OO Overkill: When Simple is Better than Not [1]

Owen Astrachan
Computer Science Department
Duke University
ola@cs.duke.edu

Abstract

Object oriented design patterns as popularized in [GHJV95] are intended to solve common programming problems and to assist the programmer in designing and developing robust systems. As first year courses increasingly emphasize object orientation, knowledge of design patterns and when to use them becomes an important component of the first year curriculum. Recent literature has focused on introducing the patterns to computer science educators, but not on the situations and contexts in which the patterns are appropriate. Design patterns and object orientation are parts of a methodology that scales to large systems. In this paper we show that these concepts do not always scale down. We analyze examples from current literature that would be simpler without patterns, and provide examples of when the same design patterns do make design and programs simpler.

1 Introduction

Object oriented design patterns as described in [GHJV95] have received a great deal of attention from practicing software designers and engineers, from academics in software engineering, and from computer science educators [Wal96, ABCM98, AW98, GGT98, Ngu98, NW99, NW00].

Software practitioners find these patterns a significant tool in developing and maintaining code. Since we started using the "Gang of Four" book [GHJV95] in 1996, the single common theme that students in our program report from job interviews is finding a copy of it on the desk of someone with whom they interview. The design patterns in this book are meant to help develop correct, re-usable, and maintainable programs. Perhaps the most important part of a written pattern is the forces that describe when the pattern may be applicable. Unfortunately, some of the recent work investigating patterns early in the curriculum [GGT98, Ngu98, NW99, NW00, NW01] has ignored the forces or any discussion of whether a pattern is appropriate in the context in which it is applied.

We do hope that this recent work will succeed in introducing design patterns to the community chiefly responsible for teaching program design and programming. At the same time we worry that many educators new to object orientation will not see the utility of patterns since the examples in this recent work are often made more complicated when patterns are used. In our experience, finally understanding these patterns after much reading and thinking causes a profound "aha and eureka" effect. To paraphrase "give someone a hammer and the whole world is a nail," we see "show someone the composite pattern and the whole world requires a polymorphic visitor."

Our primary motivation in introducing design and programming concepts is to instill best practices that are as simple as possible. There is often a tension between these goals. In this paper we report on an ongoing project to introduce design patterns into curricula and on some of the problems we have encountered. We believe strongly that design patterns are an essential part of any programming curriculum and we show contexts in which they yield good designs and programs. We also report on examples from the literature and in our classroom that are better and more simply solved without design patterns.

We first provide background on how beginning students understand advanced concepts. We then show some standard design patterns applied to problems from recent literature and how these problems may be better

[1] This work was supported by NSF grant CAREER 9702550.

solved by a more simple approach. Finally, we introduce a problem whose solution is facilitated by applying these same design patterns.

2 Novices Work with Expert Code

Until last year we used a 20-questions program with a design from [Ast98] that uses the Composite and Factory design patterns. The intent was to supply students with a design that exemplifies best practice and ask them to implement the design. Our students succeeded in implementing the design, but did not internalize the benefits that the design patterns—they could not apply the patterns in a new context. Recently we changed the assignment to allow students to develop their own design. Again students succeeded, but they noticed problems in the design and implementation of their code. We then introduced the design patterns as a solution to these struggles. Because students had experienced problems, the solution engendered by applying design patterns made sense and was appreciated. Students were then able to apply the design patterns in a new context.

These experiences are mirrored in literature distinguishing between novice and expert understanding of the programming process [Fle93]. Introducing new patterns to solve a new problem was too much for our second semester students. Instead, the *before and after* approach of applying a pattern to a solved problem was much more successful.[1] The code that results from applying patterns is simpler to someone understanding the patterns, but not to a novice programmer learning about the principal data structure of the program: a binary tree.

3 Implementing Binary Trees

Implementing binary trees using object oriented techniques has been covered in several papers[BD96, Ada96, NW99]. These papers show several techniques that are far from simple. In [NW99] the design patterns Composite, Null Object [Woo98], and Visitor make binary trees object oriented, but more difficult to understand than using a simple approach. If students already understand trees, and have covered basic tree implementations in either a functional language or in a more imperative style as shown below, these design patterns may be appropriate. The patterns would introduce a new implementation of a familiar concept, the same *before and after* technique outlined in Section 2. However, as a first introduction to trees the added complexity from using patterns makes the code more difficult to understand and extend than when a simple approach is used. This is the antithesis of of the purpose of object oriented design patterns as "simple and elegant solutions to specific problems in object oriented software design" [GHJV95]. As we show in this paper, there are many problems that are better solved by the appropriate design patterns than when the patterns are not applied. Binary tree implementations as reported in the literature are best coded with simpler techniques.

[1]In hindsight, and after reviewing literature such as [Fle93, KR91], this should have been obvious, but it was not.

3.1 Trees: Simple and Standard

Consider a binary tree node defined using what is called a *plain old data* approach.[2] This basic approach is used in both the Java Generic Library (JGL, `objectspace.com`) and the Java Collections Framework (`java.sun.com`).

```
public class PlainNode
{
    Object info;
    PlainNode myLeft, myRight;
}
```

We define a binary tree as either **empty** or a **three-tuple/node** containing information and left and right subtrees. This leads to simple code to count the number of nodes in a tree and and print the tree with an inorder traversal. Both functions have two cases reflecting the definition of a tree: one case for an empty tree and one for a non-empty tree.

```
public class TreeFunctions
{
  public static int count(PlainNode root)
  {
    if (root == null) return 0;
    return 1 + count(root.myLeft)
           + count(root.myRight);
  }
  public static void print(PlainNode root)
  {
      if (root == null) return;
      print(root.myLeft);
      System.out.println(root.info);
      print(root.myRight);
  }
}
```

Classes for binary trees are diagrammed in Figure 1.

3.2 Critiquing the Simple Approach

The functions in the class `TreeFunctions` have three characteristics that are typically criticized when object orientation is advocated. We elaborate on these criticisms below.

[2]We use Java code in this paper, but the same issues apply when C++ is used.

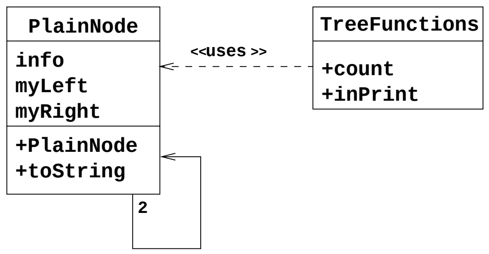

Figure 1: plain old data approach to trees

1. These functions should be methods of the tree-node classes rather than external functions.

2. Using null to indicate an empty tree is not as good as using a *null object* [Woo98].

3. Selecting which of two (or more) cases applies using control flow (the `if` statement) is not as good as using polymorphism to differentiate cases.

These can be valid criticisms of code *in general*, but they cannot necessarily be applied to the code and design for binary trees shown here. The application of a generally sound heuristic to each specific instance of the general problem is called *dicto simpliciter*, a common mistake in logical reasoning.

One tenet of object oriented programming is succinctly described as "ask not what you can do to an object, ask what an object can do to itself." In an ideal design, a binary tree would respond to a `nodeCount` message. However, it is not possible to anticipate every potentially useful function that might be applied to a tree and include that function in the tree-node interface. There is a documented tension in designing a class that is simple to use yet minimally complete [Rie96]. Thus the classes comprising a binary tree framework/module must include some operations on trees, but permit others to be added later. As we'll see, the approach of using static functions in a class `TreeFunctions` is similar to the approach with the Visitor pattern as shown in Section 4.

3.3 Forces and Patterns

Using `null` to represent an empty tree runs counter to an approach espoused in [Car98, Car00] where a *null object* is used. The forces describing when null object is appropriate [Woo98] include:

- Some instances from collaborating classes should respond to messages by doing nothing.

- The client does not have to check for nil or some other special value.

In the design outlined in Section 4 the null object class *EmptyNode* does not respond to messages and the corresponding Visitor class must include a check for the null object. In implementing binary trees the forces for null object do not apply.

As we see in the next section, the `if` statements distinguishing between empty and non-empty are not the kinds of selection statement eliminated by polymorphism. The Visitor pattern does select polymorphically, but it cannot be used unless the classes being visited are stable, that is no new classes are added to the hierarchy of visitable classes. In such a situation selecting by an if statement is nearly the same as using the Visitor pattern, both are subject to the same limitations.

4 A Principled but Flawed Approach to Implementing Trees with Patterns

In this section we discuss an implementation of binary trees using patterns from [NW99].[3] The resulting code is certainly elegant from an object oriented aesthetic, but it is more difficult to understand and extend than the simple single-node approach outlined in Section 3.

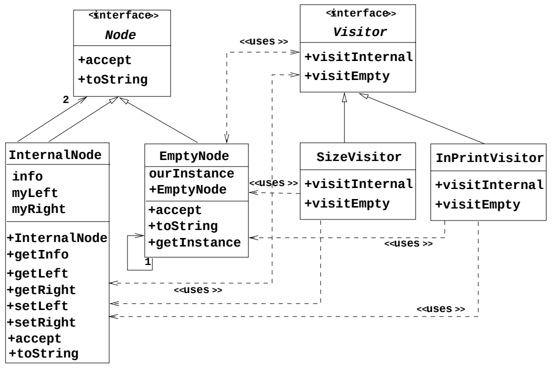

Figure 2: composite/visitor approach to trees

Our purpose in this paper is not to explain in detail how these patterns work together to implement binary trees (e.g., see [NW99] for this), but to show that this approach is not appropriate for most courses when binary trees are introduced. A simple glance at the class diagram in Figure 2 shows that the classes and their inter-relationships are much more complex than those of the plain old data approach.

4.1 The Visitor Pattern

The Visitor pattern has received attention via recent publications [NW99, NW00] and at workshops on CS2

[3]The complete code for this example is on the website that supplements this paper: `http://www.cs.duke.edu/csed/patterns`. It uses Composite, Singleton, Visitor, and Null Object patterns.

and the first year of computer science [Car98, Car00]. From [GHJV95] we find the following forces indicating the pattern may be useful.

- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
- The classes defining the object structure rarely change, but you often want to define new operations over the structure.

In the case of a binary tree there are only two classes: Empty and Internal/NonEmpty. A node does not respond to messages other than to access state since tree operations have been factored into Visitor classes. For binary trees many visitors are stateless, but for generality Visitor methods cannot be static so clients are forced to instantiate objects simply to pass a message and receive a returned value (e.g., for the size of a tree as shown below.)

Contrast the Visitor code below for counting nodes in a tree with the three-line function in Section 3.

```
public Object visitInternal(InternalNode node)
{
 Integer lcount =
    (Integer) node.getLeft().accept(this);
 Integer rcount =
    (Integer) node.getRight().accept(this);
 return new Integer(1 + lcount.intValue()
                      + rcount.intValue());
}
```

Figure 3 provides an interaction diagram for counting the number of nodes in a binary tree using a Visitor pattern when the tree has one non-empty node.

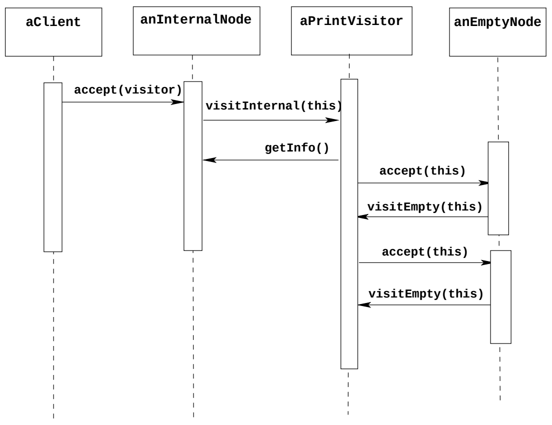

Figure 3: printing a single-node tree

Introducing binary trees using visitors is certainly overkill, visitors are more appropriately used with more complex tree-like structures as shown in Section 5.

5 A Compelling Example

The website of material used in this project contains a running example showing design patterns used in an implementation of the game Hangman. In this section we briefly discuss a more complex example that uses the Visitor pattern described earlier. Consider evaluating boolean expressions as part of implementing a toy programming language or simulating electronic circuits and gates.

Figure 4 shows an inheritance hierarchy for classes that build boolean expressions. The classes naturally illustrate the Composite design pattern where, for example, an **and expression** consists of two boolean expressions.

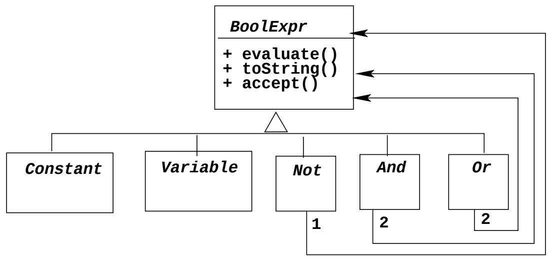

Figure 4: composite/visitor and boolean expressions

In this example, each class is responsible for evaluating itself in a context that assigns boolean values to variables (e.g., x = true).

```
class Not extends BoolExpr
{
  // some methods not shown
  public boolean evaluate(Context c)
  {
    return ! myExpr.getValue(c);
  }
  private BoolExpr myExpr
}
```

The interface for a *BoolExpr* is minimal, but the responsibility for evaluating an expression could have been factored into a visitor. Because the interface is minimal, a visitor is required, for example, to substitute an expression for a variable in a larger expression. Consider, for example, substituting `(x AND y)` for `z` in `(z AND w) or (!z AND y)` yielding `((x AND y) AND w) or (!(x AND y) AND y)`. A visitor is appropriate since the responsibility for replacing a variable in each kind of expression will be in a function associated with the expression, but residing in a visitor. It's difficult to completely understand this function without seeing all the code,[4] but the flavor for the simplicity comes through.

```
public class Replacer implements ExprVisitor
{ // methods not shown
  public Object visitOr(OrExpr or, Object o)
  {
    return new OrExpr(
       (BoolExpr) or.getLeft().accept(this,o),
       (BoolExpr) or.getRight().accept(this,o));
  }
```

6 Summary

Our project to introduce design patterns has had some success. Our greatest successes arise from showing a solution that uses patterns to a problem students have already solved. Design patterns and object orientation do not always scale down, care in choosing examples will help ensure that educators and students appreciate the power of design patterns.

References

[ABCM98] Owen Astrachan, Geoffrey Berry, Landon Cox, and Garrett Mitchener. Design patterns: An essential component of cs curricula. In *Twenty-ninth SIGCSE Technical Symposium on Computer Science Education*, pages 153–160. ACM Press, February 1998.

[Ada96] Joel Adams. Knowing your roots: Object-oriented binary search trees revisited. *SIGCSE Bulletin*, 28(4):36–40, 1996.

[Ast98] Owen Astrachan. Twenty questions/animal game. SIGCSE Symposium Presentation, 1998. `http://www.cs.duke.edu/csed/patterns/animal/`.

[AW98] Owen Astrachan and Eugene Wallingford. Loop patterns. In *Proceedings of PLoP: Pattern languages of Programming*, August 1998. Allerton Park, IL.

[BD96] A. Michael Berman and Robert Duvall. Thinking about binary trees in an object oriented world. In *Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, pages 185–189. ACM Press, 1996.

[Car98] Robert (Corky) Cartwright. Design patterns in cs2. OOPSLA Workshop on CS2, 1998.

[4]The code is available on the website that accompanies the paper.

[Car00] Robert (Corky) Cartwright. Using design patterns early. FYI 2000: Workshop on First Year Instruction, 2000.

[Fle93] Ann Fleury. Student beliefs about pascal programming. *Journal of Educational Computing Research*, 9:355–371, 1993.

[GGT98] Natasha Gelfand, Michael T. Goodrich, and Roberto Tamassia. Teaching data structures design patterns. In *Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education.* ACM Press, 1998.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[KR91] J. Koenemann and S. Robertson. Expert problem solving strategies for program comprehension. *CHI Proceedings*, pages 125–130, 1991.

[Ngu98] Dung Nguyen. Design patterns for data structures. In *Twenty-Ninth SIGCSE Technical Symposium on Computer Science Education*, pages 336–340. ACM Press, 1998.

[NW99] Dung Nguyen and Stephen B. Wong. Patterns for decoupling data structures and algorithms. In *Thirtieth SIGCSE Technical Symposium on Computer Science Education*, pages 87–91. ACM Press, 1999.

[NW00] Dung Nguyen and Stephen B. Wong. Design patterns for lazy evaluation. In *Thirty-First SIGCSE Technical Symposium on Computer Science Education*, pages 21–25. ACM Press, 2000.

[NW01] Dung Nguyen and Stephen B. Wong. Design patterns for sorting. In *Thirty-first SIGCSE Technical Symposium on Computer Science Education.* ACM Press, 2001.

[Rie96] Arthur Riel. *Object-Oriented Design Heuristics.* Addison-Wesley, 1996.

[Wal96] Eugene Wallingford. Toward a first course based on object-oriented patterns. In *Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, pages 27–31. ACM Press, 1996.

[Woo98] Bobby Woolf. The null object pattern. In Robert Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3.* Addison-Wesley, 1998.