

# The Separation Principle: A Programming Paradigm

**Yasushi Kambayashi**, *Nippon Institute of Technology*

**Henry F. Ledgard**, *University of Toledo*

**I**n the development of any technology, there's always a tendency to lose sight of the basic problems that stimulated its introduction in the first place. Technologies in software construction are no exception. One of the essential parts of software construction is the underlying programming paradigm.

In the last few decades, different software construction paradigms have evolved—for example, structured programming in the 1970s and object-oriented programming in the 1980s. Although OOP dominates current

software construction, it introduces several problems that might raise questions about its understandability and efficiency.<sup>1-6</sup>

William Cave was the first to propose and develop the Separation Principle as a programming paradigm.<sup>7,8</sup> One major contribution is that it moves most complex issues related to data sharing, parameter passing, and scope into a programming model in which simple graphics can represent access to data. The Separation Principle can simplify or even eliminate these issues. By expressing access permissions in simple, canonical drawings, the Separation Principle makes it easy to comprehend the full range of relationships between data and instructions, and

thus the code's structure. A simple way to show connectivity, a key property affecting program understandability, is by separating data from instructions. Two of Prediction Systems' flagship products, the Visual Software Environment (VSE) and the General Simulation System (GSS), embody this idea.<sup>9</sup>

Here, we discuss the Separation Principle in the context of conventional languages such as C and C++.

## A sample program

Consider a program that finds the shortest path from a start node to an end node in a given network (see Figure 1). The number given on each link represents the link's distance or cost. Such programs are often found in applications that involve routing and scheduling.

The program's general design is as follows:

1. Retrieve the network data (names of nodes and distance between each pair of nodes, and the start node and the end node) from a file.

**The Separation Principle is a simple, natural way of constructing programs. Its intuitive nature and ease of use make it useful in implementing many different software designs. A preliminary study showed that this programming paradigm improves programs' understandability.**

2. Construct a weighted-adjacency matrix out of the retrieved data to represent the network.
3. Compute the shortest path from the start to the end node using Dijkstra's algorithm.
4. Read the results in the intermediate results array backwards to produce the shortest path, and store it in an output array.

The code module needed to construct the adjacency matrix reads the input file and constructs the appropriate intermediate results array. The code module we call Dijkstra computes the shortest path to each node. Figure 2 shows the design for the shortest-path program using the Separation Principle. The connecting lines represent access permission. If a data module and a code module aren't connected, you can't access the data.

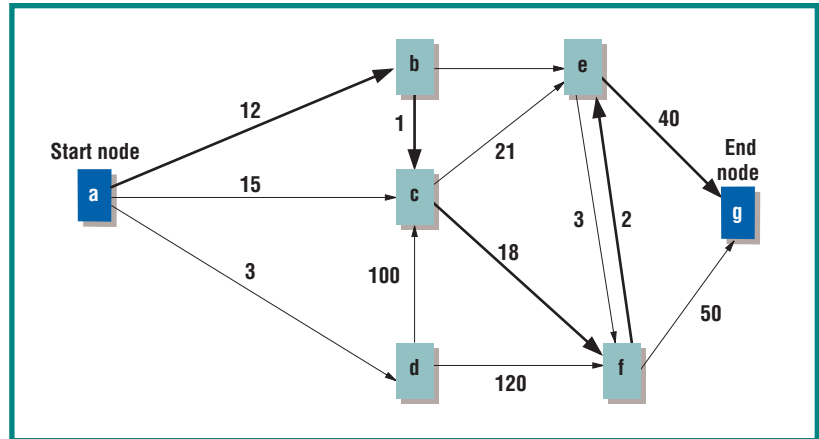
In programs that use the Separation Principle, all data is gathered in independent data modules (represented by ovals). In our design using the Separation Principle, we used a 2D matrix for the network and several arrays for storing the intermediate results, which our program uses when computing the shortest path.

The Network\_Config data module is local to the Construct\_Matrix code module. Three code modules share the Intermediate\_Results data module. The Dijkstra code module and the Output code module share the Resulting\_Path data module.

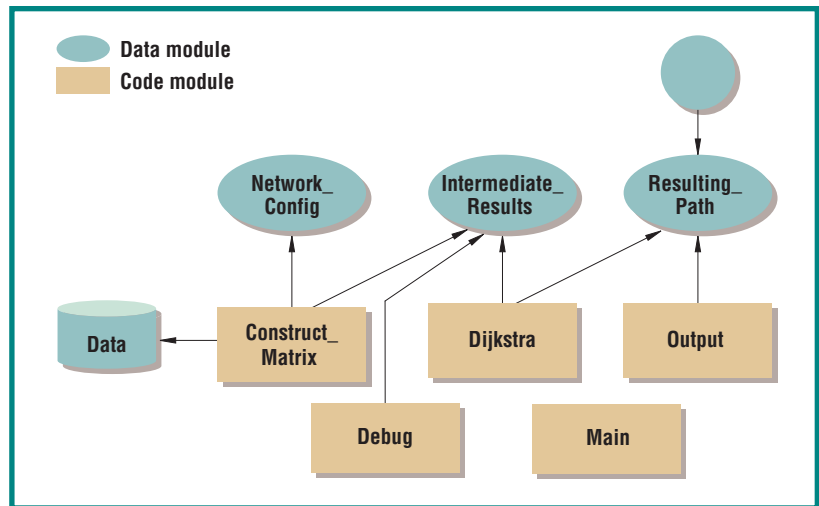
The Intermediate\_Results data module includes

- **node\_distances:** A 2D weighted-adjacency matrix to hold the distance between connected nodes
- **dist\_checked:** An array indicating whether each node's distance is a fixed or temporary value
- **dist\_from\_start:** An array that holds the distance from the start node
- **route:** An array that holds all nodes lying on the final shortest path

Figure 3 shows the Intermediate\_Results data



**Figure 1. Finding the shortest path from a to g. The number on each link represents the link's distance or cost.**



**Figure 2. Design diagram for finding the shortest path.**

```
int node_distances[NODE_NUM][NODE_NUM]; // weighted-adjacency matrix.
int dist_checked[NODE_NUM]; // whether distance is fixed or temporary.
int dist_from_start [NODE_NUM]; // distance from start node.
char route[NODE_NUM]; // previous node on path.

char startNode[2];
char endNode[2];
int numMax = 0; // maximum reachable number from start node.
int nearestNode; // nearest node from given node.
int minDistance; // minimum distance from current node.
int node; // temporary node for selecting closest node.
int flag; // true if all reachable vertices are checked.
```

**Figure 3. The Intermediate\_Results data module implementation.**

**Figure 4. The Dijkstra code module implementation with three procedures.**

```

int isComplete() {
    flag = TRUE;
    for (int i = 0; i < numMax+1; i++) flag = flag * checked[i];
    return flag;
}

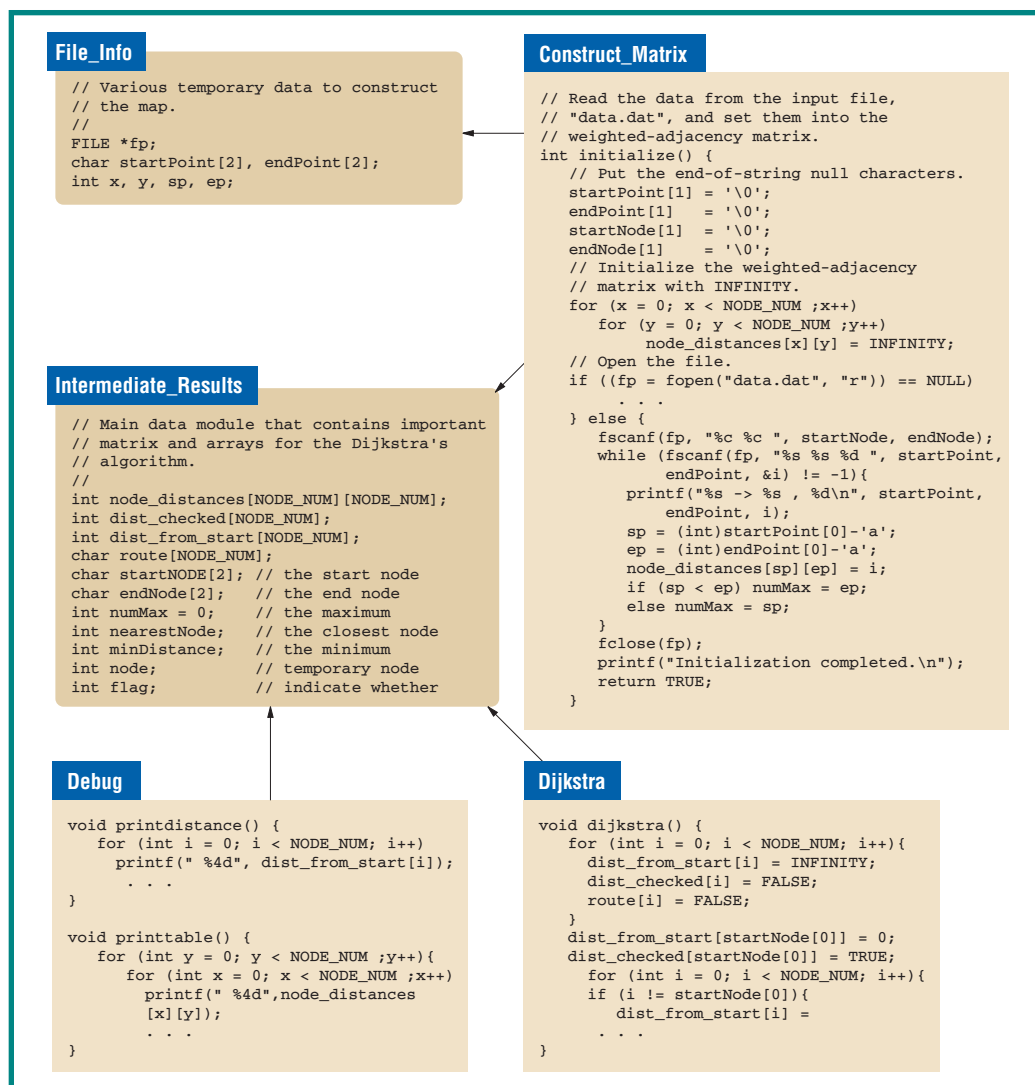
void minimumDistance() {
    nearestNode = -1;
    minDistance = INFINITY;
    for (int i = 0; i < NODE_NUM; i++){
        if (!checked[i]){
            if (minDistance > distance[i]){
                nearestNode = i;
                minDistance = distance[i];
            }
        }
    }
    return;
}

void dijkstra() {
    for (int i = 0; i < NODE_NUM; i++){
        dist_from_start [i] = INFINITY;
        dist_checked[i] = FALSE;
        route[i] = FALSE;
    }
    distance_from_start[startNode[0]] = 0;
    dist_checked[startNode[0]] = TRUE;
    for (int i = 0; i < NODE_NUM; i++){
        if (i != startNode[0]){
            distance_from_start[i] = node_distances[startNode [0]][i];
        }
    }
    while (!isComplete()) {
        minimumDistance();
        node = nearestNode;
        dist_checked[node] = TRUE;
        for (int i = 0; i < NODE_NUM; i++) {
            if (!dist_checked[i]) {
                if (dist_from_start[i] > dist_from_start[node]
                    + node_distances[node][i]) {
                    dist_from_start[i] = dist_from_start[node]
                        + node_distances[node][i];
                    route[i] = node;
                }
            }
        }
    }
    //printdistance(); // show the working array for debug.
}
}

```

module. Other data modules are Network\_Config and Resulting\_Path. Network\_Config holds the data from the file in the form of a

weighted-adjacency matrix, which represents how the system nodes are connected. Resulting\_Path stores the computed shortest path and



**Figure 5. Graphical representation of the shortest-path program.**

can be used in further computation if necessary. The code modules are defined similarly.

For example, consider the Dijkstra code module (see Figure 4), which has three procedures: One procedure checks if all reachable nodes have been considered, one selects the minimum distance node from the current candidates, and one implements Dijkstra's algorithm. We can continue this implementation method for other modules as well. We make here a key point due to Cave<sup>8</sup> and embodied in VSE and GSS. The program structure is easy to comprehend when it's shown graphically, as in Figure 5.

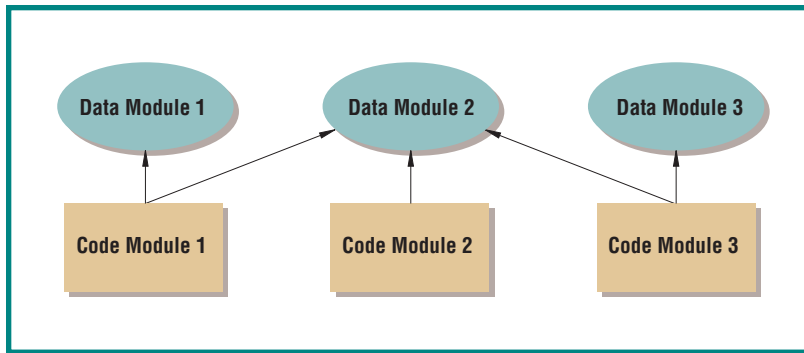
## The Separation Principle

We define the Separation Principle as follows:

- Declarations are stored in self-contained data modules.

- Executable statements are placed in self-contained code modules.
- Connecting lines are drawn between data modules and code modules that can access the data modules.
- Code modules can reference only the data modules to which they are connected.

We place all closely related data together in one data module. Code modules consist of functions that are sequences of instructions; they can contain a collection of procedures that usually relate to a common purpose. System complexity and understandability become more apparent through the lines connecting the data and code modules. The lines show that certain groups of instructions (code modules) can access related data directly. Too many connecting lines can indicate a



**Figure 6. Conceptual diagram of a program using the Separation Principle.**

high dependency between modules (and thus poor design).

With the exception of arguments to utility functions and iteration subscripts (defined in `for` statements), programs written using the Separation Principle should have no local data. We gather related data in a data module in such a way that no hierarchical structuring mechanism, such as inheritance, is needed.

Figure 6 presents a diagram of a simple program structure using this paradigm. This conceptual diagram shows data dependency only and not the function-call structure. The Separation Principle assumes that functions' visibility doesn't widely affect human understanding but that data dependency does. Data Modules 1 and 3 in the figure are local to Code Modules 1 and 3 respectively, while all the code modules share Data Module 2.

The Separation Principle gives us much more latitude to fit an application's structure to the program; we're not constrained to objects. It's a more free-flowing metaphor that we can readily use in meaningful designs.

### Simplicity of scope rules

In programs that use the structured programming principle, scope rules control the accessibility of data and procedures. These rules specify the part of the program in which the declaration of an identifier takes effect, and also to what extent that identifier is alive. The visibility rules specify where an identifier is visible or hidden within its scope.<sup>10</sup>

Scope rules contribute to a program's semantics. They're precisely defined and easily understood by parsing programs but not by human programmers. The Separation Principle makes programs less dependent on scope rules by separating data from instructions.

As far as function visibility goes, only two possibilities exist: the function exists in the

same code module, or it doesn't. There is no nested scope. If the desired function isn't in the same code module, it must be in another code module.

Even if two or more variables or functions have the same name, they're legal as long as they belong to different data or code modules. The names of variables or functions can be automatically augmented by the module names at compile time.

### Parameter passing

With conventional structured programming, it's not uncommon to find a chain of function calls in which a variable is simply passed on from one procedure to another. This situation results in a degree of excessiveness and uncertainty. The excessiveness comes with code that might repeatedly refer to the same variable. The uncertainty comes when you see a variable in a procedure: Is it a local variable, a parameter, or a global variable? Thus, the fact that there's no parameter passing in programs using the Separation Principle contributes to the programs' simplicity.

### Encapsulation

OOP provides a simple and powerful means for encapsulation. OOP languages provide a mechanism to express private and public data. The Separation Principle further pursues this distinction: public and private data are treated as different entities rather than as different attributes of the same entity.

If a data module is connected to only one code module, the data contained in that module are private to that code module. Data modules having more than one connection are shared. This simple scheme not only provides distinction between private and public data but also provides a way to control how public data are shared.

Therefore, there is no "all-public" data module. If you need an all-public data module, you have to connect that data module to all the code modules. Obviously, it's not a good idea.

In a design process, we can identify closely related code modules and then construct a data module that only those code modules use. In an OO design process using the Separation Principle, we can treat packaged collections of data modules and code modules as objects. We can combine several such components into a larger component, as Figure 7 shows.

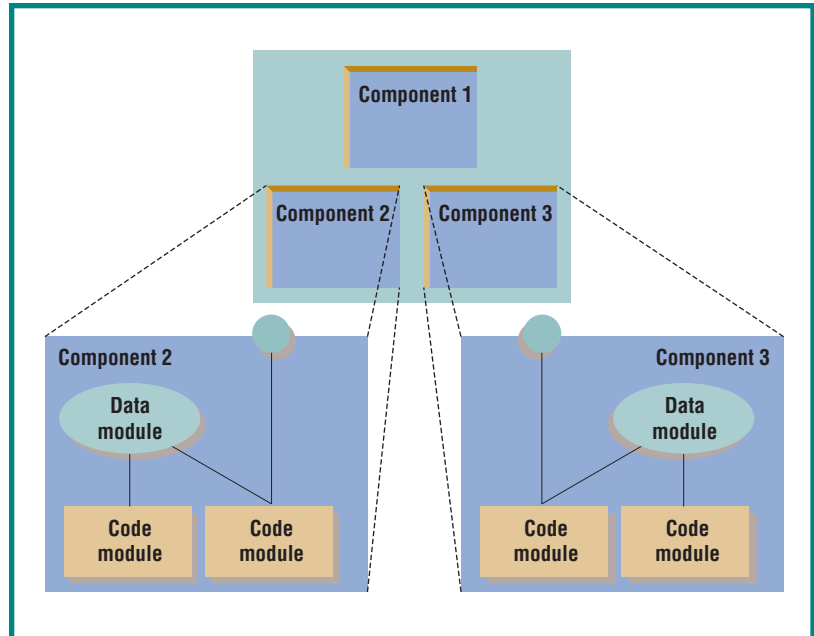
## Example airport program

Our second example is a program that simulates airplane landings and reports statistics about handling airplanes at a fictitious airport. The program assigns aircraft to gates when they arrive at an airport.

We consider three types of airplanes—a B747, an Airbus 300, and a Dash—and eight service gates. The airport simulation program behaves as follows:

- The B747, the Airbus 300, and the Dash have service time durations of two times, one time, and a half time of the nominal duration given, respectively. If all the gates are occupied, a new arrival is placed in a waiting queue.
- The three program inputs are the simulation's duration, the number of gates available, and the nominal service duration for planes.
- When the simulation ends, the program reports the number of airplanes immediately serviced, the maximum waiting period before service is completed, and the final states of the queue and the gates.

From the perspective of the Separation Principle, the implementation of this problem is intuitive and natural. For example, we know we'll need certain data describing the airport gates, so we have a data module called `Gates_Data` (see Figure 8). This module includes items such as the number of open gates, each gate's status, and an aircraft's arrival time at the gate. Other data modules are `Airplane_Data` for data on airplanes, `Queue_Data` for



**Figure 7. Data and code modules as reusable software components.**

managing the waiting queue, and `General_Data` for miscellaneous data controlling the simulation. The code modules are defined in a fashion similar to the example of finding the shortest path.

Consider the code module `Gates_Code` that operates on the gates. It has two procedures: One checks whether a plane has been completely serviced (see Figure 9a), the other moves a plane from the waiting queue to the next available gate (see Figure 9b). The variable `prog_clock` is a part of the program's `General_Data` module.

Figure 10 shows the overall design diagram for the airport program. As you might deduce, `Queue_Data` is local to `Queue_Code`, as is

```
struct gate_info {
    bool is_open;           // is gate open or not.
    int plane_id;           // ID of currently serviced plane.
    int time_arrive_gate;    // time an aircraft arrives at gate.
    int time_enter_queue;    // time an aircraft enters queue.
    int service_duration;    // duration for service.
    int time_completed;      // time aircraft completes servicing.
    int mins_unused;        // minutes that the gate is idle.
};

int num_open_gates;        // actual number of gates open.
gate_info gate[max_num_gates]; // service data for each aircraft.
```

**Figure 8. The `Gates_Data` module implementation.**

**Figure 9. The Gates\_Code module implementation. It contains two procedures: (a) checking if a plane has been completely serviced, and (b) moving a plane from the waiting queue to the next available gate.**

```
void check_gates_completed() {
    for (int i=0; i<num_open_gates ; i++) {
        if (gate[i].time_completed == prog_clock) {
            gate[i].is_open = true;
            if (prog_clock > 0) {
                cout << "aircraft " << gate[i].plane_id
                    << " completed service. ";
                cout << "gate " << (i+1) << " open.\n";
            }
        }
    }
}

(a)

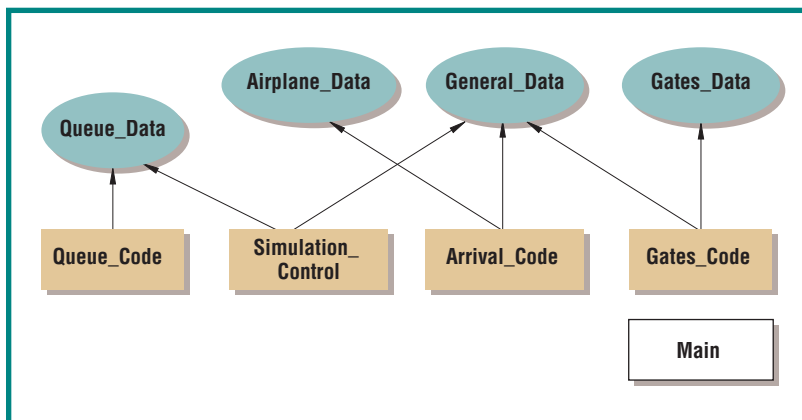
void move_planes_to_gates() {
    for (int i=0; i<num_open_gates ; i++) {
        if ((gate[i].is_open == true) && (queue_count > 0)) {
            airplane = remove();
            cout << "aircraft " << airplane->plane_id << " moved to gate "
                << (i+1) << endl;
            gate[i].plane_id = airplane->plane_id;
            gate[i].time_enter_queue = airplane->time_enter_queue;
            gate[i].time_arrive_gate = prog_clock;
            gate[i].time_completed = gate[i].time_arrive_gate +
                                    airplane->service_duration;

            gate[i].is_open = false;
            queue_time = gate[i].time_arrive_gate -
                        gate[i].time_enter_queue;

            if (queue_time > max_queue_time) {
                max_queue_time = queue_time;
            }
            if (queue_time == 0) {
                total_no_wait++;
            }
        }
    }
}

(b)
```

**Figure 10. Design diagram of the airport program.**



Gates\_Data to Gates\_Code and Airplane\_Data to Arrival\_Code. They are accessed only by the corresponding code modules (with the exception of a few calls from Simulation\_Control for the final statistics). Therefore, the data modules are securely encapsulated, and access control is simple and obvious. Figure 11 shows the airport program's graphical representation.

### Compatibility with concurrent programming

A third example is a simple concurrent program that solves the classic producer-



consumer problem. Figures 12 and 13 present a solution (a design diagram and a graphical representation, respectively) to this problem. The design diagram is surprisingly similar to the conceptual diagram of the Separation Principle (Figure 6). Here, the buffer data under mutual exclusion and the mutex variables are separately enclosed in a data module. Neither the consumer nor the producer code modules have local data, nor do they pass parameters between each other (`arg1` and `arg2` are just dummy arguments and aren't used). This ensures that data and instructions are clearly separated, and it's easy to see who's accessing the exclusive data.

One small data module is provided explicitly for data shared by both code modules. Because there's no parameter passing, the program structure is transparent and easy to understand.

## Reuse

Robert Fichman and Chris Kemerer performed extensive case studies on OO technology and software reuse.<sup>11</sup> They kept track of four industry projects from 1992 to 1996 and analyzed the problems concerning OO technology adoption. At all four sites, they found considerable challenges in transitioning to OOP and people who had difficulty grasping OOP concepts. Managers from two of the studies said they felt they needed OOP developers with years of experience to build an adequate programming team.<sup>11</sup> They also reported that OOP didn't play a major role in code reusability. They found that, when reusing code, programmers ported and salvaged code that they could use without OOP.<sup>11</sup>

OOP has high potential for reuse of components if you're willing to treat them purely as black boxes.<sup>12</sup> Many applications, however, use imperfect components. Thomas Niemann reported that in such applications, tracing program code through interconnected objects is difficult.<sup>13</sup> He found that different people had to read the program codes of these applications at different times, and they modified them repeatedly. In such applications, structural simplicity should be the first priority.

Michelle Cartwright and Martin Shepperd reported that the inheritance mechanism isn't widely used in practice, and programs with inheritance are more prone to defects than those without.<sup>14,15</sup> They also reported that programmers had more difficulty changing programs with inheritance than those without.<sup>14</sup>

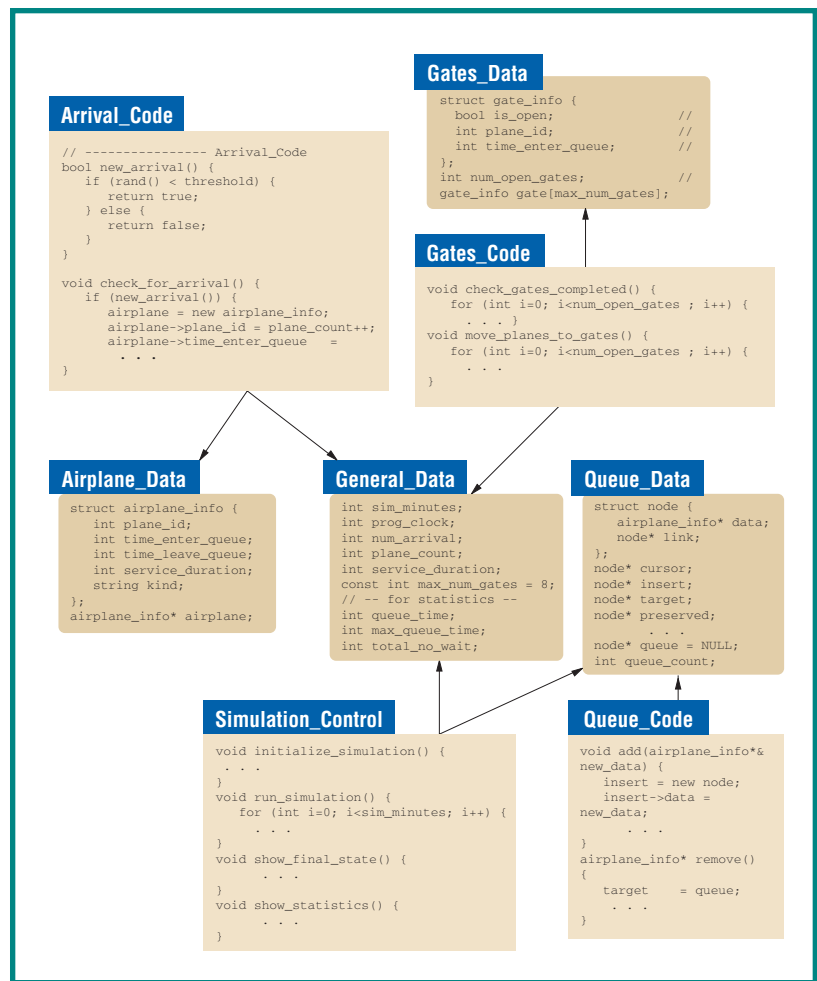


Figure 11. Graphical representation of the airport simulation program.

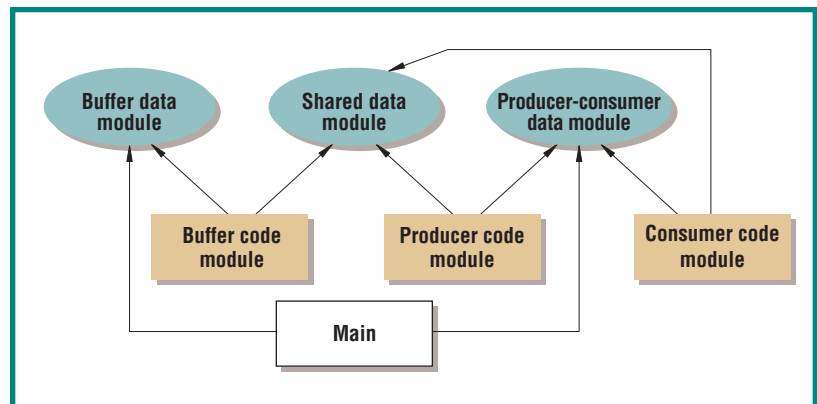


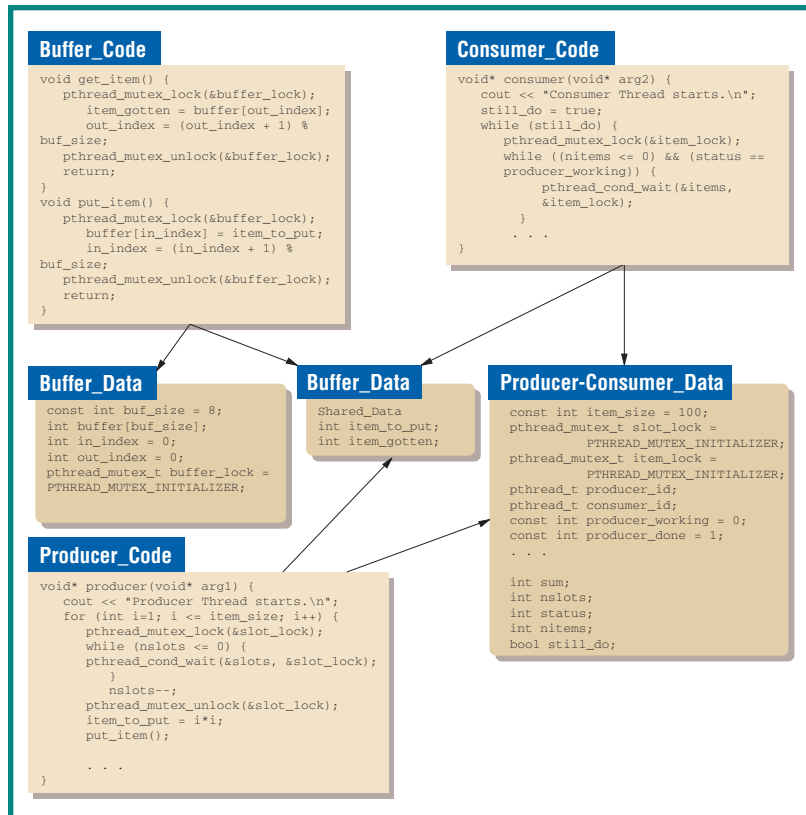
Figure 12. Design diagram of producer-consumer program.

If OOP-style reuse isn't practiced in real-world projects and if copy-and-paste reuse is still widely employed, then the visibility of data and code is clearly crucial to programmers. The Separation Principle provides such visibility.

## A limited experimental result

To see the Separation Principle's effectiveness as a programming paradigm, we conducted a





**Figure 13. Graphical representation of the producer-consumer program.**

controlled experiment comparing the understandability of programs written using the Separation Principle with those written using OOP style. Our results aren't really conclusive but are definitely suggestive.


We employed the following procedure in our experiments. We presented two semantically identical programs to two different groups of subjects (one to each group). The basic program was a business application that handled gallery management. One version of the program was written using OOP style, the other the Separation Principle. The subjects

were undergraduate students majoring in computer science and engineering. The members of each group read the programs and answered a set of questions about them. The questions were tailored so that we could ask the same set of questions for both programs. We then measured the number of correct answers for each subject and the time that the subjects needed to answer the questions. By comparing each group's scores, we identified the relative understandability of the two programs.

To analyze the experiment's results, we used the one-tailed unpaired t-test to explore hypotheses relating the programming paradigm's impact to the number of correct answers. Our data appeared to exhibit a near-normal distribution (note the similarity of the means and median in Table 1). We can

- Reject the null hypothesis that the programming paradigm in which a program is written had no impact on the number of correct answers (with 95 percent certainty) nor on the amount of time participants took to answer the questions (with 90 percent certainty)
- Conclude that the program written using the Separation Principle had a significant positive effect on the number of correct answers and on reducing the time spent to answer the questions

**O**ur human-understandability experiment, in which the subjects were relatively inexperienced programmers (college students) and the programs were relatively small, suggests the possible superiority of the Separation Principle over OOP regarding human comprehension. This shows some evidence that compels us to reconsider the belief in the OOP paradigm. The Separation Principle might provide a better programming alternative when understandability is important.

Also, we need different programming paradigms for different application areas. Our experience, and that of Prediction Systems, suggests that business applications might be well suited to the Separation Principle. 

## Acknowledgments

We thank William Cave, the Visual Software Environment's principal designer. His insights and observations are the source of many key ideas.

**Table 1**

### Number of correct answers and the number of minutes spent to answer questions

	Separation Principle		Object-oriented programming	
	No. of correct answers	No. minutes	No. of correct answers	No. minutes
Number of subjects	27	13	26	14
Mean	9.22	14.92	8.31	18.07
Median	3.00	15.00	9.00	17.00
Variance	3.564	19.58	3.102	25.30

Degree of freedom: 51 (25) Computed t-value: 1.789 (1.665)  
t distribution critical value: 1.676 (1.316 at alpha = 0.10, 1.708 at alpha = 0.05)

## References

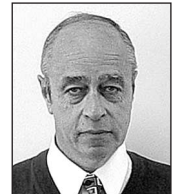
1. T.W. Pratt and M.V. Zelkowitz, *Programming Languages: Design and Implementation*, 4th ed., Prentice Hall, 1993.
2. R.W. Sebestai, *Concepts of Programming Languages*, 2nd ed., Benjamin/Cummings, 1993.
3. R. Sethi, *Programming Languages: Concepts and Constructs*, Addison-Wesley, 1989.
4. H.F. Ledgard, "The Emperor with No Clothes," *Comm. ACM*, vol. 44, no. 10, Oct. 2001, pp.126-128.
5. D. Anselmo and H.F. Ledgard, "Measuring Productivity in the Software Industry," *Comm. ACM*, vol. 46, no. 11, Nov. 2003, pp. 121-125.
6. M. Broy, "Object-Oriented Programming and Software Development—A Critical Assessment," *Programming Methodology*, A. McIver and C. Morgan, eds., Springer-Verlag, 2003.
7. Y. Kambayashi, *Separating Data from Instruction: Investigating a New Programming Paradigm*, doctoral dissertation, Dept. of Electrical Engineering and Computer Science, Univ. of Toledo, 2002.
8. W.C. Cave, *The Software Survivors*, Prediction Systems, 1995.
9. *The General Simulation System (GSS) Users Manual*, Prediction Systems, 2001.
10. R. Wilhelm and D. Maurer, *Compiler Design*, Addison-Wesley, 1995.
11. R.C. Fichman and C.F. Kemerer, "Object Technology and Reuse: Lessons from Early Adopters," *Computer*, vol. 30, no. 10, Oct. 1997, pp. 47-59.
12. B. Meyer, *Object-Oriented Software Construction*, 2nd

## About the Authors



**Henry F. Ledgard** is a professor in the Department of Electrical Engineering and Computer Science at the University of Toledo. His research interests include software and human engineering, and VSE in particular. He received his PhD in electrical engineering from Massachusetts Institute of Technology. Contact him at the Dept. of Electrical Eng. and Computer Science, Univ. of Toledo, 2801 W. Bancroft St., Toledo 43606-3398, Ohio; hledgard@eng.utoledo.edu.

**Yasushi Kambayashi** is an assistant professor in the Department of Computer and Information Engineering at the Nippon Institute of Technology. His research interests include theory of computation, theory and practice of programming languages, and intellectual property law. He received his PhD in engineering from the University of Toledo. Contact him at the Dept. of Computer and Information Engineering, Nippon Inst. of Technology, 4-1 Gakuendai, Miyashiro-cho, Saitama-ken, 345-8501, Japan; yasushi@nit.ac.jp.



ed., Prentice Hall, 1997.

13. T. Niemann, "Nuts to OOP!" *Embedded Systems Programming*, 1999, [www.embedded.com/1999/9908/9908feat1.htm](http://www.embedded.com/1999/9908/9908feat1.htm).
14. M. Cartwright and M. Shepperd, "An Empirical View of Inheritance," *Information & Software Technology*, vol. 40, no. 14, 1998, pp. 795-799.
15. M. Cartwright and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Trans. Software Eng.*, vol. 26, no. 8, Aug. 2000, pp. 786-796.

**PURPOSE** The IEEE Computer Society is the world's largest association of computing professionals, and is the leading provider of technical information in the field.

**MEMBERSHIP** Members receive the monthly magazine *Computer*, discounts, and opportunities to serve (all activities are led by volunteer members). Membership is open to all IEEE members, affiliate society members, and others interested in the computer field.

### COMPUTER SOCIETY WEB SITE

The IEEE Computer Society's Web site, at [www.computer.org](http://www.computer.org), offers information and samples from the society's publications and conferences, as well as a broad range of information about technical committees, standards, student activities, and more.

### BOARD OF GOVERNORS

**Term Expiring 2004:** Jean M. Bacon, Ricardo Baeza-Yates, Deborah M. Cooper, George V. Cybenko, Harubisha Ichikawa, Thomas W. Williams, Yervant Zorian

**Term Expiring 2005:** Oscar N. Garcia, Mark A. Grant, Michel Israel, Stephen B. Seidman, Kathleen M. Swigger, Makoto Takizawa, Michael R. Williams

**Term Expiring 2006:** Mark Christensen, Alan Clements, Annie Combelles, Ann Gates, Susan Mengel, James W. Moore, Bill Schilit

**Next Board Meeting:** 12 June 2004, Long Beach, CA

### IEEE OFFICERS

**President:** ARTHUR W. WINSTON

**President-Elect:** W. CLEON ANDERSON

**Past President:** MICHAEL S. ADLER

**Executive Director:** DANIEL J. SENESE

**Secretary:** MOHAMED EL-HAWARY

**Treasurer:** PEDRO A. RAY

**VP, Educational Activities:** JAMES M. TIEN

**VP, Pub. Services & Products:** MICHAEL R. LIGHTNER

**VP, Regional Activities:** MARC T. APTER

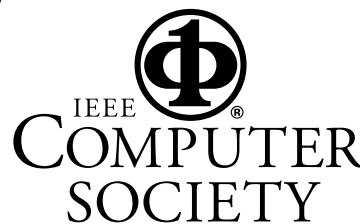
**VP, Standards Association:** JAMES T. CARLO

**VP, Technical Activities:** RALPH W. WYNDRUM JR.

**IEEE Division V Director:** GENE H. HOFFNAGLE

**IEEE Division VIII Director:** JAMES D. ISAAK

**President, IEEE-USA:** JOHN W. STEADMAN



### COMPUTER SOCIETY OFFICES

#### Headquarters Office

1730 Massachusetts Ave. NW

Washington, DC 20036-1992

Phone: +1 202 371 0101

Fax: +1 202 728 9614

E-mail: [bq.ofc@computer.org](mailto:bq.ofc@computer.org)

#### Publications Office

10662 Los Vaqueros Cir., PO Box 3014

Los Alamitos, CA 90720-1314

Phone: +1 714 821 8380

E-mail: [help@computer.org](mailto:help@computer.org)

#### Membership and Publication Orders:

Phone: +1 800 272 6657

Fax: +1 714 821 4641

E-mail: [help@computer.org](mailto:help@computer.org)

#### Asia/Pacific Office

Watanabe Building

1-4-2 Minami-Aoyama, Minato-ku

Tokyo 107-0062, Japan

Phone: +81 3 3408 3118

Fax: +81 3 3408 3553

E-mail: [tokyo.ofc@computer.org](mailto:tokyo.ofc@computer.org)



### EXECUTIVE COMMITTEE

#### President:

CARL K. CHANG\*

Computer Science Dept.

Iowa State University

Ames, IA 50011-1040

Phone: +1 515 294 4377

Fax: +1 515 294 0258

[c.chang@computer.org](mailto:c.chang@computer.org)

**President-Elect:** GERALD L. ENGEL\*

**Past President:** STEPHEN L. DIAMOND\*

**VP, Educational Activities:** MURALI VARANASI\*

**VP, Electronic Products and Services:**

LOWELL G. JOHNSON (1ST VP)\*

**VP, Conferences and Tutorials:**

CHRISTINA SCHOBER\*

**VP, Chapters Activities:**

RICHARD A. KEMMERER (2ND VP)†

**VP, Publications:** MICHAEL R. WILLIAMS†

**VP, Standards Activities:** JAMES W. MOORE†

**VP, Technical Activities:** YERVANT ZORIAN†

**Secretary:** OSCAR N. GARCIA\*

**Treasurer:** RANGACHAR KASTURIT

**2003-2004 IEEE Division V Director:**

GENE H. HOFFNAGLE†

**2003-2004 IEEE Division VIII Director:**

JAMES D. ISAAK†

**2004 IEEE Division VIII Director-Elect:**

STEPHEN L. DIAMOND\*

**Computer Editor in Chief:** DORIS L. CARVER†

**Executive Director:** DAVID W. HENNAGE†

\* voting member of the Board of Governors

† nonvoting member of the Board of Governors

### EXECUTIVE STAFF

**Executive Director:** DAVID W. HENNAGE

**Assoc. Executive Director:** ANNE MARIE KELLY

**Publisher:** ANGELA BURGESS

**Assistant Publisher:** DICK PRICE

**Director, Finance & Administration:**

VIOLET S. DOAN

**Director, Information Technology & Services:**

ROBERT CARE

**Manager, Research & Planning:** JOHN C. KEATON