A Data Abstraction Alternative to Data Structure/Algorithm Modularization

Murali Sitaraman[1], Bruce W. Weide[2], Timothy J. Long[2], and William F. Ogden[2]

[1] Computer Science and Electrical Engineering, West Virginia University
Morgantown, WV 26506-6109
murali@csee.wvu.edu

[2] Computer and Information Science, The Ohio State University
Columbus, OH 43210
{weide, long, ogden}@cis.ohio-state.edu

Abstract. Modularization along the boundaries of data structures and algorithms is a commonly-used software decomposition technique in computer science research and practice. When applied, however, it results in incomplete segregation of data structure handling and algorithm code into separate modules. The resulting tight coupling between modules makes it difficult to develop these modules independently, difficult to understand them independently, and difficult to change them independently. Object-oriented computing has maintained the traditional dichotomy between data structures and algorithms by encapsulating only data structures as objects, leaving algorithms to be encapsulated as single procedures whose parameters are such objects. For the full software engineering benefits of the information hiding principle to be realized, data abstractions that encapsulate data structures and algorithms together are essential.

1 Introduction

The dichotomy of data structures and algorithms is pervasive in computing, and this separation has been used routinely as a criterion for modularization in both structured and object-oriented software development. But the suitability of this criterion for decomposition has rarely been questioned from a software engineering perspective in the data structures and algorithms literature. Cormen, Leisersen, and Rivest, authors of probably the most widely used textbook on algorithms [2], admit to as much, as they set the following stage for their discourse on algorithms:

> We shall typically describe algorithms as programs written in pseudocode that is very much like C, Pascal, or Algol, ... not typically concerned with issues of software engineering. Issues of data abstraction, modularity, and error handling are often ignored to convey the essence of the algorithm more concisely.

Though it is arguably useful to understand the design and details of algorithms without concern for software engineering, Parnas has argued that the criteria to be used for module decomposition are at least as important in development of real software systems [8]. In the conventional data structure/algorithm decomposition, the code that stores "input" values into data structures, and the code that retrieves "output" values from data structures, resides in the calling module. The algorithm that transforms the inputs to outputs using the data structures, and which is typically a procedure whose parameters are these data structures, resides in a separate module. The major processing steps—input/output data structure handling and execution of the algorithm—have been turned into separate modules that communicate through data structures. Parnas has argued against this form of processing-based decomposition, noting that it makes it difficult to develop the modules independently, difficult to understand modules independently, and difficult to change modules independently. He has proposed "information hiding" as an alternative criterion for module decomposition [8]:

> . . . it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions that are likely to change. Each module is then designed to hide such a decision from others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in processing.

While the information hiding criterion has had some impact on hiding decisions about the data structures involved in representing "container" abstract data types such as stacks and queues, it has had little effect on modularization that respects the classical dichotomy of data structures and algorithms. The objective of this paper is to expose the weaknesses of this widely used modularization along data structure/algorithm boundaries and to discuss an alternative based on the information hiding criterion. In this approach, data structures and algorithms are encapsulated together to produce new data abstractions. The resulting modularization arguably has desirable properties of module decomposition detailed in [8], including ease of independent module development, ease of use, and ease of change. In addition, the data abstraction interface approach provides performance advantages. For a given data abstraction, it becomes possible to develop alternative plug-compatible implementations that differ both in the data structures/algorithms used in computing the results, and in whether results are computed incrementally or in batch, allowing temporal flexibility. Through their interface specifications and support for alternative implementations, reusable data abstractions of this nature supplement (functional and performance) flexibility benefits of generic programming [7], with software engineering benefits.

The rest of the paper is organized as follows. Section 2 examines a classical example from the algorithms literature from a software engineering perspective. Section 3 presents a data abstraction alternative. Section 4 discusses performance benefits of the data abstraction-based modularization. The last section contains a discussion of related work and our conclusions.

2 A Conventional Modularization That Respects the Data Structure/Algorithm Dichotomy

To illustrate the ramifications of modularization based on separation of data structures and algorithms, we begin with an instance of a shortest path problem:

Given a graph, a source vertex, and a set of destination vertices, find the shortest paths between the source and the destinations, if such paths exist, and their costs.

The best-known solution to the single source shortest paths problem is Dijkstra's greedy algorithm for finding shortest paths to all (connected) vertices from a single source [2]. Details of the data structures used in this solution are given below.

```
procedure SSSP_Dijkstra
    inputs G, w, s
    outputs S, distances, predecessors
```

Input data structures: `G` is a graph represented by an adjacency list data structure. `w` is a weight function that returns the cost of the edge joining a given pair of vertices. `s` is the source vertex.
Output data structures: `S` is a set of vertices that can be reached from `s`. `distances` is a structure that contains the cost of the shortest path from `s` to every vertex in `S`. `predecessors` is a structure that holds the penultimate vertex on a shortest path from the source to each vertex.

In this modularization, responsibility for the algorithm is given to one module (procedure `SSSP_Dijkstra`) and responsibility for storing information into and retrieving information from input/output data structures is given to the calling module. This responsibility assignment thwarts independent module development, because both modules rely on details of the data structures. The significant coupling between the calling module and the procedure that encodes the algorithm is obvious in this example. To understand and reason about the procedure, the data structures in the calling module need to be understood; and to understand the reason for choosing a particular representation for a data structure in the calling module (e.g., adjacency list representation for graphs), it is essential to understand some details of the algorithm. Any changes to details of the data structures in the calling module may affect the called procedure, and any changes to the algorithm may affect the responsibilities of the caller. Every use of the procedure requires the calling module to set up non-trivial data structures. And even when a caller is not interested in all results (paths, costs, paths to all destinations), all the data structures must be set up, if the intent is not to modify the code of the reusable procedure.

Another fundamental problem in separating data structures from algorithms arises when different algorithms for solving a problem demand different data structures. In such cases, when a client desires to switch from one algorithm to another, significant changes may be needed to the calling code. It may be essential to use a different algorithm to improve performance or because of apparently minor changes to the problem requirements. Suppose that the shortest path problem were modified slightly as below:

Given a graph, a set of source vertices, and a set of destination vertices, find the shortest paths between the sources and the destinations, if paths exists, and their cost.

One solution to this problem is to call the single source shortest paths procedure repeatedly with different sources. If the set of sources is small, this indeed may be the right choice. But if the paths need to be computed from a larger set of sources, then the Floyd-Warshall algorithm that computes efficiently shortest paths from all source vertices to all destination vertices may be more appropriate. In the Floyd-Warshall algorithm, the following data structures are used:

```
procedure ASSP_Floyd_Warshall
    input G
    outputs distances, predecessors
```

Input data structures: `G` is a graph represented by an adjacency matrix.
Output data structures: `distances` is a $|V| \times |V|$ matrix that contains distances of the shortest path from every source to every destination vertex. `predecessors` is a $|V| \times |V|$ matrix holding the penultimate vertex on a shortest path from each source to each destination vertex.

The differences in the input/output data structures and their representations between Dijkstra's algorithm and Floyd-Warshall's algorithm are significant, and they demand considerable change in the client module to switch from one algorithm to the other.

3 Modularization Based on Information Hiding

A key design decision in classical examples, such as the ones discussed in the last section, is making a suitable choice for data structures/representations for a particular algorithm. Most algorithms work efficiently only when used with particular data structures and their workings are often intricately intertwined with those data structures. Rarely are the choices so independent that arbitrary combinations are meaningful. In addition, as illustrated above, when algorithms need to be changed for performance or for software evolution reasons, the data structures are likely to change as well.

The information hiding criterion suggests that the design decision of matching proper structures with algorithms is a difficult choice that is quite likely to change (e.g., if the client decides to use a different algorithm), and it therefore must be hidden inside a module. To use a module that computes shortest paths, it should not be essential for the calling module to understand how graphs are represented or what structures are used to store inputs and results. For the shortest path problems, the interface of the data abstraction must permit a client to supply information about a graph and get answers to questions about shortest paths. Use of a data abstraction essentially decouples an implementation of the abstraction from its use, in turn facilitating independent development, ease of use, and ease of change.

The basic idea is easily explained. The abstract state space for an ADT (abstract data type) for solving the class of shortest path finding problems includes the edges of the graph under consideration. Operations on the ADT allow a graph to be defined and questions about shortest paths to be asked. One operation allows a user to input graph edges, one at a time. Other operations allow a user to ask whether there is a path between two vertices and if one exists, to request the cost of the shortest such path. The ADT also includes an operation that gives the edges on a shortest path from a source to a destination, one at a time. Other than an object of the ADT under discussion, in this design the only other parameters to operations are values of simple types such as edge information, booleans, and real numbers. No complex data structures are passed as parameters, because suitable data structures together with algorithms that manipulate them are hidden in the implementation(s) of the ADT. Clients neither need to understand nor need to set up any fancy data structures. Instead they see only an ADT and operations to manipulate objects of that type.

One interesting new issue arises in designing an interface for a data abstraction such as the one described here. Unlike data abstractions that encapsulate classical "container" data structures, where, in general, there are few or no restrictions on the order in which the operations may be called, the data abstraction described here demands that all graph information be available before the queries begin. It is easy to specify such restrictions by embellishing the abstract state model in a formal specification of the data abstraction, as explained in the next subsection.

Formal Specification of a Data Abstraction

A formal specification for a data abstraction that captures the shortest path problems in a dialect of the RESOLVE notation [10], is given in Figure 1. We have used the name "least cost path" instead of "shortest path", so weights on edges are interpreted more generally as costs instead of distances.

```
concept Least_Cost_Path_Finding_Template (
        type Edge_Index,
        constant Max_Vertex: Integer
    )

    math subtype Edge is (
            v1: integer,
            v2: integer,
            id: Edge_Index,
            cost: real
        )
        exemplar e
        constraint
            1 <= e.v1 <= Max_Vertex and
            1 <= e.v2 <= Max_Vertex and
            e.cost > 0.0

    definition CONNECTING_PATH_EXISTS (
            graph_edges: finite set of Edge,
            v1, v2: integer
        ): boolean
        = (* true iff there is a connecting path from
               v1 to v2 in graph_edges *)

    definition IS_A_LEAST_COST_PATH (
            graph_edges: finite set of Edge,
            v1, v2: integer,
            s: finite set of Edge
        ): boolean
        = (* true iff s is the set of edges on a least
               cost path from v1 to v2 in graph_edges *)

    type Path_Finder is modeled by (
            edges: finite set of Edge,
            insertion_phase: boolean
        )
        exemplar m
        initialization
            ensures m = ({}, true)

    operation Insert_Edge (
            alters m: Path_Finder,
            consumes v1: Integer,
            consumes v2: Integer,
            consumes id: Edge_Index,
            consumes cost: Real
        )
        requires m.insertion_phase and
                1 <= v1 <= Max_Vertex and
            1 <= v2 <= Max_Vertex and
            e.cost > 0.0
        ensures m.edges = #m.edges union {(#v1, #v2, #id, #cost)}
            and m.insertion_phase = #m.insertion_phase

    operation Stop_Accepting_Edges (
            alters m: Path_Finder
        )
        ensures m.edges = #m.edges and
```

```
            m.insertion_phase = false

    operation Connecting_Path_Exists (
            preserves m: Path_Finder,
            preserves v1, v2: Integer
        ) returns result: Boolean
        requires not m.insertion_phase
        ensures result = CONNECTING_PATH_EXISTS (m.edges, v1, v2)

    operation Find_Least_Cost (
            preserves m: Path_Finder,
            preserves v1, v2: Integer
        ) returns result: Real
        requires not m.insertion_phase and
            CONNECTING_PATH_EXISTS (m.edges, v1, v2)
        ensures there exists s: set of Edge
            (IS_A_LEAST_COST_PATH (m.edges, v1, v2, s) and
             result = sum e: Edge where (e is in s) (e.cost))

    operation Get_Last_Stop (
            preserves m: Path_Finder,
            preserves v1, v2: Integer,
            produces stop: Integer,
            produces id: Edge_Index,
            produces cost: Real
        )
        requires not m.insertion_phase and
            CONNECTING_PATH_EXISTS (m.edges, v1, v2)
        ensures CONNECTING_PATH_EXISTS (m, v1, v2) and
                there exists s: set of Edge
                 (IS_A_LEAST_COST_PATH (m.edges, v1, stop, s) and
                  IS_A_LEAST_COST_PATH (m.edges, v1, v2,
                      s union {(stop, v2, id, cost)}))

    operation Is_In_Insertion_Phase (
            preserves m: Path_Finder
        ) returns result: Boolean
        ensures result = m.insertion_phase

end Least_Cost_Path_Finding_Template
```

Figure 1. `Least_Cost_Path_Finding_Template`

To use the template in Figure 1, a client needs to create an instance by picking a suitable value for `max_vertex` and an identification type for Edges (e.g., names). The module provides an ADT, named `Path_Finder`. The abstract state of the type has been modeled mathematically as an ordered pair: a finite set of (graph) edges and a boolean value that is true iff edges are allowed to be inserted. In the description of this mathematical model, we have used a mathematical subtype Edge [4,9]: values of this type are constrained to be such that edge weights are positive. The essential purpose of the subtype is to make the specification easier to understand.

The **initialization ensures** clause specifies that every newly-declared object of type `Path_Finder` contains no graph edges and is ready for insertion. Edges of the graph, for which least cost paths need to be found, are inserted one at a time to through calls to `Insert_Edge` operation. The operation takes as its parameters an object m, and information on the edge that is inserted. The operation has a pre-condition as specified in its **requires** clause: `m.insertion_phase` must be true. The operation alters the state of the machine m as specified in the post-condition and consumes the edge information. In the **ensures** clause, `#m` denotes the value of the object that is input to the operation and `m` denotes its value after the operation. (In the requires clause, all parameter names refer to input values.) The values of the consumed parameters are left unspecified. They have initial values of their types after the operation.

The three operations `Connecting_Path_Exists`, `Find_Least_Cost`, and `Get_Last_Stop` require `m.insertion_phase` to be false, i.e., all edges must be available. Through these requires clauses, the specification dictates conceptually the order in which the operations can be called. In particular, the `Stop_Accepting_Edges` operation, which ensures that `m.insertion_phase` is false, must be called before the path query operations. The last operation `Is_In_Insertion_Phase` can be used to determine the insertion status of the object.

The operation `Connecting_Path_Exists` returns true iff there is a connecting path between the two given vertices in the graph. In the ensures clause, we have used a mathematical predicate `CONNECTING_PATH_EXISTS` with the obvious meaning. A formal definition of this predicate should be included in the specification, but has been omitted here for brevity. In the specification of operations `Find_Least_Cost` and `Get_Last_Stop`, we have used another predicate `IS_A_LEAST_COST_PATH (e, v1, v2, s)`. This predicate is true iff the set of edges `s` constitutes a least cost path from `v1` to `v2` in the graph whose edges are in `e`. The `Get_Last_Stop` operation returns the penultimate vertex (and the corresponding edge) in a least cost path from `v1` to `v2`. By calling this operation repeatedly, with the returned vertex as the destination, all edges on a least cost path from `v1` to `v2` can be found, incrementally. Similarly, by calling the operation with suitable parameters, the least cost paths between different sources and destinations can be found.

The specification of the data abstraction serves as the contract between modules that implement the abstraction and modules that use the abstraction. Different implementations of the abstraction hide both the data structures and algorithms used in computing the results, as well as whether the results are computed in batch or incrementally. For calling modules, finding least costs paths using the data abstraction is as simple as using a more typical ADT such as a stack or a queue. Switching from one implementation of the data abstraction to another is just as easy.

4 Performance Ramifications

Unlike the conventional modularization in Section 2 in which algorithms are encoded as batch computing procedures, the data abstraction interface in Section 3 permits incremental inputs and outputs. This distinction may not make a difference for problems where no incremental input processing is meaningful because all known algorithms require knowledge of all inputs. For some other problems, all known algorithms may need the same execution time to compute full or partial solutions, and the distinction may not matter either. But there are other problems, such as those discussed in this paper and elsewhere [8,13], where such is not the case. In these cases, the separation of processing from input/output steps leads to computation of expensive and complete solutions, even when the applications may need only a subset of outputs.

For the least cost path problem, for example, conventional modularization leads to a procedure that computes all shortest paths from the source, though the caller may be willing to abandon computation once the shortest path(s) to desired destination(s) are found. This performance problem is a result of the rigid input/process/output computing that is introduced when an algorithm is designed as a single procedure. In this case, the caller has no communication mechanism to stop computation once questions of interest have been answered. But the incremental interface of the data abstraction for the problem provides the calling module this flexibility. This is clear from considering procedure `SSSP_Dijkstra` in Section 2 and operations `Find_Least_Cost/Get_Last_Stop` in Section 3.

Consider an implementation of `Least_Cost_Path_Finding_Template` using Dijkstra's greedy algorithm. This algorithm has the property that at any time during computation, for the destination vertices in a set S, the shortest paths and costs for those paths are available in the other structures. When one of the operations `Find_Least_Cost` or `Get_Last_Stop` is called, the implementation can stop computation as soon as S contains the desired destination, and store all intermediate results. This approach can offer significant performance savings on the average. In addition, if the intermediate results are stored in a transitive closure matrix, when operations `Find_Least_Cost` or `Get_Last_Stop` are called with different sources or destinations, the information in the matrix can be used without re-computation. Such an implementation is superior to both of the procedures outlined in the last section for applications that need partial results.

Though incremental/amortized cost computing is a key benefit of the data abstraction interface, it is important to note that the interface does not preclude batch computing implementations that compute all results. For example, a different implementation of the `Least_Cost_Path_Finding_Template` may use `Floyd_Warshall`'s algorithm and compute all shortest paths, when the operation `Stop_Accepting_Edges` is called. In other words, the data abstraction interface allows the plug-compatible implementation strategies that differ both in how and when results are computed.

The least cost path problem is an illustrative example that typifies how other classical algorithms can be packaged as data abstractions. We have discussed elsewhere, for example, data abstractions for classical algorithms such as sorting, graph algorithms such as finding a minimum spanning forest of a graph, and other optimization problems [13,12]. A data abstraction that encapsulates an ordering algorithm and related data structures, for instance, allows alternative implementations that order inputs incrementally as they are inserted, or in batch after all items are inserted, or in an amortized fashion during extraction of outputs, or using a combination of strategies. We have also documented the more general impact of both duration and limited storage capacity considerations on data abstraction interface design for a graph algorithm [11]. In every case, the result is a data abstraction that allows independent team development of modules, ease of use, ease of change, and performance flexibility and incremental computation.

5 Conclusions

Classical data structure/algorithm modularization continues to dictate software modularization, despite its disadvantages for software engineering. Even modern object-based approaches typically encapsulate data structures alone. They treat data structures such as queues, lists, trees, sets, and maps as objects, but leave algorithms as procedures/methods that manipulate these objects [1,5].

Typical object-oriented code for Dijkstra's algorithm for finding a shortest path, for example, is similar to that in [2], except that it might use encapsulated graph and set objects instead of unencapsulated graph and set data structures [3,6,14]. It is especially instructive to compare the above data abstraction solution to the shortest path problem with the approach used by Weihe in [14]. Weihe shares our objective of making algorithms more reusable, without sacrificing efficiency. His approach also allows performance "tuning" of a client to use a particular algorithm (e.g., by stopping computation early), by having interface operations that take smaller steps. However, without a data abstraction approach to the problem, considerable modifications to client code are essential to use different algorithms that provide "order of magnitude" performance improvements. The traditional modularization problems that preclude independent software development, originally catalogued by Parnas, remain.

The performance issues discussed in this paper have also brought into focus the need for "online" algorithms. In the data abstraction view, these algorithms become natural alternative implementations and provide additional performance flexibility to clients. However, the interfaces need to be designed carefully to allow use of such algorithms.

In this paper, we have illustrated how new kinds of data abstractions can be developed following the principle of information hiding. Unless such data abstractions are defined and implemented to replace the traditional data structure/algorithm modularization, the full potential of the information hiding principle for software engineering cannot be realized.

Acknowledgments

We would like to thank members of the Reusable Software Research groups at The Ohio State University and at West Virginia University for discussions relating to the contents of this paper. In particular, we would like to thank Jennifer Kannen, Jen Fox, and Susan Hohenberger for implementing and experimenting with the ideas in this paper, and Greg Kulczycki for his careful reading and formatting.

We gratefully acknowledge financial support from our own institutions, from the National Science Foundation under grants CCR-9311702, DUE-9555062, and CDA-9634425, from the Fund for the Improvement of Post-Secondary Education under project number P116B60717, from the National Aeronautics and Space Administration under project NCC 2-979, from the Defense Advanced Research Projects Agency under project number DAAH04-96-1-0419 monitored by the U.S. Army Research Office, and from Microsoft Research. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U.S. Department of Education, NASA, the U.S. Department of Defense, or Microsoft.

References

1. Booch, G.: *Software Components With Ada.* Benjamin/Cummings, Menlo Park, CA (1987).
2. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: *Introduction to Algorithms.* The MIT Press, Cambridge, MA (1990).
3. Flamig, B.: *Practical Algorithms in C++.* Coriolis Group Book (1995).
4. Heym, W.D., Long, T.J., Ogden, W.F., Weide, B.W.: *Mathematical Foundations and Notation of RESOLVE*, Technical Report OSU-CISRC-8/94-TR45, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, Aug 1994.
5. Meyer, B.: *Object-Oriented Software Construction.* 2nd edn. Prentice Hall PTR, Upper Saddle River, New Jersey (1997).
6. Mehlhorn, K., Naher, S.: "LEDA: A Library of Efficient Data Structures and Algorithms" *Communications of the ACM 38*, No. 1, January 1995, 96–102.
7. Musser, D. R., Saini, A.: *STL Tutorial and Reference Guide.* Addison-Wesley Publishing Company (1995).
8. Parnas, D.L.: "On the criteria to be used in decomposing systems into modules." *Communications of the ACM 15*, No. 12, December 1972, 1053–1058.
9. Rushby, J., Owre, S., and Shankar, N.: "Subtypes for Specification: Predicate Subtyping in PVS," *IEEE Transactions on Software Engineering 24*, No. 9, September 1998, 709-720.
10. Sitaraman, M., Weide, B.W. (eds.): "Special Feature: Component-Based Software Using RESOLVE." *ACM SIGSOFT Software Engineering Notes 19*, No. 4, October 1994, 21–67.

11. Sitaraman, M: “Impact of Performance Considerations on Formal Specification Design,” *Formal Aspects of Computing 8*, 1996, 716–736.
12. Sitaraman, M., Weide, B.W., Ogden, W.F.: “On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations,” *IEEE Transactions on Software Engineering 23*, No. 3, March 1997, 157–170.
13. Weide, B.W., Ogden, W.F., Sitaraman, M.: “Recasting Algorithms to Encourage Reuse,” *IEEE Software 11*, No. 5, September 1994, 80–89.
14. Weihe, K.: “Reuse of Algorithms: Still A Challenge to Object- Oriented Programming,” *ACM SIGPLAN OOPSLA Proceedings*, 1997, 34–46.