Supporting Software Development through Declaratively Codified Programming Patterns [⋆]

Kim Mens [a,1], Isabel Michiels [a,*], Roel Wuyts [b]

[a]*Programming Technology Lab, Vrije Universiteit Brussel*

Pleinlaan 2, B-1050 Brussel, Belgium

[b]*Software Composition Group, Institut für Informatik, Universität Bern*

Neubrückstrasse 10, CH-3012 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 certain patterns in the software. As such, the programming patterns become an active part of the software development and maintenance environment.

Key words: programming patterns, logic programming, meta programming, tool support, object-oriented programming

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 (Teitelman, 1984)).

Beck (1997) argues that by using commonly accepted programming patterns it becomes much easier for programmers to communicate their intents. Well-known kinds of such patterns are *best practice patterns* (Beck, 1997), *design patterns* (Gamma et al., 1995), *design heuristics* (Riel, 1996), *bad smells* and *refactoring patterns* (Fowler, 1999). A problem with these 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 and implicit conventions.

> *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 programming 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 software development and maintenance environment. Some activities that such an environment should support are:

- *checking* whether a piece of source code matches a certain pattern;
- *finding* all pieces of source code that match a pattern;

- *searching* for all occurrences of a given pattern that were used to program a piece of source code;

- *detecting violations* of the usage of a pattern;

- *enforcing* the consistent use of some pattern throughout a program;

- *generating code* that matches a certain pattern.

In this paper, we propose to use 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 logic facts and rules at the meta level. Programming patterns are expressed as logic rules that reason about the logic 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, in this section we 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 [2] are particularly attractive to reason about

---

[2] 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 or generate prefixes and

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" (Fraser et al., 1996) which makes that Smalltalk programmers use a lot of well-known programming patterns to express important intents (Beck, 1997), but for which no explicit language constructs are available.

2.1 Setup

A DMP environment consists of four 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 use the logic language *QSOUL*, the successor of the logic language SOUL (Wuyts, 1998), to allow powerful logic reasoning about Smalltalk programs. QSOUL is implemented in Smalltalk and allows QSOUL clauses to reason about Smalltalk source code by allowing the execution of user-defined Smalltalk code as part of the logic reasoning process (Wuyts and Ducasse, 2001).

---

postfixes of a list, and so on.

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 are logic facts and rules which *reify* that construct at meta level. For example, we have a predicate class(?C) which states that ?C is a class that exists in the current Smalltalk image. (Below, we explain this predicate in more detail.)

Table 1 lists some of the predicates that constitute the representational mapping. Because our logic language is dynamically typed, in this table we use the following naming convention to indicate the types of the arguments to a predicate: a variable named ?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.

At this point, to avoid any confusion on the intended semantics of the predicates in Table 1, we stress that these predicates are ordinary Prolog-like predicates that can be used only to verify or search for information. For example, class(?C) can be used to retrieve all classes in the Smalltalk image or, when ?C is bound to a value, to check whether a certain class exists in the Smalltalk image. In Section 4, however, we will explain how we can still use these predicates as building blocks to define predicates for detecting violations of patterns, enforcing patterns or generating code from patterns.

Reification of Smalltalk language constructs at meta level is achieved by using QSOUL's symbiosis with Smalltalk. More specifically, QSOUL contains a prim-

itive construct, called "Smalltalk term", to access the Smalltalk image directly by executing a piece of Smalltalk code as part of a logic rule. "Smalltalk terms" are denoted by square brackets [. . .] that contain Smalltalk code. The actual semantics of these Smalltalk terms depends on the position where they occur: as a predication, as a logic term, or inside a generate predicate. The QSOUL rules and queries below [3], which reify the notion of Smalltalk classes, illustrate each of these possibilities. Rules that reify other Smalltalk language constructs are defined in a similar way; see Mens (2000) and Wuyts (2001) for more examples.

class(?C) **if**

    atom(?C),

    [*Smalltalk includesKey:* ?C *name*].

The above rule states what happens when the class predicate is called with a constant value. In that case, the special Smalltalk term [*Smalltalk includesKey:* ?C *name*] checks whether the value, bound to the logic variable ?C, indeed represents an existing class in the Smalltalk image. A Smalltalk term used in the position of a predication is required to return true or false. Also, all logic variables occurring in this Smalltalk term are required to be bound upon its execution, as they will be substituted by their corresponding Smalltalk value prior to evaluation of the Smalltalk expression.

class(?C) **if**

    var(?C),

    generate(?C, [*Smalltalk allClasses*]).

---

[3] In QSOUL, the keyword **if** separates the body from the head of a rule; queries are rules with an empty head; 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.

This second rule is applied when ?C is variable. In that case, a primitive generate predicate is used to unify that variable (the first argument of the predicate) one by one with each of the classes present in the Smalltalk image. This is done by executing the Smalltalk term which is provided as second argument to the generate predicate. This Smalltalk term is required to return a collection of results, each of which will be unified with the variable (one by one).

Given these rules, the query

if class([*Array*])

verifies whether *Array* is an existing class in the Smalltalk image, whereas the query

if class(?C)

subsequently unifies ?C with every class in the Smalltalk image. Note that a Smalltalk term used in the position of a logic term (as in the first query) can return any Smalltalk object. A returned Smalltalk object is automatically wrapped so that it is considered as a constant by the logic language. This kind of usage of Smalltalk terms enables QSOUL to reason about existing Smalltalk objects.

Another important part of the representational mapping is the representation of Smalltalk methods. To facilitate reasoning about methods, a method is represented as a logic data-structure that corresponds to the method's parse tree, rather than as a string containing the original Smalltalk source code. A method parse tree 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. For example, the following method of the Smalltalk class Number

```
odd
        "Answer whether the receiver is an odd number."
        ^self even == false
```

has as logic method parse tree

method([Number], [#odd],

    arguments(<>), temporaries(<>),

    statements(<return(send(

          send(variable([#self]),[#even],<>),

          [#==],

          <variable([#false]) >)) >))

To access the different parts of such a method parse tree, the representational mapping contains a set of predefined predicates: methodName, methodArguments, methodStatements and so on (see Table 1).

For more details on the symbiosis between QSOUL and Smalltalk and on the reification mechanism in particular, we refer to Wuyts (2001). In the next section, we show how best practice patterns, design patterns and other programming patterns can be encoded in QSOUL.

3   Codifying Programming Patterns

Every programming language has its set of patterns that experienced programmers follow to produce more understandable code (Beck, 1997) (Coplien, 1992). They use such patterns to make clear their intents and to improve the overall readability of the software. 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 (Beck, 1997). 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 in an object-oriented language 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 (Beck, 1997) 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:

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.

To check whether a method ?M of a class ?C is a Getting Method for some instance variable ?V, we verify that the class implements a method with the same name as the instance variable and that this method has the required form, as specified by the gettingMethodStats predicate.

gettingMethod(?C,?M,?V) **if**

    classImplementsMethodNamed(?C,?V,?M),

    instVar(?C,?V),

    gettingMethodStats(?Stats,?V),

    methodStatements(?M,?Stats).

Logic rules that codify the Setting Method pattern are very similar. See Wuyts (2001) for more details.

3.1.2   *The Constructor Method*

The Constructor Method best practice pattern indicates how you best express the creation of a class instance (Beck, 1997):

> **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:

constructorMethod(?C, ?M) **if**

   metaClass(?MC,?C),

   methodInProtocol(?MC, [#*'instance creation'*], ?M),

   returnType(?M, ?C).

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

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* (Gamma et al., 1995) have a more global scope and focus on typical class collaborations. As with best practice patterns, we codify the structure [4] of design patterns as logic meta pro-

---

[4] 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.

grams 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 1, 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 ?Element implements a method ?M called ?Accept. This method is responsible for calling a concrete visitor ?ConcreteVisitor with the actual visit operation ?VisitSelector. Finally we verify that one of the arguments of this call is the receiver (denoted by *self* in Smalltalk) and that the passed visitor ?ConcreteVisitor is an argument of the accept method:

visitor(?Visitor,?Element,?Accept,?VisitSelector) **if**

classImplements(?Visitor,?VisitSelector),

classImplementsMethodNamed(?Element,?Accept,?M),

methodStatements(?M,

     <return(send(?ConcreteVisitor,?VisitSelector,?VisitArgs))>),

member(variable([#*'self'*]),?VisitArgs),

methodArguments(?M,?AccArgs),

member(?ConcreteVisitor,?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* (Riel, 1996) and Beck and Fowler's *bad smells* (Fowler, 1999). As a typical example consider the following heuristic (Riel, 1996, 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:

abstractClassHeuristic() **if**

   forall(abstractClass(?C),baseClass(?C)),

   forall(baseClass(?C),abstractClass(?C)).

where baseClass(?C) checks whether ?C is a class from which another class inherits and abstractClass(?C) checks whether ?C 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

14

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* (Fowler, 1999):

> ### 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 (Riel, 1996), 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. To save space we only show the easiest case where two classes ?C1 and ?C2 are declared to have *common behavior* if they implement a method with the same method body.

commonBehavior(?C1,?C2,?M1,?M2) **if**
    method(?C1,?M1),
    method(?C2,?M2),
    methodStatements(?M1,?Stats),
    methodStatements(?M2,?Stats).

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

```
commonData(?C1,?C2,?V) if

    instVar(?C1,?V),

    instVar(?C2,?V),

    instVarType(?C1,?V,?Type),

    instVarType(?C2,?V,?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 explain how a programmer can use these rules to support him or her when developing or maintaining software. First of all, the rules can be used straightforwardly to *check* whether a certain pattern is satisfied or to *search* for source code that matches some pattern (4.1). But we can also use the same rules as building blocks for rules that support *detecting violations* of patterns (4.2) and even *code generation* (4.3). Finally, the rules can be used to *enforce* the consistent use of a certain pattern, but as we will see in Section 5, this is essentially a tool issue.

16

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,

if gettingMethod([*Point*],?M,[#'*x*'])

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

if gettingMethod([*Point*],?M,?V)

which finds all Getting Methods ?M together with their corresponding instance variable ?V for the class *Point*.

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*':

if gettingMethod(?C,?M,[#'*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 logic 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 third argument of the rule represents the common base class and the fourth and fifth argument are the methods to be moved.

behaviorRefactoring(?C1,?C2,?Base,?M1,?M2) **if**

    subClass(?Base,?C1),

    subClass(?Base,?C2),

    commonBehavior(?C1,?C2,?M1,?M2).

A similar rule can be made for dataRefactoring.

4.2   *Detecting Violations*

Detecting violations of patterns differs from checking or searching for patterns in the sense that we need to verify that a certain structure is *not* respected. Thus, detecting violations essentially comes down to checking the logic negation of the predicates defined in Section 3.

Getting Method   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

18

implemented in a class sends messages that have as receiver an instance variable of that class:

accessingViolator(?C,?M,?V,?Msg) **if**

    instVar(?C,?V),

    method(?C,?M),

    not(gettingMethod(?C,?M,?V)),

    isSendTo(?C,?M,variable(?V),?Msg).

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

if accessingViolator(?C,?M,?V,?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:

if   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 use a variable. The system will then deduce the name used in a specific

19

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 investigated classes and methods satisfy (the structure of) the Visitor design pattern.

4.3 Code Generation

To generate code that adheres to a given pattern, the approach is somewhat different. We need special generation predicates that allow us to generate code for Smalltalk language entities, like methods, based on a complete structural description of those entities. Of course, the necessary precautions should be taken that the entity being generated does not already exist.

Getting Method Instead of searching for Getting Methods and violations thereof, it can be useful to generate automatically the code of the Getting Method 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 low-level predicate generateMethod that uses of the strong symbiosis between QSOUL and Smalltalk to generate the source code of a method from its logic parse tree description. We repeat that a method parse tree 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.

generateAccessorCode(?C,?V) **if**

   instVar(?C,?V),

   "Verify that no method with name ?V exists"

   not(classImplements(?C,?V)),

20

"Construct the method body"

gettingMethodStats(?Stats,?V),

"Generate code from the parse tree description"

generateMethod(

      method(?C,?V,<>,<>,?Stats)).

Note that, to build the actual structural description of the method to be generated, we use the predicates of the representational mapping (Table 1) to fill in the different parts of the method parse tree, rather than merely using them for checking or searching the Smalltalk image. Again, the multi-way reasoning capabilities and the powerful unification mechanism of our logic language prove quite handy here. The rule ends with a generateMethod statement to actually generate the code for the method. Note that, when generating a method from its parse tree description, all parts have to be filled in. Due to space limitations, we will not show the detailed implementation of the generateMethod predicate; see Wuyts (2001) 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 (Fowler, 1999).

Pull Up Method

You have methods with identical results on subclasses.

Move them to the superclass.

Again 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

21

code; the mechanics of the refactoring corresponds to what is described by Fowler (1999).

pullUpMethodCode(?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,?N),

    methodStatements(?M1,?Stats),

    methodArguments(?M1,?Args),

    methodTemporaries(?M1,?Temps),

    "Verify that the common base class ?Base does

    not implement a method with the same name"

    not(classImplements(?Base,?N)),

    "Generate the new method from its parse tree description"

    generateMethod(

        method(?Base,?N,?Args,?Temps,?Stats)),

    "Delete the old methods"

    removeMethod(?M1),

    removeMethod(?M2).

In addition to the generateMethod predicate, this rule uses a special predicate removeMethod 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 QSOUL is well integrated in the Smalltalk development environment. It can reason about and manipulate Smalltalk objects directly and can even execute parameterized Smalltalk source-code fragments. Conversely, QSOUL queries can be executed from within Smalltalk itself. This symbiosis between QSOUL and Smalltalk is achieved by properly implementing QSOUL as a reflective inter- preter in Smalltalk and by using the powerful reflective capabilities of Smalltalk.

Wuyts (2001) extended QSOUL with a *synchronization framework* to build tools that rely on some kind of synchronization between design [5] and implementation. It enables the construction of tools that monitor and act upon any change to the implementation or design. For example, we can use this framework to make a tool for *enforcing* 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 would monitor all changes to methods and give an error or warning whenever a programmer accepts a method that accesses an in- stance variable directly instead of through a Getting Method. (Do note that exactly the same predicate is used as in 3.1.1 to check for 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

---

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

23

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 evaluate the corresponding query for the user. For example, the Find Application may be used to find all abstract classes that have a name matching some pattern, have a method sending some specified message and implement a getting 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 QSOUL is the 'To Do Application'. During software development or maintenance it logs all violations of certain programming patterns, conventions and heuristics in a "to do" list. This continuously updated list can be inspected at al time by the software developer to fix (or ignore) the detected problems.

A third tool (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 certain 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 QSOUL syntax and all predefined predicates can be used to construct a search query.

Also, a programmer can always add to the logic repository his or her 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 or her 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* enables finding 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 (Mens, 2000).

References

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

J. O. Coplien. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992. Reprinted with corrections february 1994.

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

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, 1996.

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

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

A. J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.

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.

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.

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, 2001.

R. Wuyts and S. Ducasse. Symbiotic reflection between an object-oriented and a logic programming language. In *Multiparadigm Programming with Object-Oriented Languages*, volume 7, pages 81–96. John von Neumann Institute for Computing, 2001.

Vitae

Kim Mens obtained the degrees of Licentiate in Mathematics, Licentiate in Computer Science and Doctor in Computer Science at the Vrije Universiteit Brussel. He started as a teaching assistant in mathematics, before becoming a teaching and research assistant in computer science at the same university. In October 2000, he

obtained his PhD while being assigned on a research project funded by Getronics and the Belgian government. The subject of his dissertation was *architectural conformance checking*, for which he used a declarative meta programming approach. After his PhD he became a post-doctoral assistent at the VUB, before starting as a computer science professor at the Université Catholique de Louvain-la-Neuve in September 2001. In addition to his current interest in *declarative meta programming* (he is part of the QSOUL development team), he is one of the founding fathers of the *reuse contract* technique for automatically detecting conflicts in evolving software. He also has a strong interest in *aspect-oriented programming* and actively participated in the organization of several workshops and a conference on that subject.

Isabel Michiels is a PhD student and teaching assistant at the Programming Technology Lab of the Vrije Universiteit Brussel (VUB), Belgium. At this university she obtained the degree of Licentiate in Computer Science. In September 1999, she successfully ended a Master of Science in Object-Oriented Software Engineering Technologies (EMOOSE), a special Master program organized by the VUB and the Ecole des Mines de Nantes, France. Her Masters thesis, carried out at Lifia, Universidad Nacional de La Plata, Argentina dealt with teaching object technology with intelligent environments. Her current research interests involve software architectures and architectural description languages. She is part of the QSOUL development team. She also organized several workshops on Teaching Object Oriented Concepts.

Roel Wuyts obtained a doctoral degree at the Programming Technology Lab of the Vrije Universiteit Brussel. In his dissertation, he investigated a framework to support the co-evolution of design and implementation. He developed the SOUL language to validate his claim that logic meta programming can relate design and im-

plementation in such a way that the one can be used to check, generate or constrain the other. Currently he is working at the Software Composition Group in Switzerland, following two main research tracks (software re-engineering and component models for embedded devices). In both tracks QSOUL, the successor of the SOUL language, is used as one of the experimentation vehicles (i.e., to extract information, and as component composition language respectively).
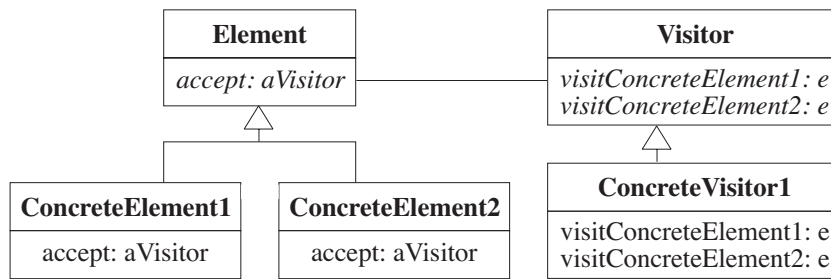
Fig. 1. Visitor Design Pattern Structure

Predicate	Meaning
class(?C)	?C is a class
classImplements(?C,?N)	class ?C implements method named ?N
classImplementsMethodNamed(?C,?N,?M)	class ?C implements method ?M
	with name ?N
method(?C,?M)	method ?M belongs to class ?C
methodArguments(?M,?Args)	method ?M has argument list ?Args
methodName(?M,?N)	method ?M has name ?N
methodStatements(?M,?Stats)	method ?M has statement list ?Stats
instVar(?C,?V)	class ?C has instance variable named ?V
isSendTo(?C1,?N,?R,?Args)	in class ?C1 there is a message send ?N
	with argument list ?Args to receiver ?R
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

Table 1

The representational mapping