Tool Support for Design Patterns based on Reference Attribute Grammars

A. Cornils*[1] & G. Hedin[2]

1: Department of Computer Science,
University of Aarhus, Denmark
apaipi@daimi.au.dk
2: Department of Computer Science
Lund Institute of Technology, Sweden
Gorel.Hedin@cs.lth.se

Abstract

Design patterns are abstract descriptions of solutions to often recurring problems. They are a means to communicate experience in design. Over the past years, along with the increase in popularity of object-oriented design patterns, some problems with the use of them have been identified. One of these lies in documenting software systems using design patterns. Experience has shown that both in the initial design, and especially in later code revisions, it is all too easy for code and documentation to diverge, rendering the documentation misleading and the code inconsistent. In this paper we present a flexible and extensible tool which enables designers to use design patterns in a safe and easy way and which semi-automatically documents and maintains the documentation of a software system. The system is implemented using reference attributed grammars (RAGs) which are capable of describing non-local dependencies. Both the programming language and the design patterns are specified using RAGs, and reference attributes are used for connecting design pattern instances to the corresponding elements in the program code.

1. Introduction

Object-oriented design patterns are descriptions of how classes can collaborate in order to solve commonly occurring design problems. The well-known catalogue by the "Gang of Four" (hereafter referred to as the "GoF book"[Gamma et al. 95]) describes 23 patterns for object-oriented design. The three basic parts of a design pattern are the *problem* description, the suggested *solution*, and the *consequences*, i.e. advantages and disadvantages of solving the problem in the suggested way.

We will use the **Decorator** pattern from the GoF book as an example throughout this paper. The problem described in **Decorator** is that a designer wants to extend the functionality of a class, but finds subclassing too inflexible. The suggested solution is to use a separate class for each piece of extending (or "decorating") functionality and to aggregate objects of these different decorator classes as needed. An advantage of this pattern is that the extended functionality of an object can be easily added or removed at run-time, and the design is thus more flexible than subclassing. A disadvantage is that the design results in many small decorator objects and thus may be more space intensive than a solution based on subclassing. Figure 1 shows the class diagram for the suggested solution. *ConcreteComponent* is the class whose functionality is to be extended. The different extensions, "decorators", are modelled by the classes *ConcreteDecoratorA*, *ConcreteDecoratorB* , and so on; all of them specialisations of the abstract class *Decorator*. The top class in the hierarchy, *Component*, models any component in this system, i.e. either a concrete component or a decorator. The extension

* This work has been supported by the Danish National Centre for IT-Research

of a component is accomplished by connecting a decorator to it and requiring the decorator to delegate all operations to the component (in addition to their added behaviour). Several concrete decorators can be attached to a concrete component in a cascade structure.

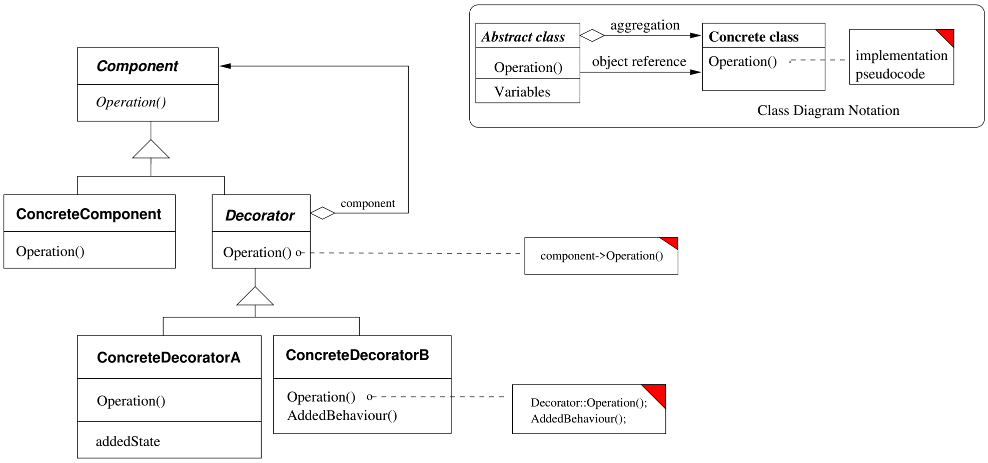

Figure 1: The **Decorator** design pattern structure

One of the benefits gained from using design patterns is that they provide documentation at an abstract level. For a reader familiar with the design patterns it may be sufficient to simply write the name of a design pattern, e.g. "Decorator", in order to document the functionality, the responsibilities, and even the motivation behind a whole subsystem of classes. Unfortunately, it happens all too often that the system is changed according to some new requirements while the documentation stays as is. For example, a design pattern may be introduced without recording it in the documentation. It is also common to make changes without consulting the documentation which may lead to the introduction of inconsistencies in the code. For example, adding a new concrete decorator, but failing to implement the delegation in the intended way, can cause existing code to work incorrectly.

If the design patterns were incorporated in the programming languages as language constructs, this problem could be solved by the compiler. Actually making language constructs out of all the design patterns is infeasible, though, for two main reasons. Firstly, as also argued in [Agerbo98], it can be worthwhile to make new language constructs out of some design patterns, but certainly not all of them since programming languages should be kept simple. Secondly, we expect design patterns to evolve, both due to the discovery of new design patterns but also when it is discovered that some design patterns are alike or applications of one another.

In [Hedin97] it is described how it might be possible to create a tool that enables designers to get the benefits of language constructs when using design patterns, but without actually changing the programming language.

The approach is based on the idea of viewing a pattern application as a language construct which identifies the program elements that play the particular roles in the pattern, and which specifies the rules that these program elements must follow. The design patterns must first be projected onto the roles and the rules of the design patterns. The roles can be of two types: *defining roles* and *derived roles*. The only classes, methods, and variables that must be explicitly marked in the code are the elements playing the defining roles. The other elements playing roles in the design pattern can be automatically derived from the defining roles, hence the name “derived roles”. In the **Decorator** design pattern the defining roles would be: *Component*, *Decorator* and the *decorated component* (the reference to a *Component* object from the *Decorator*). The roles derived from these defining roles would be: *Concrete Component(s)*, *Concrete Decorator(s)*, *Operation(s)*(the abstract operations in *Component*) and the *Implementation(s)* (the implementations of the *Operations* found in the *Decorator* and the *Concrete Decorators*)

In this paper we present the implementation of a prototype tool, $\mathcal{DPDOC}$, which supports documentation with design patterns based on the ideas discussed in [Hedin97]. The implementation technique builds on the use of *reference attributed grammars*, RAGs [Hedin99]. RAGs are stronger than conventional AGs because they allow references between nodes in the abstract syntax tree and thereby support the easy specification of non-local dependencies. The reference attributes can be freely used to access non-local information. For example, to specify name analysis, identifier uses can be directly connected to their declarations and used for type checking. For object-oriented languages, the class hierarchy can be specified explicitly by reference attributes between the classes and used for many different kinds of analyses. For the documentation of design patterns used in a program we represent the design pattern instances explicitly and use reference attributes to connect these instances with the program code. This explicit representation of design pattern instances differs from the implementation approach suggested in [Hedin97] and substantially improves the specification as will be discussed in further detail in Section 3.

$\mathcal{DPDOC}$ is based on (and built upon) an interactive language development tool, APPLAB (APPlication language LABoratory) [Bjarnason99], which supports specification by means of RAGs. APPLAB is primarily used to test grammars for new languages while they are being developed. Users can edit both programs and grammars at the same time, thus making APPLAB a highly interactive and flexible environment for language design.

APPLAB supports editing and static-semantic analysis of programs written in a subset of Java called *PicoJava*. PicoJava includes key object-oriented constructs like classes, subclassing, and methods and some basic constructs like statements, variables, and types. PicoJava has been implemented in APPLAB with the aim of providing a platform for experimentation with analyses of object-oriented languages without including all the details of a full-blown language [Hedin99]. Since $\mathcal{DPDOC}$ is built upon APPLAB it consequently supports the use of design patterns in PicoJava programs. This is sufficient for a prototype, since PicoJava contains all the language constructs necessary to apply the design patterns in the GoF book.

The implementation approach used in $\mathcal{DPDOC}$ separates the design pattern specification from the programming language specification, with only a narrow interface between them. This makes it possible to replace PicoJava with another (complete) object-oriented language without having to change the specification for the design patterns.

The design patterns are specified and implemented in the same way as language constructs: they are modelled by a grammar with syntactic and context-sensitive rules which specify the roles and rules of the available patterns. This allows the rules of the applied design patterns to be checked, similarly to the way a compiler performs compile-time checks. The tool is able to, based on a few pieces of information from the user, automatically bind elements in the program to roles in an applied design pattern. The user provides the names of the classes (or other constructs) that play the defining roles in the pattern, and the tool then finds the remaining derived roles and checks the correct use of the pattern. The design pattern grammar is split into several modules in order to support extensibility so that the set of available design patterns can be easily customised to the designer's needs. In particular, a set of general modules are provided that specify common aspects of the design patterns, thereby constituting a specification framework for the design patterns. This paper focuses on the specification of the design patterns using RAGs, whereas a companion paper describes the user interface and tool aspects in greater detail [Cornils00].

Currently, the tool supports the following patterns from the GoF book: **Decorator**, **Composite**, **Visitor**, **Observer**, **Mediator**, **Adapter**, **Bridge**, **Chain of Responsibility** and **Factory Method**. These patterns were chosen because we wanted to test different aspects of the tool. Some of them were chosen because they are challenging to check rules for. Some because they were almost alike and we wanted to show the difference in the documentation even though the design patterns had the same structure. Some because we wanted to investigate how easy it was to extend the tool with patterns that were more or less alike the existing patterns in the tool.

The rest of this paper is structured as follows. Section 2 describes $\mathcal{DPDOC}$, the architecture behind it, how it is implemented and how it can be extended and used for other languages. Section 3 describes related work in the area. Section 4 concludes the paper.

2. Implementation

In this section, we describe the architecture and the implementation of $\mathcal{DPDOC}$ and discuss why it is beneficial to implement the prototype in APPLAB by specifying the design patterns in RAGs. The user interface for the application programmer and the $\mathcal{DPDOC}$ programmer is described in relation to this.

2.1. Application- and $\mathcal{DPDOC}$-programmer interface

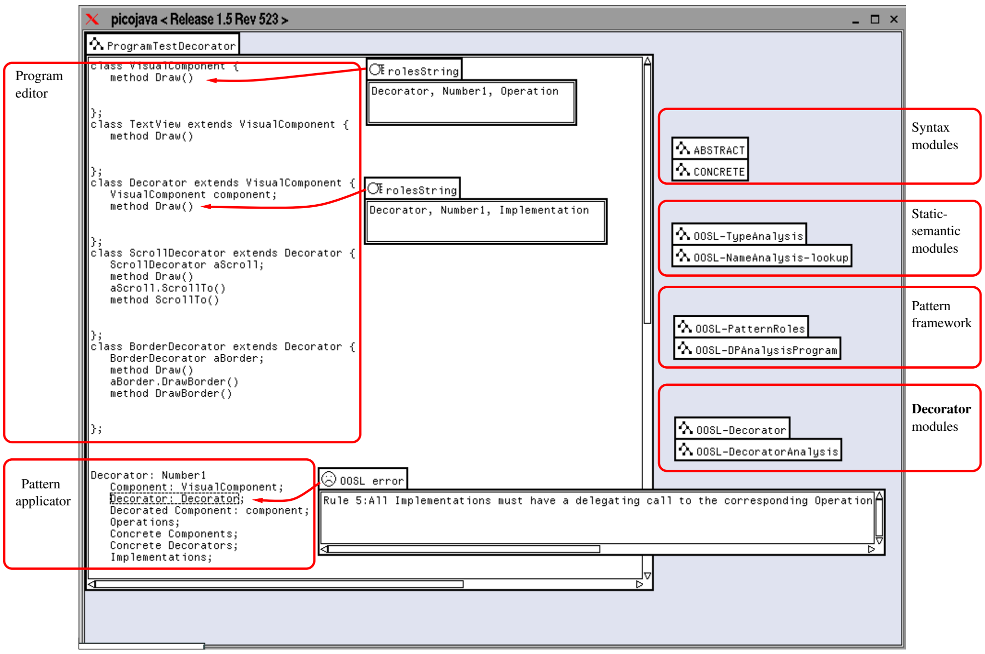

Figure 2: The user interface

Figure 2 shows the application programmer's view of $\mathcal{DPDOC}$ and a selected set of the grammar modules defining the underlying programming language and the pattern application language. The arrows and rounded boxes are not a part of $\mathcal{DPDOC}$, but additional information drawn on top of the screen-shot to emphasise the different parts of the user-interface and the grammar modules.

The application programmers view is the window called *ProgramTestDecorator* . It contains two parts: a program editor and a pattern applicator. In the program editor, the application programmer simply uses the features of the editor (in this case APPLAB) to write the programs. Below the program editor, we find the pattern applicator. In a future version, this will be placed in another window in order to make the program editor completely independent of the pattern applicator.

The editor and the pattern applicator together show a program on which the application programmer has applied the **Decorator** design pattern. Let us look at how this was done. After having written a program, the application programmer chose the design patterns they wanted to apply, in this case the design pattern **Decorator**. There are now four parameters to supply: the name of the pattern instance and the names of the elements in the program playing the three defining roles. They called the pattern instance *Number1* and let the *VisualComponent* class play the role of the *Component*, the *Decorator* class play the role of the *Decorator* and the *component* variable play the role of the *component* variable (called DecoratedComponent in $\mathcal{DPDOC}$). In supplying the names of elements playing the defining roles, there are three possibilities. Either the names could be typed, or they could be chosen from a menu containing all user defined names in the program, or the *Names* menu could be used, which lets the application programmer choose from a selected set of names. The *Names* menu is described in detail in Section 2.5.

For the $\mathcal{DPDOC}$ programmer that maintains the tool, the right part of Figure 2 is important. It shows the names of 8 grammar modules, each defining its own part of $\mathcal{DPDOC}$. These are the modules that should be changed when a $\mathcal{DPDOC}$ programmer wants to add new design patterns, modify design patterns or change $\mathcal{DPDOC}$s underlying language. The way to extend the tool with more design patterns is described in Section 2.4 and the way to change the underlying language is mentioned in Section 2.2.

In the “Syntax modules” box we find two grammar modules: “ABSTRACT” and “CONCRETE”. They define the abstract and the concrete syntax of the programming language and the design pattern application language. The “ABSTRACT” module defines the language constructs in the underlying programming language, what patterns the tool supports, and what roles these patterns have.

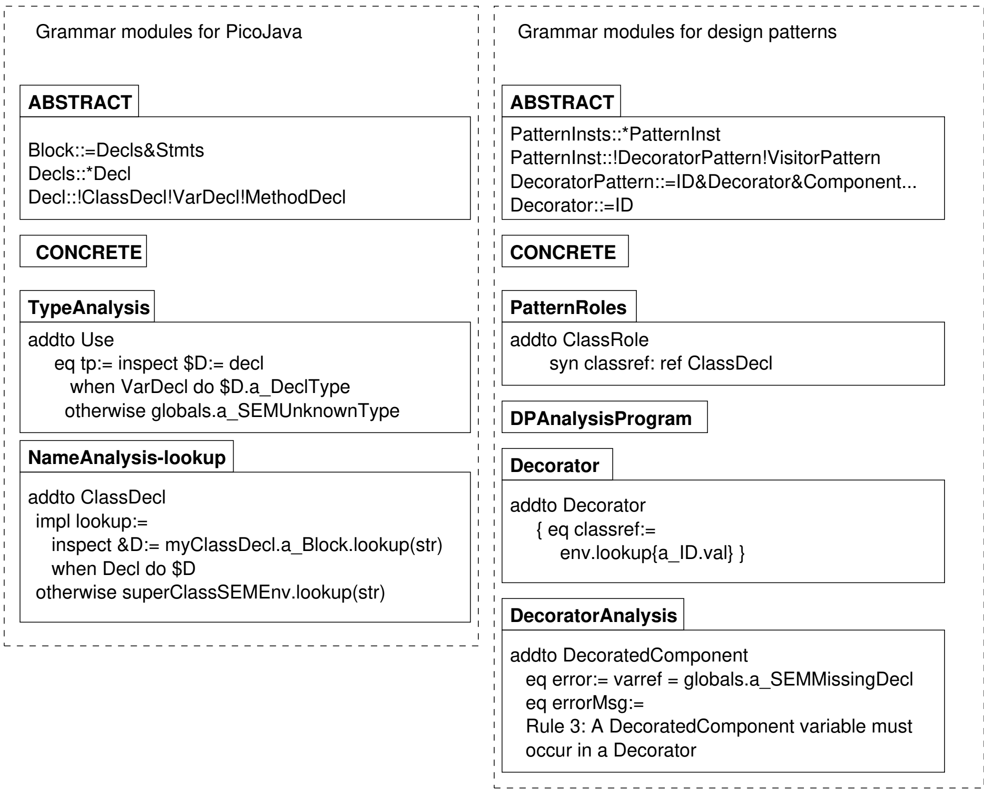

Figure 3: Excerpts from the grammar modules

Figure 3 shows a selected set of the grammar modules for $\mathcal{DPDOC}$, and for each of these, a selected set of rules. (In this figure, the syntax modules (ABSTRACT and CONCRETE) have been partitioned into different modules for the design patterns and the programming language. The tool currently supports only a single ABSTRACT and a single CONCRETE module, but in future versions we plan to support multiple syntax modules.)

The "Static semantic modules" box in Figure 2 shows a sample of the grammar modules defining the static-semantic analysis of the programs; type checking and name analysis in particular. The grammars make use of reference attributes to describe and make use of non-local dependencies. For example, in the *TypeAnalysis* module in Figure 3, a reference attribute *decl* in *Use* nodes refers to the appropriate declaration node and is used to define the *tp* attribute. (These aspects are discussed further in [Hedin99].)

The modules *PatternRoles* and *DPAnalysisProgram* in the "Pattern framework" are part of the pattern specification framework, described in detail in Section 2.3. This framework contains the implementation of the core functionality in the pattern specifications and is easy to extend with more design patterns. The module DPAnalysisProgram contains definitions that constitute the core of the documentation: For each element in the program, the semantic rules update the string attribute *rolesString* to contain which roles the element plays in which design patterns. For each element, the end-user can enquire which roles it plays, as shown in Figure 2, where the roles of two elements have been enquired.

The modules Decorator and DecoratorAnalysis specify the **Decorator** design pattern. In the Decorator module, the roles for the design pattern are specified. The DecoratorAnalysis module defines the rule checking. An example of how the rule checking works is shown in Figure 2. The error box on the right side of the pattern applicator displays a message that warns the end-user that they have broken one of the rules in the **Decorator** design pattern in his implementation. They can now choose to change the code according to the warning, so that the documentation follows the code or they can choose to ignore it. $\mathcal{DPDOC}$ will not try to change the code according to the rules of the design pattern.

The pattern application code and the user program code are connected by means of reference attributes. An example of this is shown in Figure 4 which shows a part of the syntax tree for the program and pattern applications of Figure 2. Here, the pattern part of the syntax tree contains an application of the **Decorator** design pattern, and the syntax node for the role *Component* has a reference *classref* which refers to a ClassDecl node in the program part of the syntax tree, in this program to the declaration of the class VisualComponent. This reference attribute is automatically computed according to the program and pattern grammar modules.

When the program is edited, $\mathcal{DPDOC}$ automatically builds the attributes for every node in the syntax tree, according to their definitions in the grammar. For example, for each expression in the program an attribute *tp* containing the type information is defined, as exemplified in the TypeAnalysis module in Figure 3. Likewise, the elements in the pattern applications are attributed according to the grammar. For example, when the user edits the name of an element in a pattern application, the tool locates the corresponding class declaration in the program, according to the attribute definition in the **Decorator** module in Figure 3.

2.2. The interface between the programming language and the design patterns

Figure 4 shows how the syntax tree for the pattern applications is connected to the syntax tree for the program by means of reference attributes: the *classref* attribute connects a class role like Component to the corresponding class declaration (VisualComponent in the example). To be able to declare the reference attributes, the pattern applicator grammar needs to have some knowledge of the

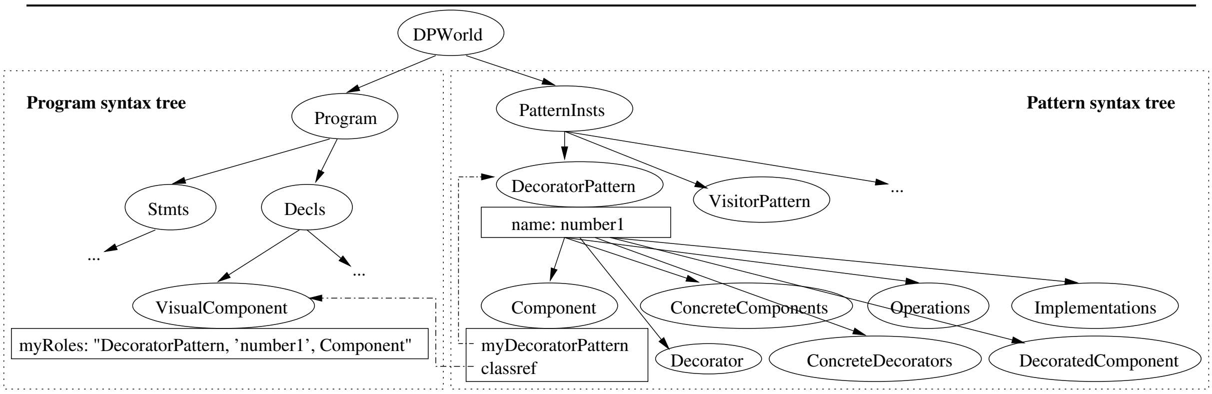

Figure 4: Reference attributes connect the design pattern applications to the program

programming language grammar. For example, the *classref* attribute has the type ClassDecl which is a production in the programming language grammar. Currently, $\mathcal{DPDOC}$ uses only the language constructs from Java that can also be found in most other object-oriented programming languages such as classes, methods and variables. The interface between the two grammars is thus narrow, and can be satisfied by most OO languages. Therefore, it is possible to change the programming language to another OO language without having to change the design pattern grammar. However, the new language may have additional constructs, e.g. multiple inheritance. To use these in the design patterns it is necessary to extend the interface accordingly.

The interface also contains attributes and functions defined by static-semantics module. This includes the environment attributes based on the block structure of the language and the lookup function of the name analysis. These attributes and functions are used for locating the appropriate declarations for class names and other names used in the pattern applications.

2.3. Pattern specification framework

RAGs are an object-oriented specification formalism where the non-terminals and productions are viewed as a class hierarchy; non-terminals correspond to superclasses and productions to subclasses. The pattern specification framework defines a number of classes capturing general aspects of design patterns, and which can be subclassed for the individual design patterns. Figure 5 shows an example of extending the framework to define the **Decorator** design pattern. The superclass *Role* defines the core functionality of a role, e.g. the name of the role. In this prototype, we have chosen to support six different kinds of roles: a single class, several classes, a single method, several methods, a single variable and several variables. Each of these kinds of roles has given rise to a class in the framework with their own default functionality common for this particular kind of role, e.g. the *classref* reference in the *ClassRole*.

In Section 1 we mentioned that we had chosen to implement certain design patterns in $\mathcal{DPDOC}$ in order to see how easy it was to extend the tool with more or less alike design patterns. Using the framework as is it was equally easy to implement design patterns no matter how much they differed from the design patterns already implemented. In a future version of the tool, when the framework is extended based on the similarities in "families" of design patterns it will be even easier to extend with more design patterns from the same "family" as design patterns already implemented in $\mathcal{DPDOC}$.

One of the main functionalities in $\mathcal{DPDOC}$ is providing the user with textual descriptions of which roles the elements of the program plays. The object hierarchy resulting from this is shown in Figure 4 (the pattern syntax tree). Most of this functionality is provided in the framework, and it is therefore very easy to add design patterns to $\mathcal{DPDOC}$, simply by adding subclasses to the framework *Role* classes.

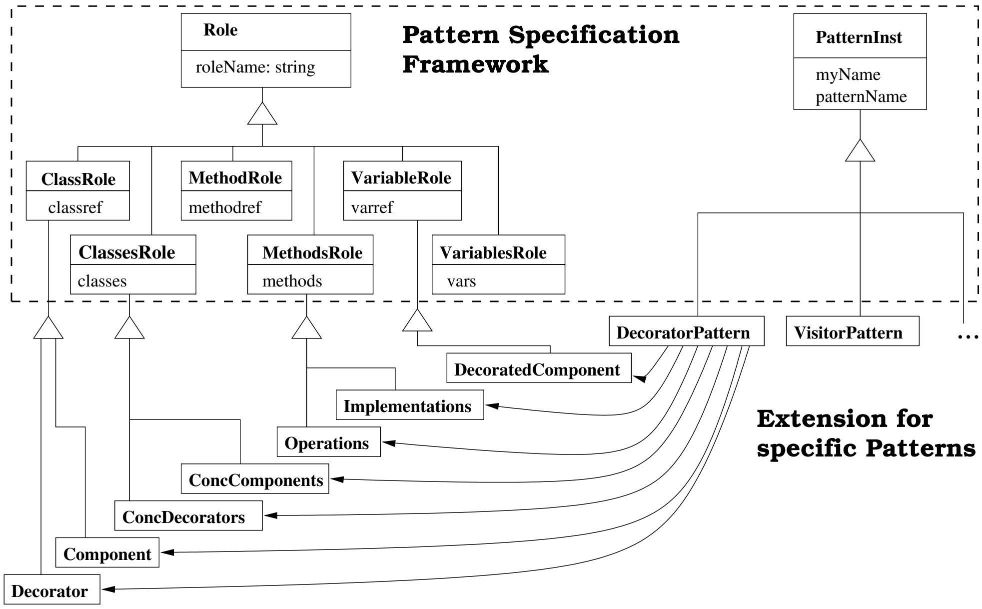

Figure 5: The pattern specification framework

2.4. Extension of the tool

To extend $\mathcal{DPDOC}$ with a design pattern, four steps must be performed:

- Extend the abstract syntax with the structure of the design pattern, i.e. the roles, and extend the concrete syntax with the end-users view of the design pattern. It is important to use the pattern specification framework by subclassing the role classes with the roles in the design pattern in order to benefit from the functionality found in the pattern specification framework.

- Extend the names facility module with the rules for naming the defining roles. This rarely amounts to much if the framework is used, since almost all of the defining roles can reuse the functionality placed as default by the framework. And most of the roles that can be restricted are already placed in the category "deriving roles".

- Extend $\mathcal{DPDOC}$ with a new module named after the design pattern. This module must contain the semantic rules for providing the references to the elements in the program. The defined roles are found by searching for the right names in the program. The references for the derived roles must be computed by the tool from the references in the defining roles. Again we stress that the use of the pattern specification framework eases this task.

- Extend $\mathcal{DPDOC}$ with a new module named after the design pattern with the suffix "analysis". This module must contain the semantic rules for checking that the rules of the design pattern are abided by in the program. The references found in the "roles"-module are followed to the program and the relationships between the elements playing the different roles are investigated. Since the rules in the design patterns differ a good deal, there is less support from the framework in this task.

An example of the modules defining the roles and the rules of a design pattern (**Decorator**) can be seen in Appendix A.

2.5. The Names Facility

$\mathcal{DPDOC}$ includes an editing facility called the *Names* menu: When the user wants to use the name of a class, method, or variable previously defined, a list is provided of names that are available at that place in the program. The list is constructed according to the scope rules of the programming language and is specified in the grammar. The *Names* facility is also used in the design pattern applicator in order to make it easier for the user to tie the defining roles to names in the user program. Here, the list of names is generated according to the rules for the design patterns.

Suppose an end-user is applying the **Decorator** design pattern to his program. If the defining role *Decorator* has already been tied to the class *VisualComponent*, the *Names* facility can be advantageously used when annotating the *component* role with a name from the program. The *Names* menu will now list the variables declared in *VisualComponent*, in accordance with the rules for **Decorator**, and the user can select one of these.

Nonterminals	**Attributes and Semantic Rules**
Roles	↑ names: **ref NodeBag** names := **new NodeBag** **son** ID.names:= names
ClassRole	names:= env.visibleNames
MethodRole	↑ myBlock: **ref** *Block* names:= **foreach** $D: *VarDecl in* myBlock.visibleNames **do** $R := (**init new NodeBag**) $R.add($D)
operation	myBlock:= componentBlock

Table 1: Names module. The syntax is described in Appendix A

Table 1 gives a simplified, but covering view of how easy it is to extend the *Names* facility in $\mathcal{DPDOC}$. The class roles have the classes in the program environment as name space. Methods and variables can as a rule only be chosen from specific classes playing a role. Some design patterns have roles that can extend this module, others not. Some design patterns (e.g. **Visitor** from the GoF book), with just defining roles, that are subclasses of the *ClassRole* will not give rise to an extension of the *Names* module.

2.6. Discussion

We chose to extend the already existing tool, APPLAB with the design pattern specifications in RAGs for these three major reasons:

Firstly, since our aim was to treat design patterns as language constructs, we had to use a tool, which could help us develop the language for describing the design patterns in an easy way. Since APPLAB is an interactive language laboratory, this tool was an obvious choice. APPLAB enables the language designer to see the consequences of design choices immediately. This interactive feature is also useful in the actual use of $\mathcal{DPDOC}$, because the roles played by the elements in the program are available at all times during the development of programs.

Secondly, the reference attributes provided by the RAG supports direct communication between nodes that are distant from each other in the syntax tree; a node c an access attributes of a distant node via a reference attribute. This is important because it enables nodes in the program part of the syntax tree to communicate directly with nodes in the design pattern part of the syntax tree. Additionally, this support for non-local communication is beneficial for specifying the underlying object-oriented language which have plenty of non-local dependencies (e.g., inheritance).

Thirdly, the object-oriented specification notation and the modularization of RAGs supported in APPLAB is beneficial in order to separate the specification of the underlying programming language from the specification of the design patterns. The narrow interface between these parts makes it possible to extend or change the programming language and the design pattern specifications almost independently from each other. In particular, support for new design patterns can easily be added by adding new grammar modules.

2.7. Future extensions

So far, we have focused the development of $\mathcal{DPDOC}$ on proving the viability of the language-based approach to integrating tool support for design patterns. There are many ways of improving the user interface in order to make it more intuitive and graphical, as discussed in [Cornils00]. From a specification point of view, the main thing to improve is the modularisation of the syntax, allowing the syntax of the programming language and the pattern application language to be defined in separate modules. (This change is in principle trivial to accomplish). In addition, it would be desirable to make the pattern specification framework independent of class names in the programming language grammar. For example, the framework currently defines a reference attribute *classref* which is a reference to a *ClassDecl* node, thereby assuming that the programming language has a nonterminal/production named *ClassDecl* . It would be desirable to have a mechanism for decoupling this dependency, allowing the programming language to use any name for its nonterminal/production modelling class declarations. This can be accomplished in several different ways. We plan to use a mechanism inspired by the "part objects" of BETA [Madsen92], but using some kind of multiple inheritance or Java-like interfaces would also be possible.

3. Related Work

RAGs were originally devised to support easy specification of static-semantics, in particular for object-oriented languages where there are more non-local dependencies than in other language paradigms. For example, RAGs allow the class hierarchy to be represented as explicit connections between syntax nodes. RAGs have also proven useful for a variety of other tasks, including specification of worst-case timing analysis [Persson99] and generation of software visualisations [Magnusson00]. In both these applications, the name analysis providing connections between identifier use and declaration sites forms a basis upon which the other analyses are built. This is the case also for the design pattern specification technique as suggested in this paper: the name analysis attributes are used in the specification of the connections between the design pattern instances to the corresponding locations in the program code.

As mentioned, our work builds on the earlier work reported in [Hedin97] which also suggested using (reference) attributed grammars as a basis for design pattern specification. However, in this earlier work, the programming language grammar was simply extended with additional rules. Our present work using an explicit representation of the design pattern instances has several advantages: First, the specification can be factored in a natural way according to the different design patterns, whereas in the earlier approach, language constructs like ClassDecl had to contain rules for all possible design pattern roles, and many conditionals were needed to handle the different cases. Second, the explicit representation of design patterns allows much of the functionality to be captured in the specification framework of role classes which can be conveniently specialised for the different design patterns. Third, the explicit representation constitutes a reification of the design patterns which makes them concrete in the user interface. Furthermore, the earlier approach was never implemented, whereas our present approach provides an implementation for all the major design patterns in the GoF book.

Several other tools for design patterns, both commercial and academic, have been developed in the last few years: [Brown96], [Budinsky et al.96], [CodeFarm], [Florijn et al.97], [Kim96], [Modelmaker], [Sane96], [Viljamaa98] (a thorough survey on tools supporting the use of design patterns can be found in [Viljamaa97]). These tools all, to some extent, help documenting code with design patterns. The way they choose to do it ranges from detecting the use of design patterns in the code to rule checking when design patterns are used. Some of them support additional functionality, e.g. code reuse, where instances of the design patterns can be downloaded from a library. Some of them support the specification of new design patterns whereas others provide only built-in support for a few patterns.

Our tool, $\mathcal{DPDOC}$, makes the use of design patterns visible on a detailed level, where every element in a program has the possibility to play a role in a design pattern. Additionally the tool perform rule checking, not just in the moment when the design patterns are applied, but consistently through the work on the program. The current version does not support code reuse or design pattern detection, but has a functionality not mentioned above: *Role derivation*. All support is specified in RAGs and the user can add support for new patterns as needed. The specification framework makes such additions comparatively easy. Also unlike other tools, the $\mathcal{DPDOC}$ approach keeps a narrow interface to the programming language and can be re-targeted by the user to other languages. The other tools are all targeted towards a specific language, some of them including vendor-supplied ports to a few languages.

The most interesting aspect to compare is the way the roles and the rule checking is specified in other tools. Of the tools we have investigated, the tool described in [Florijn et al.97] is the design pattern tool whose functionality comes closest to that of $\mathcal{DPDOC}$. Therefore, that is the tool, for which it is of relevance to describe the specification of the patterns. The tool is built on a fragment model described in [Meijers96], which allows the elements that can be used in the tool to be fragments and not just classes and methods. Fragments can represent not only the syntactic elements of a language, but also associations and inheritance relations and therefore provides numerous possibilities for describing relations between roles in design patterns and elements of a program. In the implementation the tool specifies the roles and rules of the design patterns as fragment collections of small fragments with associations implemented in Smalltalk. The way the tool is extended with a new design patterns is by reusing these types of fragments to build the design patterns with roles and rules using fragments and associations. To make an instance of a design pattern found in the tool, the specification of the design pattern (the graph of fragments) is cloned. The rule checking is expressed as compositions of predicates defined on fragment types. Much like in $\mathcal{DPDOC}$ some fragments have query operations, that checks whether e.g. a class playing a certain role contains a method playing a certain role. [Florijn et al.97] also supplies support for rule-checking after the code is glued into the program.

[Florijn et al.97] and $\mathcal{DPDOC}$ differ in some important points. Firstly, the ability to let other elements in the program than classes and methods play roles in design patterns is mirrored in our tool. But $\mathcal{DPDOC}$ enables the $\mathcal{DPDOC}$-programmer to let *everything* that can be represented as a node in an abstract syntax tree play a role in the design pattern, because the design patterns are specified in their own grammar module which can contain references to any kind of node in an abstract syntax tree. Secondly, $\mathcal{DPDOC}$ is developed to be language-independent, which makes it more flexible than [Florijn et al.97], which can only be applied to Smalltalk programs. Last, but not least, the rule-checking in [Florijn et al.97] is only done on the roles that are hard coded into the program by the user. The system is not able to derive which other roles from the design pattern are played by elements in the program, which is a feature in our tool. And to the best of our knowledge ours is the only tool with this feature.

4. Conclusion

DPDOC is a working prototype of a tool which can make it safe and easy for beginners to use design patterns and automatically maintain valid documentation of software. The tool supports design pattern visibility, rule checking, and automatic role derivation. *Design pattern visibility* is supported by reifying design pattern applications into explicit language constructs and allowing program elements to be tied to the reified design pattern roles. This allows the user to see explicitly which design patterns are applied in the code, and what roles are played by the different elements. This documentation of the applied design patterns is tied directly to the code via reference attributes, and the documentation does therefore not become out of date when the program is changed. Automatic *rule checking* is supported by checking that the rules for applying each pattern are abided by. Finally, *role derivation* is supported by automatically deriving many of the roles in a design pattern application, based on a small set of defining roles.

The tool is built on APPLAB and therefore inherits some functionality from this, like the *Names* facility, the syntax-directed editing and static-semantic check according to the grammar. Since we have developed *DPDOC* as an extension of APPLAB by specifying the design patterns as language constructs in a grammar, it makes use of the built-in static-semantic analysis to check a program. This way the rules for using design patterns are implemented exactly as rules for using language constructs and *DPDOC* works as a *lint* for design pattern applications. Because we have used RAGs it is possible to specify the design patterns in a very detailed and thus, very precise manner.

The object-oriented nature of the grammar formalism allows common aspects of the specification to be factored out into a pattern specification framework, making the addition of support for new patterns simple.

Since the architecture of the tool consists of two largely separate parts, the programming language and the design pattern applicator, with a narrow interface between them, it would be possible and even relatively easy to change the underlying programming language and the tool would be immediately useful for other languages than PicoJava.

By using modularisation and RAGs in the development of *DPDOC* we have created an extensible and strong prototype of a design pattern tool.

5. Acknowledgements

The paper is supported in part by COT (Centre for Object Technology). We would like to thank Erik Corry, Jørgen Lindskov Knudsen and all the anonymous reviewers for the constructive criticism, which we have attempted to follow. We would also like to thank the students, that are currently testing *DPDOC*: Tabita Enig, Henrik Kjær Nielsen and Lars Mellergaard.

Bibliography

[Agerbo98] Ellen Agerbo and Aino Cornils (1998): *How to Preserve the Benefits of Design Patterns.* Proceedings of OOPSLA'98.

[Bjarnason99] E. Bjarnason, G. Hedin, K. Nilsson (1999): *Interactive Language Development for Embedded Systems.* Nordic Journal of Computing 6(1999), 36-55.

[Brown96] K. Brown (1996): *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk. http://www2.ncsu.edu/eos/info/tasug/kbrown/thesis2.htm*

[Budinsky et al.96] F.J.Budinsky, M.A. Finnie, J.M. Vlissides, P.S. Yu (1996): *Automated code generation from design patterns.* IBM Systems Journal vol. 35, No. 2, 1996 – Object technology. *http://www.research.ibm.com/journal/sj/budin/budinsky.html*

[CodeFarm] *http://www.CodeFarms.com*

[Cornils00] Aino Cornils and Görel Hedin (2000): *Static-Semantic Checked Documentation with Design Patterns.* Proceedings of TOOLS Europe 2000.

[Florijn et al.97] G. Florijn, M. Meijers, P. van Winsen (1997): *Tool support for object-oriented patterns.* Proceedings of ECOOP'97.

[Gamma et al. 95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995): *Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company.

[Hedin97] Görel Hedin (1997): *Language Support for Design Patterns using Attribute Extension.* Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP '97.

[Hedin99] Görel Hedin (1999): *Reference Attributed Grammars.* WAGA'99. Second Workshop on Attribute Grammars and their Applications. Amsterdam, The Netherlands, March 26, 1999.

[Javadoc] *The Javadoc Tool Homepage http://java.sun.com/products/jdk/javadoc*

[Kim96] J. Kim et al. (1996): *An Experience Using Design Patterns: Lessons Learned and Tool Support.* Theory and Practice of Object Systems (TAPOS), Vol 2, No. 1, 1996, pp. 66-74

[Madsen92] O. L. Madsen, B. Møller-Pedersen (1992): *Part-objects and their location.* Proceeding of TOOLS '92 pp. 283-297.

[Magnusson00] E. Magnusson and G. Hedin (2000): *Program Visualization using Reference Attributed Grammars.* Submitted to NWPER'2000 (The Ninth Nordic Workshop on Programming and Software Development Environment Research), Lillehammer, Norway, May 28-30 2000.

[Meijers96] Marco Meijers (1996): *Tool Support for Object-Oriented Design Patterns.* Master's thesis, INF-SCR-96-28, Department of Computer Science, Utrech University, The Netherlands.

[Modelmaker] *http://www.modelmaker.demon.nl*

[Persson99] Patrik Persson and Görel Hedin (1999): *Interactive Execution Time Predictions using Reference Attributed Grammars.* Second Workshop on Attribute Grammars and their Applications WAGA99.

[Sane96] A. Sane, M. Sefika, R. H. Campbell (1996): *Monitoring Compliance of a Software System With Its High-Level Design Models.* ICSE-96 (http://choices.cs.edu/sane/patlint.pdf)

[Viljamaa97] J. Viljamaa (1997): *Tools supporting the Use of Design Patterns in Frameworks* Report C-1997-25, University of Helsinki, Department of Computer Science.

[Viljamaa98] J. Viljamaa (1998): *Tools supporting the Use of Design Patterns in Frameworks* Proceedings of the ECOOP'98 Workshop on Object-Oriented Software Architecture (OOSA'98), Brussels, Belgium, July 1998. *http://ide.hk-r.se/∼bosch/oosa98*

A. Appendix

This appendix shows some example specifications. Tables 2–3 correspond to the grammar module OOSL-Decorator of Figure 2 and show the specification of the roles of the **Decorator** pattern. Tables 4–5 correspond to the OOSL-DecoratorAnalysis module and show the specification of the rules for the pattern.

The tables should be read as follows:
The left column shows the name of the node being extended and the right column describes the attributes and their equations. The nodenames are *slanted* and the keywords are in **boldface**. The attributes can be of three types; synthesized declared by use of an $\uparrow$, inherited (in the attribute grammar sense, not in the object-oriented sense) declared by use of a $\downarrow$, and local declared by use of a $\rightarrow$. Synthesized attributes are used to propagate information upwards in the syntax tree, inherited attributes are used to propagate information downwards in the syntax tree and local attributes are just used within the node. The values of the attributes can be defined by equations, which are described by: “attribute name” := “expression”.

Nonterminals	**Attributes and Semantic Rules**
DecoratorPattern	**son** Roles.myPattern := **this** *DecoratorPattern* myName := a-ID.val patternName := “Decorator” **son** Roles.env := env **son** Roles.globals := globals a-DecOperations.myComponent := a-Component.classref a-ConcComponents.myComponent := a-Component.classref a-ConcDecorators.myDecorator := a-Decorator.classref a-ConcComponents.myDecorator := a-Decorator.classref a-ConcDecorators.operations := a-DecOperations.methods a-ConcComponents.myConcDecorators := a-ConcDecorators.classes a-DecoratedComponent.DecVar := a-Decorator.classref.a-Block a-DecImplementations.Implementations2 := **foreach** $M: *MethodDecl* **in** a-DecOperations.methods **do** $R := (**init new NodeBag**) $R.union(a-ConcDecorators.findMethod($M)); a-DecImplementations.Implementations1 := **foreach** $M: *MethodDecl* **in** a-DecOperations.methods **do** $R := (**init new NodeBag**) $R.union($M.matchListForMethod (a-Decorator.classref.a-Block.a-Decls)));
Component	classref := env.lookup(a-ID.val) roleName := “Component”

Table 2: Specification of roles for the **Decorator** design pattern, part 1

Nonterminals	**Attributes and Semantic Rules**
ConcComponents	↓ myDecorator: **ref** *ClassDecl* ↓ myComponent: **ref** *ClassDecl* ↓ myConcDecorators: **ref NodeBag** classes := myComponent.bagOfSubclasses.diff(myConcDecorators.add(myDecorator)) roleName := "Concrete Component"
Decorator	classref := env.lookup(a-ID.val) → TempBag: **ref NodeBag** roleName := "Decorator"

ConcDecorators	↓ myDecorator: **ref** *ClassDecl*
	classes := myDecorator.bagOfSubclasses
	↓ operations: **ref NodeBag**
	findMethod: **func ref NodeBag**
	        (M: **ref** *MethodDecl*) :=
	            foreach $C: *ClassDecl* **in** classes **do**
	            $R := (**init new NodeBag**)
	            $R.union(M.matchListForMethod($C.a-Block.a-Decls))
	roleName := "Concrete Decorator"

DecOperations	↓ myComponent: **ref** *ClassDecl*
	methods :=
	        foreach $M: *MethodDecl* **in** myComponent.a-Block.a-Decls **do**
	        $R := (**init new NodeBag**)
	        $R.add($D)
	roleName := "Operation"

DecImplementations	↓ Implementations1: **ref NodeBag** ↓ Implementations2: **ref NodeBag** methods:= Implementations1.union(Implementations2) roleName := "Implementation"
DecoratedComponent	↓ DecVar: **ref** *Block* varref:= DecVar.lookup(a-ID.val) roleName := "Decorated Component"

Table 3: Specification of roles for the **Decorator** design pattern, part 2

Nonterminals	**Attributes and Semantic Rules**
DecoratorPattern	a-Decorator.myImplementations := a-DecImplementations.Implementations1 a-Decorator.myDecComp := a-DecoratedComponent.varref a-Decorator.operations := a-DecOperations.methods

Decorator	$\uparrow$ error: **boolean**
	error := error1 **or**
	(error2 **or**
	(error3 **or**
	error4));
	$\uparrow$ errorMsg: **string**
	errorMsg := **if** error1 **then** errorMsg1
	else if error2 **then** errorMsg2
	else if error3 **then** errorMsg3
	else if error4 **then** errorMsg4
	else nostring;
	$\uparrow$ error1: **boolean** ;
	$\uparrow$ errorMsg1: **string** ;
	$\downarrow$ myComponent: **ref** Component ;
	error1 := **not** classref.subclassOf(myComponent.classref);
	errorMsg1 :="Rule 1:A Decorator should be a subclass of Component";
	$\uparrow$ error2: **boolean** ;
	$\uparrow$ errorMsg2: **string** ;
	error2 := classref.a-Block.lookup(myDecComp.a-ID.val)
	=globals.a-SEMMissingDecl;
	errorMsg2 :=
	"Rule 2:A Decorator should have a DecoratedComponent variable";
	$\uparrow$ error3: **boolean** ;
	$\uparrow$ errorMsg3: **string** ;
	$\downarrow$ myImplementations: **ref NodeBag**
	$\downarrow$ myDecComp: **ref** *VarDecl*
	$\rightarrow$ referenceval: **string** ;
	referenceval := myDecComp.a-ID.val;
	error3 :=
	foreach $M: *MethodDecl* **in** myImplementations **do**
	$R := (**init false**)
	$R **or** (
	foreach $S: MethodCallStmt **in** $M.a-Block.a-Stmts **do**
	$B := (**init true**)
	$B **and not** ((myDecComp.a-ID.val=(
	inspect $U1 := $S.a-Use
	when QualUse **do**
	inspect $U2 := $U1.a-Use
	when SimpleUse **do** $U2.a-ID.val
	otherwise nostring
	otherwise nostring)) **and**
	($M.a-ID.val=(
	inspect $U1 := $S.a-Use

Table 4: Specification of rules for the **Decorator** design pattern, part 1

Nonterminals	**Attributes and Semantic Rules**
Decorator	**when** QualUse **do** **inspect** $U2 := $U1.a-UnQualUse **when** MethodUse **do** $U2.a-ID.val **otherwise nostring** **otherwise nostring**)))); errorMsg3 := "Rule 5:All Implementations must have a delegating call to the corresponding Operation"; ↓ operations: **ref NodeBag**; ↑ error4: **boolean** ; ↑ errorMsg4: **string**; error4 := **not** operations.size(-)=myImplementations.size(-); errorMsg4 := "Rule 6: A Decorator must have an Implementation of all the Operations"
ConcDecorators	↑ error: **boolean** ↑ errorMsg: **string** errorMsg:= "Rule 4: A ConcreteDecorator must have an implementation of every operation" error := **foreach** $C: *ClassDecl* **in** classes **do** $R := (init false) $R **or** (**foreach** $M: *MethodDecl* **in** operations **do** $S := (init false) **if** (findImplementation($M.a-ID.val , $C)=$M) **or** (findImplementation($M.a-ID.val , $C)=**none**) **then** ($S **or true**) **else** ($S **or false**)); findImplementation: **func ref** *MethodDecl* (str: **string**,cls: **ref** *ClassDecl*):= **inspect** $M := cls.a-Block.lookup(str) **when** *MethodDecl* **do** $M **otherwise** **if** (cls.a-SuperOpt.superClass.a-Block.lookup(str) **in** *MethodDecl*) **then** cls.a-SuperOpt.superClass.a-Block.lookup(str) **else none** ;
DecoratedComponent	↑ error: **boolean** error:= varref=globals.a-SEMMissingDecl ↑ errorMsg: **string** errorMsg:= "Rule3: A DecoratedComponent variable must occur in a Decorator"

Table 5: Specification of rules for the **Decorator** design pattern, part 2