Codifying Structural Regularities of Object-Oriented Programs

Roel Wuyts[1] and Kim Mens[2]

[1] Software Composition Group
University of Bern, Switzerland
roel.wuyts@iam.unibe.ch
[2] Département d'Ingénierie Informatique
Université Catholique de Louvain-la-Neuve, Belgium
kim.mens@info.ucl.ac.be

Abstract. Well-written object-oriented programs exhibit many structural regularities ranging from naming and coding conventions, through design patterns, to architectural constraints. Tools and environments that aid a software developer in constructing, understanding or modifying object-oriented programs should be able to reason about and manipulate such regularities. We codify structural regularities of object-oriented programs as predicates in a logic meta language on top of Smalltalk. These predicates can be used not only to enforce regularities in the software, but also to search the source code, detect violations against certain regularities and to generate code fragments that exhibit a certain regularity. On the basis of two case studies we illustrate how the use of such predicates supports a developer in various activities throughout the software development life-cycle.

1 Introduction

As argued by Minsky [16], object-oriented systems exhibit many *regularities* that improve the comprehensibility and manageability of the software. This is especially important for large systems. Unfortunately, due to their intrinsic global nature, it is inherently hard to make these regularities explicit in the software. To capture these software regularities formally, Minsky explicitly describes them as 'laws of the system' in a declarative programming language that is integrated with the development environment. As such, the declared laws can be enforced by this environment easily.

Minsky's laws essentially regulate interactions among *objects*. They are ideal to declare dynamic constraints, such as interaction protocols and business rules that constrain the *message sending* behaviour between objects. Depending on the kinds of regularities expressed they can be enforced by the environment dynamically (at run-time) or statically (at compile-time).

Unfortunately, Minsky's approach is somewhat less suited when it comes down to handling more *structural* regularities. Such regularities are used by experienced programmers (and object-oriented programmers in particular) to structure their programs. The most common structural regularities are recorded in literature under the form of a wide variety of 'patterns'. Well-known kinds are best practice patterns [1], design patterns [9], design heuristics [18], bad smells and refactoring patterns [8]. An advanced software development environment should be able to reason about and manipulate these and other kinds of structural regularities in the program code.

Moreover, it does not suffice that the environment merely supports 'enforcement' of structural regularities. As we already discussed in an earlier paper [13], an advanced software development environment — which aids a software developer in comprehending, building and maintaining object-oriented programs — also requires support for:

- *checking* whether (a part of) the source code exhibits a certain structural regularity;
- *finding* all code fragments that exhibit a certain structural regularity;
- *detecting* those parts of the source code where a certain regularity is *not* followed;
- *enforcing* structural regularities;
- *generating code* (templates) that exhibit some structural regularity;
- *transforming code* based on structural regularities.

Minsky's 'law-governed systems' do not support all these activities. His main goal is to implement regularities reliably, in other words, to have them "*enforced* by some kind of higher authority" [16].

An environment that supports the above kind of activities requires the ability to reason about and manipulate source code at a sufficiently fine-grained level. An approach that can only reason at run-time about objects and message sends is clearly not sufficient. For example, to codify the structure of a design pattern we need to describe the structure of the different classes and methods that play a role in this pattern. For this, we need to be able to reason about the full parse trees of those methods. (In [13, 22], we show how to codify the Visitor design pattern and [21] explains the Composite pattern.)

To achieve these goals we codify *structural regularities* of object-oriented programs as predicates in a declarative meta language on top of the object-oriented language under consideration, in our case: Smalltalk. Our approach shows many similarities with Minsky's, but emphasizes structural regularities rather than more dynamic ones. Another important difference lies in the way that our logic language is integrated with and reasons about the underlying Smalltalk system. Instead of having the environment enforce the regularities automatically, we deliberately adopt a more liberal approach. The logic predicates are offered as a tool to the software developer who can use them for a variety of different tasks ranging from navigating and searching the source code, through verification of certain regularities, to generating source code fragments.

The goal of this paper is *not* to explain our *declarative meta programming* approach in detail, nor the logic meta language *SOUL* we use for this purpose. Instead we present two case studies that illustrate how the approach can support a software developer in his or her every-day software engineering activities of program understanding, program construction and program maintenance.

The remainder of this paper is structured as follows. Section 2 sketches the context by introducing the two case studies we conducted to validate our declarative meta programming approach. Section 3 explains our declarative meta programming environment and illustrates how to codify some structural regularities. Section 4 and section 5 discuss the case studies. From these case studies, section 6 draws some overall conclusions regarding the practical usability of our approach. Related work is discussed in section 7 and section 8 concludes the paper.

2 Introducing the Case Studies

Our declarative meta programming approach has been the subject of earlier works [13, 21, 22]. We summarize the important details of the approach in section 3. The main purpose of this paper, however, is to show the practical usability and scalability of the approach and how it can be used to codify structural regularities of large object-oriented programs. To this extent, we conducted two case studies. The general setup of these case studies was rather informal: we just used our approach to help us with some real software engineering activities with which we were faced.

The first case study was performed on *HotDraw* [2, 4, 5, 10], a well-known object-oriented application framework written in Smalltalk. HotDraw can be used to build structured drawing editors for designing specialized drawings such as schematic diagrams, blueprints or program designs. The elements of these drawings can have constraints among them, they can react to user commands and they can be animated.

We used HotDraw to build a graphical editor for constraint networks. This involved a series of activities. First of all, we had to *understand* the framework and learn how to build applications with it. While doing so, we explicitly *documented* the discovered structural regularities of the framework by means of logic predicates. Not only is this information important to future framework users, the predicates also proved beneficial to us in a later phase when we wanted to verify whether the constructed application satisfied the demands of our framework. Next, we actually customized the framework to build our graphical editor. Finally, we performed some *checks* to verify whether the application conformed to the structural regularities of the framework, and modified the application where necessary. In section 4, we report in detail on this case study.

A larger-scale industrial case study was performed at *MediaGeniX*, a company that produces customized software for television broadcast stations. Due to space limitations, we do not elaborate on this case study. Section 5 present the main results only. For more details on either of these case studies we refer to [22].

3 Declarative Meta Programming

We now briefly explain the *SOUL* language and environment [21] we used to conduct our case studies. We only discuss those aspects of the language that are relevant to this paper and present the implementation of some predicates that were used in our case studies. For more examples and details on how to use SOUL, see [21, 13, 22].

Essentially, SOUL is a *Prolog*-like logic programming language that is implemented in *Smalltalk*[1]. Although the SOUL syntax closely resembles Prolog syntax, there are some notable differences. The keywords **Rule**, **Fact** and **Query** denote logical rules, facts and queries respectively. Instead of using :- to separate the head from the body of a rule, the keyword **if** is used. Logic variables start with a question mark instead of with a capital letter. Lists are delimited by <> instead of by square brackets. Other syntactic constructs remain the same as in Prolog, e.g., a comma still denotes logical conjunction.

But SOUL is more than merely a logic programming language: it also has a tight symbiosis with the underlying Smalltalk language and environment. In fact, SOUL is a meta language as it allows SOUL clauses to query the Smalltalk source code by making *direct* meta calls to the Smalltalk image. There is no need of first representing the source code explicitly as facts in a logic repository. This is accomplished by introducing a special logic term called "*Smalltalk term*" and a primitive predicate called "*generate predicate*" for evaluating Smalltalk expressions as part of logic rules. The following two subsections describe these special constructs. Syntactically, *Smalltalk terms* will be represented in SOUL as Smalltalk expressions (that may reference logic variables that are bound in the surrounding scope) between square brackets.

3.1 Smalltalk Term

The language construct in SOUL that enables the symbiosis with Smalltalk is the *Smalltalk term*. It allows us to use Smalltalk objects as logic constants. So whereas Prolog only allows symbols, strings and numbers as constants, SOUL can use any Smalltalk object. Consider for example the following query:

```
Query collection([Array])
```

The argument of collection is a *Smalltalk term*. Between the square brackets we find the description of the term, in this example Array. Array is a Smalltalk constant, namely the object that represents the class Array. Of course, this example is very simple. In fact, the expression between square brackets can be any Smalltalk expression. The semantics of a *Smalltalk term* is that the enclosed Smalltalk expression is evaluated when the query or predicate in which it occurs is interpreted. The returned Smalltalk value is then considered as value of the *Smalltalk term*.

To enhance the symbiosis between SOUL and Smalltalk even further, the expressions in *Smalltalk terms* may contain logic variables. We illustrate this with a query that first binds a logic variable ?C to a Smalltalk value Array (by unifying it with the *Smalltalk term* [Array]), and then uses another *Smalltalk term* containing a Smalltalk code fragment that evaluates to a Boolean value.

```
Query    equals(?C, [Array]),
         [?C selectors isEmpty]
```

The Smalltalk expression ?C selectors isEmpty is parametrised by the logic variable ?C. It first sends the parameterless message selectors to the Smalltalk value bound to the logic variable ?C. Since ?C is bound to Array this message send returns a Smalltalk collection that contains all the names of methods in the Array class. This collection object is then sent the message isEmpty and returns true when the collection is empty or false otherwise. In other words, interpreting this query will succeed if the class *Array* has methods, and will fail otherwise.

Note that, although semantically they are interpreted in exactly the same way, a *Smalltalk term* used in the position of a predication (i.e., used as a 'subquery', as is the case for the *Smalltalk term* [?C selectors isEmpty] above) is always required to return a Boolean value.[2] A *Smalltalk term* used in the position of a

[1] Links to the download location can be found at http://prog.vub.ac.be/poolresearch/dmp/, under the *artefact* section.

[2] In its current implementation, the SOUL interpreter will simply abort with an error when no Boolean value is returned.

logic term (as is the case for the *Smalltalk term* [Array] above) does not have this restriction and may return any value.

We stress that for the above query to succeed, we did not have to declare any logic facts expressing implementation details about Array. This information was retrieved transparently by reflectively accessing the Smalltalk system.

3.2 Generate Predicate

Smalltalk terms support a symbiosis between the logic code and the Smalltalk environment, by allowing us to evaluate Smalltalk expressions during logic interpretation, and wrapping the resulting Smalltalk object so that it can be used as a constant in the logic language. One interesting characteristic of logic languages, however, is that one query might produce many different results. Suppose, for example that we want to write a predicate simpleSelector(?C,?S) that expresses the relationship between a class ?C and the names ?S of the methods (i.e., the selectors) that it implements. Calling this predicate with an unbound variable ?S would yield multiple answers: one for each valid method name. In first approximation, we could implement this predicate as follows:

```
Rule simpleSelector(?C, ?S) if
        class(?C),
        equals(?Selectors, [?C selectors]),
        collectionToLogicList(?Selectors, ?SelectorsList),
        member(?S, ? SelectorsList)
```

The reason we need to use the member predicate above is because the *Smalltalk term* [?C selectors] returns a collection of selectors, whereas we need them one by one. Furthermore, because member only works on logic lists, we first have to transform the collection of selectors bound to ?Selectors into a logic list ?SelectorsList. Since this situation — i.e., a *Smalltalk term* returns a collection but in the logic language we need the elements of the collection one by one — occurs so frequently, the SOUL language offers a primitive predicate called the "*generate predicate*" which does precisely this. Using this generate predicate, the predicate above would become:

```
Rule simpleSelector(?C, ?S) if
        class(?C),
        generate(?S, [?C selectors])
```

The semantics of generate is straightforward. The second argument is a *Smalltalk term* that is required to return a *collection of solutions*[3]. The first argument of the *generate predicate* specifies the logic variable to bind the results to. The generate predicate will subsequently bind, one by one, each of the elements of the collection to the variable. When the generate predicate is evaluated, it results in *n* solutions for the variable, where *n* is the number of elements in the collection. For example, the query

```
Query simpleSelector([Array], ?S)
```

yields 15 results. Each result is a different binding of the variable ?S to the name of a method of the class Array.

3.3 Declarative Framework

The *SOUL* system comes with a *declarative framework*, i.e., a layered library of rules for reasoning about the structure of Smalltalk programs. Although the current version of the declarative framework does not enforce the layering, the idea is that rules in one layer only rely on rules in lower layers. Each layer groups rules with a similar or related functionality:

- The *logic layer* contains rules that address core logic programming functionality, such as *list handling*, *arithmetic*, *program control* and *repository handling*.
- The *representational layer* reifies concepts of the underlying object-oriented language: *classes*, *methods*, *instance variables* and *inheritance*.

[3] The SOUL interpreter will abort with an error when no collection is returned.

– On top of the representational layer the *basic layer* defines a range of auxiliary predicates to facilitate reasoning about the implementation structure. As predicates in the representational layer are very primitive, the basic layer is needed to interact at a reasonable level of abstraction with the logic meta programming language.
– Various higher-level layers are defined on top of these layers. For example a 'design layer' which groups all rules that express particular design patterns and heuristics, or an 'architecture layer' which defines rules for reasoning about the architectural structure of a program.

Since it is not focus of this paper we do not go into the details of these layers. The important thing to remember is that a structured library of predefined predicates is available and that the library is completely open so that users can add their own predicates or modify existing predicates to their specific needs.

3.4 Codifying Structural Regularities

To illustrate the SOUL language we show how to codify some interesting structural regularities, needed for the case studies, under the form of logic predicates. We also explain the different ways in which these predicates can be used.

Accessor methods Accessor methods are parameterless methods that get the value of an instance variable of a class. Mutator methods set the value of an instance variable. By consequently using accessor and mutator methods throughout an implementation, every access to the state of an object is hidden by a message send. As such, the distinction between state and behaviour is made more transparent [1].

Let us now codify the structure of accessor methods. Accessor methods can be implemented in several ways. The first one is the direct implementation scheme (see the Smalltalk code below) that merely returns the instance variable:

```
var
    ↑ var
```

Another implementation scheme uses *lazy initialization*, the rationale being that an instance variable does not need a value unless it is actually used. Therefore lazy initialization is built into the accessor method. The accessor method first verifies whether the variable was already initialized or not (by checking whether its value is *nil*). If the variable was not yet initialized (its value is nil) then it is initialized and returned, otherwise the value is returned. The Smalltalk code representing this scheme is given below[4]:

```
var
    ↑var isNil
        ifTrue: [var := some initial value]
        ifFalse: [var]
```

The first 'simple' implementation scheme is codified by a rule which describes that the method's body just consists of a return statement that returns an instance variable:

```
Rule accessorForm(?Method, ?Var, simple) if
    methodStatements(?Method,<return(variable(?Var))>)
```

The rule that codifies the second 'lazy' implementation scheme is:

```
Rule accessorForm(?Method, ?Var, lazy) if
    methodStatements(?Method,
        <return(send(   ?NilCheck,
                        [#ifTrue:ifFalse: ],
                        <?TrueBlock,?FalseBlock>))>),
    nilCheckStatement(?NilCheck,?Var),
    blockStatements(?TrueBlock,<assign(?Var,?VarInit)>),
    blockStatements(?FalseBlock,<?Var>)
```

[4] For an alternative implementation of this scheme, see [22].

Finally, we write a predicate which codifies the Smalltalk *naming convention* that the accessor methods of a class typically have the name of the instance variable they are accessing:

Rule accessor(?Class,?Method,?VarName) **if**
instVar(?Class,?VarName),
classImplementsMethodNamed(?Class,?VarName,?Method),
accessorForm(?Method,?VarName,_)

Now that we have codified this structural regularity, we can use it directly to find all accessor methods in an implementation. Or we can use a slightly more complex query to check whether all instance variables actually have a corresponding accessor method:

Query forall(instVar(?Class,?VarName), accessor(?Class,?Method,?VarName))

Or we can use it in a rule that produces all violations of this regularity, i.e., it searches all non-accessor methods that directly access an instance variable (see [13, 22]).

Composite design pattern Next, we give an example of a predicate codifying the structure of the Composite *design pattern* [9]. To save space we do not show its implementation but explain how it can be used only. The predicate compositePattern relates a class ?Composite to a class ?Component and can be used in 4 different ways.

- When we pass two actual classes, the predicate returns wether the relationship holds for these two classes. For example, compositePattern([Figure], [CompositeFigure]) can be used to check whether the classes Figure and CompositeFigure are in a composite pattern relationship.
- When we only pass the component class (for example Figure), then we can infer all classes that play the role of *composite class*: compositePattern([Figure], ?Composite).
- We can also pass the composite class and infer all component classes for that composite class: compositePattern(?Component, [CompositeFigure]).
- When we pass no information (but only variables), all possible combinations of components and composite classes as described by the relation are found:
compositePattern(?Component, ?Composite).

Other design pattern structures that exist as predefined predicates in the declarative framework are the Visitor (also see [13, 22]), Singleton, Bridge and Factory Method design patterns. Each of the corresponding predicates can be used in a variety of ways, depending on whether some or all of their arguments are instantiated or not.

UML class diagrams As *UML class diagrams* [20] are often used to describe the structure of object-oriented programs it would be useful to have some predicates that help us in reasoning about and manipulating such diagrams. The declarative framework contains some predefined predicates to express the basic concepts of UML class diagrams: *classifiers* (with *operations* and *attributes*) and the *generalization* and *association* relationships. We chose to map classifiers to classes, operations to methods, attributes to instance variables and generalization to inheritance.[5] Mapping the association relationship is not as simple. Because Smalltalk is dynamically typed, this mapping heavily uses some predicates for inferring the types of instance variables of Smalltalk classes. It is outside the scope of this paper to discuss the details of these predicates here; see [22].

Most UML predicates can be used in a variety of ways. One popular usage is to *extract* class diagrams from Smalltalk source code. In fact, a predefined predicate assertUMLDiagram(?Class,?Repository) exists to generate a UML diagram for the class hierarchy with ?Class at its root. This diagram contains all classifiers and generalization and association relationships relevant for that class hierarchy. The generated information is stored in the form of facts in a logic repository ?Repository and is also rendered as a drawing on screen. In addition to this generative predicate the declarative framework also contains predicates for *checking conformance* of source code to a diagram to find out where and how they differ.

[5] Other mappings are possible, but this one-to-one mapping is often used in real-world development and in mainstream modelling tools such as Rational Rose or TogetherJ.

Generating accessor methods In addition to searching for accessor methods and checking violations of the convention that instance variables can be read only by accessor methods, sometimes we want to *generate* accessor methods for instance variables of a class. This can be done rather easily by combining the rule defining the 'simple' accessorForm with a primitive predicate cpgMethod for generating the code of a method[6]:

Rule generateAccessor(?Class, ?VarName, ?AccessorMethod) **if**
instvar(?Class, ?VarName),
accessorForm(?AccessorMethod, ?VarName, simple),
methodClass(?AccessorMethod, ?Class),
methodName(?AccessorMethod, ?VarName),
cpgMethod(?AccessorMethod)

4 HotDraw Case

Let us now turn our attention to the *HotDraw* case study announced in section 2. We addressed four activities:

1. *Understand the framework* and how to build applications with it.
2. *Document the framework* by explicitly codifying its structural regularities and the regularities it imposes on applications built with it.
3. *Customize the framework* into a graphical editor for constraint networks.
4. *Check conformance* of the constructed application to the structural regularities that were codified earlier on.

Obviously, these four activities were partially overlapping and not entirely sequential. For example, understanding and documenting the framework occurred pretty much in parallel. Also, when checking conformance, we encountered some inconsistencies in the customized application that had to be corrected. Nevertheless, to structure the discussion, in the next subsections we discuss each activity as if had occurred separately.

4.1 Understand framework

We were not familiar with HotDraw when we started experimenting with it. Hence, we used our declarative meta-programming approach to better understand the implementation of the framework. This was a kind of reverse engineering activity, where we used existing documentation of HotDraw as blueprints of queries that were checked against the source code. In other words, this activity was primarily concerned with *extracting structural information* from the framework implementation.

Being novices at the implementation of the HotDraw framework, we started by reading the documentation and papers mentioned on the website[7]. We soon came to the conclusion that most documents described an outdated version of HotDraw. However, they provided a sufficient overview of the implementation to get us started. One of the first queries we ran extracted the *root classes* of HotDraw by getting a list of all classes in the HotDraw package (hotdrawClasses), and removing every class that was a subclass of another class in that list (stripHierarchyClasses):

Query hotdrawClasses(?ClassList),
stripHierarchyClasses(?ClassList, ?Roots)

This resulted in 18 root classes. Based on the names of these classes and on a closer investigation of the HotDraw implementation, we discovered three classes that were of general interest: DrawingEditor, Tool and Figure. The Figure hierarchy proved very extensive, so in order to familiarize ourselves with it we extracted a simple UML class diagram (shown in Figure 1), using the predefined predicate assertUMLDiagram provided by the declarative framework:

Query initializeDrawing(),
assertUMLDiagram([Figure], [HotDrawDesign])

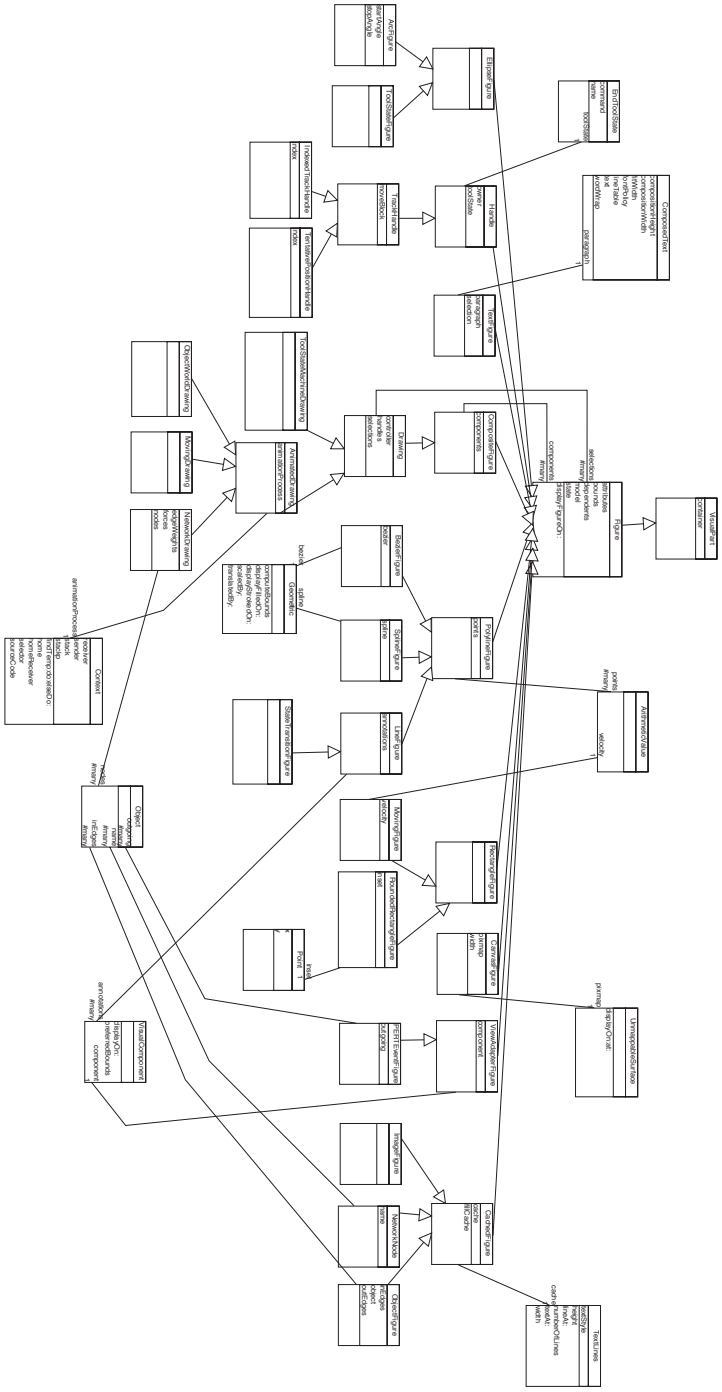

Fig. 1. Extracted UML class diagram for class Figure.

In the above query, initializeDrawing() just performs some initialization before a drawing can be generated. The second part assertUMLDiagram([Figure], [HotDrawDesign]) actually draws the extracted UML diagram and stores its description in a logic repository HotDrawDesign[8] which contains a set of facts describing the design structure of HotDraw.

In addition to extracting this UML information, we also ran queries to check what 'accessor methods' were used where, and whether we could detect occurrences of certain design patterns. The results are summarized in table 1. For those queries that returned many results we only show the number of results found.

Structural regularity	**Discovered occurrences**
Composite pattern	compositePattern([Figure],[LineFigure]) compositePattern([Figure], [Drawing]) compositePattern([Figure], [CompositeFigure])
Visitor pattern	none
Singleton pattern	none
Bridge pattern	none
Factory Method pattern	base level: 41, meta level: 3
Accessor methods	base level: 56, meta level: 1

Table 1. *Extracting structural regularities from HotDraw.*

This subsection illustrates how a lot of interesting structural information can be extracted from the implementation, helping a developer unfamiliar with the implementation to gain a better insight in it. This does *not* eliminate the need to read the documentation nor to browse the source code manually to study certain parts of the implementation. It merely complements it by providing advanced tools that allow browsing the system on different levels of abstraction. In the next subsection we see how we refined the extracted information into a more specific framework documentation.

4.2 Document framework

In addition to understanding the HotDraw framework and how it can be customized, we wanted to codify part of these insights explicitly in the form of logic predicates. During this activity we played the role of *framework developer*, documenting the framework with logic meta programs in an effort to keep the framework documentation and the framework implementation tied closely together. In particular we focussed on the hierarchy of composite figures and on the relationships between the *editor* (the actual application), the *tools* (that create and manipulate figures) and the *figures* themselves. We complemented the structural regularities discovered in the previous activity with more specific HotDraw design information regarding editors and composite figures.

Figure 2 depicts a standard HotDraw editor with some default figures. The toolbar of the editor window displays a series of buttons to draw, select and delete figures. Not shown is the context sensitive menu associated with each figure to set properties such as fill colour and line width. The implementation of an editor uses three main classes: DrawingEditor, Figure and Tool, amongst which we found some essential, yet undocumented structural regularities. All these regularities are more or less implicit in the HotDraw source code. They are implemented using naming conventions, hardcoded references and Smalltalk meta-programming techniques. As they are undocumented and scattered throughout the source code, we only discovered these regularities by using a combination of standard Smalltalk browsers and SOUL queries. To make them more explicit we decided to codify them as logic predicates. Below we discuss some of the more interesting structural regularities. Note that the predicates are prefixed with *hd*, an abreviation for *HotDraw*.

[6] The acronym 'cpg' means 'check and perhaps generate' because the predicate will not generate anything if the class to be generated already exists.

[7] http://st-www.cs.uiuc.edu/ brant/HotDraw/

[8] As we implemented SOUL entirely in Smalltalk, logic repositories are represented by classes.

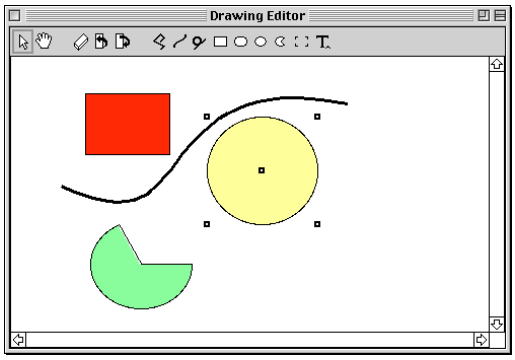

Fig. 2. Screenshot of the standard HotDraw editor.

hdToolNamesMethod A first structural regularity is that an editor should override the method *toolNames* to describe all tools it uses. There are two straightforward implementation schemes to implement this method: one that simply enumerates all tools and one that gets a tool list from the superclass and adapts it for adding extra tools.

The predicate hdToolNamesMethod codifies the general structure of the toolNames method for a certain editor. It uses the hdEditorClass predicate to verify that ?Editor is indeed an editor class. It then selects the method ?M named toolNames in this class and verifies whether it has the correct format described above. (The actual format of the method body is defined by two auxiliary rules, named hdToolNamesStatements, one for each implementation scheme.)

Rule hdToolNamesMethod(?Editor, ?M, ?ToolNamesList, ?Kind) **if**
hdEditorClass(_, ?Editor),
classImplementsMethodNames(?Editor, toolNames, ?M),
methodArguments(?M, <>),
methodTemporaries(?M, <>),
hdToolNamesStatements(?Statements, ?ToolNamesArray, ?Kind),
array2List(?ToolNamesArray, ?ToolNamesList)

We can use hdToolNamesMethod to extract the names of all tools for a certain editor, to find all editors using a particular tool, to check whether a particular editor uses a particular tool, and so on. We cannot use it to generate the code for such a method (given a list of tool names). It is not difficult to implement a predicate cpgToolNamesMethod(?Editor,?ToolNamesList) that generates an appropriate toolNames method when it does not exist yet, or that rewrites an existing toolNames method when it does exist but lacks certain tools we would like to offer in the editor. We refer to [22] for the implementation of this predicate.

hdIconMethod Another structural regularity of HotDraw is that every tool offered by the editor should have a corresponding toolbar icon. In other words, every tool name enumerated by toolNames should have a corresponding class method that returns its icon. A predicate hdIconMethod codifies this structural regularity. We also defined a predicate cpgIconMethod that generates the necessary method to provide an icon in the case where it is absent.

hdToolMethod The Tool class uses a state machine to implement the interaction between the user and the editor. Each tool offered by the editor has to be added to this state machine. When the user selects a tool in the buttonbar, and then clicks on the drawing area, the figure defined by the tool can be created.

To build the state machine defined by class Tool, all methods from the method protocol *tool states* of Tool's metaclass are enumerated. Every method in this protocol adds certain states and transitions that are used by Tool whenever the mouse is moved or clicked within the HotDraw editor. Therefore, every toolname listed in the toolNames method should occur in at least one method in the *tool states* method protocol. Also important is that these methods can be used to associate tool names and figures, since the transitions are responsible for creating figures. We have codified this knowledge in a predicate hdToolMethod (see below)

and again provided a predicate cpgToolMethod to generate a default initialization method adding states to create and select the figure.

```
Rule hdToolMethod(?ToolName, ?Figure, ?InitMethod) if
        methodInProtocol([Tool class],[#'tool states'],?InitMethod),
        classesUsed(?InitMethod, ?ClassList),
        member(?Figure, ?ClassList),
        hierarchy([Figure], ?Figure),
        isSendTo( ?InitMethod,
                  send(variable([#Tool]),[#states],<>),
                  [#at:put:],
                  <literal(?ToolName), _>),
        patternMatch(?ToolName, postfix(['Tool']))
```

The hdToolMethod predicate relates three variables: the tool used, the figure and the tool where the states are added. In other words, it describes how several classes from completely different hierarchies interact. This is a major difference with some of the other predicates that only seem to express local properties of methods or classes. While this predicate uses similar techniques, it does describe a more global interaction among classes in the HotDraw framework.

When we actually used hdToolMethod to find out which figures are initialized by which methods, we discovered that there exist HotDraw figures that are not created by tools in editors. When looking more closely at the roles they play in the application, we found that some of those are example figures. But some other figures (such as LineFigure, TrackHandle, IndexedTrackHandle, TentativePositionHandle or CompositeFigure) are never created with tools. They are instead created by manipulating figures in the drawing area (such as grouping which creates a CompositeFigure or selecting figures which creates handles).

This example illustrates that, by explicitly codifying structural regularities and using them to check conformance of the source code, we can make some *quality assessments*. In this case it shows places where refactoring of code is probably needed, or where more explanations are needed on why these classes form exceptions and are not created by tools.

Predicates dealing with composite figures In addition to the structural regularities that deal with the interplay among the Tool class, the editor and the figures, we also codified some structural regularities regarding composite figures. For example, there exists a predicate cpgCompositeFigure to generate the code of a composite figure class.

4.3 Customize framework

In the third activity, we played the role of a *framework user* who tries to use the framework, as well as the documentation that describes it, to customize it into an actual application. The previous activities already codified documentation under the form of structural regularities that constrain the possible customizations of the HotDraw framework. We now explain how this information can help in guiding the actual customization of the framework into an application.

First of all, we can use it to *detect* parts of the implementation that still need to be customized in order to obtain an actual application. This comes down to a straightforward use of the predicates that codify the structural regularities to detect all violations of those regularities in the implementation. For example, the structural regularity may be that each class of a certain kind must have a certain method. All classes that do *not* have such a method clearly need to be customized further.

Secondly, we can automatically *generate code* for those violations. For example, whenever the structural regularities declare that some class or method should exist, but it is not found in the implementation, it is generated. (Depending on the situation the generated implementation may only be a code template, in the sense that we will probably have to add some extra detail to it later.)

For example, we provide a new predicate cpgEditorClass that generates code for a HotDraw editor class when needed:

```
Rule cpgEditorClass(?Name, ?Editor) if
        cpgClass(?Name, [DrawingEditor]),
        className(?Editor, ?Name)
```

This predicate is defined in terms of the auxiliary cpgClass predicate. This first checks whether a class named ?Name already exists. If not, it generates such a class as a direct subclass of *DrawingEditor*. If an editor class named ?Name *did* already exists in the hierarchy of DrawingEditor, nothing is generated and the predicate succeeds. If a class named ?Name did exist, but not in the hierarchy of DrawingEditor, the predicate fails. In case of success, as a result of calling the cpgEditorClass predicate, the actual editor class will be bound to the variable ?Editor.

Likewise, we have implemented other predicates to generate code according to the structural regularities codified in the previous sections, such as the cpgToolNamesMethod, cpgIconMethod or cpgCompositeFigure mentioned before. We use these predicates to generate customizations of the framework that can then be completed by hand. For example, reconsider our customization of the HotDraw framework to implement an editor for drawing constraint networks. In a constraint network we have a notion of constraint variables that are depicted by ellipses with the variable name inside them. So, in our customization of the framework we represent a constraint variable as a composite figure called ConstraintVariableFigure that has two component figures, varName and ellipse. We also provide an editor, called ConstraintEditor, together with a tool that can draw ConstraintVariableFigures. To relate this application-specific structural information to the framework-specific information, we write the following query:

```
Query cpgCompositeFigure(   [#ConstraintVariableFigure], <[#varName], [#ellipse]>),
      cpgEditor(    [#ConstraintEditor], ?Editor,
                    <['ConstraintVariableFigure Tool'] >,
                    <['ConstraintVariableFigure'] >)
```

Note that in this query the predicates contain the framework-specific information and the arguments contain the application-specific information. The predicates were already explained above. In the case of this particular query, neither the composite figure class, nor the editor class existed yet. Hence, they were completely generated, including their methods and the appropriate methods on the class Tool. Since no types were specified for the component figures, simple TextFigures containing their name were generated. This is the default behaviour, which we obviously had to change manually afterwards to get the desired behaviour. But still, the net result of this query was that the knowledge contained in the structural regularities was used to generate code for those parts of the implementation mentioned in the query.

4.4 Check conformance

In the previous subsection we explained how the framework could be customized into an application using a combination of declaring application-specific information, automatic code generation and manual code editing. Once the application has been constructed, it is still useful to do some conformance checks to verify that we did not forget to customize important parts of the framework, or that some structural regularities were invalidated due to manual changes. In contrast with the previous activity where a default action is provided that generates parts of the implementation, a consistency check results in a report that shows where the (modified) application is inconsistent with the structural regularities imposed by the framework.

Such a conformance check typically consists of two parts that can be combined. The first is to check whether the structural regularities still hold for the implementation. The second is to check whether the new implementation yields new or changed regularities.

Checking whether the structural regularities are respected throughout the implementation is straightforward. Since all regularities are codified as logic clauses (i.e., facts or rules), we just have to launch each of them as queries to verify whether the implementation still satisfies them. This holds for the rules describing framework-specific structural regularities (such as those described in subsection 4.2), as well as for the facts describing the framework structure (see subsection 4.1). We are interested in all queries that return false, since they correspond to the regularities that were invalidated. This is illustrated in the following query, which first retrieves all clauses from our logic repository (using the repositoryClause predicate) and then checks all of them against the current implementation (by calling them as queries using the call predicate). The failed ones are enumerated in a list, which is the result of this query.

```
Query findall(  ?FailedInfo,
                and(  repositoryClause(?FailedInfo, [HotDrawDesign]),
                      not(call(?FailedInfo))),
```

```
        ?FailedList)
```

This query returns a list containing all failed clauses. This list can be processed manually (to check why a piece of design information does not hold anymore, using all facilities offered by the development environment and our tools), can be displayed more appropriately using a browser, or can form the foundation for a sophisticated tool that suggests solution strategies.

In addition to checking conformance of the implementation to existing regularities, we can also check whether the modification has actually changed the structure of the implementation. In that case we need to update the structural information that is contained in the logic repository. The general approach to do so is to regenerate (parts of) the design information, and check whether they are contained in the repository. For example, we can extract the information regarding composite design patterns, and find all possible new instances:

```
Query findall( newCompositePattern(?Comp, ?Composite, ?Sel),
               and( compositePattern(?Comp, ?Composite, ?Sel),
                    not(repositoryClause( compositePattern(?Comp, ?Composite, ?Sel),
                                          [HotDrawDesign]))),
               ?NewComposites)
```

This query again returns a list containing all possible new occurrences of composite design patterns. These can then be checked by the user and asserted in the logic repository, or fed into a browser or a tool.

4.5 Conclusions

The case study of the *HotDraw* framework illustrates, amongst other things, how our declarative meta programming approach can be used to:

- extract structural information (class diagrams, design pattern structures) from the implementation;
- generate implementation (source code) from structural regularities;
- check conformance of implementation to structural regularities, showing discrepancies between both;
- guide the implementation by checking implementation changes against structural regularities, and possibly react on detected violations by re-generating code.

We have also showed how we used the approach to codify explicitly some undocumented and hard to find structural regularities among the *DrawingEditor*, *Figure* and *Tool* classes. These regularities were hardcoded in a number of methods on these three classes, and used several naming conventions and low-level dependencies. Using SOUL we made these structural regularities explicit and used them to guide development. Some more general conclusions regarding the usability of our approach are deferred until section 6, after we have discussed the MediaGeniX case study.

5 MediaGeniX Case

In addition to the smaller-scale case study on HotDraw, we tested the usability and scalability of our approach to codify structural regularities of object-oriented programs on a large-case industrial framework. This real-world test was conducted at MediaGeniX, a Belgian company that develops tailor-made broadcast management systems for television stations [6]. Our case study concerned the *Media Management* module of *Whats'On* and handles everything that has to do with the actual management of the media used for broadcasting, such as tapes. This module had recently been rewritten, and consisted of 441 classes. Since the *Media Management* module is one of the newer parts of *Whats'On*, it was one of the first to use the *MediaGeniX Application Framework* (MAF). The MAF is MediaGeniX' framework for building applications.

We performed two sets of experiments. The first was to check conformance of the existing UML diagrams for the *Media Management* module with the actual structure of the implementation. The second was to make explicit the structural regularities that MAF applications should comply with and to check existing applications built with the framework for conformance with these regularities.

Due to space limitations we cannot go into the details of this case study. Therefore, we summarize the main results only (see [22] for more details). In a limited period of time (less then a week) we managed to successfully codify some structural regularities of the Media Management module and the MediaGeniX Application Framework and to verify conformance of the implementation to these regularities:

- We checked conformance of the released implementation to the existing UML diagrams and discovered some important discrepancies between the two. Some classes and relations in the UML diagrams did not exist in the implementation. We also succeeded in complementing the UML diagrams with extra structural information that was extracted from the implementation, such as role names for associations.
- We extracted some structural regularities underlying the MAF from the main architect and codified them. As in the HotDraw case, most structural regularities of the MAF were implicit. Hence we first asked the main architect to write down a list of regularities. We then codified these regularities, but soon realized that they were incomplete or inconsistent. We then asked the architect to clarify some issues. Several iterations were needed over the codified regularities, trying them out on small parts of the implementation, before we and the architect were satisfied. In fact, this extraction process proved interesting for the architect as well, as it uncovered some hidden information that he was not aware off. The output of this experiment was a set of structural regularities codifying programming conventions used in the MAF.
- We then searched the implementation of applications using the MAF for violations against the structural regularities imposed by the MAF. We discovered a number of clear errors in some parts of the implementation that did not respect these conventions, as well as a number of potential bugs. These potential bugs were 'dirty' code fragments for which it was unclear whether they conformed to the MAF architecture or not, and that had to be checked manually afterwards.

6 Discussion

The case studies illustrate how our approach is used to codify structural regularities, and how this supports the software development process. We now discuss what we have learned from the cases as well as some advantages and disadvantages of our approach. The next subsections discuss its usability, performance, expressivity and scalability. We then address some open issues and how we plan to resolve those.

6.1 Usability

We successfully applied our approach in two different contexts to make undocumented and sometimes hard to find structural regularities explicit. In the case of HotDraw, many regularities were hardcoded in a number of methods on three different base classes, and relied on various naming conventions and low-level dependencies. Using SOUL we codified these regularities explicitly and use them to guide development. This illustrates the practical usability and relevance of our approach, even for a 'mature' framework such as HotDraw. In the MediaGeniX case, we expressed some of the structural regularities of their application framework. There the actual process of writing down the rules proved worthwhile in itself.

While it is already important to make explicit the structural regularities of a program, once they have been codified these regularities can support the software development process in a variety of ways. One of the main uses is to check conformance of program code to the 'documentation' codified in the regularities. For example, in the MediaGeniX case study we uncovered some discrepancies between the documented UML class diagrams and the actual implementation. In addition to conformance checking, the case studies illustrated how the codified regularities can be used for high-level searching and browsing of the source code, for extracting design information from undocumented code and even for generating source code.

6.2 Performance

SOUL was originally conceived as an experimental language to allow us to experiment with the integration of a logic programming language and an object-oriented programming language. Therefore, not so much the performance, but the extensibility and expressivity of the language were important.

On our 250 mhz Macintosh G3 portable, inferring the type of an instance variable of a class takes about one minute and a half. Extracting the UML class diagram for the complete Figure hierarchy in HotDraw (around 60 classes) took about three hours. Note that, since Smalltalk is dynamically typed, most of this time is spent in typing the instance variables. This typing is so slow because it not only uses classic inferencing techniques but takes programming conventions into account. From these numbers we conclude that even using the current, non-optimized implementation of SOUL, querying the source code using a logic meta programming language is feasible. While very complex queries may take a few hours, most queries take on the order of minutes. Although this is too slow to be truly interactive, it is promising because of the current experimental nature of SOUL.

To address the performance question we have recently reimplemented SOUL (and renamed it *QSOUL*). QSOUL is essentially the same as SOUL but is optimized for speed. It is still a research language but its stack-based implementation already yields a performance gain of a factor 5.

Another remark is that some queries will always last long, even when we would have a very efficient logic language. For example, doing a full conformance check to make sure that the implementation follows all structural regularities inherently requires a lot of time. Such queries could always be run in batch overnight, producing a report of issues to be addressed. This approach was also adopted in [12], where a declarative meta programming is adopted for checking architectural conformance of an implementation.

6.3 Expressivity

Logic programming has long been identified as very suited to meta programming and language processing in general. In particular, logic meta programming proved to be very expressive for reasoning about structural regularities of programs thanks to its declarative nature, its expressive power (including recursion), its capacity to support multi-way queries and the powerful built-in mechanisms of unification and backtracking.

Of course, there is always an important trade-off between expressivity and speed. For example, regular-expression based search tools (such as SmallLint or grep) are much faster than our approach, but can only express a subset of the regularities we can express. To combine the best of both worlds we are planning to extend the SOUL language and environment with alternative search and inference strategies. Like that we can always choose the most efficient strategy for the particular task at hand.

6.4 Scalability

Contrary to what one might have expected, from the case studies (the smaller-scale HotDraw case, as well as the larger-scale MediaGeniX case) we can conclude that our declarative meta-programming approach scales well to real-world contexts. We identified two reasons for this.

First of all, the logic language is tightly integrated with the Smalltalk environment on which it relies to reason about Smalltalk source-code entities. Reasoning about a small program or about a large one is possible because the Smalltalk environment itself scales very well in that respect. For example, there is little difference between asking the Smalltalk image for all implementations of methods in HotDraw, or doing the same for the complete Smalltalk system.

Another key point that makes the approach efficient in a real world context is reduction of scope. In practice, we do not often need to ask queries about all source code in the entire Smalltalk image. We typically reduce the scope of complex queries quite effectively by making use of certain programming conventions, and by pre-filtering irrelevant information using a coarse grained (but efficient and inexpensive) approach and then finecombing these results with the more expensive full logic programming approach. This has worked out nicely in all the experiments we performed.

6.5 Open Issues

In the case studies, we used declarative meta-programming in a rather ad hoc way. One important thing that is still missing in the approach to make it more usable in practice, is a clear *methodology* that tells us how, where and when to use it.

Another issue we are currently looking into is to extend the SOUL environment to be able to deal with *other object-oriented languages*. We are currently extending the language and declarative framework so that they can work over Java source code too.

As opposed to Minsky's approach (see sections 1 and 7), our approach is a static one and is not good at regulating the interactions among objects and constraining the message-sending behaviour at run-time. Technically, however, it is possible to extend our approach with more dynamic capabilities.

7 Related Work

SOUL is a logic meta programming language that can reason about the static structure of Smalltalk programs. By writing rules, facts and queries in this language, we reason about and manipulate various structural regularities and as such aid software developers in a variety of activities. Two characteristics of SOUL set it apart from some of the related work discussed below. First of all, structural regularities expressed in SOUL are essentially *global*, although they can often be applied more locally by restricting their scope. Secondly, SOUL does not restrict the ways in which the rules may be used. It supports querying, conformance checking and even generation of source code.

Minsky's work on 'law-governed' systems was discussed extensively in section 1. While it focuses on enforcing software regularities *dynamically*, we focus on reasoning about structural software regularities that can be checked *statically*. Also, whereas Minsky's approach is essentially restricted to *enforcement* of regularities, we do not have this restriction. But both approaches have in common that they try to support software developers by providing explicit support for *global* software regularities.

NéOpus [17] is a run-time decision support system based on a forward chainer integrated with Smalltalk. Like Minsky it emphasizes dynamic enforcement of rules. Another difference with our approach is that their inference engine uses *forward chaining* whereas we use *backward chaining*. Forward chaining seems to be a good choice in the context of decision support, but when building an advanced querying facility for Smalltalk, a backward chainer is better suited to the task. Still, for some applications (e.g., actively guiding a software developer to restructure code) forward chaining could be better. We currently research how to integrate a forward chainer with our approach.

A whole range of related work exists that focus on enforcing or constraining the structure of object-oriented programs. For C++ there is *CCEL* [15] or *Astlog* [7]. For Java, *CoffeeStrainer* [3] is a preprocessor that allows to express structural constraints on Java code. Like law-governed systems and NéOpus, all these approaches are restricted to constraining the structure of a program but provide little or no support for querying and code generation. Also, because CCEL and CoffeeStrainer embed rules inside the programs, they are essentially geared towards enforcing local structural constraints. Eiffel [14] is comparable to the systems described above, but uses run-time checks. Structural assertions take the form of boolean Eiffel expressions, and can be used as pre- and postconditions of routines or as class and loop invariants. Hence their scope is local and determined by their position in the source code.

An interesting related approach that allows *conformance checking* of source code to structural constraints is *Lint* [11]. Originally, Lint was used to check C code for common programming mistakes. Its regular expression based search engine allows one to express fairly sophisticated string patterns. An interesting port of Lint is *SmallLint* [19], which supports regular expression searches on Smalltalk parse trees. *Lint* and its derivatives are good examples of lightweight approaches to express simple, string-based programming conventions. They sacrifice expressivity (abstraction facilities and recursion) to obtain better performance. Such an approach may be ideal to complement our approach.

8 Conclusion

In this paper we presented our declarative meta programming approach and how it can be used to codify structural regularities of object-oriented programs. These regularities have an intrinsic global nature. A declarative language offers a natural way of describing them. Furthermore, describing them in a meta programming language on top of another programming language allows us to use these structural regularities for searching and browsing the source code, checking conformance, enforcing regularities and generating

source code. As such, the structural regularities serve as 'active' documentation of a program and effectively support a software developer in various activities throughout the software life-cycle.

To validate the usefulness of our approach in practice, we conducted two extensive case studies. The first one was the well-known HotDraw framework for graphic editors. The second case study was a larger industrial framework for building television broadcast management systems. We used our approach successfully to extract class diagrams, occurrences of design patterns and other structural regularities from source code; to generate source code from structural regularities; to check conformance of source code to structural regularities, showing the discrepancies between both; to guide the implementation by checking implementation changes against structural regularities and possibly reacting on detected violations by re-generating code.

Whereas these results certainly convinced us of the practical usability, performance, scalability and expressivity of our approach, we felt that the approach was sometimes a bit too ad-hoc. This is partly caused by the fact that the integration with the Smalltalk development environment can still be improved in many ways, and partly by the fact that a good methodology for using our approach is still missing.

Acknowledgments. We thank everyone who contributed to this paper, in particular MediaGeniX (for the experiments) and the *QSOUL* team at the Programming Technology Lab.

References

1. K. Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, 1997.
2. Kent Beck and Ralph Johnson. Patterns generate architectures. In *Proceedings ECOOP'94*, volume 821 of *LNCS*, pages 139–149. Springer-Verlag, July 1994.
3. Boris Bokowski. Coffeestrainer: Statically-checked constraints on the definition and use of types in java. In *Proceedings of ESEC/FSE'99*. Springer-Verlag, September 1999.
4. John Brant. Hotdraw. Master's thesis, University of Illinois, 1992.
5. Ian Chai. How to create a new figure with constraints. Technical report, University of Illinois, 1994.
6. Wim Codenie, Koen De Hondt, Patrick Steyaert, and Arlette Vercammen. Evolving custom-made applications into domain-specific frameworks. *Communications of the ACM*, 40:71–77, October 1997.
7. Roger F. Crew. Astlog: A language for examining abstract syntax trees. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, 1997.
8. Martin Fowler. *Refactoring: Improving the Design of Existing Programs*. Addison-Wesley, 1999.
9. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addisson-Wesley, 1994.
10. Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, volume 27-10, pages 63–76, 1992.
11. S. C. Johnson. Lint, a C program checker. *Computing Science TR*, 65, December 1977.
12. K. Mens. *Automating Architectural Conformance Checking by means of Logic Meta Programming*. PhD thesis, Department of Computer Science, Vrije Universiteit Brussel, Belgium, October 2000.
13. K. Mens, I. Michiels, and R. Wuyts. Supporting software development through declaratively codified programming patterns. In *SEKE 2001 Proceedings*, pages 236–243. Knowledge Systems Institute, 2001. International conference on Software Engineering and Knowledge Engineering, Buenos Aires, Argentina, June 13-15, 2001.
14. Bertrand Meyer. *Eiffel : The Language*. Prentice Hall, 2000.
15. Scott Meyers, Carolyn K. Duby, and Steven P. Reiss. Constraining the structure and style of object-oriented programs. Technical Report CS-93-12, Department of Computer Science, Brown University, Box 1910, Providence, RI 02912, April 1993.
16. N. H. Minsky. Law-governed regularities in object systems; part 1: An abstract model. *Theory and Practice of Object Systems (TAPOS)*, 2(1), 1996.
17. F. Pachet. On the embeddability of production rules in object-oriented languages. *Journal of Object-Oriented Programming*, 8(4):19–24, 1995.
18. Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison Wesley Publishing Company, April 1996.
19. D. Roberts, J. Brant, R. Johnson, and B. Opdyke. An automated refactoring tool. In *Proceedings of ICAST '96, Chicago, IL*, April 1996.
20. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
21. R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *Proceedings TOOLS USA'98, IEEE Computer Society Press*, pages 112–124, 1998.
22. Roel Wuyts. *A Logic Meta-Programming Approach to Support the Co-Evolution of Object-Oriented Design and Implementation*. PhD thesis, Vrije Universiteit Brussel, January 2001.