Statically Checked Documentation with Design Patterns

Aino Cornils
Computer Science Department
University of Aarhus
Denmark
e-mail: apaipi@daimi.au.dk

Görel Hedin
Department of Computer Science
Lund Institute of Technology
Sweden
e-mail: Gorel.Hedin@cs.lth.se

Abstract

Over the past years, along with the increase in popularity of design patterns, some problems with the use of design patterns have been identified. The so-called tracing problem *describes the difficulty in documenting software systems using design patterns. Initial approaches to solving the tracing problem have focused on guidelines for documenting design pattern roles and rules within a system, but 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. In this paper we present a flexible and extensible tool which enables designers to use design patterns in a safe and efficient way, which checks the design pattern rules, and which semi-automatically documents, and maintains the documentation of, a software system.*

1: Introduction

Design patterns are descriptions of abstract solutions to often recurring problems. The most well-known catalogue of design patterns can be found in [Gamma95], hereafter called the GoF book. In some ways we want to be able to treat design patterns like language constructs. Both in the sense that they are visible in the code and that we want to be warned when we misuse them. 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. Since we only refer to the type of patterns called *design patterns* in this paper, we will hereafter say patterns when we mean design patterns.

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 patterns, but without actually changing the underlying language. The idea is to annotate the program to identify the program elements that play particular roles in a pattern application and to check that the pattern rules are followed, i.e. that the roles are applied in the intended way. In the present work we have implemented this approach in a tool, $\mathcal{DPDOC}$, and we have gone one step further in reifying the pattern applications by representing them as explicit syntactic constructs separate from the program, but connected to the program by a kind of links. The patterns are specified in the same way as language constructs, but separately from the underlying programming language, and it is easy to extend the tool with support for new patterns in order to customise it to a designer's needs. This approach is more elegant and flexible than the one proposed in [Hedin97]. Our tool uses *reference attributed grammars* [Hedin99], which is an extension to attribute grammars that allows attributes of syntax nodes to be references to other nodes. This mechanism is used to connect the pattern instantiations to the syntax tree of the program.

Several pattern tools, both commercial and academic, have been developed in the last few years. The developers of these tools have focused on different characteristics.

The task of making the use of patterns visible in the code, is called *the tracing problem* [Soukup95], [Bosch97]. It is desirable for pattern tools to help solve this problem by improving what we shall call *pattern visibility*, enabling the user to see the use of the patterns in the code. Such tools can be helpful in the documentation of the system. Empirical results from documenting design patterns in code have showed that pattern documentation can speed up program changes as well as improve their quality [Prechelt et al.97]. These tests were performed with "static" handwritten documentation of the code with patterns. With a tool to automatically maintain the documentation, we claim that these results could be even better. *Rule checking* is another characteristic found in some tools. The tool developers have interpreted the abstract solutions in the design patterns to find the rules that must then be abided by, when the patterns are used and the tools are able to check whether these rules are followed. These checks can catch a certain class of errors that would otherwise call for extensive writing of test programs. We expect such checks to result in improved code quality and increased programmer's confidence in the code, similar to as reported for automated testing as in eXtreme Programming [Beck99].

$\mathcal{DPDOC}$ helps pattern visibility and performs rule checking and it has a functionality not found in other tools: *Role derivation*. The tool is able to, based on a few pieces of information from the user, automatically bind elements in the program to roles in the pattern. $\mathcal{DPDOC}$ also differs from other tools in its language-based approach to specifying the patterns which we claim provides a general and precise basis for building this kind of tools.

The rest of this paper is structured as follows. Section 2 contains an overview of our approach. Section 3 contains a description of $\mathcal{DPDOC}$ seen from the perspective of the end-user, i.e., the user that uses the tool to design and implement a system. Section 4 describes the architecture behind the tool, how it is implemented and how it can be extended and used for other languages. Section 5 reports on the current status of the tool. Section 6 gives a more detailed account of other pattern tools and how they relate to $\mathcal{DPDOC}$. Section 7 concludes the paper.

2: Overview of the approach

$\mathcal{DPDOC}$ helps designers use patterns in a safe and efficient way, almost as if they were language constructs. If a user misuses a construct in the programming language, the compiler answers with appropriate warnings. Similar to this, $\mathcal{DPDOC}$ checks whether the program abides by the rules of the patterns and warns the user if it doesn't. In a program, language constructs are easily visible due to keywords and the structure of the code. In contrast, an applied pattern normally has no explicit visible representation in the program – rather, it corresponds to a specific combination of classes, methods, and other constructs in the program. In $\mathcal{DPDOC}$, however, the applied patterns are reified and given an explicit representation, making it possible to see which patterns are applied in the program, and which program elements play the roles of each of these patterns. This approach provides a way of documenting the code by patterns and supports both visibility and rule checking. $\mathcal{DPDOC}$ proves particularly useful in relation to object-oriented *frameworks*. Since frameworks are semi-implemented solutions, applying a framework means specialising its functionality, i.e., subclassing the framework classes or aggregating them into application classes. To do this in the intended way, the user has to follow some rules for applying the framework. These rules can be expressed as patterns that $\mathcal{DPDOC}$ can check.

A program element that participates in a pattern is said to play a certain *role* in that pattern. In the GoF book, the class roles are called the *participants* of the pattern. For the precise definition of a pattern, any kind of program element can be used for a role: classes, methods, variables, etc. In our approach we distinguish between two kinds of roles: *defining roles* and *derived roles*. The programmer must supply information about which elements play the defining roles and the system will then automatically compute which elements play the derived roles. For example, for class roles, a superclass will typically play the defining role and its subclasses will play derived roles. For a given pattern, there are often certain rules that must be abided by. For example, there may be rules for how the defining roles must relate to each other, or rules for method call delegation, for example saying that certain method implementations must contain calls to certain other methods.

3: The end user's perspective

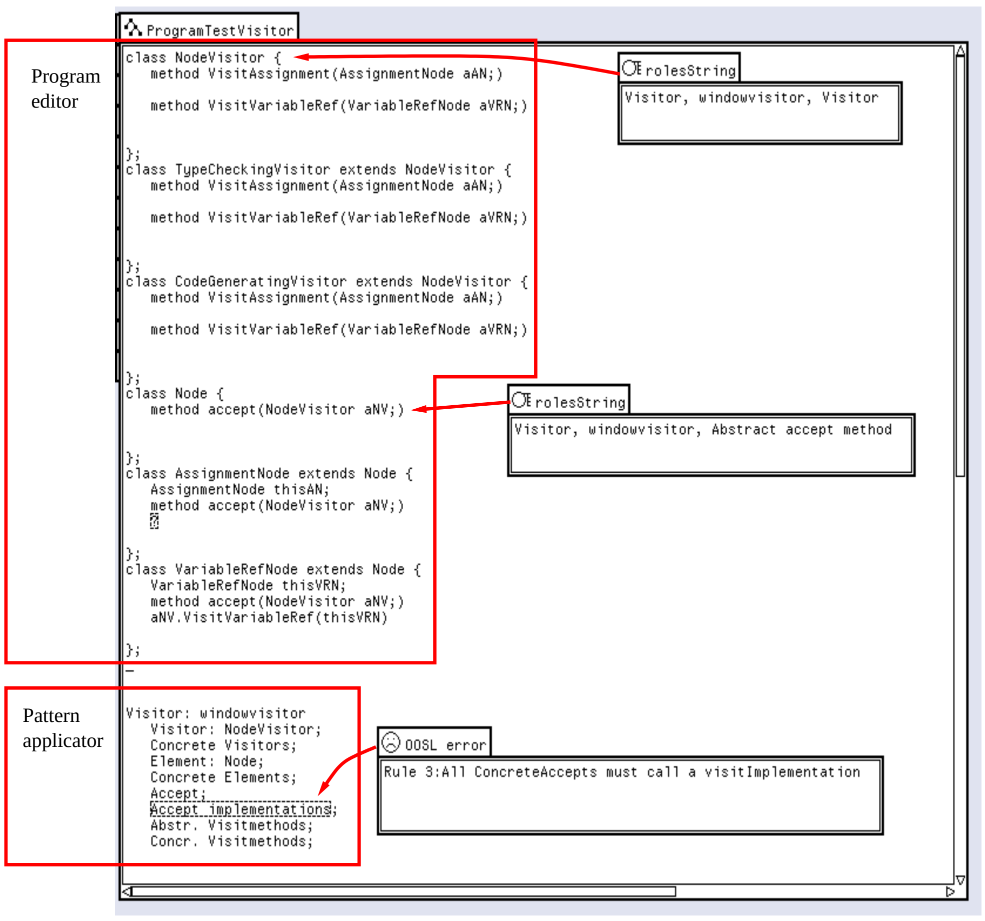
In this section we will describe the system seen from the perspective of the end user, i.e., the user that uses $\mathcal{DPDOC}$ to develop or document a system with the help of patterns.

Figure 1 shows an example screenshot from the tool. The workspace is separated into two parts: The *program editor* where the user writes program code, and the *pattern applicator* where the user uses a special-purpose language to document the patterns that are used in the program. In both parts of the workspace, the development is eased by various language-based editing support, like structure-oriented editing.

The example used in the figure is that of the **Visitor** pattern, as described in the GoF book. The problem of this pattern is how to define new operations on the objects in a polymorphic tree structure without changing the classes of those objects. The solution is to add the new operations in a new class in a separate "visitor" class hierarchy, and to connect to the original classes via an "accept" method.

In the pattern applicator part, the user creates applications (instances) of design patterns and ties these to the program code. The user selects a pattern from a list of supported design patterns and a template will then appear in the pattern applicator, showing the name of the design pattern and the roles of the new design pattern application. In the **Visitor** example, the roles are: *Visitor*, *Concrete Visitors*, *Element*, *Concrete Elements*, *Accept methods*, *Implementations of Acceptmethods*, *Abstract Visitmethods* and *Concrete visitmethods*. Some of these roles are defining roles and the user must tie these to the program by filling out placeholders with the names of the classes (or methods or variables) that play those roles. There are two defining roles in this example: *Visitor* and *Element*. The user ties these roles to elements in the program by filling in the corresponding class names: *NodeVisitor* and *Node*. The remaining six roles are derived and the tool automatically ties these roles to the appropriate elements in the program. Often, a derived role is tied to a whole set of program elements. For example, the derived role *ConcreteVisitor* is automatically tied to the classes *TypeCheckingVisitor* and *CodeGeneratingVisitor*, i.e., to the subclasses of *NodeVisitor* (the class playing the role of *Visitor*).

The pattern applicator also supports checking that the user program follows the rules of the applied patterns. When errors are discovered, the user is warned with a message box as shown in the bottom of Figure 1. In this example the system has detected an error for one of the methods playing the derived role of *Accept implementations*. The message box makes the user aware that a call to the method playing the role of *concrete visit method* is missing in one of these methods. The user can now choose to change the program or ignore the message, *DPDOC* will not try to change the code according to the rules of the pattern.



```
class NodeVisitor {
   method VisitAssignment(AssignmentNode aAN;)

   method VisitVariableRef(VariableRefNode aVRN;)


};
class TypeCheckingVisitor extends NodeVisitor {
   method VisitAssignment(AssignmentNode aAN;)

   method VisitVariableRef(VariableRefNode aVRN;)


};
class CodeGeneratingVisitor extends NodeVisitor {
   method VisitAssignment(AssignmentNode aAN;)

   method VisitVariableRef(VariableRefNode aVRN;)


};
class Node {
   method accept(NodeVisitor aNV;)


};
class AssignmentNode extends Node {
   AssignmentNode thisAN;
   method accept(NodeVisitor aNV;)


};
class VariableRefNode extends Node {
   VariableRefNode thisVRN;
   method accept(NodeVisitor aNV;)
   aNV.VisitVariableRef(thisVRN)

};
```

```
Visitor, windowvisitor, Visitor
```

```
Visitor, windowvisitor, Abstract accept method
```

```
Visitor: windowvisitor
   Visitor: NodeVisitor;
   Concrete Visitors;
   Element: Node;
   Concrete Elements;
   Accept;
   Accept implementations;
   Abstr. Visitmethods;
   Concr. Visitmethods;
```

```
Rule 3:All ConcreteAccepts must call a visitImplementation
```

Figure 1. The use of Visitor

The program editor can make use of information in the pattern applicator. As an example, for each element in the program it is possible to see which roles it plays and in which pattern applications. This is shown in the top of Figure 1 where the message boxes labelled “rolesString” shows the roles played by the class *NodeVisitor* (uppermost box) and the roles played by the method *accept* (middle box).

3.1: The Names Facility

DPDOC includes a language-based 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 point in the program. The list is constructed according to the scope rules of the programming language. The Names facility is also used in the 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 patterns. In the **Visitor** example, the *Names* menu for the defining role *Visitor* would contain all the names of the classes in the program, since this role, according to the rules of the pattern, should be played by a class. But the menu could be constrained by more complicated rules too. Imagine there was an extra defining role in **Visitor**, that should be played by a method, and according to the rules, this method should be declared in the class playing the role of *Visitor*. Then the *Names* menu for this defining role, would contain all the methods found in the class playing the role of *Visitor*.

4: Behind the Scenes

The main goal for *DPDOC* is to support pattern applications being explicitly represented and tied to the code so that they can be used as a means for documentation and error detection. Additionally, we wanted the tool to be easily extensible with new patterns and customisable to different object-oriented programming languages.

We have accomplished the customisation to other languages by representing the pattern applications explicitly as code in a specialised language for pattern applications (see the pattern applicator in Figure 1). This allows a loose coupling between the patterns and the underlying object-oriented language. The narrow interface between the implementation of the pattern application language and the implementation of the programming language enables the user to change the underlying language easily. This also works the other way around: the specialised pattern application language can easily be extended to support new patterns without affecting the underlying object-oriented language. We find it very important that these pattern definitions can be changed and extended since there are often variations on how a particular pattern can be implemented.

To implement our approach we needed a way to make the tool act like a compiler for our design-pattern language together with the compiler for the language. For this we used an interactive language development tool, APPLAB (APPlication language LABoratory) [Bjarnason99], as the basis for *DPDOC*. 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. This makes APPLAB a highly interactive and flexible environment for language design.

A detailed description of the implementation of *DPDOC* can be found in a companion paper [Cornils00].

4.1: Architecture

Figure 1 shows the end user's view of *DPDOC*. In addition, *DPDOC* contains a meta level where the underlying programming language and the pattern application language are defined. These definitions are given in two grammars: the program grammar and the pattern grammar. Each of these grammars is split into several modules, each extending the tool with a new piece of functionality. For example, the program grammar contains modules for name analysis, type checking, and so on. The pattern grammar contains modules for the different patterns, and also general modules constituting a specification framework for the design patterns. The user can change the program grammar to support another programming language, or extend or change the pattern grammar to support new patterns or pattern variations.

The specification technique used is reference attributed grammars, an object-oriented extension of attribute grammars that models the grammar as an inheritance hierarchy and permits the use of so called references [Hedin99]. References are attributes of syntax nodes that can be defined to refer to other nodes in the syntax tree. In $\mathcal{DPDOC}$, reference attributes are used for connecting elements in the pattern application code to elements in the user program code. An example of this is shown in Figure 2 which shows parts of the syntax trees for the program and patterns of Figure 1 . Here, the pattern syntax tree contains an application of the Visitor pattern, and the syntax node for the role Visitor has a reference *classref* which refers to a *ClassDecl* node in the program syntax tree, i.e., to the declaration of the class *NodeVisitor*. This reference attribute is automatically computed according to the program and pattern grammars.

All elements in the program contain a textual description of all the roles they play. This information can be used interactively, or for generating documentation for the code. The textual description contains a list of strings each carrying three pieces of information; which kind of pattern they play a role in, what instance of the pattern, and what role played in that pattern. This is contained in a variable called *myRoles* as can be seen in Figure 2.

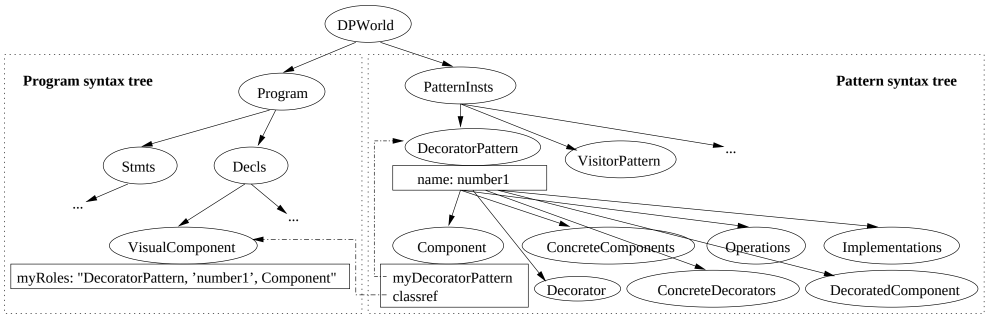
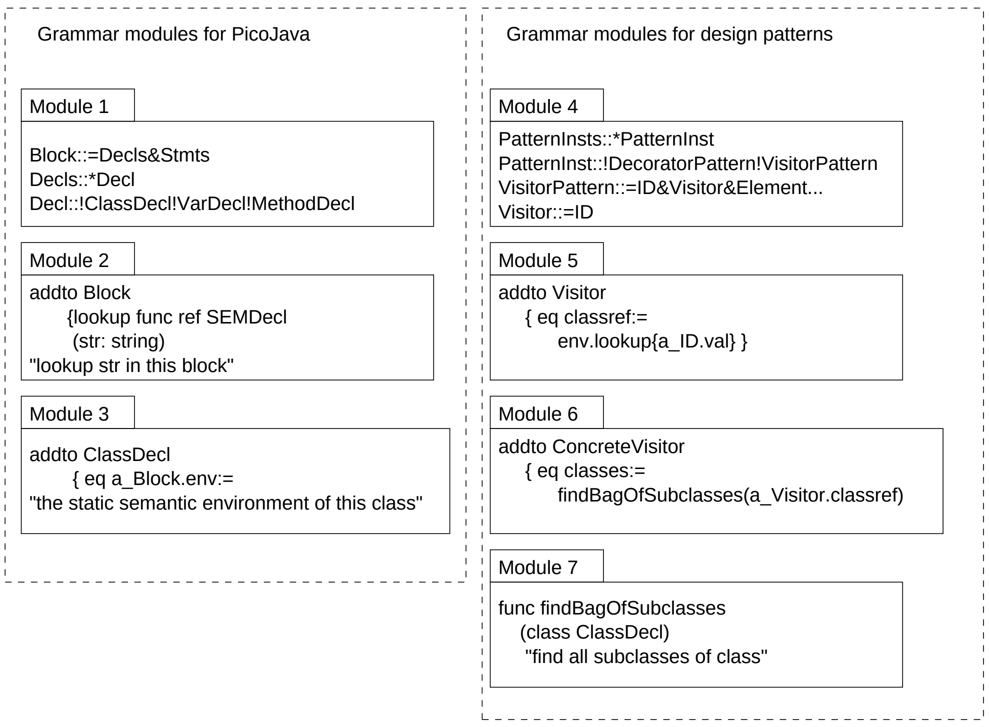


Figure 2. The reference attributes from the pattern applications to the program

Some of the grammar modules are shown in Figure 3 in a simplified form. To the left, part of the program grammar is shown. **Module 1** contains a description of the abstract syntax of the programming language. **Module 2** shows an example of a general function in the semantic analysis where the argument, a name, is looked up in the appropriate environment and a reference to the corresponding declaration node is returned. **Module 3** shows the definition of one of the environment attributes.

To the right in Figure 3, part of the pattern grammar is shown. **Module 4** contains a description of the abstract syntax of the pattern application language. This module defines what patterns the tool supports and what roles these patterns have. **Module 5** shows the definition of the *classref* attribute of the *Visitor* role, which is defined using the *lookup* function defined in the program grammar in **Module 2**. **Module 6** shows the definition of an attribute *classes* in a derived role. It is defined as the set of subclasses of the *Visitor* role class. This equation uses a function *findBagOfSubclasses* which is part of the pattern specification framework. It computes all subclasses for a given class and is defined in **Module 7**. Other modules (not shown in the figure) contain analysis that check if the rules for using the patterns are followed.


Grammar modules for PicoJava
Grammar modules for design patterns
Module 1
Module 2
Module 3
Module 4
Module 5
Module 6
Module 7

```
Module 1
Block::=Decls&Stmts
Decls::*Decl
Decl::!ClassDecl!VarDecl!MethodDecl

Module 2
addto Block
      {lookup func ref SEMDecl
       (str: string)
"lookup str in this block"

Module 3
addto ClassDecl
       { eq a_Block.env:=
"the static semantic environment of this class"

Module 4
PatternInsts::*PatternInst
PatternInst::!DecoratorPattern!VisitorPattern
VisitorPattern::=ID&Visitor&Element...
Visitor::=ID

Module 5
addto Visitor
    { eq classref:=
         env.lookup{a_ID.val} }

Module 6
addto ConcreteVisitor
    { eq classes:=
         findBagOfSubclasses(a_Visitor.classref)

Module 7
func findBagOfSubclasses
   (class ClassDecl)
    "find all subclasses of class"
```

Figure 3. The grammar modules

It is not coincidental that the grammars are made following the same principles. This is done since the patterns should be treated as language constructs in this tool. To use a language construct from the programming language the grammar for that language has to be followed. Similarly, to apply a design pattern, the grammar for the "language" of the design pattern applications must be followed.

When the program is edited, $\mathcal{DPDOC}$ rebuilds the syntax tree for the program and the pattern instances. The tool automatically builds the attributes for every point in the code, according to their definitions in the grammar. For example, for each class in the program, the environment attribute defined in **Module 3** in Figure 3 is built. Likewise, the elements in the design pattern applications are attributed according to the grammar. For example, when the user edits the name of an element in a design pattern application, the tool locates the corresponding class declaration in the program, according to the attribute definition in **Module 5** in Figure 3.

5: Current status and future extensions

$\mathcal{DPDOC}$ is a prototype tool. It was (and is being) developed to study how a tool can support the use of patterns and documentation in the best way. We use it in our research to experiment and evaluate possible characteristics in pattern tools. $\mathcal{DPDOC}$ is currently being evaluated by students from *The University of Aarhus* with respect to both functionality and user-friendliness.

The version of APPLAB that $\mathcal{DPDOC}$ is built on uses a subset of the grammar for Java called PicoJava [Hedin99]. PicoJava includes key object-oriented constructs of Java like classes, subclassing, and methods, but currently lacks support for, e.g., interfaces and exceptions. The constructs are, however, sufficient for experimenting with the original patterns as described in the GoF book. Changes to the underlying language of $\mathcal{DPDOC}$ is supported by the modularity of the tool, and $\mathcal{DPDOC}$ is thus capable of scaling its features to, e.g., full Java, if need be. However, when changing the underlying language, one might also want to change the pattern application language in order to take advantage of specific constructs in the new language. If the underlying language is expanded to full Java, one might want to change also the pattern application language to use the pattern solutions as suggested in [Grand98]. Similarly, if the underlying language is changed to Smalltalk, the pattern variations suggested in [Alpert et al.98] might be desired. These changes can easily be made in our tool by changing the specification of the pattern application language. In fact, the tool includes a pattern specification framework for easily specifying new pattern variations. A detailed description of the range and the implementation of the pattern specification framework can be found in a companion paper [Cornils00].

Currently $\mathcal{DPDOC}$ supports the use of 7 patterns from the GoF book: **Decorator**, **Observer**, **Mediator**, **Composite**, **Adapter**, **Factory Method** and **Visitor**. We have chosen to start with these patterns because some of them are fundamental patterns, according to the paper [Agerbo98], and therefore the most challenging to specify. From the specifications of these patterns we have factored out the general parts into a pattern specification framework which supports the common aspects of pattern roles. We will extend $\mathcal{DPDOC}$ with the remaining patterns from the GoF book, and based on our experience with extending $\mathcal{DPDOC}$, we expect this to be straight-forward. This mostly because of the pattern specification framework, but also because most of the remaining patterns can be seen as variations on the ones we have already specified. For instance, extending $\mathcal{DPDOC}$ with **Decorator** when **Composite** was already implemented was easily done.

So far, we have focused the development of $\mathcal{DPDOC}$ on proving the viability of the language-based approach to integrating tool support for patterns. An interesting way to continue the work might be to provide support for a more intuitive and graphical user interface. We have several ideas for how to do this within the language-based approach. One idea is to improve the user interface for connecting the pattern applications to the user program: As we have described in this paper, the connection is currently done by entering class names, method names, etc. in the pattern application code. A more intuitive way would be to support user-defined reference attributes that could be set by direct-manipulation, e.g. by clicking on the appropriate class or method in the user program.

We also intend to support navigation in terms of the pattern applications and roles. A simple extension to APPLAB to allow navigation along a reference attribute will allow us to specify navigational links, for example to navigate from the program code to the pattern application code or vice versa, or between different points in the program code that belong to the same pattern.

Another way of improving the user interface would be to provide visualisations of the pattern applications, e.g. in the form of UML-like diagrams similar to the solution diagrams in the GoF book. Visualisations can be integrated with the APPLAB system as described in [Magnusson00].

An interesting area to explore would be that of generating design documentation from programs, similar to how many systems, e.g. JavaDoc [Javadoc] and the Eiffel *short* mechanism [Meyer88] support the generation of API documentation. The strength of $\mathcal{DPDOC}$ in this setting would be that it is easy to specify the generation of different kinds of documentation from the program, either visual or textual.

6: Related Work

In this section we describe the characteristics of some pattern tools, both commercial and academic, and relate them to $\mathcal{DPDOC}$. The functionality of the tools range from simple detection of the use of patterns in a program to rule checking tools with a functionality that can be compared to that of $\mathcal{DPDOC}$. In these tools, the most interesting aspect to compare is the way the roles and the rule checking is specified. We also find the issue of extensibility and flexibility interesting. A thorough survey on tools supporting the use of patterns can be found in [Viljamaa97]. Some tools, e.g. [Brown96], perform pattern detection by pattern matching the code with the pattern structures to find instances of patterns in the code. These tools are not able to tell two patterns with the same structure apart and are little help in relation to documentation of the system.

Another group of pattern tools focus on code reuse, that is, these tools contain implementations of patterns, which can be glued into the users code, where they choose it. Examples are [CodeFarm](containing a library of C++ templates), [Modelmaker] (generating code in Delphi) and [Budinsky96] (which automatically generates code in C++). [Budinsky96] is a little more sophisticated than the first two, since it is possible to choose between various trade-offs in the pattern solution before the code is glued to the rest of the system. The focus in the development of these tools is code reuse, not documentation or rule checking.

A growing group of tools does perform rule checking, that is, they help the user use the patterns correctly by checking some rules made by the tool developers from the guidelines in the design patterns.

In "Monitoring Compliance of a Software System With Its High-Level Design Models" [Sane96], Sane et al. describe how their tool, "μchoices", can confirm whether the C++ implementation of a system maintains its expected design models and rules. The new approach seen in this paper is that the tool checks both statically and dynamically by both checking the code and the runtime performance. The static rule checking is specified by implementation of rules in Prolog. These rules are expressions of the authors' formalisation of patterns in positive evidence and violations. When the user claims that certain classes play certain roles in a pattern, positive evidence that they are in fact following the pattern is checked. If they are, the implementation is checked for violations of the rules. A Prolog inference engine matches the program data to the design model rules to produce relations that can demonstrate whether the rules in the patterns are abided by. By checking dynamically the tool can help to reassure that the rules are followed when the system is run even if the code appears to allow a deviation from the design. This approach exhibits advantages and disadvantages similar to those shown by languages with dynamic run-time type checking. There is no immediate support in "μchoices" for changing the underlying programming language to something else than C++ or extending the number of patterns, for which rule checking is supported. It is possible that the system would be able to derive roles in the program, as $\mathcal{DPDOC}$ does it, because of the way the rule checking is implemented, but it is not described how it could be done.

In [Kim96] Kim and Benner describe a C++-oriented tool, POE (Pattern Oriented Environment), which can help designers implement patterns in the intended way. The design pattern instances are represented as lists of instances of *Class*. The *Class* type contains the constraints connected to the use of it. Among these, the *Properties*, which are the relations and operations connected to the *Class* instance. When a design pattern is instantiated, the user can make bindings between the design pattern classes and the user defined classes. POE implements validation algorithms to ensure that different pattern instances and role bindings are used properly. It is possible to extend the tool with more patterns using the *Class*, *Relations* and *operations* types. Thus the means to specify patterns is limited to these types.

FRED (FRamework EDitor) as described in [Viljamaa98] is a development environment especially designed for framework development and specialisation in Java. A pattern is described using templates, which are sets of possible implementations with constraints that all its instances must conform to. One limitation of FRED is that constraints only apply to method and field signatures, data types and larger constructs, and thus can not verify that code within a method body is valid. The implementation of the role binding functionality is influenced by POE.

In the paper: "Tool Support for Object-Oriented Design Patterns" [Florijn et al.97], Florijn et al. describe their pattern tool. 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 the tool provides more possibilities for describing relations between roles in patterns and elements of a program. The tool specifies the roles and rules of the design patterns as collections of small fragments, with associations implemented in Smalltalk. The way the tool is extended with a new pattern is by reusing these fragments to build the patterns with roles and rules using fragments and associations. To instantiate a pattern found in the tool, the specification of the 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 check whether e.g. a class playing a certain role contains a method playing a certain role. [Florijn et al.97] provides three different ways of instantiating and binding patterns to the programs: *top-down*, where the tool generates all the program elements needed for the application of the pattern, *bottom-up*, where all the elements that should play roles from the pattern already exist in the program but must be bound to the pattern and *mixed*, where some elements of the design pattern are found in the program and the rest are generated by the tool. [Florijn et al.97] also supplies support for rule-checking after the code is glued into the program.

This tool is the one which comes closest in functionality to $\mathcal{DPDOC}$, but nevertheless they differ in some important points. Firstly, the ability to let other elements in the program than classes and methods play roles in patterns is mirrored in our tool. But $\mathcal{DPDOC}$ enables the user who adds patterns to let *everything* that can be represented as a node in an abstract syntax tree play a role in the pattern. 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 set explicitly by the user. The system is not able to derive which other roles from the pattern are played by elements in the program, which is an important feature in our tool.

7: Conclusion

$\mathcal{DPDOC}$ is a language-based prototype tool supporting documentation of program code with patterns. The tool supports pattern visibility, rule checking, and automatic role derivation. *Pattern visibility* is supported by reifying pattern applications into explicit language constructs and allowing program elements to be tied to the reified pattern roles. This allows the user to see explicitly which patterns are applied in the code, and what roles are played by the different elements. This documentation of the applied patterns is tied directly to the code, 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. This is useful in program evolution in order to make certain that existing pattern applications are not broken when the program is evolved. Rule checking is also useful when applying frameworks in order to extend them as intended, since patterns are often used in providing the variability or hot spots of the framework. Finally, *role derivation* is supported by automatically deriving many of the roles in a pattern application, based on a small set of defining roles. This is useful because it allows the user to tie the program code to a pattern using only a few key program elements (the defining roles). In the framework setting, the defining roles are typically located in the framework whereas the derived roles are located in the framework application code. This allows the user of the framework to take advantage of pattern visibility and rule checking without having to explicitly identify any roles in the application code.

The language-based approach for implementing $\mathcal{DPDOC}$ has several advantages. It allows support for new patterns to be added by specifying the roles and rules of those patterns in a declarative way, thus providing an extensible system. Due to its basis in reference attributed grammars, it provides the possibility of adding rule checks for literally anything the user could think of. It is as precise as a compiler, but takes much less work to build and extend. The object-oriented nature of the grammar formalism allows common aspects of the specification to be factored out into a specification framework, making the addition of support for new patterns simple. The language-based approach also provides language independence: By changing the programming language grammar, the system can be tailored to different programming languages.

By implementing seven of the most challenging of the GoF patterns in $\mathcal{DPDOC}$, we have proven the viability of the technique, and we think the approach constitutes a sound basis for pattern tool support.

8: Acknowledgements

The paper is supported in part by COT (Centre for Object Technology). We would like to thank Erik Corry, Ellen Agerbo, 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 $\mathcal{DPDOC}$: Tabita Enig, Henrik Kjær Nielsen and Lars Mellergaard.

References

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

[Alpert et al.98] Sherman R. Alpert, Kyle Brown and Bobby Woolf (1998): *The Design Patterns Smalltalk Companion.* Addison-Wesley Publishing Company.

[Beck99] Kent Beck (1999): *Extreme Programming Explained: Embrace Change.* Addison-Wesley Publishing Company. 1999.

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

[Bosch97] Jan Bosch (1997): *Design Patterns & Frameworks: On the Issue of Language Support.* Workshop on Language Support for Design Patterns and Object-Oriented Frameworks (LSDF), ECOOP '97.

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

[Budinsky96] F.J.Budinsky, M.A. Finnie, J.M. Vlissides, P.S. Yu (1996): *Automated code generation from design patterns.* IBM System Journal vol. 35, No. 2, 1996-Object technology. *www.research.ibm.com/journal/sj/budin/budinsky.html*

[CodeFarm] *www.CodeFarms.com*

[Cornils00] Aino Cornils and Görel Hedin (2000): *Tool Support for Design Patterns using Specification with Reference Attributed Grammars.* Submitted to WAGA'00. Third Workshop on Attribute Grammars and their Applications.

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

[Gamma95] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1995): *Elements of Reusable Object-Oriented Software.* Addison-Wesley Publishing Company.

[Grand98] Mark Grand (1998) *Patterns in Java. Vol.1* John Wiley and Sons.

[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

[Magnusson00] E. Magnusson and G. Hedin (2000): *Program Visualization using Reference Attributed Grammars.* To appear in 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.

[Meyer88] Meyer, B, (1988): *Object-Oriented Software Construction.* Prentice-Hall.

[Modelmaker] *www.modelmaker.demon.nl*

[Prechelt et al.97] (1997): Lutx Prechelt, Barbara Unger and Michael Philippsen *Documenting Design Patterns in Code Eases Program Maintenance.* ICSE-97 workshop on Proces Modeling and Empirical Studies of Software Evolution pp. 72-76, Boston, MA, May 18. 1997

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

[Soukup95] Jiri Soukup (1995): *Implementing Patterns.* Pattern Languages of Program Design. Eds. Coplien and Schmidt. Addison-Wesley Publishing Compagny.

[Viljamaa97] J. Viljamaa (1997): *Tools supporting the Use of Design Patterns in Frameworks* Report C-1997-25, University of Helsinki, Departement of Computer Science.*http://www.cs.Helsinki.FI/research/fred/reports.html*

[Viljamaa98] J. Viljamaa (1998): *Pattern-Oriented Framework Engineering Using FRED* Proceedings of the ECOOP'98 Workshop on Object-Oriented Software Architecture (OOSA'98), Brussels, Belgium, July 1998.