Synthesizing Object-Oriented and Functional Design to Promote Re-use*

Shriram Krishnamurthi, Matthias Felleisen, and Daniel P. Friedman**

Department of Computer Science
Rice University

Abstract. Many problems require recursively specified types of data and a collection of tools that operate on those data. Over time, these problems evolve so that the programmer must extend the toolkit or extend the types and adjust the existing tools accordingly. Ideally, this should be done without modifying existing code. Unfortunately, the prevailing program design strategies do not support both forms of extensibility: functional programming accommodates the addition of tools, while object-oriented programming supports either adding new tools or extending the data set, but not both. In this paper, we present a composite design pattern that synthesizes the best of both approaches and in the process resolves the tension between the two design strategies. We also show how this protocol suggests a new set of linguistic facilities for languages that support class systems.

1 Evolutionary Software Development

Programming practice frequently confronts programmers with the following design dilemma. A recursively defined set of data must be processed by several different tools. In anticipation of future extensions, the data specification and the tools should therefore be implemented such that it is easy to

1. add a new variant of data and adjust the existing tools accordingly, and
2. extend the collection of tools.

Ideally, these extensions should not require any changes to existing code. For one, source modification is cumbersome and error-prone. Second, the source may not be available for modification because the tools are distributed in the form of object code. Finally, it may be necessary to evolve the base program in several different directions, in which case code modifications are prohibitively expensive because the required duplication would result in duplicated maintenance costs.

This dilemma manifests itself in many different application areas. A particularly important example arises in the area of programming languages. Language

* This research was partially supported by NSF grants CCR-9619756, CCR-9633109, CCR-9708957 and CDA-9713032, and a Texas ATP grant.
** Permanent address: Computer Science Department, Indiana University.
grammars are typically specified via BNFs, which denote recursively defined data sets. Language-processing tools recursively traverse sentences formed from the grammar. In this scenario, a new form of data means an additional clause in the BNF; new tools must be able to traverse all possible elements of the (extended) grammar. Unfortunately, prevailing design strategies do not accommodate the required evolution:

- The "functional" approach, which is often realized with conventional procedural languages, implements tools as procedures on recursive types. While this strategy easily accommodates the extension of the set of tools, it requires significant source modifications when the data set needs to be extended.
- The (standard) "object-oriented" approach defines a recursive set of data with a collection of classes, one per variant (BNF clause), and places one method per tool in each class. In the parlance of object-oriented design patterns [13], this approach is known as the Interpreter pattern. The problem it poses is dual to the problem of the functional approach: variants are easy to add, while tool additions require code modifications.
- If the collection of tools is large, the designer may also use the Visitor pattern, a variant of the Interpreter pattern, which collects the code for a tool in a single class. Roughly speaking, the Visitor pattern emulates the functional approach in an object-oriented setting. As a result, it suffers from the same problem as the functional approach.

In short, the two design styles suffer from a serious problem. Each style accommodates one form of extension easily and renders the other nearly impossible.¹

This paper presents the Extensible Visitor pattern, a new composite design pattern [28], which provides an elegant solution to the above dilemma. The composite pattern is a combination of the Visitor and Factory Method patterns. Its implementation in any class-based object-oriented programming language is straightforward. In addition, the paper introduces a linguistic abstraction that facilitates the implementation of the Visitor and Extensible Visitor patterns. The abstraction syntactically synthesizes the best of the functional and the object-oriented design approaches. Using the abstraction, a programmer only specifies the necessary pieces of the pattern; a translator assembles the pattern implementation from these pieces. We consider this approach a promising avenue for future research on pattern implementations.

Section 2 introduces a simple example of the design dilemma and briefly discusses the functional approach and the standard object-oriented approach (based on the Interpreter pattern) to extensible software. Section 3 analyzes the problems of the Visitor pattern and then develops the Extensible Visitor pattern in the context of the same running example. Section 4 describes some of the type-checking issues that arise when using this pattern. Section 5 presents a

¹ Cook [4] devotes his tutorial to this problem, which was first anticipated by Reynolds [27].