In Proceedings of the Workshop on Language Support for Design Patterns and Frameworks (LSDF'97), held in conjunction with ECOOP'97, Jyväskylä, Finland, June 1997. http://ide.hk-r.se/~bosch/lsdf

Language Support for Design Patterns using Attribute Extension

Görel Hedin
gorel@daimi.aau.dk

Computer Science Department, Aarhus University
Ny Munkegade , DK-8000 Aarhus C , Denmark

May,1997

Abstract We present a technique for formalizing design patterns using a technique based on attribute grammars. The technique allows design pattern applications to be identified in the source code, and supports automatic checking that the pattern is applied correctly. We expect the technique to be particularly useful when specializing frameworks built using design patterns.

1 Introduction

Design patterns in catalogs like [Gamma et al. 1994] and [Buschmann et al. 1996] describe recurring structures of collaborating objects. Each pattern involves a number of participating objects playing different roles in the pattern. Although the pattern descriptions are based on semi-formal class diagrams which allow for many implementation variations, they are sufficiently precise so that when selecting a particular implementation, the pattern could be formalized and form the basis of some kind of programming language support.

When a design pattern is applied, this can be viewed as a introducing a number of rules, i.e. subsequent updates to the program should be done without breaking the rules for the pattern. For example, the Decorator pattern uses a clever way of adding functionality to Component objects: a Decorator object wraps a Component and forwards all messages to its Component (in addition to possibly doing some functionality of its own). When introducing the Decorator pattern we implicitly introduce a rule which says that whenever we introduce a new operation on Components, the Decorators must implement this operation and forward it to its Component.

Thus, we can think of a pattern application as a kind of language construct which identifies the objects that play the particular roles in the pattern, and which specifies some rules that these objects must follow. We can also note that a given object may participate in several patterns, playing different roles.

Patterns are evolving concepts. New patterns and variations on how to implement specific patterns are reported every year at conferences, in books and journals, and in discussion groups on the net. While we might expect the set of general-purpose patterns like those reported in [Gamma et al. 1994] to converge to some fairly stable catalog, domain-specific patterns will probably continue to be identified over time. For this reason, it is not sufficient to support a number of predefined patterns by language constructs. Instead, we need a mechanism which allows new patterns to be both specified and applied. I.e., if we view each pattern as a language construct, we need a mechanism for extending the language.

In this paper, we propose the use of attribute extension [Hedin 1997] to support the specification and application of design patterns. This is a technique which allows the static-semantics of a language to be extended, allowing conventions to be enforced, but which keeps the syntax of the base language. A key advantage of this technique is that it is easy to integrate with existing languages and environments.

Other authors that have investigated language support for design patterns include Soukup who has suggested a technique based on macros [Soukup 1995] and Bosch who uses a filters-based object model (LayOM) to extend a language with pattern support [Bosch 1997]. Bosch identifies several problems with implementing design patterns: *traceability*, *the self problem*, *reusability*, and *implementation overhead*. Of these problems, this paper deals only with traceability (the identification of patterns in the code). In particular, we do not deal with implementation overhead (reducing the amount of coding a programmer has to do to implement a pattern). Instead, we deal with another problem: *rule enforcement* (how to make sure the code adheres to the rules of a pattern).

The rest of this paper is organized as follows. Section 2 gives an overview of the attribute extension technique. Section 3 discusses how we view the formalization of a pattern, taking Decorator from [Gamma et al. 1994] as a running example. Section 4 discusses how the pattern can be specified using attribute extension (an appendix gives the complete extension grammar). Finally, section 5 gives a concluding discussion of the benefits of the technique, and how it can be developed further.

2 Attribute Extension

Attribute extension is a technique for describing and enforcing programming conventions [Hedin 1997]. The technique is based on attribute grammars, describing conventions by declarative semantic rules. A convention checker can be automatically generated from an extension grammar in a similar way as an attribute evaluator can be generated from an attribute grammar. The technique makes use of three kinds of specification:

- A *base grammar interface*, which is a context-free grammar for the base language, extended with functions for basic static-semantic information such as name bindings and type information. The functions may return references to other nodes in the syntax tree.
- An *extension grammar*, which is an attribute grammar describing the programming conventions, making use of the base grammar interface to avoid specifying basic information from scratch.
- *Attribute comments*, which are special comments used to annotate an application program. Technically, an attribute comment `/*= a =*/` defines a boolean attribute `a` to have the value `true`.

The use of references in the base grammar interface is important because it allows name declarations and use sites to be bound together by direct references. This in turn allows the extension grammar to propagate information directly from declarations to use sites, without having to base the solution on the large complex symbol-table attributes that are necessary in standard attribute grammars.

The base grammar interface is specified in an object-oriented notation with a *node class* for each language construct. A simplified base grammar interface is shown below with nodeclasses like Declaration, ClassDecl, MethodDecl, VarDecl, Statement, and MethodCall. The interface provides functions which link identifier use sites to declaration sites. For example, the MethodCall nodeclass provides a function receiverDeclBinding to return the ClassDecl of the receiver, and a function selectorDeclBinding to return the MethodDecl of the selector. Similarly, the ClassDecl nodeclass provides a function superClassBinding which returns the ClassDecl of the superclass.

```
nodeclass Declaration ::= { }

nodeclass ClassDecl extends Declaration ::=
   ( optional SuperClassId, ClassId, DeclList ) {
   ClassDecl function superClassBinding()
}

nodeclass MethodDecl extends Declaration ::=
   ( optional ReturnType, MethodId, FormalParamList, MethodBody ) { }

nodeclass VarDecl extends Declaration ::= ( VarId, VarType) { }

nodeclass VarType ::= (ClassId) {
   ClassDecl function typeDeclBinding()
};

nodeclass Statement ::= { }

nodeclass MethodCall extends Statement ::=
   ( RecevierId, SelectorId, ActualParamList) {
   ClassDecl function receiverDeclBinding();
   MethodDecl function selectorDeclBinding()
}

nodeclass WhileStmt extends Statement ::= (Expression, Statement) { }
```

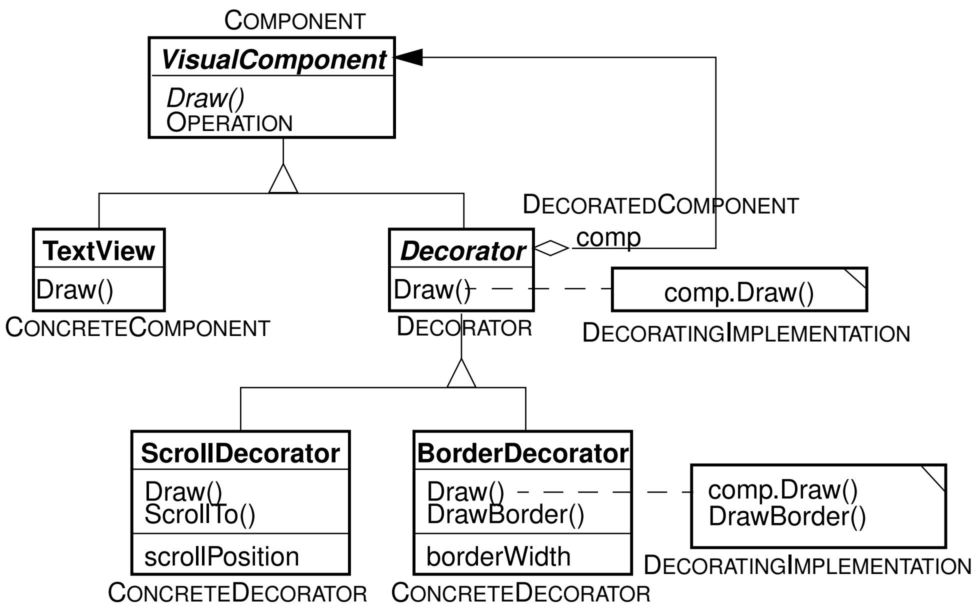
3 Language support for a pattern

To support a pattern on the language level, we need first of all to identify the different roles in the pattern and then formulate the rules for these pattern roles.

3.1 Pattern roles

The most important roles are played by classes, but some roles may also be played by operations or variables. Typically, the roles correspond to the generic names used in the pattern structure diagrams. For example, in the Decorator pattern [Gamma et al. 1994], we can identify the following roles (see also the figure below):

- COMPONENT. An abstract class for components to be decorated.
- OPERATION. An operation belonging to the interface of COMPONENT.
- CONCRETE COMPONENT. An ordinary subclass of the COMPONENT.
- DECORATOR. A special subclass of the COMPONENT serving as an abstract decorator. Has a DECORATED COMPONENT and possibly some DECORATING IMPLEMENTATIONS.
- DECORATED COMPONENT. A variable declared in a DECORATOR which denotes the decorated COMPONENT.
- CONCRETE DECORATOR. A subclass of the DECORATOR which may contain DECORATING IMPLEMENTATIONS.
- DECORATING IMPLEMENTATION. An implementation of an OPERATION which delegates the call to the DECORATED COMPONENT, and optionally also performs additional actions before and/or after the delegating call.



To support the identification of a pattern in the source code, we annotate the source code with pattern roles, using attribute comments. However, it is not necessary to mark *all* the roles in the source program, because many of the roles can be derived from the other roles. For the decorator, it is sufficient to explicitly mark the COMPONENT, DECORATOR, and DECORATED COMPONENT roles. We call these roles the *defining roles*. Other roles are called *derived roles*. For the Decorator pattern, the roles can be identified in the source program according to the following table.

Table 1 Definition of Decorator roles

Role	*Kind*	*Definition*
COMPONENT	defining	A class marked in the source code
OPERATION	derived	Any method declared in a COMPONENT
CONCRETE COMPONENT	derived	Any subclass of a COMPONENT which is *not* a DECORATOR or subclass to a DECORATOR
DECORATOR	defining	A class marked in the source code
DECORATED COMPONENT	defining	An instance variable marked in the source code
CONCRETE DECORATOR	derived	Any subclass of a DECORATOR
DECORATING IMPLEMENTATION	derived	Any implementation in a DECORATOR or CONCRETE DECORATOR of an OPERATION

We could get by with even fewer defining roles by for example demanding that the DECORATED COMPONENT variable must have a specific name, or even that the COMPONENT and DECORATOR must have specific names. However, in our opinion that would be too restrictive to the programmer.

In general, the use of derived roles ties the pattern closer to a specific implementation. For example, the fact that we view all methods in COMPONENT as playing the OPERATION role rules out the possibility for the programmer to have some methods in COMPONENT which are outside of the DECORATOR pattern. Another example is CONCRETE COMPONENT which here is defined as any subclass to DECORATOR. This forces all subclasses of DECORATOR to be implemented as concrete (instantiable) classes where DECORATING IMPLEMENTATIONS must exist for all OPERATIONS, i.e. ruling out the possibility of having abstract subclasses to DECORATOR. This is not a severe restriction, because a programmer can easily make DECORATOR and all its subclasses concrete by giving default implementations of all OPERATIONS in DECORATOR. But these two examples illustrate the fact that derived roles reduce the flexibility of the implementation. On the other hand, derived roles reduces the burden on the programmer to explicitly state the pattern roles.

An application program making use of the Decorator pattern is annotated by the defining roles, using attribute comments as in the example below. Since many role names are generic, like COMPONENT and OPERATION, we use the following naming convention: /*= *Pattern_Role* =*/.

```
(*= DECORATORPATTERN_COMPONENT =*)
class VisualComponent {
   void draw() { };
};

class TextView extends VisualComponent {
   void draw() {...};
};

(*= DECORATORPATTERN_DECORATOR =*)
class Decorator extends VisualComponent {
   (*= DECORATORPATTERN_DECORATEDCOMPONENT =*)
   VisualComponent comp;
};

class BorderDecorator extends Decorator {
   void draw() {drawBorder(); comp.draw()}
};
```

Patterns may overlap so that a given class plays different roles in different patterns. This is no problem with this approach, since a class (or method, etc.) may be marked by several attribute comments to indicate its different roles.

There may be several applications of a given pattern in the same program. For example, a complex system might use Decorators both in the window subsystem and in the file handling subsystem. To be able to distinguish different pattern applications, we therefore need to recognize that the different roles are related to each other in specific ways. We have already discussed how derived roles are related to the defining roles. In addition, the defining roles are also related to each other. For example, in the case of the Decorator pattern, a DECORATOR will be a subclass of the COMPONENT, and the DECORATEDCOMPONENT variable will be declared in the DECORATOR class. For other patterns, class roles can be related in other ways. For example, in the Object Adapter pattern [Gamma et al. 1994], the ADAPTER and the ADAPTEE are related by an instance variable and not by subclassing. Here, the ADAPTER would typically be a defining role, and ADAPTEE a derived role.

Pattern applications often cross module boundaries. In particular, it is common that a framework implements some of the roles in a pattern application and that the application program using the framework supplies the other roles. For example, the COMPONENT and the DECORATOR may be part of a window-system framework, whereas the CONCRETE COMPONENTS and the CONCRETE DECORATORS may be part of an application using the framework. We expect language support for design patterns to be particularly beneficial for frameworks since it allows rules for how to use the framework to be formalized and checked automatically.

3.2 Pattern rules

The rules for applying a pattern can be expressed in terms of the pattern roles. The identified roles must be consistent with each other. We refer to rules which express such consistency as *role rules*. For the Decorator pattern we have the following role rules:

- R1: A DECORATOR should be a subclass of a COMPONENT.
- R2: A DECORATOR should have a DECORATEDCOMPONENT variable.
- R3: A DECORATEDCOMPONENT variable must only occur in a DECORATOR.

Note that using defining roles instead of derived roles often leads to more role rules, due to the redundancy introduced. For example, if CONCRETE DECORATOR was a defining role rather than a derived one, we would need an additional role rule to check that each class marked as a CONCRETE DECORATOR is a subclass of a DECORATOR.

If the role rules are satisfied, the pattern application is sufficiently complete to make it possible to go on with checking the collaborations between the different roles. To check that the collaborations occur according to the pattern, we formulate a number of *collaboration rules*. For the Decorator pattern we have the following:

- R4: A CONCRETE DECORATOR must have a DECORATINGIMPLEMENTATION for each OPERATION declared in the COMPONENT. The DECORATINGIMPLEMENTATION may be declared in CONCRETE DECORATOR or in any of its superclasses.
- R5: A DECORATINGIMPLEMENTATION must contain a delegating call to the corresponding OPERATION of its DECORATEDCOMPONENT.

The collaboration rules are usually more interesting than the role rules in that they are more easily broken by mistake, and therefore more interesting to enforce. For example, if we are working with a window system applying the Decorator pattern, it is easy to forget to update the CONCRETE DECORATOR classes with delegating operations each time a new OPERATION in the COMPONENT is added. This error might not show up immediately, since applications which do not make use of the decorating objects will work fine. A system which enforces the pattern rules would detect such errors at compile-time.

4 Specifying a pattern

To specify the roles and rules for a pattern, an extension grammar is written which extends the base grammar interface with attribute declarations and equations defining the attribute values. The appendix A gives an extension grammar for the Decorator pattern.

To support that a program is annotated with defining pattern roles, a *program-defined attribute* is declared for each of these roles. A program-defined attribute is an attribute with a default value which can be overridden by a value given in an *attribute comment* of the form /*= *attribute* = *value* =*/. Because program-defined attributes are often boolean with the default value false, we allow

/*= *attribute* =*/ as a shorthand for /*= *attribute* = true =*/. For example, the comment /*= DecoratorPattern_Component =*/ in the program example in section 3.1 defines the DecoratorPattern_Component attribute for the class VisualComponent as having the value true, whereas for all other classes, the attribute will be false. Section A.1 shows the definition of the program-defined attributes for the Decorator pattern.

The derived roles are represented by boolean synthesized attributes and computed with the help of the defining roles. For internal (non-program-defined) attributes the names are local to the extension grammar, and need not be prefixed by the pattern name. In order to avoid using the long names internally for the defining role attributes, it is useful to define additional synthesized attributes also for them. Thus, the extension grammar specifies a boolean synthesized attribute for each of the Decorator pattern roles, as shown in section A.2. The specification of the defining role attributes is trivial. For example, the Component attribute is equal to the DecoratorPattern_Component attribute. The specification of the derived attributes is more complex, often making use of recursive attributes. For example, the definition of ConcreteDecorator uses the ConcreteDecorator attribute of the superclass.

To check rules we make use of *error attributes* which are string-valued synthesized attributes. When checking a program for rule violations, all error attributes are evaluated and for those with a non-empty string value, the string is printed as an error message. Thus, an error attribute should be defined as the empty string if there is no error, and as an appropriate error message otherwise. Section A.3 shows the specification of the rules for the Decorator pattern.

5 Concluding discussion

The language support for design patterns we have outlined here supports both the identification of patterns in source code (traceability), and automatic checkability that the patterns are applied consistently, according to given rules. The technique requires that the programmer annotates the source code with some defining pattern roles. For our formalization of the Decorator pattern, the programmer needs to give three such annotations, whereas all other roles (potentially quite a large number since there may be any number of CONCRETE DECORATORS and CONCRETE COMPONENTS), can be derived automatically.

We find it reasonable that the programmer explicitly annotates the program to define which patterns are applied, rather than trying to recognize pattern applications from the source code only. The reason for this is that several patterns have similar structures, and it is only the programmer who can know what pattern is intended to be used. Furthermore, if the programmer by mistake breaks the pattern rules, it may be impossible to recognize the pattern if there are no annotations.

The use of formalized patterns naturally has a cost in flexibility: Any pattern formalization will pin down precise rules for the pattern, and it might be difficult to foresee all reasonable implementation variations of the pattern. However, just as standardized libraries and frameworks are developed for a given implementation language, we might expect standardized pattern specifications to be developed for a given language.

We expect one of the most beneficial uses of formalized patterns to be for frameworks where the framework supplies some roles, and the application program the other roles. It is then very important that the application program follows the pattern rules, because otherwise the complete system may fail. It is well known that frameworks are difficult to use and that application programmers need to have detailed knowledge of the framework implementation in order to be able to use it correctly. A solution to this problem could be to formulate the requirements on using the framework in terms of patterns, and to formalize and enforce these pattern rules as discussed in this paper.

In this paper we have only discussed the Decorator pattern, and the question naturally arises to what extent other patterns can be described using the same technique. Although Decorator is the only example we have worked through in detail so far, we expect that other patterns can be treated in the same way. All of the patterns in [Gamma et al. 1994] use roles which would benefit from being traceable in the code. Many of the patterns in addition have collaboration rules similar to the ones we listed for Decorator, and which would be useful to enforce. For example, in the Observer pattern any change to the SUBJECT state should lead to a NOTIFY call; in the Visitor pattern, each VISITOR class should have a VISITCONCRETEELEMENT operation for each CONCRETEELEMENT class; etc.

Attribute extension is a general and powerful technique for describing compile-time rules like the ones we have discussed here for patterns. As discussed in [Hedin 1997] it is more powerful than some other related techniques for describing compile-time rules like Meyers' CCEL language [Meyers et al. 1993] and Minsky's Law-Governed Architecture [Minsky 1995]. However, when applied to a particular problem like the formalization of design patterns, one could imagine developing a more specialized notation, more specifically oriented towards the support of design patterns, making the specification both shorter and more readable. One idea which might prove fruitful would be to construct a projected syntax tree containing only the roles of the pattern, and casting these roles into an abstract syntax specific to the design pattern. I.e., each role would be a syntactic category in that language. In effect, this would map the roles and the role rules into a syntax of the design pattern, and the collaboration rules into a static-semantics for it.

References

[Bosch 1997]
Bosch, J. Design Patterns as Language Constructs. To appear in *Journal of Object-Oriented Programming*.

[Buschmann et al. 1996]
Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. *Pattern-Oriented Software Architecture. A System of Patterns*. Wiley. 1996.

[Gamma et al. 1994]
Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[Hedin 1997]
Hedin, G. Attribute Extension - a Technique for Enforcing Programming Conventions. *Nordic Journal of Computing* 4 (1997), 93-122.

[Meyers et al. 1993]
Meyers, S., Duby, C. K., and Reiss, S. P. Constraining the Structure and Style of Object-Oriented Programs. *Proceedings of the Workshop on Principles and Practice of Constraint Programming*, April 1993.

[Minsky 1995]
Minsky, N. H. Law-Governed Regularities in Object Systems; Part 1: an Abstract Model. *TAPOS* 4, 1, 1995.

[Soukup 1995]
Soukup, J. Implementing Patterns. In *Pattern Languages of Program Design*, Coplien, Schmidt (eds), 395-412. Addison-Wesley. 1995.

Appendix A: Extension grammar for Decorator

A.1 Defining roles

To support the annotations of the defining roles COMPONENT, DECORATOR, and DECORATEDCOMPONENT, we declare three program-defined attributes as follows.

```
addto ClassDecl {
   progdef boolean DecoratorPattern_Component = false;
   progdef boolean DecoratorPattern_Decorator = false;
}
addto VarDecl {
   progdef boolean DecoratorPattern_DecoratedComponent = false;
}
```

A.2 Role attributes

Role COMPONENT

```
addto ClassDecl {
   syn boolean Component = DecoratorPattern_Component;
}
```

Role DECORATOR

```
addto ClassDecl {
   syn boolean Decorator = DecoratorPattern_Decorator;
}
```

Role DECORATEDCOMPONENT

```
addto VarDecl {
   syn boolean DecoratedComponent =
      DecoratorPattern_DecoratedComponent;
}
```

Role CONCRETEDECORATOR

```
addto ClassDecl {
   syn boolean ConcreteDecorator =
      if superClassBinding == null
```

```
        then false
        else
          superClassBinding.Decorator or
          superClassBinding.ConcreteDecorator;
    }
```

Role ConcreteComponent

```
    addto ClassDecl {
      syn boolean ConcreteComponent =
        if superClassBinding == null then false
        else if superClassBinding.Decorator then false
        else
          superClassBinding.Component or
          superClassBinding.ConcreteComponent
    }
```

Role Operation

The definition of the Operation role below makes use of the **enclosing**(*T*) construct which returns the closest ancestor in the syntax tree of type *T*.

```
    addto MethodDecl {
      syn boolean Operation = enclosing(ClassDecl).Component
    }
```

Role DecoratingImplementation

To be able to define the derived role DecoratingImplementation, we need to find the Component, given a Concrete Decorator. This is done by an additional attribute, ComponentOfAnyDecorator, which returns the Component of a Decorator or a Concrete Decorator.

```
    addto ClassDecl {
      syn ClassDecl ComponentOfAnyDecorator =
        if Decorator then
          if superclassBinding == null then null
          else if superclassBinding.Component then superclassBinding
          else null;
        else if ConcreteDecorator
          then superclassBinding.ComponentOfAnyDecorator
        else null;
    }
```

With the above attribute, the final role DecoratingImplementation can be defined. This definition makes use of the construct

match *list* **for** *tree*

which returns the set of syntax subtrees in *list* which match the structure given in *tree*.

```
addto MethodDecl {
   syn boolean DecoratingImplementation =
      if enclosing(ClassDecl).Decorator or
         enclosing(ClassDecl).ConcreteDecorator
      then
         let comp = enclosing(ClassDecl).ComponentOfAnyDecorator
         in if comp == null then false
            else
               cardinal (
                  match comp.DeclList
                  for MethodDecl(ReturnType, MethodId, FormalParamList, *)
               ) = 1
      else false
}
```

A.3 Rules

Rule 1: A DECORATOR should be a subclass of the COMPONENT

This can be checked by an error attribute as follows:

```
addto ClassDecl {
   error string missingComponentRole =
      if DecoratorPattern_Decorator and
         superclassBinding() != null and
         not superclassBinding().Component
      then "Decorator pattern: Missing Component role for Decorator."
      else "";
}
```

Rule 2: A DECORATOR should have a DECORATEDCOMPONENT variable

We first define an attribute DecoratedComponentOfDecorator which computes the DECORATEDCOMPONENT (if it exists) for a DECORATOR class. This definition uses a more advanced version of the match operator with the following form:

match *list* **for** *x*: *tree* **with** *boolean-expression*

This construct searches the *list* for subtrees matching the structure of *tree*, and if such a tree *x* is found it is included in the resulting set only if *boolean-expression* holds, where *boolean-expression* may access attributes in *x*.

```
addto ClassDecl {
   syn VarDecl DecoratedComponentOfDecorator =
      if Decorator
      then
```

```
    let dc =
      match DeclList
      for v: VarDecl(ComponentOfDecorator, *)
      with v.DecoratedComponent
    in
      if cardinal(dc) = 1
      then dc(1)
      else null
  else null
}
```

The specification of the rule is now straight-forward:

```
addto ClassDecl {
  error string missingDecoratedComponent =
    if Decorator and DecoratedComponentOfDecorator == null
    then "Decorator Pattern: Missing correct declaration of
DecoratedComponent"
    else "";
}
```

Rule 3: A DECORATEDCOMPONENT variable must only occur in a DECORATOR.

This rule says that if a variable is marked as a DECORATEDCOMPONENT, it must be an instance variable declared in a DECORATOR.

```
addto VarDecl {
  error string misplacedDecoratedComponent =
    if  DecoratedComponent and
      enclosing(Declaration) == enclosing(ClassDecl) and
      enclosing(ClassDecl).Decorator
    then ""
    else "Decorator Pattern: Misplaced DecoratedComponent";
}
```

Rule 4: A CONCRETEDECORATOR must have a DECORATINGIMPLEMENTATION for each OPERATION declared in the COMPONENT. The DECORATINGIMPLEMENTATION may be declared in CONCRETEDECORATOR or in any of its superclasses.

To check this we first compute the complete set of signatures for DECORATINGIMPLEMENTATIONS in a CONCRETEDECORATOR (including those declared in superclasses). This is done by computing the local set of DECORATINGIMPLEMENTATIONS, then extracting the signature information, and finally merging with the signature set of the superclass.

```
addto ClassDecl {
  syn set of (ReturnType, MethodId, FormalParamList)
    DecoratingImplementationSignatures =
      if ConcreteDecorator or Decorator
```

```
        then
          let localSignatureSet =
            extract (ReturnType, MethodId, FormalParamList)
            from
              match DeclList
              for m:MethodDecl
              with m.DecoratingImplementation
          in
            if Decorator
            then localSignatureSet
            else
              localSignatureSet union
              superclassBinding.DecoratingImplementationSignatures
        else empty
  }
```

We also need to compute the set of signatures for the OPERATIONS in a COMPONENT :

```
  addto ClassDecl {
    syn set of (ReturnType, MethodId, FormalParamList)
      OperationSignatures =
        if Component
        then
          extract (ReturnType, MethodId, FormalParamList)
          from
            match DeclList for m.MethodDecl with m.Operation
        else empty
  }
```

The rule can now be checked by taking the relative complement of these sets using the **diff** operator .

```
  addto ClassDecl {
    error string missingDecoratingImplementation =
      if ConcreteDecorator
      then
        let missingOps =
          ComponentOfDecorator.OperationSignatures diff
          DecoratingImplementationSignatures
        in
          if missingOps == empty
          then “”
          else “Decorator Pattern: Missing implementations of operations ” +
            missingOps + “ in ConcreteDecorator”;
      else “”
  }
```

Rule 5: A DECORATINGIMPLEMENTATION must contain a delegating call to the corresponding OPERATION of its DECORATEDCOMPONENT

We first need to find the DECORATEDCOMPONENT of a CONCRETE DECORATOR. This is done by the following recursive attribute:

```
addto ClassDecl {
   syn VarDecl DecoratedComponentOfConcreteDecorator =
      if ConcreteDecorator
      then
         if superclassBinding.Decorator
         then superclassBinding.DecoratedComponentOfDecorator
         else superclassBinding.DecoratedComponentOfConcreteDecorator
      else null
}
```

The rule can now be checked by using the match operator to find delegated calls in the method body as follows.

```
addto MethodDecl {
   error string missingDelegation =
      if DecoratingImplementation
      then
         let concDec =
            enclosing(ClassDecl).DecoratedComponentOfConcreteDecorator
         in
            let delegatingCalls =
               match methodBody.StmtList
               for MethodCall(concDec.MethodId, MethodId, *)
            in
               if cardinal(delegatingCalls) = 0
               then
                  "Decorator Pattern: Missing delegating call."
               else ""
      else ""
}
```