ch
Software Composition Group
Institut für Informatik
Universität Bern, Switzerland

Abstract

In current-day software development, programmers often use programming patterns to clarify their intents and to increase the understandability of their programs. Unfortunately, most software development environments do not adequately support the declaration and use of such patterns. To explicitly codify these patterns, we adopt a declarative meta-programming approach. In this approach, we reify the structure of an (object-oriented) program in terms of logic clauses. We declare programming patterns as logic rules on top of these clauses. By querying the logic system, these rules allow us to check, enforce and search for occurrences of patterns in the software. As such, the programming patterns become an active part of the development and maintenance environment.

Keywords: programming patterns, logic programming, meta-programming, tool support

1 Introduction

Contemporary software development practice regards software construction as an incremental and continuous process that involves large development teams. In such a context, it is crucial that the software is as readable as possible. One cannot afford that programmers have to wade through piles of documentation and code to understand the software or to discover the intents of the original programmers. Instead, they should spend their precious time to tackle the real problem (that is, the task of programming itself, i.e. conceptualizing, designing, implementing and maintenance [7]).

By using commonly accepted programming (and design) patterns, it becomes much easier for programmers to communicate their intents [1]. A problem with ad-hoc patterns, however, is that they are not supported by the programming language nor by the development environment. For example, whether or not a certain programming pattern is consistently used throughout a program solely depends on the programmers' discipline.

> By relieving the mind of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.
>
> *Alfred North Whitehead*

To allow programmers to gain maximum profit from the extra information that is encoded in patterns, there is a need for tools that support the use of such patterns. We envision the patterns as becoming an explicit and active part of the development environment. Some activities that such an environment should support are:

- *checking* whether a piece of source code matches a pattern;
- *finding* all pieces of code that match a pattern;
- *searching* for all occurrences of patterns that were used to program a piece of source code;
- *detecting violations* of the usage of a pattern;
- *enforcing* the consistent use of a pattern throughout a program;
- *generating code* that matches a pattern.

This paper advocates the use of a declarative meta-language for expressing and reasoning about programming patterns in object-oriented programs.

2 Declarative Meta-Programming

Declarative meta-programming (DMP) is an instance of hybrid language symbiosis, merging a declarative language at meta-level with a standard (object-oriented) base language. Base-level programs are expressed in terms of facts and rules at the meta-level. Programming patterns are expressed as rules that reason about the clauses representing those base-level programs. By querying the logic system, the rules can be used to check, detect, search for occurrences of and even generate code fragments from programming patterns. Before discussing what the programming pattern rules look like, we first elaborate on the base and meta-language.

As *declarative meta-language*, we use a Prolog-variant. Logic programming has long been identified as very suited to meta-programming and language processing in general. Prolog's expressive power (e.g. unification and backtracking) and its capacity to support multi-way reasoning[1] are particularly attractive to reason about patterns.

Although DMP can be applied to programs written in any programming language, in this paper we take the object-oriented language Smalltalk as *base language*. One reason for choosing Smalltalk for our experiments is that there exists a "Smalltalk culture" which makes that Smalltalk programmers use a lot of well-known patterns to express important intents [1, 3], but for which no explicit language constructs are available.

2.1 Setup

A DMP environment consists of 4 main elements. In a *logic language*, we declare programming patterns as logic meta-programs that reason about programs written in an (object-oriented) base language. The logic meta-programs are stored in a *logic repository*. The base-level language constructs are stored in an *implementation repository* that can be accessed from within the logic language, by means of a *meta-level interface*.

For the experiments in this paper, we used the logic language *SOUL* [8] to allow powerful logic reasoning about Smalltalk programs. SOUL was implemented in Smalltalk and contains a primitive construct, called "Smalltalk term", for evaluating Smalltalk expressions as part of logic rules. This allows SOUL clauses to reason about Smalltalk source code by making direct meta-calls to the Smalltalk image.

[1]A prototypical example is the append/3 predicate, which can be used to append two lists, check whether a list is the concatenation of two others, check for and generate prefixes and postfixes of a list, and so on.

2.2 The Representational Mapping

The *representational mapping* defines the meta-level interface between the declarative meta-language and the object-oriented base language. For each base-language construct we want to reason about at meta-level, there is a logic fact or rule which *reifies* that construct at meta-level. Figure 1 lists some of the predicates that constitute this representational mapping.[2]

Reification is achieved by using SOUL's symbiosis with Smalltalk to access the Smalltalk image directly by executing a piece of Smalltalk code as part of a logic rule. E.g., the SOUL rules below reify the notion of 'classes': [3]

Rule class(?Class) **if**
  constant(?Class),
  [*Smalltalk includes:* ?Class *name*].

Rule class(?Class) **if**
  variable(?Class),
  generate(?Class, [*Smalltalk allClasses*]).

The first rule declares what happens when the class predicate is called with a constant value. In that case, the special Smalltalk term [*Smalltalk includes:* ?Class *name*] checks whether the value represents an existing class in the Smalltalk image. Note that a Smalltalk term used in the position of a predication is required to return true or false. The second rule is applied when ?Class is variable. In that case, a primitive generate predicate is used to unify that variable (the first argument of the predicate) with each of the classes present in the Smalltalk image. This is done by executing the Smalltalk expression between square brackets, which is provided as second argument to the generate predicate. A Smalltalk expression used in a generate predicate is supposed to return a collection of results.

Given these rules, the query **Query** class([*Array*]) verifies whether *Array* is an existing class in the Smalltalk image, whereas the query **Query** class(?Class) unifies ?Class with a class in the Smalltalk image. Note that a Smalltalk term used in the position of a logic term can return any value.

Other rules that reify Smalltalk language constructs are defined in a similar way; see [5, 9] for more examples. In the next section, we show how best practice patterns, design patterns and other programming patterns can be encoded in SOUL.

[2]In this table, a variable ?C represents a Smalltalk class, ?M a method parse tree, ?N a method name, ?V an instance variable name, ?P the name of a Smalltalk method protocol, ?MC a Smalltalk meta-class, ?Stats a list of Smalltalk statements and ?Args a list of names of argument variables.

[3]In SOUL, the keyword **if** separates the body from the head of a **Rule** ; logic variables start with question marks; a comma denotes logical conjunction; lists are delimited with <> and terms between square brackets represent Smalltalk expressions that may contain (bound) logic variables.

Predicate	Meaning
class(?C)	?C is a class
classImplementsMethod(?C,?N,?M)	class ?C implements method ?M with name ?N
methodArguments(?M,?Args)	method ?M has argument list ?Args
methodClass(?M,?C)	method ?M belongs to class ?C
methodName(?M,?N)	method ?M has name ?N
methodStatements(?M,?Stats)	method ?M has list of statements ?Stats in its body
instVar(?C,?V)	class ?C has instance variable with name ?V
isSentTo(?C1,?C2,?N,?Args)	in class ?C1 message ?N with argument list ?Args is sent to receiver ?C2
metaClass(?C,?MC)	class ?C has meta-class ?MC
methodInProtocol(?C,?P,?M)	method ?M of class ?C belongs to method protocol ?P
subClass(?C1,?C2)	class ?C1 has subclass ?C2

Figure 1. The representational mapping

3 Codifying Programming Patterns

Every programming language has its set of patterns that experienced programmers follow to produce more understandable code. They use such patterns to make clear their intents and to improve the overall readability of the software. Well-known kinds of such patterns are *best practice patterns* [1], *design patterns* [4], *design heuristics* [6], *bad smells* and *refactoring patterns* [2]. In this section, we illustrate some of these patterns and show how they can be codified in a DMP medium.

3.1 Best Practice Patterns

Beck's "Smalltalk best practice patterns" capture commonly accepted programming conventions for Smalltalk [1]. They suggest how to choose clear names for objects, instance variables and methods, how to communicate the programmer's intents through code, how to write understandable methods, etc. As concrete examples we discuss the *Getting Method* and *Constructor Method* best practice patterns.

3.1.1 Getting Method

One way to make the distinction between state and behavior more transparent is by hiding every access to the state of an object by a message send. This is the motivation behind the idea of accessing methods. An *accessing method* is responsible for getting or setting the value of an instance variable. All references to an instance variable should be made by calling these methods. Methods that get the value of a variable are *Getting Methods*; methods that set the value of a variable are *Setting Methods*. The Getting Method best practice pattern [1] states:

> **Getting Method** How do you provide access to an instance variable?
>
> *Provide a method that returns the value of the variable. Give it the same name as the variable.*

One possible DMP implementation for representing the structure of a Getting Method is given below. It declares that the statement list of a Getting Method consists of a single statement, which merely returns the value of the instance variable ?V:

Fact gettingMethodStats(<return(variable(?V))>,?V).

Note that the above fact expresses only the simplest form of a Getting Method. Other forms of Getting Methods can be codified by adding similar facts or rules. E.g., a Getting Method that uses 'lazy initialization' has an extra statement to initialize the value of the variable the first time the variable is retrieved. Due to space limitations, we did not include these other forms here.

The predicate gettingMethodStats only defines the statement list of a Getting Method. To check whether a method of a class is a Getting Method for some instance variable, we need to verify that the instance variable belongs to that class, that the method has the same name as the variable and that the method returns that particular instance variable.[4]

Rule gettingMethod(?Class,?Method,?InstVar) **if**
    classImplementsMethod(?Class,_,?Method),
    instVar(?Class,?InstVar),
    methodName(?Method,?InstVar),
    gettingMethodStats(?Stats,?InstVar),
    methodStatements(?Method,?Stats).

This gettingMethod predicate states that ?Class has an instance variable ?InstVar and a Getting Method ?Method that retrieves the value of that variable.[5]

[4]The underscore symbol '_' denotes a special variable of which the actual value is unimportant.

[5]To facilitate reasoning about method statements, a ?Method is represented as a logic data-structure that corresponds to the method's parse tree,

3.1.2 The Constructor Method

The Constructor Method best practice pattern indicates how you best express the creation of a class instance [1]:

> **The Constructor Method.** How do you represent instance creation?
> *Provide methods that create well-formed instances. Pass all required parameters to them. (Put Constructor Methods in a method protocol called "instance creation".)*

The fact that all Constructor Methods are, by convention, put in the *instance creation* method protocol, makes it very easy to codify this pattern:

```
Rule constructorMethod(?Class,?Meth) if
    metaClass(?Class,?Meta),
    methodInProtocol(?Meta,[#'instance creation'],?Meth),
    returnType(?Meth,?Class).
```

In Smalltalk, Constructor Methods are defined on meta-classes. Hence, we retrieve the meta-class and verify that ?Meth belongs to its 'instance creation' method protocol. As an extra consistency check, we verify that the Constructor Method returns an instance of the correct type ?Class, by using an auxiliary predicate returnType(?Meth,?Class).

This typing predicate returnType only 'guesses' the type because Smalltalk is dynamically typed. To infer the type of the expression that is returned by the method, we look at all messages that are sent to that expression (in the context where it occurs). A class is a possible type for that expression if it understands all these messages (if not, a 'message not understood' error may occur at run-time).

3.2 Design Patterns

Whereas best practice patterns define programming conventions at the level of single classes, methods or instance variables, *design patterns* [4] have a more global scope and focus on typical class collaborations. As with best practice patterns, we codify the structure[6] of design patterns as logic meta-programs that reason about the structure of a base-level program. As an illustration, we codify the Visitor design pattern (structure).

The general idea of the Visitor design pattern is to separate the *structure* of some elements from the *operations* that can be applied on these elements. This separation makes it easier and more cost-effective to add new operations, because the classes that describe the element structure do not need to be changed. Separating the nodes of a parse tree from the different operations performed on those nodes (such as generating code, pretty printing, optimizing) is the typical example where the Visitor design pattern offers a solution.

As shown in Figure 2, in the Visitor design pattern structure there is a hierarchy describing the elements and there is a separate hierarchy implementing the operations. Assume that *Element* is the root class of a hierarchy on which the subclasses of the class *Visitor* define operations. Every *Element* class defines a method *accept* that takes a *Visitor* as argument and calls this visitor. This call is in general unique for that element. The *Visitor* hierarchy consists of the classes that define operations on the *Element* classes. They just need to implement the calls made by the different element classes.

The rule describing the structure of the Visitor design pattern is fairly straightforward. First of all, it declares that ?Visitor is a class that implements the visit method ?VisitSelector. In the same way, the class ?El implements a ?Method called ?AcceptM. This method is responsible for calling the visitor ?V with the actual visit operation ?VisitSelector. Finally we need to verify that one of the arguments of this call is the receiver (denoted by *self* in Smalltalk) and that the passed visitor ?V is an argument of the accept method:

```
Rule visitor(?Visitor,?El,?AcceptM,?VisitSelector) if
    classImplementsMethod(?Visitor,?VisitSelector,_),
    classImplementsMethod(?El,?AcceptM,?Meth),
    methodStatements(?Meth,
            <return(send(?V,?VisitSelector,?VisitArgs))>),
    member(variable([#'self']),?VisitArgs),
    methodArguments(?Meth,?AccArgs),
    member(?V,?AccArgs).
```

3.3 Other Programming Patterns

Next to best practice patterns and design patterns, other patterns exist that check whether or not the software is well designed or well structured. Examples are Riel's *design heuristics* [6] and Beck and Fowler's *bad smells* [2]. As a typical example consider the following heuristic [6, Heuristics 5.6 and 5.7]:

> *All abstract classes must be base classes and all base classes should be abstract classes.*

This heuristic can be codified as follows:

```
Rule abstractClassHeuristic() if
    forall(abstractClass(?Class),baseClass(?Class)),
    forall(baseClass(?Class),abstractClass(?Class)).
```

rather than as a string containing the original Smalltalk source code. The predicate methodStatements/2 matches a list of Smalltalk statements with the statements occurring in such a parse tree.

[6]Note that a design pattern captures more than only the *structure* of a class collaboration. It also has a *motivation*, *intent*, *applicability*, as well as relationships with other design patterns. In this paper, however, we only focus on the *structure* of design patterns.

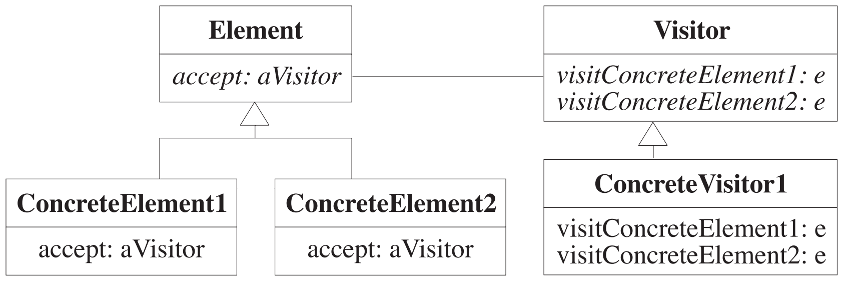

Figure 2. Visitor Design Pattern Structure

where baseClass(?Class) checks whether ?Class is a class from which another class inherits and abstractClass(?Class) checks whether ?Class is abstract by verifying that it contains at least one abstract method. In Smalltalk, abstract methods can be recognized because they make a *subclassResponsibility* self send. In other words, we check whether their statement list matches the following pattern:
<send(variable([*#'self'*]),[*#'subclassResponsibility'*],<>)>

A second example of a programming pattern for detecting ill-designed code is the Duplicated Code *bad smell* [2]:

> ***Duplicated Code***
> *...A common duplication problem is when you have the same expression in two sibling subclasses. ...*

This 'bad smell', together with its proposed solution, is similar to Riel's heuristic 5.10 [6], which suggests when and how to refactor two classes that implement the same state and behavior:

> *If two or more classes have common data and behavior (i.e. methods) then those classes should each inherit from a common base class which captures those data and methods.*

Below, we codify two rules that check for a common expression in two classes.[7] Two classes ?Class1 and ?Class2 have *common behavior* if they implement a method with the same method body.

```
Rule commonBehavior(?Class1,?Class2,?Met1,?Met2) if
   classImplementsMethod(?Class1,_,?Met1),
   classImplementsMethod(?Class2,_,?Met2),
   methodStatements(?Met1,?Statements),
   methodStatements(?Met2,?Statements).
```

Having *common data* is codified as having a common instance variable ?InstVar of the same type.

```
Rule commonData(?Class1,?Class2,?InstVar) if
   instVar(?Class1,?InstVar),
   instVar(?Class2,?InstVar),
   instVarType(?Class1,?InstVar,?Type),
   instVarType(?Class2,?InstVar,?Type).
```

Similar to the returnType predicate, our lightweight type inference rules guess the type of an instance variable by looking at all messages sent to that variable (in the scope of its class) and computing all classes that understand all these messages. In addition, initialization of variables, as well as factory methods and getting and setting methods are taken into account.

4 Supporting Software Development

In the previous section we used DMP to declare many kinds of programming patterns. In this section we elaborate on how a programmer can use these rules to support him when developing or maintaining software. The rules can be used in different ways: checking whether a certain pattern is satisfied, searching source code that matches some pattern, detecting violations of patterns, and even code generation.

4.1 Checking and Searching

Due to the multi-way reasoning capability of our logic language, most predicates can be used in multiple ways. To illustrate this, let us elaborate on the gettingMethod predicate of Subsection 3.1.1. When calling the predicate with constant arguments, it merely *checks* whether a given method of a given class is a Getting Method for a given instance variable. When the query contains variables, we *search* for all values that satisfy the pattern. For example,

Query gettingMethod([*ApplicationModel*],?M,[*#'builder'*])

returns the Getting Method for the variable '*builder*' of the Smalltalk class *ApplicationModel*. We can even use more than one logic variable, as in

[7] To save space we only show the simplest implementation.

```
Query gettingMethod([ApplicationModel],?M,?InstVar)
```

which finds all Getting Methods together with their corresponding instance variable for the class *ApplicationModel*.

We can also use the predicate in the opposite way to find all classes that have a Getting Method for a given instance variable ‘*name*’:

```
Query gettingMethod(?Class,?Method,[#'name'])
```

Again, this query returns several results (one for each of the classes that implements such a Getting Method).

Finally, we can call the predicate with variables only, in which case all classes in the entire Smalltalk image are searched for Getting Methods. Computing such a query may take a very long time, however.

A similar reasoning can be made for all other predicates that were defined in Section 3. As a second example of "checking and searching" we revisit the commonBehavior rule of Subsection 3.3 that tells us when to move common behavior in sibling subclasses to their common base class. We can use the rule below to find all classes ?C1 and ?C2 that should be refactored or to detect whether two classes have some behavior in common, and so on. The rule also returns the common base class and the methods to be moved.

```
Rule behaviorRefactoring(?C1,?C2,?Base,?M1,?M2) if
    subClass(?Base,?C1),
    subClass(?Base,?C2),
    commonBehavior(?C1,?C2,?M1,?M2).
```

4.2 Detecting Violations

Getting Method Coming back to the Getting Method pattern, in addition to checking whether a method is a Getting Method and searching the image for occurrences of Getting Methods, we can also write queries that check the source code for *violations* of the Getting Method pattern.

Methods that violate the encapsulation imposed by the Getting Method programming pattern are methods that directly send messages to instance variables (with the exception of Getting Methods themselves, because they are the only ones allowed to do so). The rule for detecting such violations verifies whether no method implemented in a class sends messages that have as receiver an instance variable of that class:

```
Rule accessingViolator(?Class,?Meth,?IV,?ViolMsg) if
    instVar(?Class,?IV),
    classImplementsMethod(?Class,_,?Meth),
    not(gettingMethod(?Class,?Meth,?IV)),
    isSentTo(?Class,variable(?IV),?ViolMsg,_).
```

We can then invoke the query below to find all violations of the Getting Method pattern. It returns the violating method ?Meth that directly accesses some instance variable ?IV, together with the class ?Class it belongs to and the violating message ?Msg it sends to the instance variable.

```
Query accessingViolator(?Class,?Meth,?IV,?Msg)
```

Visitor Design Pattern As an illustration of how to use the visitor predicate of Subsection 3.2 for detecting violations, consider some class hierarchy with root class *ParseTreeElement* representing a parse tree. We want to detect all non-abstract parse tree elements that do not comply to the Visitor pattern. To do so, we select all subclasses of *ParseTreeElement* that are not abstract, and for each of those we find the ones that do not comply to the visitor rule:

```
Query  hierarchy([ParseTreeElement],?Node),
       not(abstractClass(?Node)),
       not(visitor(?Visitor,?Node,[#'doNode:'],?VisSel))
```

The last line in this query mentions the name of the visit-method (i.e., ‘*doNode:*’) used by the visitor to visit the nodes. When we do not know the name of this method, we can leave it variable. The system will then deduce the name used in this specific instance of the visitor pattern.

The results of this query contain the methods that do not comply to the Visitor design pattern, and that might need to be reimplemented. If the query fails, this means that all classes and methods satisfy (the structure of) the Visitor design pattern.

4.3 Code Generation

Getting Method Instead of searching for Getting Methods and violations thereof, it can be useful to generate automatically the code of the Getting Methods for some instance variable of a class. This can be done by combining the gettingMethodStats predicate describing the body of a Getting Method with a primitive predicate generateMethod that generates the source code of a method from its logic parse tree description.[8]

```
Rule generateAccessor(?Class,?InstVar) if
    instVar(?Class,?InstVar),
    "Verify that no method with name ?InstVar exists"
    not(classImplementsMethod(?Class,?InstVar,_)),
    gettingMethodStats(?Stats,?InstVar),
    "Generate code from the method parse tree description"
    generateMethod(
        method(?Class,?InstVar,<>,<>,?Stats)).
```

[8]A method parse tree description consists of five parts: the method's class, the name of the method, its argument list, a list of temporary variables and a statement list.

Due to space limitations, we do not show the detailed implementation of the generateMethod predicate. It is a meta-predicate that makes use of the strong symbiosis between SOUL and Smalltalk to add the method (in parse-tree format) it takes as input to the Smalltalk image; see [9] for more details.

Behavior refactoring As a second example of code generation, we reconsider the predicate behaviorRefactoring of Subsection 4.1. It only searches the image for common methods to be refactored. To perform the actual refactoring, we codify the Pull Up Method refactoring pattern [2].

> ***Pull Up Method***
> You have methods with identical results on subclasses.
> *Move them to the superclass.*

We only show the easiest case where two methods have exactly the same body (typically as a result of "copy and paste" programming). The rule below defines how to do the refactoring. The comments (between parentheses) explain the code; the mechanics of the refactoring corresponds to what is described in [2].

```
Rule pullUpMethod(?C1,?C2) if
   "Check that ?C1 and ?C2 have common behavior"
   behaviorRefactoring(?C1,?C2,?Base,?M1,?M2),
   "Retrieve information about the common method"
   methodName(?M1,?Name),
   methodStatements(?M1,?Stats),
   methodArguments(?M1,?Args),
   methodTempVars(?M1,?Temps),
   "Verify that the common base class ?Base does
    not implement a method with the same name"
   not(classImplementsMethod(?Base,?Name,_)),
   "Generate code for the new method"
   generateMethod(
        method(?Base,?Name,?Args,?Temps,?Stats)),
   "Delete code of the old methods"
   removeMethod(?M1),
   removeMethod(?M2).
```

In addition to the generateMethod meta-predicate, this rule uses a removeMethod meta-predicate to remove a given method from the Smalltalk image.

Being able to generate code has the important advantage that a programmer gains time to concentrate on more intellectually-rewarding development or maintenance activities. Straightforward coding tasks can be performed partially. For example, we might imagine having some kind of design pattern tool where we just select some pattern from which a code template is automatically generated for the programmer to fill in.

In the next section, we further elaborate on possible tool support and on how to integrate the DMP language with an existing development environment.

5 Tool Support

Our logic meta-language SOUL is well integrated in the Smalltalk development environment. It can reason about and manipulate Smalltalk entities directly and can even execute parameterized Smalltalk source-code fragments. Conversely, SOUL queries can be executed from within Smalltalk itself. All this is achieved by implementing SOUL in Smalltalk and by using the powerful reflective capabilities of Smalltalk to obtain a good symbiosis with the logic language.

More recently, SOUL was extended with a *synchronization framework* to build tools that rely on some kind of synchronization between design[9] and implementation [9]. It enables the construction of tools that monitor and act upon any change to the implementation or design. For example, we can make a tool to *enforce* the use of certain patterns in the implementation. Suppose that we want to enforce the consistent usage of the Getting Method best practice pattern throughout a program. The tool monitors all changes to methods and gives an error or warning whenever a programmer accepts a method that accesses an instance variable directly instead of through a Getting Method.

In our experiments we worked directly at the level of the logic meta-language. We defined our own logic rules and used logic queries directly to reason about patterns. However, for programming patterns to become an explicit and active part of the development environment we need well-integrated and user-friendly support tools in that environment.

One of the already developed tools is the 'Structural Find Application', a sophisticated search engine. This tool transparently uses logic queries to allow searching for methods or classes in the Smalltalk image using complex search patterns. The user only needs to fill in one or more simple selection fields and the Find Application will automatically generate and interpret the corresponding query for the user. For example, the Find Application may be used to find all classes that have a name matching some pattern, have a method sending some specified message and implement a method with some name. The results of the search are presented in a user-readable format.

A second interesting tool that has been implemented on top of SOUL is the 'To Do Application'. During implementation of a program it logs all violations of certain programming patterns, conventions and heuristics in a "to do" list. This list can be inspected later by the programmer to fix (or ignore) the detected problems.

[9]or other high-level descriptions on top of the implementation

A third example (which is currently being developed) is a tool for visualizing and manipulating design patterns. It supports the definition of design patterns, generating code templates, searching for occurrences of design patterns in the source code, checking consistency of design pattern instances, evolution and transformation of design patterns, detecting and resolving conflicts and so on.

Tools like the above hide the details of the logic meta-language from the programmer. However, they do not prohibit a programmer to access the logic meta-language. Instead of using the provided high-level tools, a (power-)user can always use the query engine directly to reason about the software. For example, although the Find Application supports very powerful search queries, it is restricted to some fixed set of selection fields. By using the query engine directly, even more powerful searches can be performed, because the full SOUL syntax and all predefined predicates can be used to construct a search query.

Also, a programmer can always add to the logic repository his own specific rules to declare some pattern. All available tools on top of the logic language should be open enough so that they automatically provide support for these additional patterns as well.

6 Conclusion

We discussed the importance of using programming patterns to support software development and maintenance. Especially in a context of continuously evolving software, large development teams and a high turn-over rate, advanced tools to support the software development process are crucial. Current-day software development environments and tools, however, provide little or no support to declare and use best practice patterns, design patterns, design heuristics, bad smells and refactoring patterns.

In this paper, we proposed declarative meta-programming as a basis for building sophisticated development tools that aid a programmer in his programming tasks. We illustrated this by expressing different kinds of programming patterns as rules in a DMP language and by showing how these rules could be used to search for occurrences of, to check, to detect violations of and to enforce programming patterns and even to generate code. DMP proved to be an ideal medium for expressing and using such rules, because:

- it is *declarative* (hence intuitive and readable);
- the specific benefits of *logic* languages: *multi-way reasoning* allows one and the same rule to be used in many different ways; *unification* provides a powerful pattern matching mechanism; *backtracking* can find all possible solutions of a query;
- it is *base-language independent*: the rules that describe the patterns can, to a certain extent, also be used for other object-oriented languages;
- it is *customizable*: *user-defined rules* can easily be expressed. A programmer can declare and use his own set of rules that support his particular development and maintenance activities.

Finally, if we can rely on the fact that, in a given piece of software, certain programming patterns are consistently used throughout the code, we effectively reach a higher level of abstraction of the code. This makes it possible to reason about even more powerful concepts, like architectural abstractions [5].

References

[1] K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall PTR, 1997.

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

[3] S. Fraser, A. Cockburn, L. Brajkovich, J. Coplien, L. Constantine, and D. West. OO anthropology: Crossing the chasm (panel 3). In *Proceedings of OOPSLA 1996 Conference*, volume 31(10) of *ACM SIGPLAN Notices*, pages 286–291. ACM Press, October 1996.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

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

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

[7] W. Teitelman. Automated programmering: The programmer's assistant. In D. R. Barstow, H. E. Shrobe, and E. Sandewall, editors, *Interactive Programming Environments*, pages 232–239. McGraw-Hill, 1984.

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

[9] R. Wuyts. *A Logic Meta-Programming Approach to Support Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, January 2001.