A Programming Environment for the Separation Principle

Yasushi Kambayashi Shigeru Kawano
Nippon Institute of Technology

1 Introduction

The separation principle is a programming paradigm that is first proposed by Cave [1], and refined and validated by Kambayashi [2]. The separation principle has two concepts: 1) separating architecture from program code and 2) separating data from instructions. Preliminary experiments on the understandability of programs written using the separation principle and programs written using object-oriented style revealed that programs written using the separation principle were easier to understand than those written using object-oriented style [2].

In order to take full advantages of the separation principle, a programming environment is required. In this paper, the authors report the design and implementation of such environment.

2 The Separation Principle

The separation principle is a programming paradigm that constraints programmers to construct their programs in separated data modules and instruction modules. In object-oriented programming paradigm, programs consist of multiple *objects*. Each object consists of data and instructions that manipulate the data.

In programs that use the separation principle, all the data are gathered in independent data modules. There is no local data, and certain groups of instructions can access related data directly, thus no parameter passing is required. Related data are gathered in a data module. A conceptual diagram of the structure of a program using the separation principle is shown in Figure 1. A notable point is that it shows only data dependency.

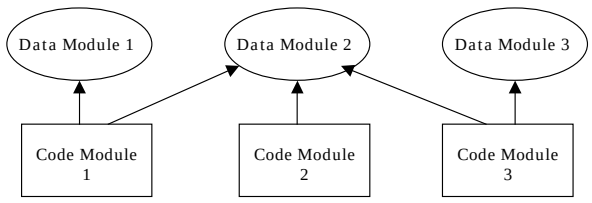

Figure 1. Conceptual Diagram of a Program Using the Separation Principle

3 The Programming Environment

The user of this programming environment is expected first to construct the architecture of the software he or she wishes to construct. This phrase is achieved through the *architecture editor* of the environment. In the design process, one can identify closely related instruction modules, and then one can construct data modules that are used by the instruction modules. Such combined modules can be treated as components. Several such components can further combined into a larger component. This process is achieved by the architects, and should be supported by the architecture editor. The conceptual diagram of such editor is shown in Figure 2.

Two separate methodologies that construct software from components are established. One is for the structured design and the other is for the object-oriented design. They are almost identical and only differ in minor ways. The architecture editor is expected to support to construct software hierarchically from components. A few closely related data modules and instruction modules are grouped into higher-level modules. Then these kinds of modules are further combined, forming a larger, structured component.

After architecture is set (then the target software is decomposed into several set of components), the programmer precedes the detailed design by using the programming environment. The designer has to identify the data modules and instruction modules and their dependency. Data/instruction dependency is represented as a set of connecting lines. Each data/instruction module is edited in a separated window. Each instruction module has explicit connections to certain data modules. If a data module

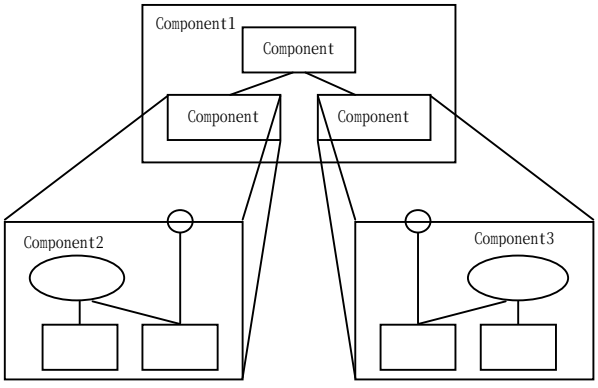

Figure 2. Architecture Support Graphical Environment

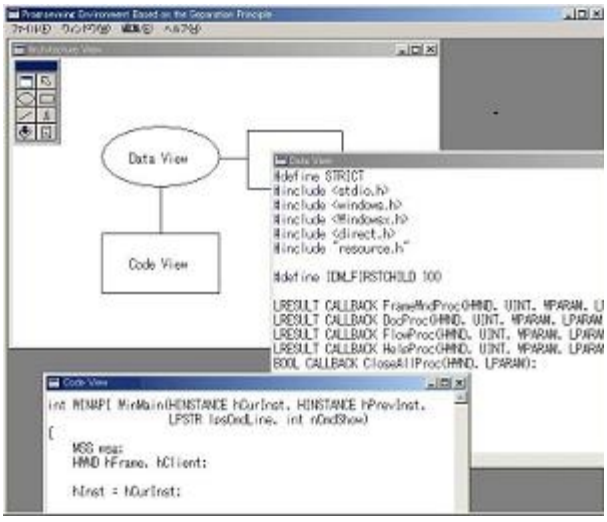

Figure 3. Programming Environment Based on the Separation Principle

has three connections to three instruction modules, that data module is public to three instruction modules, and not public to any other instruction modules. The programmer is expected to control this kind of public/private relation, and the programming environment supports such activity through graphical representation and intuitive manipulations. In summary, connected data modules and instruction modules consists of a package that provides a mechanism for grouping and data sharing like Ada packages. Figure 3 shows the programming environment that is editing such a package (component). This component consists of one data module and two instruction modules, and each module is edited in a separate window. (At this moment, each component is decomposed into several data modules and instruction modules.)

Equipping two different editors: one for architects with graphic representation and one for programmers with text representation is, the authors believe, provides a good example of balancing text and graphics [3].

4 The Compiler

Although Programs produced by this programming environment have C-like syntax, they have different scope rules. The scope of each identifier is explicitly shown by the connections between the data and instruction modules. The effect of the declaration is not controlled by the scope rules, because the instruction modules contain no data declarations. The visibility of variables is explicitly expressed by the connections. This simplicity of the scope rules contributes the simplicity of the compiler the environment supplies. The compiler performs the semantic analysis by using these connections.

The compiler is supposed to produce C source programs for optimized C compiler available in the market as well as intermediate code for the virtual machine the programming environment furnishes. The virtual machine is a simple stack machine that reads the intermediate code and produces the computation results. Since the purposes of the virtual machine are evaluation and debugging, the authors keep the machine simple and follow the design formulated by Wirth [5].

5 Conclusions and Future Directions

A programming environment based upon the separation principle is presented. Even though as mentioned in section three, the architecture editor has not been implemented, the programming environment is completed and used to support the programming paradigm, separating data from instructions. The architecture editor should be a competent feature of this environment and support the separation between design phase and implementation phase. This feature should contribute a clear separation between them. It should be a future research direction to explore the separation principle in the object-oriented analysis methodology.

The environment is planned to be extended to provide a run-time facility including debugger. This run-time environment not only provide the virtual machine on which the produced intermediate code run but also bi-directional pointers between data and instructions so that the debugger can identify which instructions access which data and provide enough information for run-time anomaly.

Because each instruction module should be short, straight and simple, it may be possible to be produced by a program generator. Exploring the instruction generator is another possible future directions of this project.

References

[1] W. C. Cave, *The Software Survivors*, Prediction Systems, Inc., 1995.

[2] Y. Kambayashi, *Separating Data from Instructions: Investigating a New Programming Paradigm*, PhD Dissertation, University of Toledo, May 2002, published by Dissertation.com, 2002.

[3] Y. Kambayashi, Balancing Text and Graphics in a Programming Environment, *Proc. of Winter Workshop on Software Engineering in Kobe*, IPSJ, January 2003, to appear. In Japanese.

[4] J. B. Rosenberg, *How Debuggers Work*, John Wiley & Sons, Inc., 1997.

[5] N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.