Using the Divide and Conquer Strategy to Teach Java Framework Design

H. Conrad Cunningham[1], Yi Liu[1], Cuihua Zhang[2]
[1] Computer & Information Science, University of Mississippi
[2] Computer & Information Systems, Northwest Vista College
cunningham@cs.olemiss.edu

ABSTRACT

All programmers should understand the concept of program families and know the techniques for constructing them. This paper describes a case study that can be used to introduce students in a Java software design course to the construction of program families using software frameworks. The example is the family of programs that use the well-known *divide and conquer* algorithmic strategy.

1. INTRODUCTION

Parnas noted that a "software designer should be aware that he is not designing a single program but a family of programs." [8] Therefore, it is desirable to have future programmers, i.e. students, develop this awareness early, e.g., in advanced programming or software design courses.

A *program family* is a set of programs with so many common properties that it is worthwhile to study the set as a group [7]. Object-oriented frameworks form a foundation upon which members of program families can be built [1]. This paper describes one way to introduce students to software frameworks by using the commonly known *divide and conquer* algorithmic strategy as the basis for a family.

In an advanced Java programming or software design course, this topic can be organized as a few lectures followed by a programming assignment. The lectures assume that the students have completed an introductory computing science sequence using Java and understand concepts such as inheritance, delegation, recursion, and sorting. They do not assume prior study of frameworks.

2. FRAMEWORKS

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. We need to shift their focus to techniques for building a program family. Since a program family is a set of programs that have many common properties, "it is advantageous to study the common properties of the programs before analyzing individual members." [7] To design and implement a program family, we first find the properties that are common to all programs; and then, based on the characteristics of these commonalities, we make certain design decisions to realize software reuse.

A *framework* is a generic application that allows the creation of different specific applications from a family [9]. It is an abstract design that can be reused within a whole application domain. In a framework the common aspects of the family are represented by a set of abstract classes that collaborate in some structure. Common behaviors are implemented by concrete *template methods* in a base class. A variable aspect of the system, sometimes called a *hot spot* [9], is represented by a group of abstract *hook methods*. The hook methods are realized by concrete methods in a *hot spot subsystem* in an application of the family. Figure 1 shows a framework with a hot spot.

There are two principles for framework construction—unification and separation [2]. 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. The *separation principle* uses delegation to implement the hot spot subsystem. The template methods are implemented in a concrete client 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.

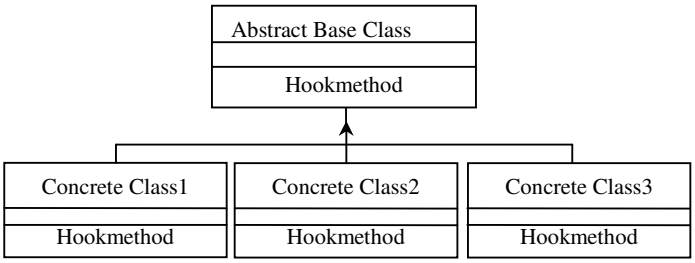

Figure 1. Framework with a hot spot

A framework is a system that is designed with generality and reuse in mind; and *design patterns*, which are well-established solutions to program design problems that commonly occur in practice, are the intellectual tools to achieve the desired level of generality and reuse. In our lectures, two patterns, corresponding to the two framework construction principles, are used to implement 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." [3] 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 [2]. We also structure a framework with the *Strategy pattern*, which uses the separation principle. That approach is not shown here because of limitations on the length.

3. DIVIDE AND CONQUER FRAMEWORK

To illustrate the design of a framework, we use the family of *divide and conquer* algorithms as an example of a program 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 to its meaningful influence in algorithms, serves well the purpose of examining a program family.

The pseudo-code for the *divide and conquer* technique for a problem `p` is as follows:

```
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);
}
```

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.

If the Template Method pattern is 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.

Figure 2 shows a design for a *divide and conquer* program family expressed as a *Unified Modeling Language* (UML) 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`).

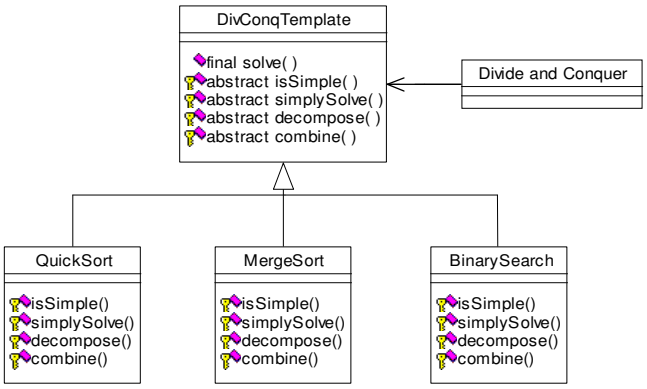

Figure 2. Template Method for divide

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) as follows:

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

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

```
abstract public class DivConqTemplate
{     final public 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);
    }
```

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

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 follows:

```
    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;
        private int   first, last;
    }
```

Given the definitions for base class `DivConqTemplate` and auxiliary class `QuickSortDesc`, we can implement the concrete subclass `QuickSort` as shown below:

```
    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;
    }
}
```

In lectures on 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 more 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.

4. DISCUSSION

This paper describes a simple example designed to help teach computing science students both the use and construction of software frameworks. The example is 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 program families.

Some advocate that use of frameworks be integrated into the introductory computing science sequence, e.g., into the data structures course [10]. 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 the task of using standard components to build systems than 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 [4], they describe a 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 Methods patterns, they extend this framework to enable lazy evaluation of the linear structures [5]. In work similar to the example in this paper, Nguyen and Wong use the Template Method and Strategy patterns and the *divide and conquer* algorithmic approach to develop a generalized sorting framework [6]. 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."

The goal of the *divide and conquer* framework described in this paper differs from the goal of Nguyen and Wong's sorting framework. This paper focuses on teaching framework use and construction. It 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 techniques illustrated by Nguyen and Wong.

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.

5. CONCLUSION

Software frameworks and design patterns are important concepts that students should learn in a Java 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 shows that the *divide and conquer* algorithmic strategy can be used as an example to provide a familiar, simple and understandable environment in which students can better understand these concepts. The Template Method pattern is illustrated through the design of this simple framework. Since students are familiar with the algorithms 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.

6. 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)."

7. REFERENCES

[1] M. E. Fayad, D. C. Schmidt, and R. E. Johnson. Building Application Frameworks: Object-Oriented Foundations of Framework Design, Wiley, 1999.

[2] M. Fontoura, W. Pree, and B. Rumpe. *The UML Profile for Framework Architectures.* Addison-Wesley, 2002.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[4] D. Nguyen and S. B. Wong. "Patterns for Decoupling Data Structures and Algorithms," In *Proceedings of ACM SIGCSE Technical Symposium*, pp. 87-91, March 1999.

[5] D. Nguyen and S. B. Wong. "Design Patterns for Lazy Evaluation," In *Proceedings of ACM SIGCSE Technical Symposium*, pp. 21-25, March 2000.

[6] D. Nguyen and S. B. Wong. "Design Patterns for Sorting," In *Proceedings of ACM SIGCSE Technical Symposium*, pp. 263-267, February 2001.

[7] D. L. Parnas. "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, Vol. SE-2, pp. 1-9, March 1976.

[8] D. L. Parnas. "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, Vol. SE-5, pp. 128-138, March 1979.

[9] H. A. Schmid. "Systematic Framework Design by Generalization," *Communications of the ACM*, Vol. 40, No. 10, pp. 48-51, October 1997.

[10] J. Tenenberg. "A Framework Approach to Teaching Data Structures," *Proceedings of the ACM SIGCSE Technical Symposium*, pp. 210-214, February 2003.