Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 59 (2006) 147–169
Science of Computer Programming
www.elsevier.com/locate/scico

Using classic problems to teach Java framework design

H. Conrad Cunningham[a,*], Yi Liu[a], Cuihua Zhang[b]

[a] *Department of Computer and Information Science, University of Mississippi, 201 Weir Hall, University, MS 38677, USA*
[b] *Department of Computer and Information Systems, Northwest Vista College, AB 135, San Antonio, TX 78251, USA*

Received 4 October 2004; received in revised form 22 February 2005; accepted 21 March 2005
Available online 8 August 2005

Abstract

All programmers should understand the concept of software families and know the techniques for constructing them. This paper suggests that classic problems, such as well-known algorithms and data structures, are good sources for examples to use in a study of software family design. The paper describes two case studies that can be used to introduce students in a Java software design course to the construction of software families using software frameworks. The first is the family of programs that use the well-known divide and conquer algorithmic strategy. The second is the family of programs that carry out traversals of binary trees.

Keywords: Software family; Software framework; Hot spot; Design pattern; Divide and conquer; Tree traversal

1. Introduction

In a classic paper [17] David Parnas observes, "Variations in application demands, variations in hardware configurations, and the ever-present opportunity to improve a program means that software will *inevitably* exist in many versions". Parnas proposes that development of a program should therefore be approached as the development of a whole

* Corresponding author. Tel.: +1 662 915 5358; fax: +1 662 915 5623.
E-mail address: cunningha@cs.olemiss.edu (H.C. Cunningham).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2005.07.009

family of related programs. He defines a program family as a set of programs "whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members". If programmers can recognize and exploit these "common aspects and predicted variabilities" [24], the resulting software can be constructed to reuse code for the common parts and to enable convenient adaptation of the variable parts for specific circumstances. In a 2001 article [18], Parnas observes that there is "growing academic interest and some evidence of real industrial success in applying this idea," yet "the majority of industrial programmers seem to ignore it in their rush to produce code". He warns [18], "if you are developing a family of programs, you must do so consciously, or you will incur unnecessary long-term costs". If software families are to become pervasive, future industrial programmers (i.e., students) need to learn to design and construct them effectively. This is an important challenge for computing science and software engineering curricula.

How can we respond to this challenge within a college course? The general form of software family is called a *software product line*. A software product line is "a collection of systems sharing a managed set of features constructed from a common set of core software assets" [1]. These assets include a common software architecture shared by the products and a set of reusable software components [10]. Software product lines in their full generality are difficult to teach in the setting of a college course because their design may require extensive knowledge of the application domain and use of special-purpose languages and tools [24]. However, the form of software family called a *software framework* is more accessible. A framework is essentially the reusable skeleton of a software product line implemented entirely in an object-oriented programming language. The common aspects are expressed by a set of abstract and concrete "classes that cooperate closely with each other and together embody a reusable solution" [2] to problems in the application domain. The framework can be customized to a specific member of the family by "plugging in" appropriate subclasses at the supported points of variability. Frameworks are more accessible to students because the techniques build upon standard object-oriented concepts that students are taught in undergraduate courses.

How can we introduce students to the concept of software frameworks? Some advocate that teaching of frameworks be integrated into the introductory computing science sequence. For example, they might be used to introduce a generalization of sorting algorithms [15] or to provide a new approach to teaching the standard introductory data structures material [23]. They might also provide interesting programming examples and exercises to reinforce object-oriented programming concepts and introduce design patterns into the introductory sequence [13,14,16]. Some textbooks use standard Java libraries such as the Collections, Swing, and input/output frameworks and case studies such as drawing pads as examples to illustrate the concepts and techniques [12].

There are at least four levels of understanding of software frameworks that students need to develop. First, because frameworks are normally implemented in an object-oriented language such as Java, students must understand the applicable language concepts, which include inheritance, polymorphism, encapsulation, and delegation. Second, they need to understand the framework concepts and techniques sufficiently well to use frameworks to build their own custom applications. Third, students should be able to do detailed design and implementation of frameworks for which the common and variable aspects

are already known. Fourth, they need to learn to analyze a potential family, identify its possible common and variable aspects, and evaluate alternative framework architectures.

In teaching the framework concepts, instructors must devise appropriate case studies. They wish to use several interesting and realistic, but well-focused, examples and exercises to illustrate the framework techniques. However, building a good framework requires an extensive understanding of the application domain addressed by the framework. Because students come to a course with diverse backgrounds and experiences, it may take considerable time for students to come to a sufficient understanding of an application domain to design a framework. This paper takes the view that various classic problems, such as standard algorithms and data structures, are useful in introducing framework concepts and programming techniques when little time is available to spend on the domain analysis. This approach might be used in a dedicated course on software families [5,7] or in teaching modules within an advanced Java programming or software design course.

Sections 2 and 3 of this paper seek to address aspects of the second and third levels of understanding noted above—teaching the concepts so that students can use an existing framework and so that they can develop their own frameworks given an analysis of the points of commonality and variability in the family. Section 2 introduces the technical concepts and techniques for construction and use of frameworks. Section 3 illustrates these concepts and techniques using a case study that develops a framework for the family of divide and conquer algorithms and applies it to develop a quicksort application [6]. The case study assumes that the students have a basic understanding of object-oriented programming using Java and understand concepts such as inheritance, polymorphism, delegation, recursion, and sorting.

Sections 4 and 5 seek to address aspects of the fourth level of framework understanding—teaching students how to analyze potential families and identify the common and variable aspects. Section 4 introduces the techniques for systematically generalizing an application to discover the points of variability. Section 5 illustrates these concepts using binary tree traversals as the basis for a family [11]. This family of applications includes members ranging from standard preorder, postorder, and in-order traversals to more complicated computations carried out by navigating through binary tree structures in a custom manner.

Section 6 discusses related work, and Section 7 summarizes the paper and gives a few observations about use of the techniques in a college course.

2. Framework construction and use

In beginning programming classes students are taught to focus on a specific problem and write a program to solve that problem. This is appropriate because beginning students need to learn a particular programming language and grasp specific, concrete programming skills. However, as students gain more experience in programming, they should be taught to work at higher levels of abstraction. Instructors need to shift the students' focus to techniques for building a software family.

In building a software family, it is important to separate concerns. We must separate the aspects of the design that are common to all family members from those aspects that are

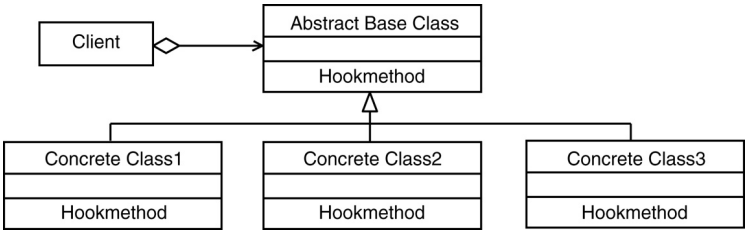

Fig. 1. Hot spot subsystem.

specific to one family member. Furthermore, we must separate the various common and variable aspects from each other and consider them independently, one at a time. We use the terms *frozen spot* to denote a common (or shared) aspect of the family and *hot spot* to denote a variable aspect of the family [22].

A *software framework* is a generic application that allows the creation of different specific applications from a family [21]. It is an abstract design that can be reused within a whole application domain. In a framework, the frozen spots of the family are represented by a set of abstract and concrete base classes that collaborate in some structure. A behavior that is common to all members of the family is implemented by a fixed, concrete *template method* in a base class. A hot spot is represented by a group of abstract *hook methods*. A template method calls a hook method to invoke a function that is specific to one family member.

A hot spot is realized in a framework as a *hot spot subsystem*. A hot spot subsystem typically consists of an abstract base class, concrete subclasses of that base class, and perhaps other related classes [22]. The hook methods of the abstract base class define the interface to the alternative implementations of the hot spot. The subclasses of the base class implement the hook methods appropriately for a particular choice for a hot spot. Fig. 1 shows a UML class diagram of a hot spot subsystem.

There are two principles for framework construction—unification and separation [8]. The *unification principle* uses inheritance to implement the hot spot subsystem. Both the template methods and hook methods are defined in the same abstract base class. The hook methods are implemented in subclasses of the base class. In Fig. 1, the hot spot subsystem for the unification approach consists of the abstract base class and its subclasses. The *separation principle* uses delegation to implement the hot spot subsystem. The template methods are implemented in a concrete context class; the hook methods are defined in a separate abstract class and implemented in its subclasses. The template methods thus delegate work to an instance of the subclass that implements the hook methods. In Fig. 1, the hot spot subsystem for the separation approach consists of both the client (context) class and the abstract base class and its subclasses.

A framework is a system that is designed with generality and reuse in mind; and design patterns [9], which are well-established solutions to program design problems that commonly occur in practice, are the intellectual tools for achieving the desired level of generality and reuse. Two design patterns, corresponding to the two framework construction principles, are useful in implementation of the frameworks.

The *Template Method pattern* uses the unification principle. In using this pattern, a designer should "define the skeleton of an algorithm in an operation, deferring some steps to a subclass" to allow a programmer to "redefine the steps in an algorithm without changing the algorithm's structure" [9]. It captures the commonalities in the template method in a base class while encapsulating the differences as implementations of hook methods in subclasses, thus ensuring that the basic structure of the algorithm remains the same [8].

The *Strategy pattern* uses the separation principle. In using this pattern, a designer should "define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it" [9]. It extends the behavior of a client class by calling methods in another class. The common aspects (template methods) are captured in the concrete methods of the client; the variable aspects (hook methods) are declared in the abstract Strategy class and implemented by its subclasses. The behavior of the client class can thus be changed by supplying it with instances of different Strategy subclasses.

What is the primary difference between the two construction principles in practice? To introduce new behaviors for hook methods, the unification principle requires programmers to implement a new subclass of the base class defining the template methods. This kind of extension by overriding often requires detailed knowledge of the base class, but it is otherwise relatively straightforward for programmers to understand and implement. The unification principle results in efficient but inflexible execution. To introduce new hook method behavior in a framework that uses the separation principle, the client code needs to instantiate an object from a class that has hook methods with the desired behaviors and supply it to the class containing the template methods. If the needed hook behaviors have been implemented previously, then the programmer must just choose an appropriate implementation from the component library. If the needed hook behaviors have not been implemented, then the programmer must implement an appropriate new class. This class is sometimes more difficult to implement than the equivalent unification solution, but its implementation usually requires less knowledge of the internal details of the class containing the template methods. An application of a framework that uses the separation principle may execute slightly less efficiently than a unification-based framework, but separation may enable the application to adapt itself at runtime by merely changing object references [8]. In the next section, we examine a simple software family and consider framework designs based on each of these design principles.

3. Divide and conquer framework

To illustrate the construction and use of a framework, we can use the family of divide and conquer algorithms as an example of a software family. The *divide and conquer* technique solves a problem by recursively dividing it into one or more subproblems of the same type, solving each subproblem independently, and then combining the subproblem solutions to obtain a solution for the original problem. Well-known algorithms that use this technique include quicksort, mergesort, and binary search. Since this algorithmic strategy can be applied to a whole set of problems of a similar type, divide and conquer, in addition

```
function solve (Problem p) returns Solution
{   if isSimple(p)
        return simplySolve(p);
    else
        sp[] = decompose(p);
    for (i= 0; i < sp.length; i = i+1)
        sol[i] = solve(sp[i]);
    return combine(sol);
}
```

Fig. 2. Divide and conquer pseudo-code.

to its meaningful influence in algorithms, serves well the purpose of examining a software family.

The pseudo-code for the divide and conquer technique for a problem `p` is shown in Fig. 2, as it might be presented in an undergraduate algorithms textbook. In this pseudo-code fragment, function `solve()` represents a template method because its implementation is the same for all algorithms in the family. However, functions `isSimple()`, `simplySolve()`, `decompose()`, and `combine()` represent hook methods because their implementations vary among the different family members. For example, the `simplySolve()` function for quicksort is quite different from that for mergesort. For mergesort, the `combine()` function performs the major work while `decompose()` is simple. The opposite holds for quicksort and binary search.

The remainder of this section illustrates the construction and use of a divide and conquer framework. First, we examine how to construct a framework using the unification principle; then we apply this framework to develop an application using the quicksort algorithm. Finally, we look at how the framework can be implemented using the separation principle.

3.1. Constructing a framework using unification

If the unification principle and Template Method pattern are used to structure the divide and conquer framework, then the template method `solve()` is a concrete method defined in an abstract class; the definitions of the four hook methods are deferred to a concrete subclass whose purpose is to implement a specific algorithm.

Fig. 3 shows a design for a divide and conquer framework expressed as a Unified Modeling Language (UML) class diagram. The family includes three members: `QuickSort`, `MergeSort`, and `BinarySearch`. Method `solve()` is a final method in the base class `DivConqTemplate`. It is shared among all the classes. Hook methods `isSimple()`, `simplySolve()`, `decompose()`, and `combine()` are abstract methods in the base class; they are overridden in each concrete subclass (`Quicksort`, `MergeSort`, and `BinarySearch`).

To generalize the divide and conquer framework, we introduce the two auxiliary types `Problem` and `Solution`. `Problem` is a type that represents the problem to be solved by the algorithm. `Solution` is a type that represents the result returned by the algorithm. In Java, we define these types using tag interfaces (i.e., interfaces without any methods)

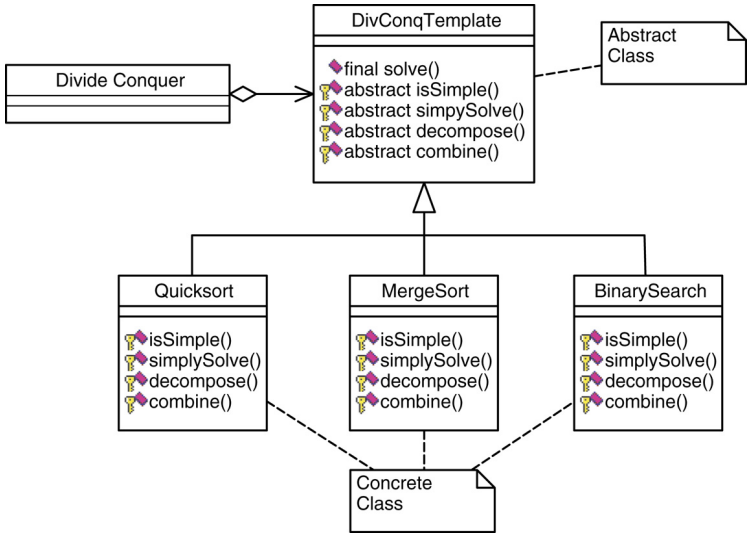

Fig. 3. Template method for divide and conquer.

```
abstract public class DivConqTemplate
{   public final Solution solve(Problem p)
    {   Problem[] pp;
        if (isSimple(p)){ return simplySolve(p); }
        else            { pp = decompose(p); }
        Solution[] ss = new Solution[pp.length];
        for(int i=0; i < pp.length; i++)
        {   ss[i] = solve(pp[i]);   }
        return combine(p,ss);
    }
    abstract protected boolean isSimple (Problem p);
    abstract protected Solution simplySolve (Problem p);
    abstract protected Problem[] decompose (Problem p);
    abstract protected Solution combine(Problem p,Solution[] ss);
}
```

Fig. 4. Template method framework implementation.

as follows:

```
public interface Problem {};
public interface Solution {};
```

Given the auxiliary types above, we define the abstract Template Method class `DivConqTemplate` as shown in Fig. 4. We generalize the `combine()` method to take both the description of the problem and the subproblem solution array as arguments. The divide

```
public class QuickSortDesc implements Problem, Solution
{   public QuickSortDesc(int[]arr, int first, int last)
    {   this.arr = arr; this.first = first; this.last = last; }
    public int getFirst () { return first; }
    public int getLast ()  { return last;  }
    private int[] arr;          // instance data
    private int   first, last;
}
```

Fig. 5. Quicksort `Problem` and `Solution` implementation.

and conquer framework thus consists of the `DivConqTemplate` class and the `Problem` and `Solution` interfaces. We can now consider an application built using this framework library.

3.2. Building an application of the framework

In using a traditional procedure or class library, a client's program is in control of the computation; it "calls down" to code from the library. However, frameworks usually exhibit an *inversion of control*. The framework's code is in control of the computation; its template methods "call down" to the client-supplied hook methods. This section illustrates the use of the divide and conquer framework to build a quicksort application.

Quicksort is an in-place sort of a sequence of values. The description of a problem consists of the sequence of values and designators for the beginning and ending elements of the segment to be sorted. To simplify the presentation, we limit its scope to integer arrays. Therefore, it is sufficient to identify a problem by the array and the beginning and ending indices of the unsorted segment. Similarly, a solution can be identified by the array and the beginning and ending indices of the sorted segment. This similarity between the `Problem` and `Solution` descriptions enables us to use the same object to describe both a problem and its corresponding solution. Thus, we introduce the class `QuickSortDesc` to define the needed descriptor objects as shown in Fig. 5. Given the definitions for base class `DivConqTemplate` and auxiliary class `QuickSortDesc`, we can implement the concrete subclass `QuickSort` as shown in Fig. 6.

In a teaching module using this case study, both the framework (i.e., the abstract class) and the framework application (i.e., the implementation of quicksort) can be presented to the students so that they can discern the collaborations and relationships among the classes clearly. However, a clear distinction must be made between the framework and its application. As an exercise, the students can be assigned the task of modifying the quicksort application to handle more general kinds of objects. Other algorithms such as mergesort and binary search should also be assigned as exercises. The amount of work that each hook method has to do differs from one specific algorithm to another. In the quicksort implementation, most of the work is done in the `decompose()` method, which implements the splitting or pivoting operation of quicksort. In mergesort, however, more work will be done in the `combine()` operation because it must carry out the merge phase of the mergesort algorithm.

```
public class QuickSort extends DivConqTemplate
{   protected boolean isSimple (Problem p)
    {   return ( ((QuickSortDesc)p).getFirst()  >=
                 ((QuickSortDesc)p).getLast() );
    }
    protected Solution simplySolve (Problem p)
    {   return (Solution) p ;   }
    protected Problem[] decompose (Problem p)
    {   int first = ((QuickSortDesc)p).getFirst();
        int last  = ((QuickSortDesc)p).getLast();
        int[] a   = ((QuickSortDesc)p).getArr ();
        int x     = a[first]; // pivot value
        int sp    = first;
        for (int i = first + 1; i <= last; i++)
        {   if (a[i] < x) { swap (a, ++sp, i); }  }
        swap (a, first, sp);
        Problem[] ps = new QuickSortDesc[2];
        ps[0] = new QuickSortDesc(a,first,sp-1);
        ps[1] = new QuickSortDesc(a,sp+1,last);
        return ps;
    }
    protected Solution combine (Problem p, Solution[] ss)
    {   return (Solution) p;   }
    private void swap (int [] a, int first, int last)
    {   int temp = a[first];
        a[first] = a[last];
        a[last]  = temp;
    }
}
```

Fig. 6. Quicksort application.

3.3. Constructing a framework using separation

As an alternative to the above design, we can use the separation principle and Strategy pattern to implement a divide and conquer framework. The UML class diagram for this approach is shown in Fig. 7. The template method is implemented in the (concrete) context class `DivConqContext` as shown in Fig. 8. The hook methods are defined in the (abstract) Strategy class `DivConqStrategy` as shown in Fig. 9. The context class delegates the hook method calls to a reference to the instance of the Strategy class that it stores internally. Note that the Strategy approach is more flexible than the Template Method approach in that it is possible to switch Strategy objects dynamically by using the `setAlgorithm()` method of the context class. Constructing an application of the Strategy-based framework for Quicksort requires that we implement a subclass of the abstract class `DivConqStrategy` that is quite similar to the `QuickSort` class used in the unification framework (shown in Fig. 6).

The divide and conquer family of algorithms is a simple example that can be used to illustrate both approaches to framework design. It consists of a set of algorithms that should be known to the students. Hence, the application domain should be easy to explain. In the associated project, students can be given the framework and asked to construct

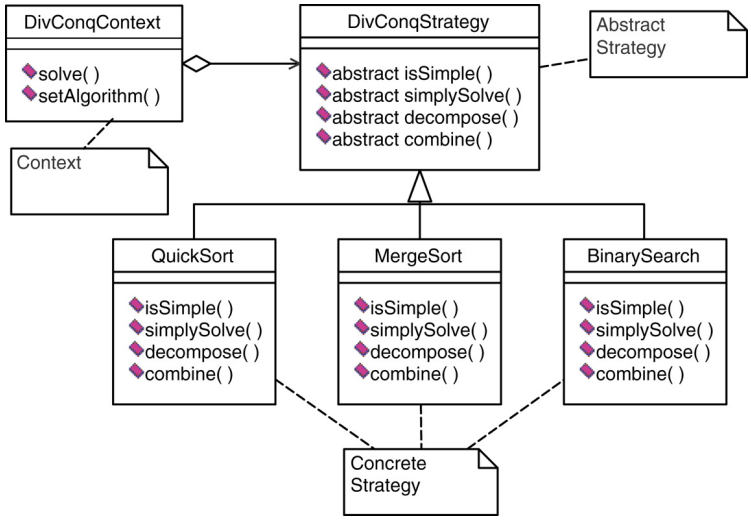

Fig. 7. Strategy pattern for divide and conquer framework.

```
public final class DivConqContext
{   public DivConqContext (DivConqStrategy dc)
    {   this.dc = dc;   }
    public Solution solve (Problem p)
    {   Problem[] pp;
        if (dc.isSimple(p)) { return dc.simplySolve(p); }
        else                { pp = dc.decompose(p);     }
        Solution[] ss =  new Solution[pp.length];
        for (int i = 0; i < pp.length; i++)
        {   ss[i] = solve(pp[i]);   }
        return dc.combine(p, ss);
    }
    public void setAlgorithm (DivConqStrategy dc)
    {   this.dc = dc;   }
    private DivConqStrategy dc;
}
```

Fig. 8. Strategy context class implementation.

```
abstract public class DivConqStrategy
{   abstract public boolean   isSimple (Problem p);
    abstract public Solution  simplySolve (Problem p);
    abstract public Problem[] decompose (Problem p);
    abstract public Solution  combine(Problem p, Solution[] ss);
}
```

Fig. 9. Strategy object abstract class.

applications. This requires an understanding of the framework's design at a level sufficient for using it, without requiring the students to develop their own framework abstractions. However, the students also need experience in identifying the hot spots and developing the needed framework abstractions. We consider those generalization steps in the following sections.

4. Framework development by generalization

Framework design involves incrementally evolving a design rather than discovering it in one single step. Typically, this evolution is a process of examining existing designs for family members, identifying the frozen spots and hot spots of the family, and generalizing the program structure to enable reuse of the code for frozen spots and use of different implementations for each hot spot. This generalization may be done in an informal, organic manner as codified by Roberts and Johnson in the Patterns for Evolving Frameworks [19] or it may be done using more systematic techniques.

Schmid's *systematic generalization* methodology is one technique [22] that seeks to identify the hot spots a priori and construct a framework systematically. This methodology identifies the following steps for construction of a framework [22]:

- creation of a fixed application model,
- hot spot analysis and specification,
- hot spot high-level design,
- generalization transformation.

In Schmid's approach, the fixed application model is an object-oriented design for a specific application within the family. Once a complete model exists, the framework designer analyzes the model and the domain to discover and specify the hot spots. The designer begins by asking which of the features of the application are characteristic of all applications in the domain (i.e., frozen spots) and which need to be made flexible (i.e., hot spots). Guided by appropriate design patterns [9], the designer then replaces a fixed, specialized class at a hot spot by an abstract base class. The hot spot's features are accessed through the common interface of the abstract class. However, the design of the hot spot subsystem enables different concrete subclasses of the base class to be used to provide the variant behaviors.

Another systematic approach is *function generalization* [20]. Where Schmid's methodology generalizes the class structure of the design for an application, the function generalization approach generalizes the functional structure of a prototype application to produce a generic application [4]. It introduces the hot spot abstractions into the design by replacing concrete operations by more general abstract operations or perhaps by replacing concrete data types by more abstract types. These abstract entities become "parameters" of the generalized functions. That is, the generalized functions are higher-order, having explicit or implicit parameters that are themselves functions. After generalizing the various hot spots of the application, the resulting generalized functions are used to generate a framework in an object-oriented language such as Java.

The case study in the next section explains the thinking process that a designer may use in analyzing and designing a framework. It uses an informal technique motivated by Schmid's systematic generalization and by function generalization.

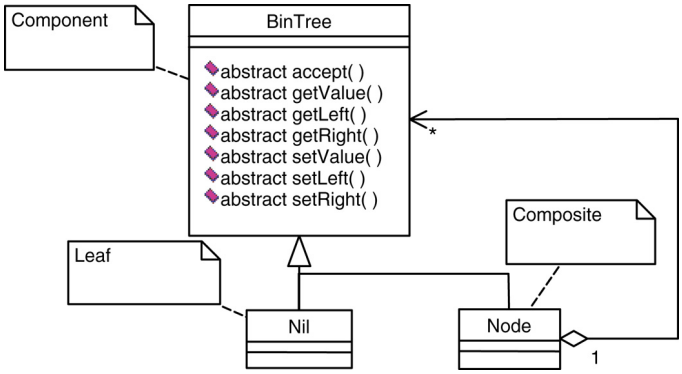

Fig. 10. Binary tree using Composite design pattern.

5. Binary tree traversal framework

As a case study on framework generalization, consider another classic problem, a binary tree traversal [11]. This case study seeks to address aspects of the fourth level of framework understanding described in Section 1—learning to analyze potential software families to identify the frozen and hot spots—as well as reinforcing and extending the students' understanding of the principles and techniques for constructing frameworks.

A *binary tree* is a hierarchical structure that is commonly taught in a lower-level undergraduate data structures course in a computing science curriculum. In this case study, we implement the binary tree with the `BinTree` class hierarchy, which is a structure designed according to the *Composite design pattern* [9] as shown in Fig. 10. The Composite pattern "lets clients treat individual objects and compositions of objects uniformly" [9]. Class `BinTree` has the Component base-class role in the pattern implementation, subclass `Node` has the Composite role, and subclass `Nil` has the Leaf role. `Nil` is also implemented according to the *Singleton pattern* [9], which guarantees exactly one instance exists. Fig. 11 shows the Java code for the `BinTree` class hierarchy.

A *traversal* is a systematic technique for "visiting" all the nodes in a tree. One common traversal technique for a binary tree is the *preorder traversal*. This is a depth-first traversal, that is, it accesses a node's children before it accesses the node's siblings. The preorder traversal can be expressed by a recursive procedure as follows:

```
procedure preorder(t)
{   if t null, then return;
    perform visit action for root node of tree t;
    preorder(left subtree of t);
    preorder(right subtree of t);
}
```

The visit action varies from application to another. The `BinTree` hierarchy in Fig. 11 supports a simple preorder traversal operation `preorder()` that merely prints a node's value when it is visited.

```
abstract public class BinTree
{   public void setValue(Object v)  { }        // mutators
    public void setLeft(BinTree l)  { }        //  default
    public void setRight(BinTree r) { }
    abstract public void preorder();           // traversal
    public Object  getValue() { return null; } // accessors
    public BinTree getLeft()  { return null; } //   default
    public BinTree getRight() { return null; }
}

public class Node extends BinTree
{   public Node(Object v, BinTree l, BinTree r)
    {   value = v; left = l; right = r; }
    public void setValue(Object v)  { value = v; }  // mutators
    public void setLeft(BinTree l)  { left = l;  }
    public void setRight(BinTree r) { right = r; }
    public void preorder()                          // traversal
    {   System.out.println("Visit node with value:  " + value);
        left.preorder(); right.preorder();
    }
    public Object  getValue() { return value; }     // accessors
    public BinTree getLeft()  { return left;  }
    public BinTree getRight() { return right; }
    private Object  value;                          // instance data
    private BinTree left, right;
}

public class Nil extends BinTree
{   private Nil() { }   // private to require use of getNil()
    public void preorder() { };                       // traversal
    static public BinTree getNil() { return theNil; } // Singleton
    static public BinTree theNil = new Nil();
}
```

Fig. 11. Binary tree class hierarchy.

Building a software framework for binary tree traversals involves the general principles for framework design. We begin with the simple preorder operation and tree structure given in Fig. 11 and consider the domain of the family and identify the frozen spots and hot spots.

What is the scope of the family of binary tree traversals? The family should include at least the standard kinds of depth-first traversals (e.g., preorder, postorder, and in-order) and allow flexible visit actions on the nodes. In general, the visit action will be a function on the node's attributes and on the accumulated state of the traversal computed along the sequence of all the nodes accessed to that point in the computation. The framework should enable traversal orders other than the depth first. The framework should also support binary search trees, but it is not necessary that it support multiway trees or general graphs.

What, then, are the commonalities, that is, frozen spots, that all members of the family exhibit? Considering the scope and examining the prototype application, we choose the following frozen spots:

(1) The structure of the tree, as defined by the `BinTree` hierarchy, cannot be redefined by clients of the framework.
(2) A traversal accesses every element of the tree once, unless the computation determines that it can stop before it completes the traversal.
(3) A traversal performs one or more visit actions associated with an access to an element of the tree.

What are the variabilities—the hot spots—that exist among members of the family of binary tree traversals? Again considering the scope and examining the prototype application, we identify the primary hot spots to be the following:

(1) Variability in the visit operation's action. It should be a function of the current node's value and the accumulated result of the visits to the previous nodes in the traversal.
(2) Variability in ordering of the visit action with respect to subtree traversals. That is, the client should be able to select preorder, postorder, in-order, etc.
(3) Variability in the tree navigation technique. That is, the client should be able to select node access orders other than left-to-right, depth-first, total traversals.

Now, given these variabilities, we examine how it can be introduced into a framework by generalizing the prototype application.

5.1. Generalizing the visit action

In this case study, hot spot #1 requires making the visit action a feature that can be customized by the client of the framework to meet the specific application's needs. The visit action, in general, varies from one application to another. The fact that there are visit actions associated with the access to an element is a common behavior of the framework. The visit action itself is the variable behavior that is to be captured in a hot spot subsystem. As we see in Section 2, we can introduce the variable behavior into a framework using either the unification principle (e.g., using the Template method pattern) or the separation principle (e.g., using the Strategy pattern). Because the `BinTree` structure is a frozen spot (i.e., cannot be changed by the framework user), we choose to use the Strategy pattern to implement variable visit behavior. This allows different visit actions to be used with the same tree structure.

We introduce this hot spot into the traversal program by generalizing the classes in the `BinTree` hierarchy to have the Context role in the Strategy pattern. We generalize the `BinTree` method `preorder` to be a template method and define a Strategy interface `PreorderStrategy` for objects that implement the hook methods. The new implementation of `preorder` must capture the general concept of a preorder traversal but delegate the specific preorder visit action to a method in a Strategy object. We specify method `visitPre` on interface `PreorderStrategy` as the hook method to encapsulate the preorder visit action. Furthermore, we define the `preorder` method (which had no arguments in the prototype program) to take two arguments: a "state" object that accumulates the relevant aspects of the traversal as the nodes are accessed and an instance of the `PreorderStrategy` Strategy object. Fig. 12 shows the changes made to the code in Fig. 11 for this generalization step.

```
abstract public class BinTree
{   ...
    abstract public Object preorder(Object ts, PreorderStrategy v);
    ...
}

public class Node extends BinTree
{   ...
    public Object preorder(Object ts,PreorderStrategy v) //traversal
    {   ts = v.visitPre(ts, this);
        ts = left.preorder(ts, v);
        ts = right.preorder(ts, v);
        return ts;
    }
    ...
}

public class Nil extends BinTree
{   ...
    public Object preorder(Object ts, PreorderStrategy v)
    {   return ts; }
    ...
}

public interface PreorderStrategy
{   abstract public Object visitPre(Object ts, BinTree t); }
```

Fig. 12. Binary tree with generalized visit action.

5.2. *Generalizing the visit order*

In this case study, hot spot #2 requires making the "order" of the visit actions a feature that can be customized for a specific application. That is, it must be possible to vary the order of a node's visit action with respect to the traversals of its children. The framework should support preorder, postorder, and in-order traversals and perhaps combinations of those. A good generalization of the three standard traversals is one that potentially performs a visit action on the node at any of three different points—on first arrival (i.e., a "left" visit), between the subtree traversals (i.e., a "bottom" visit), and just before departure from the node (i.e., a "right" visit). This is sometimes called an *Euler tour traversal* [11].

We enable the needed variability for this hot spot by generalizing the hot spot subsystem introduced in the previous step instead of introducing a new hot spot subsystem. We generalize the behavior of the `preorder` method in the `BinTree` hierarchy and replace it by a method `traverse` that encodes the common features of all Euler tour traversals but delegates the visit actions at the three possible visit points to hook methods defined on the Strategy object. We also observe that there was no visit action associated with a `Nil` subtree in the previous versions of the program. In some applications, it might be useful to have some action associated with a visit to a `Nil` subtree. So we add a fourth hook method for handling this as a special case. In the framework, we thus replace the `PreorderStrategy`

```
abstract public class BinTree
{   ...
    abstract public Object traverse(Object ts, EulerStrategy v);
    ...
}

public class Node extends BinTree
{   ...
    public Object traverse(Object ts, EulerStrategy v) // traversal
    {   ts = v.visitLeft(ts,this);     // upon arrival from above
        ts = left.traverse(ts,v);
        ts = v.visitBottom(ts,this);   // upon return from left
        ts = right.traverse(ts,v);
        ts = v.visitRight(ts,this);    // upon completion
        return ts;
    }
    ...
}

public class Nil extends BinTree
{   ...
    public Object traverse(Object ts, EulerStrategy v)
    {   return v.visitNil(ts,this); }
    ...
}

public interface EulerStrategy
{   abstract public Object visitLeft(Object ts, BinTree t);
    abstract public Object visitBottom(Object ts, BinTree t);
    abstract public Object visitRight(Object ts, BinTree t);
    abstract public Object visitNil(Object ts, BinTree t);
}
```

Fig. 13. Binary tree with Euler traversal.

interface from the previous program with a new `EulerStrategy` interface that defines the new hook methods. Fig. 13 shows the changes made to the code in Fig. 11 to incorporate the Euler tour traversal order for the visit actions.

5.3. *Generalizing the tree navigation*

In this case study, hot spot #3 requires making the navigation of the tree structure a feature that can be customized to meet the needs of a specific application. In particular, it should enable variability in the order in which nodes are accessed. For example, the framework should support breadth-first traversals as well as depth-first traversals. As we consider this generalization step, two of the frozen spots are of relevance:

(1) The `BinTree` hierarchy cannot be modified by clients of the framework.
(2) A traversal must access every element of the tree once, unless the computation determines that it can stop before it completes the traversal.

In the previous version of the binary tree traversal framework, the traversal technique is implemented directly by the `traverse` method of the `BinTree` hierarchy. The navigation technique implemented by this method must be made a customizable feature of the framework. However, because a client of the framework cannot modify the `BinTree` class or its subclasses, we must use the separation principle to implement the tree navigation subsystem. We could again use the Strategy pattern. However, another design pattern is more applicable to this situation—the *Visitor pattern* [9].

The intent of the Visitor pattern is to enable the functionality of an object structure to be extended without modifying the structure's code. The Visitor pattern does this by putting the new functionality in a separate class. Objects of this Visitor class access the elements of the object structure to carry out the desired new computation. An element of the object structure then calls back to the Visitor's method corresponding to the element's type. This "double-dispatching" uses polymorphism to avoid explicit checks on the type of an object. The Visitor pattern is quite compatible with object structures designed according to the Composite design pattern.

In the binary tree traversal framework design, we assign the `BinTree` class hierarchy the role of the Element hierarchy in the Visitor pattern's description [9], and we introduce a `BinTreeVisitor` interface to take on the role of the Visitor class in the description. We also generalize the `traverse` method of the `BinTree` hierarchy and replace it by the `accept` method for the Visitor pattern. The `accept` method of a `BinTree` element takes a `BinTreeVisitor` object and delegates the work of the traversal back to an appropriate method of that visitor object. This method applies the appropriate binary tree visit actions and navigates through the tree as needed for the application. The `BinTreeVisitor` interface has methods named `visit` with overloaded implementations for each subclass in the `BinTree` hierarchy. The constraint on the framework given by frozen spot #2 (i.e., to access each node once) becomes a requirement upon the designer of the visitor classes that implement `BinTreeVisitor`. Fig. 14 illustrates the class structure of a traversal program based on the Visitor design pattern. Fig. 15 shows the Java code for the traversal program.

The Visitor framework has two levels. The upper level of the framework is characterized by the Visitor pattern as described above. However, the specific designs for the Visitor objects themselves may be small frameworks. Consider a program to carry out an Euler tour traversal. We can choose to design a concrete class `EulerTourVisitor` that implements the `BinTreeVisitor` interface. Similar to the design for hot spot #2's `traverse` method, this class delegates the specific traversal visit actions to a Strategy object of type `EulerStrategy`. Fig. 16 illustrates the class structure of this lower-level design. Fig. 17 shows its implementation in Java.

The binary tree traversal framework is quite general. It supports a large set of binary tree algorithms. For example, it is possible using this framework to implement a "mapping" operation on trees. That is, it is feasible to implement a program that changes the value stored at every node of a tree by applying a mapping function to the previous value. Such a program can either be implemented directly as a `BinTreeVisitor` or as a customization of the Euler tour traversal framework. It is also possible to implement a breadth-first traversal operation by implementing an appropriate `BinTreeVisitor` class. Other interesting applications of the framework might be to use it to implement programs for binary search trees.

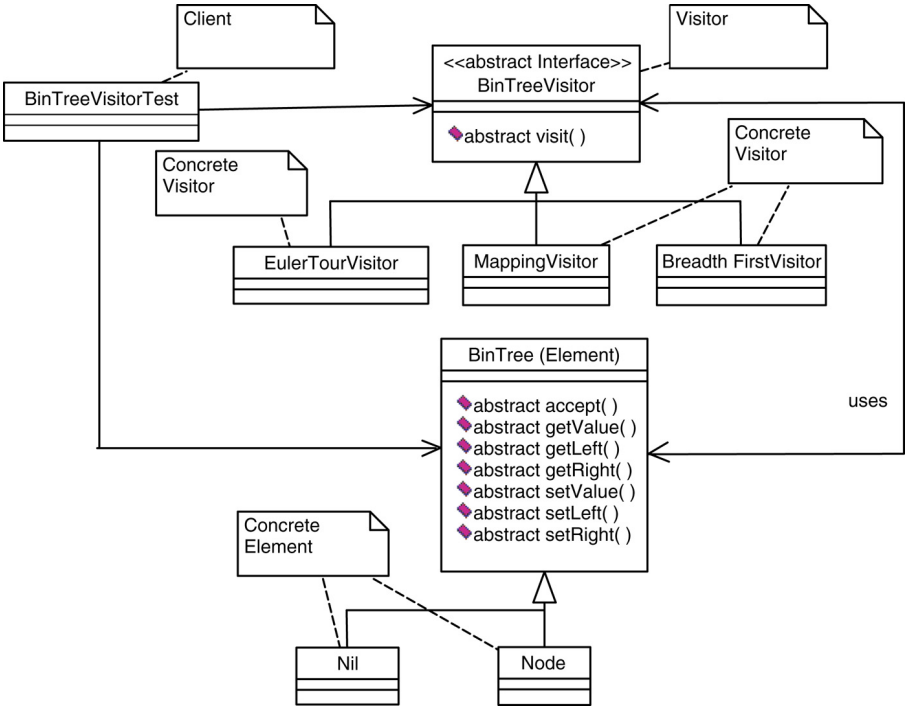

Fig. 14. Binary tree Visitor framework.

Programming projects accompanying use of this case study in a course can require development of various applications or require the design of new kinds of `BinTreeVisitor` subsystems. Instructors can also ask the students to apply the analysis and design techniques to other possible families.

As with the divide and conquer algorithms, binary tree structures and algorithms are well known to computing science and software engineering students. Use of this case study in an upper-level undergraduate course should not require an extensive explanation of the domain of the framework. However, this case study, and the application of the techniques to other problems, does require considerable thought and analysis on the part of the students. It is not a trivial activity for students to discover a sequence of generalization steps and effective hot spot abstractions that are appropriate for a large family of programs.

6. Related work

The thesis of this paper is that classic problems, such as those related to classic algorithms and data structures, are helpful examples for instructors to use in teaching computing science and software engineering students techniques for the design of software families. This paper describes two relatively simple examples designed to help teach both the use and construction of the type of software family called a software framework. The

```
abstract public class BinTree
{   public void setValue(Object v)  { }        // mutators
    public void setLeft(BinTree l)  { }        //  default
    public void setRight(BinTree r) { }
    abstract public void accept(BinTreeVisitor v); // accept Visitor
    public Object  getValue() { return null; } // accessors
    public BinTree getLeft()  { return null; } //   default
    public BinTree getRight() { return null; }
}

public class Node extends BinTree
{   public Node(Object v, BinTree l, BinTree r)
    {   value = v; left = l; right = r; }
    public void setValue(Object v)  { value = v; }  // mutators
    public void setLeft(BinTree l)  { left = l;  }
    public void setRight(BinTree r) { right = r; }
    // accept a Visitor object
    public void accept(BinTreeVisitor v) { v.visit(this); }
    public Object  getValue() { return value; }    // accessors
    public BinTree getLeft()  { return left;  }
    public BinTree getRight() { return right; }
    private Object  value;    // instance data
    private BinTree left, right;
}

public class Nil extends BinTree
{   private Nil() { }   // private to require use of getNil()
    // accept a Visitor object
    public void accept(BinTreeVisitor v) { v.visit(this); }
    static public BinTree getNil() { return theNil; } // Singleton
    static public BinTree theNil = new Nil();
}

public interface BinTreeVisitor
{   abstract void visit(Node t);
    abstract void visit(Nil t);
}
```

Fig. 15. Binary tree using Visitor pattern.

examples are aimed at advanced Java programming or software design courses in which students have not been previously exposed to frameworks in a significant way. The goal is to improve the students' abilities to construct and use abstractions in the design of software families.

Some advocate that use of frameworks be integrated into the introductory computing science sequence, e.g., into the data structures course [23]. In this approach, the understanding and use of standard data structure frameworks replace many of the traditional topics, which focus on the construction of data structures and algorithms. The availability of standard libraries such as the Java Collections framework makes this a viable approach. The argument is that when students enter the workplace, they more often face

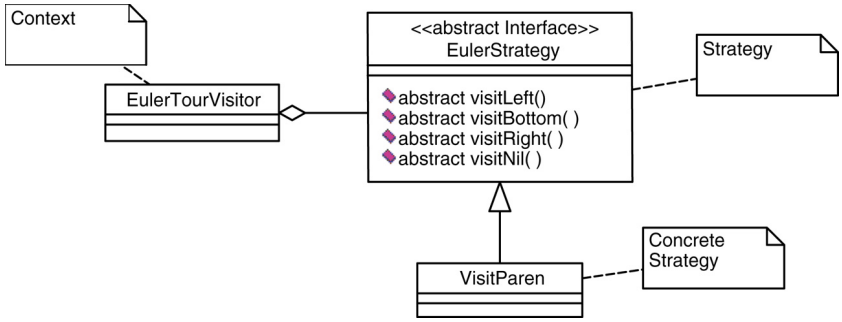

Fig. 16. Euler tour traversal Visitor framework.

```
public class EulerTourVisitor implements BinTreeVisitor
{   public EulerTourVisitor(EulerStrategy es, Object ts)
    {   this.es = es;  this.ts = ts; }
    public void setVisitStrategy(EulerStrategy es) // mutators
    {   this.es = es; }
    public void setResult(Object r) { ts = r; }
    public void visit(Node t)       // Visitor hookimplementations
    {   ts = es.visitLeft(ts,t);    // upon first arrival from above
        t.getLeft().accept(this);
        ts = es.visitBottom(ts,t);  // upon return from left
        t.getRight().accept(this);
        ts = es.visitRight(ts,t);   // upon completion of this node
    }
    public void visit(Nil t) { ts = es.visitNil(ts,t); }
    public Object getResult(){ return ts; }  // accessor
    private EulerStrategy es;  // encapsulates state changing ops
    private Object ts;         // traversal state
}

public interface EulerStrategy
{   abstract public Object visitLeft(Object ts, BinTree t);
    abstract public Object visitBottom(Object ts, BinTree t);
    abstract public Object visitRight(Object ts, BinTree t);
    abstract public Object visitNil(Object ts, BinTree t);
}
```

Fig. 17. Euler tour traversal Visitor.

the task of using standard components to build systems than that of writing programs in which they re-implement basic data structures and algorithms. Although it is appropriate that we cultivate the use of high-level abstractions, we should be careful not to abandon teaching of the intellectual fundamentals of computing science in a desire to train better technicians.

Others have constructed small software frameworks that are useful in pedagogical settings. Of particular interest is the work by Nguyen and Wong. In work similar to the

divide and conquer example in this paper, they use the Template Method and Strategy patterns and the divide and conquer algorithmic approach to develop a generalized sorting framework [15]. They believe that their design not only gives students "a concrete way of unifying seemingly disparate sorting algorithms but also" helps them understand the algorithms "at the proper level of abstraction". In an interesting design, they extend their framework to measure algorithm performance in a non-intrusive way by using the Decorator design pattern.

The goal of the divide and conquer framework in this paper differs from the goal of Nguyen and Wong's sorting framework. This paper focuses on teaching framework use and construction. The case study seeks to support any divide and conquer algorithm, not just sorting. The use of sorting algorithms to demonstrate the framework was incidental. However, future development of the divide and conquer framework can benefit from the design techniques illustrated by Nguyen and Wong.

In [13], Nguyen and Wong describe an interesting framework design that decouples recursive data structures from the algorithms that manipulate them. The design uses the State and Visitor design patterns to achieve the separation. In subsequent work, using the Strategy and Factory Method patterns, they extend this framework to enable lazy evaluation of the linear structures [14].

Nguyen and Wong's binary search tree framework in [13] has some similarities to the binary tree traversal framework in this paper. Their work seeks to teach students in introductory data structures courses to encapsulate "variant and invariant behaviors" in separate classes and use well-defined "communication protocols" to combine them into an application program. The use of design patterns, such as Visitor and State, is central to their design technique. The binary tree traversal case study in this paper has a similar goal in the context of teaching students how to design and construct frameworks in general. However, this paper approaches the design as the systematic application of a sequence of generalizing transformations to a prototype application. This systematic technique first identifies a point of variation and then chooses a design pattern that is effective in providing the needed flexibility.

While this paper uses design patterns in teaching the construction of frameworks, Christensen approaches the task from the other side [3]. He expresses concern that the conventional "catalogue-like" approaches to teaching design patterns "leave the impression that they are isolated solutions to independent problems". To overcome this misconception, he advocates the use of well-designed frameworks to teach the effective use of design patterns. He emphasizes that "a framework makes it clear that design patterns work together, and that patterns really define roles" rather than classes. He laments that "the subject of frameworks is sadly overlooked in teaching". The work in this paper seeks to help remedy that situation.

7. Conclusion

The first author has used the divide and conquer example and related programming exercises three times in Java-based courses on software architecture. They are effective in introducing students to the basic principles of framework construction and use if care

is taken to distinguish the framework from its application. However, other exercises are needed to help students learn to separate the variable and common aspects of a program family and to define appropriate abstract interfaces for the variable aspects.

The binary tree traversal framework case study and a similar case study on a cosequential processing framework [4,20] are designed to illustrate techniques that can help expand the ability of students to discover appropriate framework abstractions. The first author has used the cosequential processing problem (but not the case study) as the basis for a term project in a Java-based course on software engineering [5,7]. It proved to be a problem that challenged the students. However, the students' feedback indicated that more explicit attention should be paid to teaching systematic techniques for hot spot analysis and design.

In summary, software frameworks and design patterns are important concepts that students should learn in an advanced programming or software design course. These concepts may seem very abstract to the students, and, therefore, we need to start with familiar, non-daunting problems. This paper suggests the use of classic problems such as divide and conquer algorithms and binary tree traversals as examples to provide a familiar, simple and understandable environment in which students can better understand the framework concepts. Design patterns, such as the Template Method pattern and the Strategy pattern, are illustrated through the design of these simple frameworks. Since students are familiar with the algorithms and data structures and may have implemented them, they can concentrate on the design process more instead of the coding process and thus learn more effectively how to design a framework and build a program family.

Acknowledgements

The work of Cunningham and Liu was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)". Liu's work was also supported by University of Mississippi Graduate School Summer Research and Dissertation Fellowships. The authors thank Will Vaughan, Pallavi Tadepalli, and anonymous referee #1 for making several comments and suggestions that led to improvements in this paper.

References

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, Addison-Wesley, 1998.
[2] T. Budd, An Introduction Object-Oriented Programming, 3rd edition, Addison-Wesley, 2002.
[3] H.B. Christensen, Frameworks: Putting design patterns into perspective, in: Proceedings of the SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE, ACM, 2004, pp. 142–145.
[4] H.C. Cunningham, P. Tadepalli, Using function generalization to design a cosequential processing framework, Tech. Rep. UMCIS-2004-22, Department of Computer and Information Science, University of Mississippi, December 2004.
[5] H.C. Cunningham, Y. Liu, C. Zhang, Keeping secrets within a family: Rediscovering Parnas, in: Proceedings of the Software Engineering Research and Practice (SERP) Conference, CSREA Press, 2004, pp. 712–718.
[6] H.C. Cunningham, Y. Liu, C. Zhang, Using the divide and conquer strategy to teach Java framework design, in: Proceedings of the International Conference on the Principles and Practice of Programming in Java, PPPJ, 2004, pp. 40–45.

[7] H.C. Cunningham, P. Tadepalli, Y. Liu, Secrets, hot spots, and generalization: Preparing students to design software families, Journal of Computing Sciences in Colleges 20 (6) (2005) 74–82.
[8] M. Fontoura, W. Pree, B. Rumpe, The UML Profile for Framework Architectures, Addison-Wesley, 2002.
[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
[10] G.C. Gannod, R.R. Lutz, An approach to architectural analysis of product lines, in: Proceedings of the 22nd International Conference on Software Engineering, ICSE 00, 2000, pp. 548–557.
[11] M.T. Goodrich, R. Tamassia, Data Structures and Algorithms in Java, 3rd edition, Wiley, 2004.
[12] X. Jia, Object-Oriented Software Development using Java: Principles, Patterns, and Frameworks, Addison-Wesley, 2000.
[13] D. Nguyen, S.B. Wong, Patterns for decoupling data structures and algorithms, in: Proceedings of ACM SIGCSE Technical Symposium, 1999, pp. 87–91.
[14] D. Nguyen, S.B. Wong, Design patterns for lazy evaluation, in: Proceedings of ACM SIGCSE Technical Symposium, 2000, pp. 21–25.
[15] D. Nguyen, S.B. Wong, Design patterns for sorting, in: Proceedings of ACM SIGCSE Technical Symposium, 2001, pp. 263–267.
[16] D. Nguyen, S.B. Wong, Design patterns for games, in: Proceedings of ACM SIGCSE Technical Symposium, 2002, pp. 126–130.
[17] D.L. Parnas, On the design and development of program families, IEEE Transactions on Software Engineering SE-2 (1) (1976) 1–9.
[18] D. Parnas, Software design, in: D.M. Hoffman, D.M. Weiss (Eds.), Software Fundamentals: Collected Papers by David L. Parnas, Addison-Wesley, 2001, pp. 137–142.
[19] D. Roberts, R. Johnson, Patterns for evolving frameworks, in: R. Martin, D. Riehle, F. Buschmann (Eds.), Pattern Languages of Program Design 3, Addison-Wesley, 1998, pp. 471–486.
[20] P. Tadepalli, H.C. Cunningham, Using function generalization with Java to design a cosequential framework, in: Proceedings of the Conference on Applied Research in Information Technology, Acxiom Laboratory for Applied Research, 2005, pp. 95–101.
[21] H.A. Schmid, Systematic framework design by generalization, Communications of the ACM 40 (10) (1997) 48–51.
[22] H.A. Schmid, Framework design by systematic generalization, in: M.E. Fayad, D.C. Schmidt, R.E. Johnson (Eds.), Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley, 1999, pp. 353–378.
[23] J. Tenenberg, A framework approach to teaching data structures, in: Proceedings of ACM SIGCSE Technical Symposium, 2003, pp. 210–214.
[24] D.M. Weiss, C.T.R. Lai, Software Product-Line Engineering: A Family-Based Software Development Process, Addison-Wesley, 1999.