Appears in:

INFORMATION PROCESSING '92

Proceedings of the 12th World Computer Congress

Madrid, Spain, 7-11 September 1992

Volume 1: ALGORITHMS, SOFTWARE, ARCHITECTURE

Edited by J. van Leeuwen

Component Enhancement: An Adaptive Reusability Mechanism for Groups of Collaborating Classes

Karl J. Lieberherr

Center for Software Sciences, College of Computer Science, Northeastern University,
161 Cullinane Hall, 360 Huntington Ave., Boston MA 02115
lieber@corwin.CCS.northeastern.EDU

Abstract

We enhance the Demeter Method for object-oriented software development with the component model for describing the evolution of groups of collaborating classes. The model is based on class dictionary graphs [LBSL91, Ber91] and reusable behavior descriptions which are expressed by propagation in class dictionary graphs [LXSL91]. Our experience demonstrates that the component model lifts programming to a higher level of abstraction, it significantly reduces the size of programs and it makes them resilient to change and therefore we propose a reusability mechanism for components. Components are useful for the development of reusable software libraries, for application development, as well as for recording the history of object-oriented programs. An implementation of propagation patterns, which are an important part of components, and related papers are available by ftp [LBH+91].

Keyword Codes: D.1.5, D.2.2, D.3.3
Keywords: Object-oriented Programming, Tools and Techniques, Language Constructs and Features.

1 Introduction

In the treaty of Orlando paper [SLU87], Stein, Lieberman and Ungar provide the motivation for this paper: "Occasions arise when we would like to specialize or extend not just a single object, but an entire hierarchy at once. No existing object-oriented language provides a mechanism for this that does not require the modification of previously existing code. ..." In this paper we propose a mechanism, called *components*, for reusing a group of evolving classes.

This paper makes a contribution to reusability of software artifacts with minimal "from scratch" development. We focus on a reuse mechanism at the design/source code level with the following applications:

- Development of reusable component libraries which will be distributed in source code form. For example, libraries proprietary to a company fall into this category.
- Reuse within the same project. We introduce a mechanism to reuse part-of and kind-of relations between classes to formulate the behavior of classes.

The adaptiveness of components comes from specifying methods without hard-wiring them to classes. The method specifications adjust themselves in a useful way to the environment they live in.

A component (in the sense used in this paper) consists essentially of a class dictionary graph (a set of classes with relations) and a list of propagation patterns. A **class dictionary graph** is a directed graph in which vertices represent classes, and edges represent the part-of and kind-of relations on those classes. Labels on edges give names to parts of a class. The classes are partitioned into construction classes and alternation classes. The construction classes are instantiable classes while the alternation classes are abstract, uninstantiable classes. A class dictionary graph is given by a tuple $(VC, VA, \Lambda; EC, EA)$, where VC is a set of construction vertices and VA is a set of alternation vertices. Λ is a set of labels naming part-of relations; EC is a set of labeled edges (triples), called *construction edges*, defining part-of relations between classes; while EA is a set of edges (pairs), called *alternation edges* defining kind-of relations between classes. We use the following graphical notation which is explained in more detail in [LBSL91, Ber91] and demonstrated by example in Fig. 1: construction edges: single-shafted arrows: $\longrightarrow$, alternation edges: double-shafted arrows: $\Longrightarrow$, construction vertices: □, alternation vertices: ⬭.

A **propagation pattern** consists of an interface specification (in C++ notation in this paper) which is propagated to a set of classes along the class relationships. These classes are called implementor classes of the interface. Each class which is reached by the propagation gets a custom generated body, unless it is overridden with a primary code fragment, or modified with before and after code fragments. Inside code fragments, programming language statements are shown in C++ notation between `(@` and `@)`. Propagation patterns take advantage of the connectivity of a class dictionary graph to specify propagation of messages in a concise manner. The propagation pattern tool uses propagation pattern specifications to generate uninteresting methods automatically. The simplest propagation pattern that can be written for any class dictionary graph is one that generates a pre-order object traverser: `*interface* void traverse() *from* Start`. `Start` is the name of the class from which the traversal wants to be started. Running the propagation pattern tool on this propagation pattern results in a method (specifically, a C++ member function) being generated for each class for which there is a path from class `Start`. Each method calls the traverse function for all the parts of the method's class and is implemented complying with the strong Law of Demeter[LHR88]; indeed, propagation patterns can be viewed as a tool to easily comply with the Law of Demeter.

To write a propagation pattern, the interface of the propagated method is specified first, then the class or classes which are the sources of the propagation. If no target classes are specified, every class that is at the end of any path from any of the source classes is considered, as is the case for the traverser. Optionally, a number of edges can be specified as bypassing or through edges. Bypassing edges cause paths in the graph to be eliminated. Through edges cause paths in the graph to be exclusively taken. A formal introduction to propagation patterns and a definition of their semantics is given in [LXSL91].

A simple example of a component is shown in Fig. 2. It implements the depth-first-traversal algorithm which we use as example throughout this paper to discuss components and their enhancement. We describe graphs by the data model given in Fig. 1 using the graphical notation explained above. The component in Fig. 2 implements the depth-first

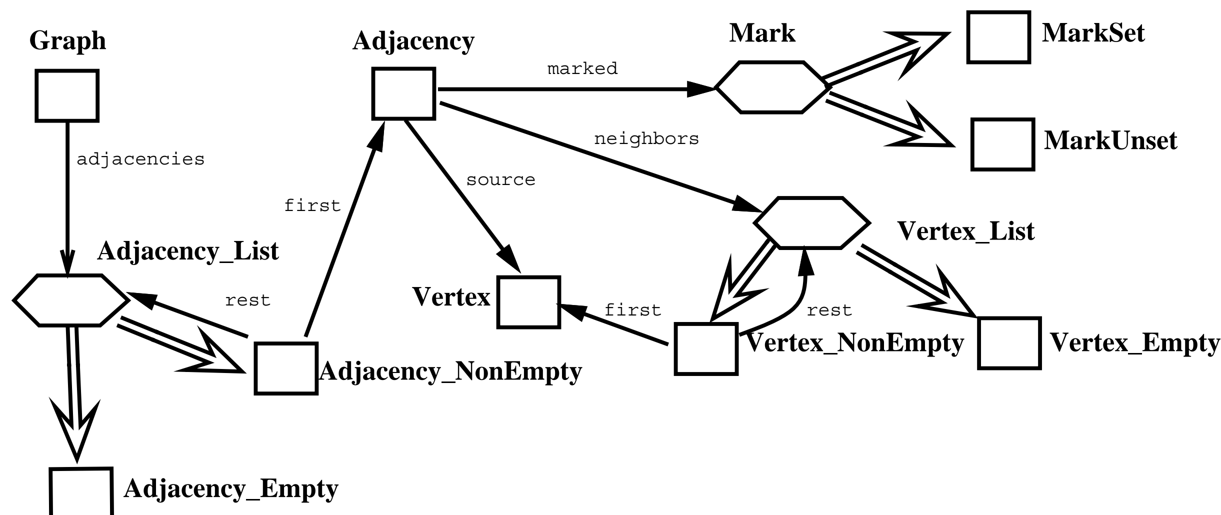

Figure 1: Class dictionary graph for graphs

traversal algorithm in terms of the data model in Fig. 1. We assume that the input graph object is properly initialized.

The algorithm is activated by calling function dft for an Adjacency-object (lines 3+4). This function will eventually call the unconditional version of dft (called uncond_dft, on line 8), if the adjacency is not marked, by using the collaboration of the classes Mark and MarkUnset (line 6). The condition whether an adjacency is marked, is not checked explicitly by the algorithm. Instead an Adjacency-object is passed (lines 4+5) to Mark and MarkUnset and only in MarkUnset, the unconditional version of dft is called (line 8). To accomplish the unconditional traversal, we need classes from Adjacency to Vertex to cooperate (line 10). It is important not to include the source part of Adjacency and therefore we formulate the graph with two from-to clauses.

To implement the find operation, we need the cooperation of classes from Graph to Adjacency (line 13). It is interesting to observe that this searching algorithm could be made more efficient by using the *cut* feature of propagation patterns which is not discussed in this paper. A nicer implementation of the algorithm could be achieved by using a derived edge from Vertex to Adjacency with name find.

```
1  *component* DFT
2    *cd* see Fig. 1
3    *interface* void dft(Graph* g)
4      *primary* Adjacency (@ marked->dft(g, this); @)
5    *interface* void dft(Graph* g, Adjacency* adj)
6      *from* Mark *to* MarkUnset
7      *primary* MarkUnset (@ adj->g_print();
8        adj->set_marked(new MarkSet()); adj->uncond_dft(g); @)

9    *interface* void uncond_dft(Graph* g)
10     *from* Adjacency *to* Vertex_List *from* Vertex_List *to* Vertex
11     *primary* Vertex (@ g->find(g, this); @)

12   *interface* void find(Graph* g, Vertex* v)
13     *from* Graph *to* Adjacency
14     *primary* Adjacency (@ if (v->g_equal(source)) adj->dft(g); @)
15 *end* DFT
```

Figure 2: Depth-first traversal

2 Component enhancement

Next we generalize the depth-first traversal algorithm to a graph cycle-checking algorithm. It maintains a path of vertices from the start vertex to the current vertex. If the next vertex during the traversal is a vertex which is already on the path, a cycle has been found. We use a parameterized Stack class which keeps track of the path. To get the cycle checker, we enhance the DFT component as follows (we omit the implementation of the push, pop and contains_duplicate functions of the stack class):

```
*component* CYCLE
  *reuse-cd* DFT
  *reuse-pp* DFT
    *code-change*
      *interface* void dft(Graph* g)
        *add-arguments*  Stack<Adjacency>* s
        *add-fragments*
          *before* Adjacency (@ s->push(this);
            if (s->contains_duplicate(this)){
              s->g_print(); cout << "cycle found";} @)
          *after* Adjacency (@ s->pop(); @)
*end* CYCLE.
```

In the above dft function we only changed the interface dft(Graph* g) and we added a before and an after code fragment. Extra formal and actual arguments for the definitions and

calls of the other functions are provided automatically. This is an elegant reuse of the basic traversal algorithm and compares favorably with corresponding discussions in algorithm textbooks. One key advantage of the approach given here is that no parameterization of the dft algorithm is needed: we just add new information and replace old one.

The before and after code fragments go before and after the primary code fragment of class Adjacency. Note how convenient it is just to provide additions to the generated primary code fragment, instead of writing the complete primary code fragment for Adjacency.

2.1 Adaptiveness of components

Components are more reusable than standard object-oriented software. We demonstrate their flexibility by changing the class structure for the traversal problem. Instead of traversing graphs having only one kind of edges, we traverse now graphs with two kinds of edges and we want the traversal to be done with respect to both kinds of edges. The new graph data model is shown in the class dictionary graph in Fig. 3. To adjust the

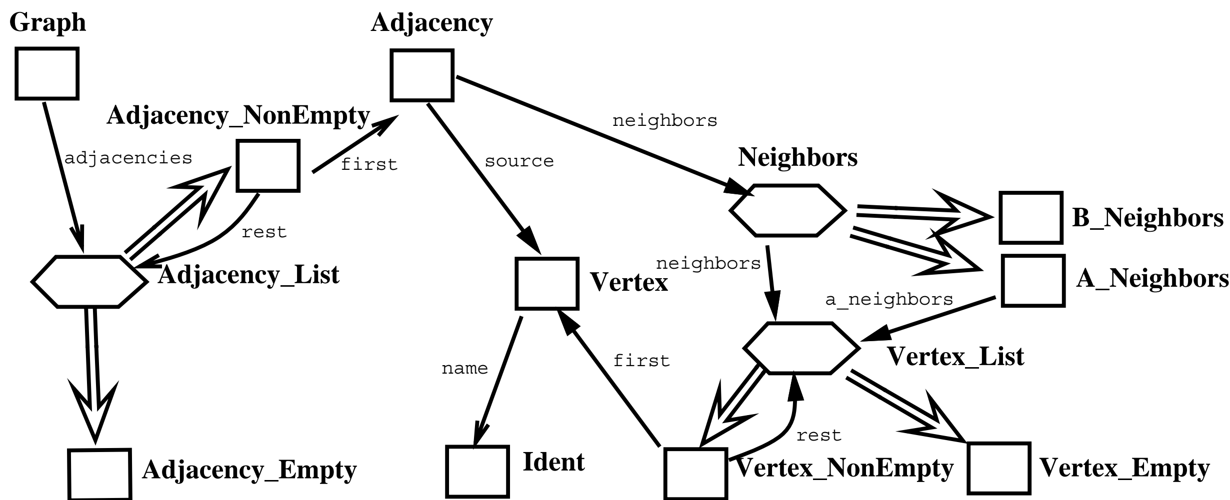

Figure 3: Extended graph data model

algorithm to the new requirements, nothing needs to be changed, i.e., the propagation patterns in Fig. 2 stay invariant. Although we have introduced the new classes Neighbors, A_Neighbors and B_Neighbors, we do not have to write code for them. The propagation patterns in the component will do it for us.

3 Related work

A component defines groups of collaborating objects and therefore a component can be viewed as a generalization of a contract [HHG90, Hol92]. The first distinguishing feature of our notation is that we localize interfaces: we first define an interface and then we list the classes which implement that interface. The second distinguishing feature is that we propagate interfaces to groups of classes and that we generate default bodies. This leads

to a classification of participant obligations into important and unimportant ones. The unimportant ones use the default code body and they are not explicitly mentioned in a component, while a contract would include them explicitly. Component enhancement is related to contract refinement, contract inclusion and contract conformance. Components integrate the useful features of class dictionaries, propagation patterns and contracts into a powerful abstraction mechanism for object-oriented software development.

Programming environments which support reusability are addressed in [Hab88]. [Deu83] addresses reusability questions in object-oriented systems. Deutsch distinguishes between algorithm reusability across data structures and framework reusability across applications.

Clichés and the plan calculus of [RW86] are related to components. [SE83] supports the claim that programmers reuse programming plans. The Demeter Method follows this idea and views a class dictionary as a first approximation to a programming plan. The programming plan is further refined with propagation patterns.

4 Conclusions

Components have a good potential to significantly improve designer and programmer productivity for numerous application domains. Others and we have accumulated significant experience with using components. Components have been used at a major bank to validate parts of a business model. Components have also been shown to be programming language independent and useful for implementing a graphical user interface for the Demeter Tools an a PC using Turbo Pascal. Sägesser, who implemented the user interface, confirmed our experience that components reduce typing significantly, they reduce dependencies between classes and they make changes (especially interface changes) easier.

Motivated by the early success of propagation patterns, we have shown in this paper how components can be reused to serve as a powerful abstraction mechanism for object-oriented system development. The discussion has been informal and ignoring many technical details, but a formal treatment will be given in the journal version of this paper.

Acknowledgements: I would like to thank Cun Xiao for his maintenance of the propagation pattern tool without which it would be impossible to do this work. Walter Hürsch and Ignacio Silva-Lepe support other vital parts of the Demeter System/C++ in which propagation patterns live and to which components will be added.

The 45 graduate students in my fall 1991 object-oriented systems class, coached by Ph.D. students Paul Bergstein, Ian Holland, Walter Hürsch, Ignacio Silva-Lepe, Greg Sullivan, Linda Seiter, and Cun Xiao, have given me valuable feedback.

This work is supported in part by IBM, Mettler-Toledo, Citibank and the National Science Foundation under grants CCR-9102578 (Software Engineering) and CDA-9015692 (Research Instrumentation). Demeter is a trademark of Northeastern University.

References

[Ber91] Paul Bergstein. Object-preserving class transformations. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 299–313, Phoenix, Arizona, 1991. ACM Press. SIGPLAN Notices, Vol. 26, 11 (November).

[Deu83] Peter Deutsch. Reusability in the Smalltalk-80 Programming System. In *Proc. Workshop on Reusability in Programming*, pages 72–82, Newport, RI, 1983.

[Hab88] A. N. Habermann. Programming environments for reusability. In *Proceedings of the Annual Hawaii International Conference on System Sciences*, pages 1–10, 1988.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 169–180, Ottawa, 1990. ACM Press. Joint conference ECOOP/OOPSLA.

[Hol92] Ian M. Holland. Specifying reusable components using contracts. In *European Conference on Object-Oriented Programming*, pages 287–308, Netherlands, 1992. Springer Verlag.

[LBH+91] Karl Lieberherr, Paul Bergstein, Ian Holland, Walter Hürsch, Ignacio Silva-Lepe, Linda Seiter, and Cun Xiao. Demeter papers and license agreement. In *FTP distribution from corwin.ccs.northeastern.edu or 129.10.8.150 in pub/demeter*. Northeastern University, 1991.

[LBSL91] Karl J. Lieberherr, Paul Bergstein, and Ignacio Silva-Lepe. From objects to classes: Algorithms for object-oriented design. *Journal of Software Engineering*, 6(4):205–228, July 1991.

[LHR88] Karl J. Lieberherr, Ian Holland, and Arthur J. Riel. Object-oriented programming: An objective sense of style. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, number 11, pages 323–334, San Diego, CA., September 1988. A short version of this paper appears in IEEE Computer, June 88, Open Channel section, pages 78-79.

[LXSL91] Karl Lieberherr, Cun Xiao, and Ignacio Silva-Lepe. Propagation patterns: Graph-based specifications of cooperative behavior. Technical Report NU-CCS-91-14, Northeastern University, September 1991.

[RW86] Charles Rich and Richard C. Waters. *Readings in Artificial Intelligence and Software Engineering*. Morgan Kaufmann Publishers, 1986.

[SE83] Elliot Soloway and Kate Ehrlich. What do programmers reuse? In *Proc. Workshop on Reusability in Programming*, pages 184–191, Newport, RI, 1983.

[SLU87] Lynn Andrea Stein, Henry Lieberman, and David Ungar. The Shared View of Sharing: The Treaty of Orlando. In Won Kim and Frederick H. Lochovsky, editors, *Object-oriented Concepts, Databases, and Applications*, pages 31–47. ACM Press, 1987.