In Bosch and Mitchell (Eds.): Object-Oriented Technology. ECOOP'97 Workshop Reader, pp137-140. LNCS 1357, Springer Verlag, 1997.

Language Support for Design Patterns Using Attribute Extension

Görel Hedin

Dept. of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
Gorel.Hedin@dna.lth.se

Abstract This paper presents a technique based on attribute grammars for formalizing design pattern solutions. 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 design pattern solutions 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 can be formalized and form the basis 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 of the pattern. For example, the Decorator pattern uses a clever way of adding functionality to Component objects: a Decorator object “wraps” a Component, i.e., all clients of the Component refer to the wrapping Decorator instead of directly to the Component. The Decorator forwards all messages to its Component in addition to possibly adding some behavior of its own. An application of the Decorator pattern implies some rules, for example that whenever a new operation in introduced on Components, each Decorator 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.

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. For this reason, it is not sufficient to support a number of predefined patterns by built-in 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 a] to support the specification and application of design patterns. Attribute extension allows the static-semantics of a language to be extended, allowing programming conventions to be enforced, but 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.

Attribute extension 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: 1) 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. 2) 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. 3) *Attribute comments*, which are special comments used to annotate an application program.

In the following we give an overview over how the attribute extension technique can be used to formalize design pattern solutions. For a detailed example and a more thorough discussion of the technique, the reader is referred to the full paper [Hedin 1997 b].

2 Language support for a pattern

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

2.1 Pattern roles

The most important roles are played by classes, but some roles may also be played by methods or variables. For example, in the Decorator pattern [Gamma et al. 1994], we can identify the following roles: COMPONENT (class), OPERATION (method), CONCRETECOMPONENT (class), DECORATOR (class), DECORATEDCOMPONENT (variable), CONCRETEDECORATOR (class), DECORATINGIMPLEMENTATION (method).

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 DECORATEDCOMPONENT roles. We call these roles the *defining roles*. Other roles are called *derived roles*. It is possible to formalize the pattern using more or fewer defining roles, but too few defining roles may give a too restrictive implementation (for example relying on naming conventions of classes), and too many defining roles may put an unnecessary burden on the programmer to explicitly identify the roles.

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 CONCRETECOMPONENTS and the CONCRETEDECORATORS 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.

2.2 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 example, in our formalization of the Decorator pattern we will have a role rule stating that a DECORATOR must be a subclass of a COMPONENT.

If the role rules are satisfied, the pattern application is sufficiently complete to make it possible to go on with checking *collaboration rules*. For example, to formalize the Decorator pattern we could have a collaboration rule stating that 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.

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 CONCRETEDECORATOR 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.

2.3 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. For each defining role, a special "program-defined" attribute is declared, whose default definition can be overridden by an attribute comment. A derived role is represented by a synthesized attribute. Finally, the role rules and collaboration rules are represented by special string-valued "error" attributes. The value of an error attribute is defined as an error message if the rule is violated, and the empty string if the rule is satisfied. To allow the definition of these attributes, additional synthesized and inherited attributes may be introduced.

3 Conclusion

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.

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 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. An aid in solving 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.

References

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

[Hedin 1997 b]
Hedin, G. Language Support for Design Patterns using Attribute Extension. In Proceedings of LSDF'97, *Workshop on Language Support for Design Patterns and Frameworks*, held in connection to ECOOP'97. Full paper available electronically at http://www.ide.hk-r.se/~bosch/lsdf/.