# Adaptive Object-Oriented Programming using Graph-Based Customization

Karl J. Lieberherr, Ignacio Silva-Lepe and Cun Xiao
Northeastern University, College of Computer Science
Cullinane Hall, Boston MA 02115
lieber@ccs.neu.edu, phone: (617) 373 2077

**Abstract**

Object-oriented programs are easier to extend than programs which are not written in an object-oriented style, but object-oriented programs are still very rigid and hard to adapt and maintain. In this article, we introduce adaptive object-oriented programming as an extension to conventional object-oriented programming. Adaptive object-oriented programming facilitates expressing the elements - classes and methods - that are essential to an application by avoiding to make a commitment on the particular class structure of the application. Adaptive programs are specified using propagation patterns which specify sets of related constraints on class structures. An adaptive program denotes an entire family of programs, as many programs as there are class structures which satisfy its constraints. A class structure which satisfies the constraints of an adaptive program is said to customize the program. Adaptive programming, realized by the use of propagation patterns, offers a new paradigm, extending the object-oriented paradigm by lifting programming to a higher level of abstraction.

**Keywords and Phrases:** Object-oriented design and programming, reusable software, class dictionaries, class evolution, Law of Demeter, software engineering methods.

## 1 Introduction

Object-oriented programs are easier to extend than programs which are not written in an object-oriented style, but object-oriented programs are still very rigid and hard to adapt and maintain. A key feature of most popular approaches to object-oriented programming is that methods are attached to classes - C++, Smalltalk, Eiffel, Beta - or to groups of classes - CLOS. This feature is both a blessing and a curse. On the brighter side, attaching methods to classes is at the core of (1) objects being able to receive messages, (2) different classes of objects responding differently to a given message, and (3) the ability to define standard protocols. On the darker side, by explicitly attaching every single method to a specific class, the details of the class structure are encoded into the program unnecessarily. This leads to programs which are hard

to evolve and maintain. In other words, today's object-oriented programs often contain more redundant application specific information than is necessary, thus limiting their reusability.

Does this mean that we have to either take the curse in order to enjoy the blessing or give up the blessing altogether? Analyzing the problem we realize that not all is lost. What we need is to be able to specify only those elements that are essential to an object-oriented program and then specify them in a way that allows them to adapt to new environments.

What do we mean by specifying only those elements - classes and methods - that are essential to an object-oriented program? In [WH91], Wilde and Huitt point out: "There is a general impression that object-oriented programs may tend to be structured rather differently than conventional programs. For many tasks very brief methods may be written that simply 'pass through' a message to another method with very little processing." Such 'traversal, pass through' methods we regard as non-essential. But more importantly, we intend to focus on classes and methods that are essential not only to a particular application but also potentially to a family of related applications.

How can we identify such generic classes and methods? Consider the following "paradox of the inventor" posed by mathematician George Polya[Pol49]. Polya observed that it is often easier to solve a more general problem than the one at hand and then to use the solution of the general problem to solve the specific problem. The hard work consists of finding the appropriate generalization. Polya uses the following example to demonstrate the technique: Given are a line and a regular octahedron. Find a plane that contains the line and that cuts the volume of the octahedron in half. What is important about the regular octahedron to provide for an easy solution? The fact that it is a symmetric body. Given any symmetric body, the solution consists of choosing the plane which contains the given line and the center of symmetry. The general solution is easily applied to solve the specific octahedron problem. Applying Polya's paradox of the inventor to object-oriented program design results in more adaptive programs being written, programs that adjust gracefully to specializations of the generalization they were designed for.

In this article we introduce *adaptive object-oriented programming* as an extension to conventional object-oriented programming. Adaptive object-oriented programming facilitates expressing the elements - classes and methods - that are essential to an application by avoiding to make a commitment on the particular class structure of the application. Adaptive object-oriented programs specify essential classes and methods by constraining the configuration of a class structure that attempts to customize the adaptive program, without spelling out all the details of such a class structure. This way, adaptive object-oriented programmers are encouraged to think about families of programs by finding appropriate generalizations, in the spirit of Polya.

This article is organized as follows. Section 2 introduces adaptive programs, describing their structure. Adaptive programs[1] are specified using *propagation patterns*, which express program constraints. Propagation patterns are introduced in section 3. An adaptive program denotes an entire family of programs, as many programs as there are class structures which satisfy its constraints. A class structure which satisfies the constraints of an adaptive program is said to

[1] In the remainder of this article we refer to adaptive object-oriented programs simply as adaptive programs.

customize the program, and is specified as a *class dictionary graph*. Class dictionary graphs and customization of adaptive programs are introduced in section 4. Finally, section 5 describes related work and section 6 concludes this article.

## 2 Adaptive Programming

Conventional object-oriented programs consist of a structural definition in which a class structure is detailed, and a behavioral definition where methods attached to the classes in the class structure are implemented. Likewise, adaptive programs are defined structurally and behaviorally. What makes an adaptive program different is that class structures are described only partially, by giving a number of constraints that must be satisfied by a customizing class structure. In addition, behavior is not implemented exhaustively. That is, methods in an adaptive program are only specified when they are needed, when they implement an essential piece of behavior. Constraint-based partial specifications can be satisfied by a vast number of class structures which, when annotated with essential methods and automatically generated methods, denote a potentially infinite family of conventional object-oriented programs. This situation is illustrated in Fig. 1.

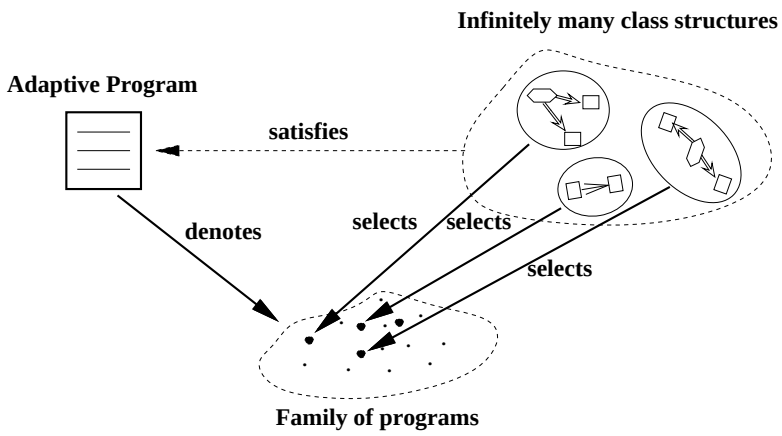


Figure 1: An infinite family of programs denoted by an adaptive program

Let us further illustrate the process of writing an adaptive program with an example. Let us assume that we are interested in computing the salaries of the officers in a conglomerate of companies. In fact, the process of writing an adaptive program can be seen as a process of making assumptions. These assumptions are expressed as constraints in the class structures which customize an adaptive program. Such constraints specify groups of collaborating classes in the customizing class structures.

What is important about the computeSalary problem? We assume there is a Conglomerate object, which contains somewhere inside of it a Salary object. These assumptions imply that for any class structure to successfully customize the computeSalary adaptive program, it must define a Company class which contains a nested Salary class. In addition, we require that the computeSalary program must not consider officers in subsidiary companies of the conglomerate. This turns into an assumption that the adaptive program must somehow bypass the

relationship subsidiaries of any company in the conglomerate. Thus, the structural section of an adaptive program should specify a number of constraints, expressed using class-valued and relation-valued variables. *Class-valued* variables itemize assumptions on the existence of classes in a customizing class structure. *Relation-valued* variables further restrict customizing class structures by excluding or forcibly including relationships among classes.

Behaviorally, the computeSalary program requires only one essential element, a method which accumulates the salary values. Nevertheless, every other method that constitutes a denoted object-oriented program should share a common signature. In particular, we would like an accumulator totalSalary to be handed to the specified method for update and to be accessible at the completion of the program. This can be done by using a modifiable argument, defined by each method in the program. Thus, the behavioral section of an adaptive program should define a common signature for its methods, and the code fragments that implement the required essential methods, attached to the appropriate class-valued variables.

The table in Fig. 2 describes informally the structure of the computeSalary adaptive program. Getting this adaptive program up and running involves the following steps: (1) Formally specify the program using a new notation which extends existing object-oriented languages; section 3 introduces specification of adaptive programs using the *propagation pattern* notation. (2) Customize the adaptive program with a particular class structure that satisfies the program's constraints; customization is discussed in section 4.

| *Structural Constraints Section* | | |
|---|---|---|
| *Variables* | | *Constraints* |
| *Type* | *Value* | Find all Salary-objects which are contained in Conglomerate objects but not reachable through the subsidiaries relation. |
| Class | Conglomerate | |
| | Salary | |
| Relation | subsidiaries | |
| *Behavioral Section* | | |
| Signature | void computeSalary( int& totalSalary ) | |
| Methods | *Attached to* | *Code fragment* |
| | Salary | totalSalary = totalSalary + *(this->get_value()); |

Figure 2: Informal description of computeSalary adaptive program

Adaptive programming introduces a new form of polymorphism, *adaptive polymorphism*. In [Ber93], Berard defines polymorphism as a measure of the degree of difference in how each item in a specified collection of items must be treated at a given level of abstraction, and provides the following examples of different types of polymorphism.

**General polymorphism** - where the same method is used regardless of the type (or form) of the operands.

**Ad hoc polymorphism** - where a different method is selected based on the type (or form) of the operands.

**Parameterized polymorphism** - where the user of the polymorphic abstraction must supply some information, in the form of parameters, to the abstraction.

In adaptive polymorphism an adaptive program may be used on different class structures. Thus, adaptive polymorphism is a new kind of parametric polymorphism. An adaptive program as a polymorphic entity specifies the information to be supplied by a user implicitly, as opposed to using explicit parameters, by means of constraints. Berard further points out that polymorphism is increased when any unnecessary differences, at any level of abstraction, within a collection of items are eliminated. Adaptive polymorphism increases the level of abstraction over existing types of parameterized polymorphism by allowing methods to be bound to classes at the time an adaptive program is customized, eliminating the need to attach methods to classes when the adaptive program is written.

Adaptive programming, as would be expected, is realized by delayed binding. We read in the Encyclopedia of Computer Science: "Broadly speaking, the history of software development is the history of ever-later binding time ..." Indeed, in the early days, machine language programmers used to bind variables to memory locations, while assembly language programmers left this task to the assembler. Later on, Pascal programmers bound function calls to code, while C++ programmers now have the choice to delay this decision until run-time. Adaptive programming introduces a subsequent degree of delayed binding. While conventional object-oriented programmers bind methods explicitly to classes, adaptive programmers delay binding of methods until a class structure customizer is provided.

## 3 Propagation Patterns

An adaptive program is specified using a collection of propagation patterns, each of which specifies a set of related constraints in the adaptive program. Adaptive programs, as we have pointed out, are customized by class structures. Although we cannot assume the composition of a specific customizing class structure[2], it seems reasonable to assume that it conforms to some given representation. Propagation patterns take advantage of this, assuming that customizing class structures are represented as graphs, specifically, as *class dictionary graphs*[3]. Assumptions, like there is a class Conglomerate which contains a nested Salary class, are represented in a propagation pattern as constraints of the form: the traversal from vertex Conglomerate to vertex Salary must be possible in any class dictionary graph that customizes this propagation pattern.

Given a customizing class dictionary graph, a propagation pattern produces an object-oriented program in the family denoted by the adaptive program it specifies. The object-oriented program is produced in two steps: (1) First we generate a subgraph of the class dictionary graph,

[2]That is, how many classes of what kind it has and with how many parts.

[3]Class dictionary graphs are introduced in section 4.

denoting the set of collaborating classes specified by the structural constraints in the adaptive program. (2) Then, a method is attached to each vertex in the generated subgraph, sharing the signature given by the adaptive program in its behavioral section. (3) Finally, each method specification in the behavioral section - class and code fragment - is used to either fill in or annotate some generated method.

Consider again the adaptive program for computing salaries of officers outlined in the previous section, and summarized in Fig. 2. The propagation pattern in Fig. 3 specifies this adaptive program, using the following elements.

```
operation void computeSalary( int& totalSalary )
  traverse
    from Conglomerate
      bypassing -> *, subsidiaries, *
    to Salary
  wrapper Salary
    prefix   begin totalSalary = totalSalary + *(this->get_value()); end
```

Figure 3: Propagation pattern for the computeSalary adaptive program

1. An operation clause. The signature void computeSalary( int& totalSalary ), shared by every method that implements the compute salary adaptive program, is specified with the keyword **operation**.

2. A traversal clause. The class-valued variables in the clauses **from** Conglomerate, **to** Salary specify vertices delimiting a traversal in a customizing class dictionary graph. The relation-valued variable, which in the clause **bypassing** -> *, subsidiaries, * represents an edge in a customizing class dictionary graph, further constrains the traversal to only those paths that do not include the edge. Given a customizing class dictionary graph, the traversal specified by this clause induces a set of vertices representing classes which includes: (1) the classes in the customizing class dictionary graph that match the class-valued variables in this traversal clause, and (2) any class contained in any path denoted by this traversal clause. Each class in such induced set of classes gets a method generated automatically, all of which define one object-oriented program in the family denoted by the adaptive program computeSalary.

3. A code fragment clause. The class-valued variable in **wrapper** Salary, indicates that the code **begin** totalSalary = totalSalary + *(this->get_value()); **end** fills in the body of the method generated automatically for class Salary.

In general, a propagation pattern consists of an operation clause, a traversal clause, and a set of code fragment clauses. A traversal clause is defined as a set of propagation directives, each of which is a 4-tuple composed of the following elements.

1. A non-empty set of source vertices from which a traversal starts, indicated by the keyword **from**.

2. A possibly empty set of target vertices where a traversal ends, indicated by the keyword **to**.

3. A possibly empty set of through edges, out of which each path denoted by a traversal is required to include at least one. Through edges are indicated by the keyword **through**.

4. A possibly empty set of bypassing edges, none of which may be included in any path denoted by a traversal. Bypassing edges are indicated by the keyword **bypassing**. Through and bypassing edges are specified with relation variables.

A **wrapper** code fragment is associated to a class-valued variable or to a relation-valued variable and can be either **prefix**, or **suffix** with **prefix** being the default. Wrapper code fragments are prepended or appended to the code that is generated automatically to properly implement traversals, depending on whether the code fragments are **prefix** or **suffix**, respectively. A more formal definition of propagation patterns and their semantics is given in [LXSL93, LX93a].

# 4 Customization

Adaptive programs, specified using the propagation pattern notation, exist at a higher level of abstraction than conventional object-oriented programs, much in the same way as parameterized classes exists at a different level than non-parameterized or, for that matter, instantiated classes. To select a particular object-oriented program for execution from the family denoted by an adaptive program, the adaptive program must be customized or instantiated, the same way a parameterized class is instantiated. As we have indicated, propagation patterns expect customizing class structures to be represented as class dictionary graphs.

Class dictionary graphs represent class structures at a programming language independent level using vertices to represent classes and edges to represent relationships between classes. An example of a class dictionary graph is illustrated in Fig. 4. There are three kinds of vertices in a class dictionary graph, construction, alternation, and repetition vertices. The vertex labeled Conglomerate is a construction vertex. A *construction* vertex, represented as a rectangle (□ ), is an abstraction of a class definition in a typical statically typed programming language (e.g., C++).

The vertex labeled Officer is an alternation vertex. *Alternation* vertices define union classes, and are represented as ⬭. When modeling an application domain it is natural to take the union of sets of objects defined by construction classes. Alternation vertices are implemented as abstract classes and their alternatives as subclasses through inheritance. In our example, the vertices labeled Ordinary and ShareHolding are the alternatives of Officer and define classes which inherit from class Officer. The alternative relationship is indicated using *alternation edges* ($\Longrightarrow$ ), outgoing from an alternation vertex into either a construction or another alternation vertex.

Construction and alternation vertices can have outgoing *construction edges* ($\longrightarrow$ ), which represent parts. Part is a high level concept which might be implemented as a method, not necessarily

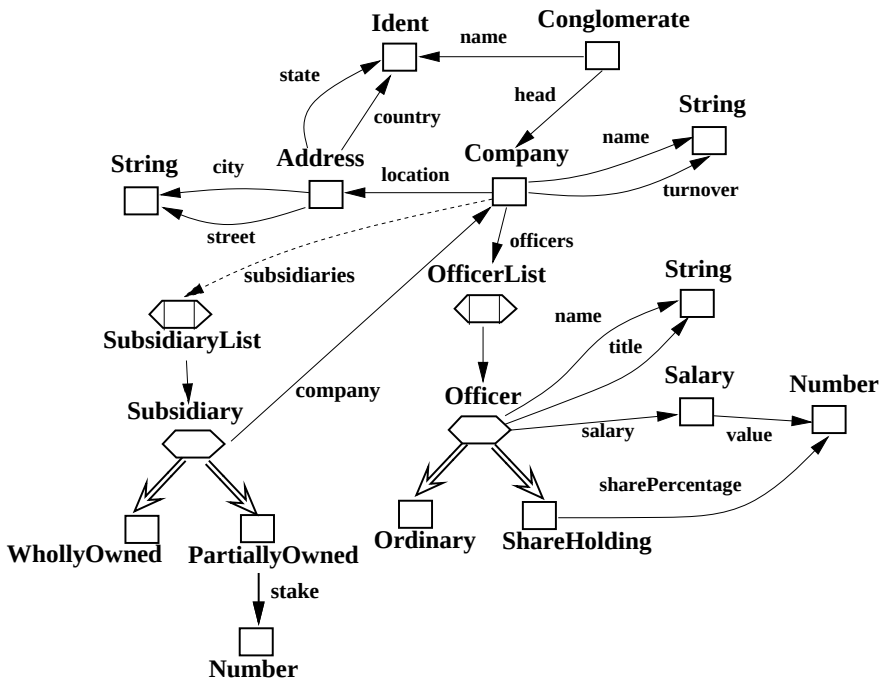


Figure 4: Class dictionary graph representing conglomerates of companies

as an instance variable. Construction edges outgoing from alternation vertices indicate common parts, inherited by each alternative of the alternation.

Finally, the vertex labeled SubsidiaryList is a repetition vertex. Repetition vertices represent container classes which have as their instances collections of objects from a given "repeated" class. Two important advantages of using repetition vertices are (1) the designer need not be concerned with a class belonging to a collection when designing such a class and (2) all the functionality common to container classes, such as iteration, appending, and element count can be abstracted into a single class. Class dictionary graphs are defined formally in [LBSL91, LX93b].

Let the class dictionary graph in Fig. 4 be a customizer for the propagation pattern in Fig. 3, which specifies the computeSalary adaptive program. First, we verify that the class dictionary graph satisfies the constraints in the adaptive program. The class dictionary graph does define classes Conglomerate and Salary in such a way that the traversal specified by the propagation pattern is possible.

When a propagation pattern is customized with a class dictionary graph, its traversal specifications induce a set of paths as follows. Every path from each **from** to each **to** vertex in the class dictionary graph is taken. In our example, some of those paths are:

1. $\mathsf{Conglomerate} \xrightarrow{head} \mathsf{Company} \xrightarrow{officers} \mathsf{OfficerList} \longrightarrow \mathsf{Officer} \xrightarrow{salary} \mathsf{Salary}$
2. $\mathsf{Conglomerate} \xrightarrow{head} \mathsf{Company} \xrightarrow{subsidiaries} \mathsf{SubsidiaryList} \longrightarrow \mathsf{Subsidiary} \xrightarrow{company} \mathsf{Company} \xrightarrow{officers} \mathsf{OfficerList} \longrightarrow \mathsf{Officer} \xrightarrow{salary} \mathsf{Salary}$

The set of paths is restricted to those paths that contain at least one **through** edge and that do not contain any **bypassing** edge. In our example, the path Conglomerate $\xrightarrow{head}$ Company $\xrightarrow{subsidiaries}$ SubsidiaryList $\longrightarrow$ Subsidiary $\xrightarrow{company}$ Company $\xrightarrow{officers}$ OfficerList $\longrightarrow$ Officer $\xrightarrow{salary}$ Salary would be eliminated, since it contains the edge $\xrightarrow{subsidiaries}$ , which must not be included. The resulting set of paths defines a subgraph of the customizing class dictionary graph referred to as the *propagation graph* of the customization. The propagation graph induced by the propagation pattern in our example is shown in Fig. 5.

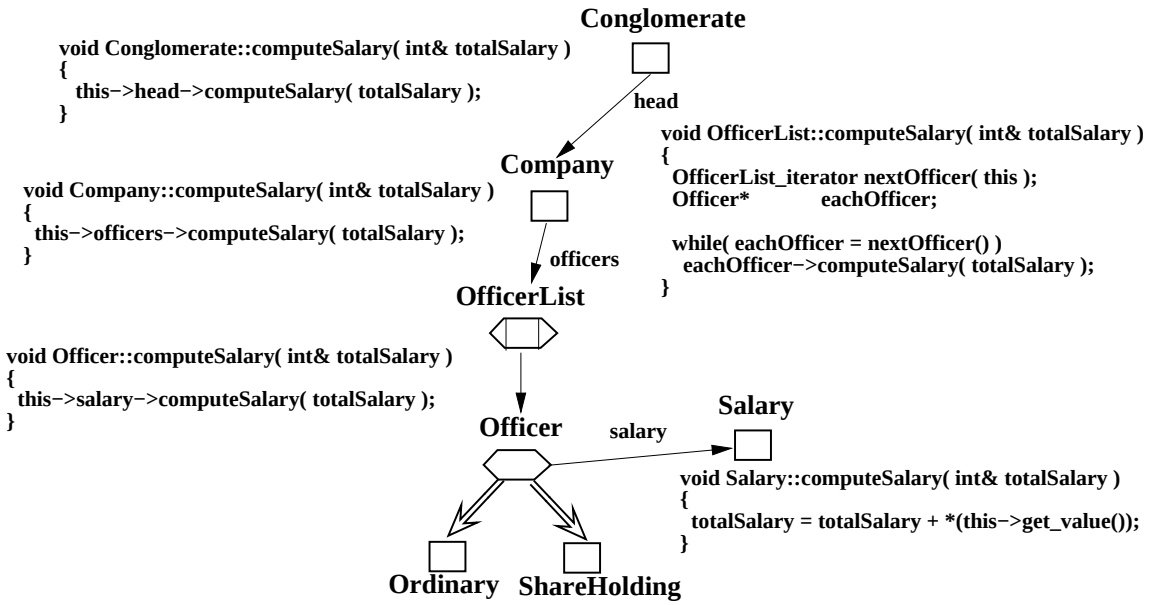


Figure 5: Propagation graph for a customization of the computeSalary adaptive program

This propagation graph also shows the code that defines the object-oriented program selected by the customizing class dictionary graph of Fig. 3. Once the propagation graph for a customization is computed, the code attached to it is generated as follows. For each vertex in the propagation graph a method is created with the signature given by the **operation** specification in the propagation pattern. The body for this method contains as many calls as the given vertex has outgoing construction edges in the propagation graph[4]. Each call is made to the method with the same signature attached to the vertex target of the corresponding construction edge. If a vertex has an incoming alternation edge as well as at least one outgoing construction edge, the body of the method for that vertex also contains a call to the method attached to the alternation vertex source of the alternation edge. In other words, if a construction vertex must redefine an inherited generated method, the redefining method makes sure that the redefined method gets executed as well. Finally, each **wrapper** code fragment in the propagation pattern is prepended or appended to the generated code for the vertex or edge it specifies. When a wrapper is associated to a class-valued variable, the code fragment is prepended or appended to the entire method generated for the class the variable stands for. Relation-valued variables, implemented as edges, get code generated of the form of a message send to the target of the

[4]Notice, in the propagation graph, as opposed to the class dictionary graph.

edge. Wrappers associated to relation-valued variables, prepend or append their code to this message send code. [LXSL93] formally defines the rules for computing a propagation graph and for generating the code attached to a propagation graph.

To further illustrate the adaptiveness of the propagation pattern in Fig. 3, consider the class dictionary graph in Fig. 6, a second customizer for the propagation pattern. In this customizer, conglomerates have lists of companies with simpler subsidiaries and officers. Again, we verify that this customizer satisfies the constraints posed by the parameters for the adaptive program specified by the propagation pattern. There is a vertex Conglomerate from which a traversal is possible to a vertex Salary. Hence, this second customizer induces the propagation graph of Fig. 7, which is also annotated by the code generated for each vertex.

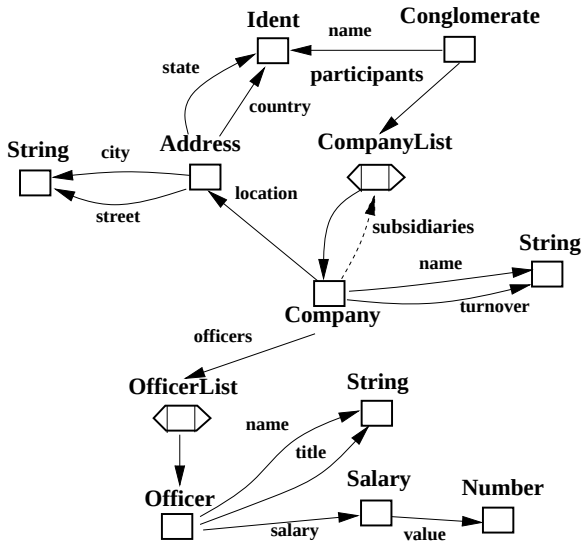


Figure 6: Another representation for conglomerates of companies

So far we have discussed customization of adaptive programs using class dictionary graphs. Another possibility is to use sample objects to automatically generate a customizing class dictionary graph. In [LBSL91], a method is introduced to generate class dictionary graphs automatically from object samples. This method first generates some class dictionary graph that describes at least those objects given as samples. In a second stage, the method optimizes the generated class dictionary graph by eliminating redundant parts and reducing the amount of multiple inheritance, while at the same time preserving the set of objects described by the class dictionary graph.

## 5 Related Work

To show the relationship to previous work, we sketch the history of software development from procedural to object-oriented to adaptive. Adaptive programming is a similar improvement over object-oriented programming as object-oriented programming is over procedural programming (see Fig. 8). Object-oriented programming introduces the binding of functions to data structures while adaptive object-oriented programming introduces the binding of functions to

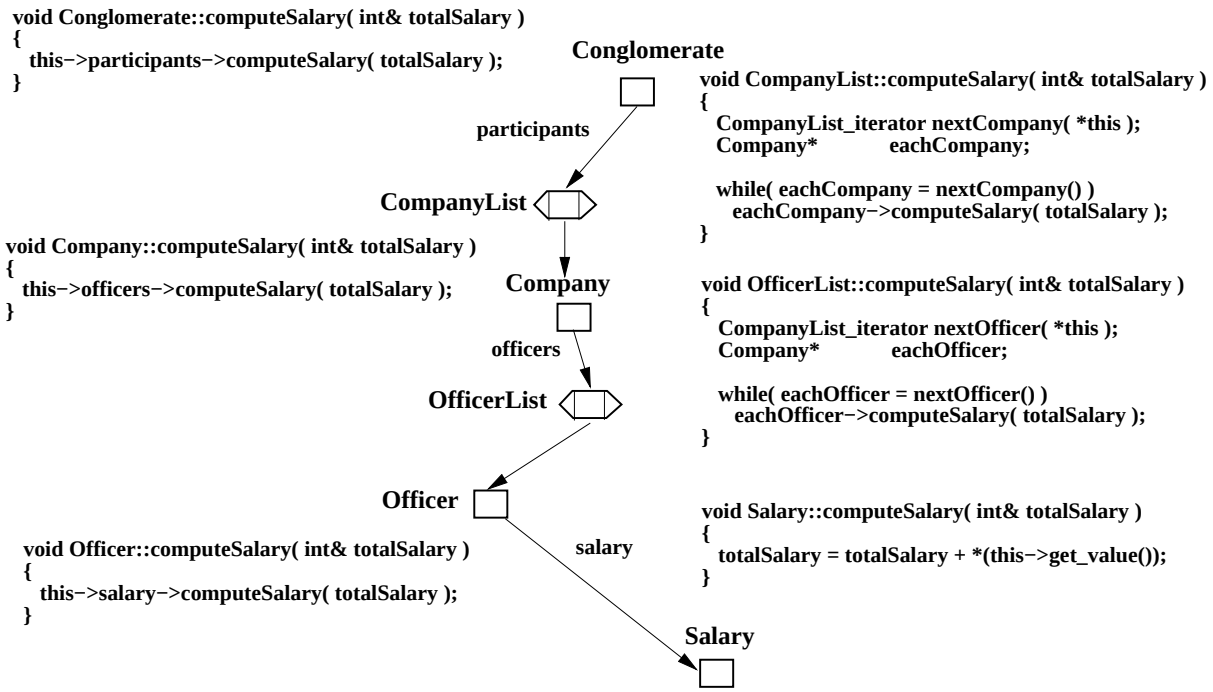


Figure 7: Propagation graph with code for second customization

constraint-induced partial data structures. Both bindings are done at write-time. Object-oriented programming introduces inheritance to express programs at a higher level of abstraction while adaptive programming uses partial data structures to lift the level of abstraction. Object-oriented programming employs inheritance to delay the binding of calls to code from compile-time to run-time while adaptive programming uses partial data structures to delay the binding of methods to classes from write-time to compile-time. Adaptive programming is therefore a natural step in the evolution of software development methods.

| *Paradigm* | *Write-time association* | *Resulting delayed binding* | *Due to* |
|---|---|---|---|
| Procedural | *Function calls* $\rightarrow$ *code* | | |
| Object-oriented | *Functions* $\rightarrow$ *Data structures* | *Function calls* $\xrightarrow{run-time}$ *code* | Inheritance |
| Adaptive | Functions $\rightarrow$ Partial data structures | *Functions* $\xrightarrow{compile-time}$ *data structures* | Partial data structures |

Figure 8: Comparison of programming paradigms

Rumbaugh [Rum88] has proposed an operation propagation mechanism which he found of practical value. The most significant difference between his and our approach is that his mechanism uses an explicit specification of the propagation scope, while we use a succinct notation to describe the scope. In addition, Rumbaugh's mechanism is run-time based while

our mechanism is compile-time based and geared towards a new method for developing object-oriented software. In Rumbaugh's method, for each relation between an object and one of its parts, a propagation attribute must be supplied (unless the default of "None" is desired). If an operation is to propagate along the links of a deeply-nested part-of relation, then the "Propagate" attribute has to be provided at each link. In our system, we need only indicate the target class, and propagation automatically follows all links. This has the further consequence that if class relations are modified in our system, the propagation pattern is usually untouched. In Rumbaugh's scheme, the introduction of an intermediate class along a part-of chain would require a new propagation attribute to maintain the propagation.

In [GTC+90] we read: "As a result, the class hierarchy may become a rigid constraining structure that hampers innovation and evolution". This observation is the driving force behind our work on adaptive programming. What we actually do is to parameterize our propagation pattern programs with a class hierarchy (part-of and inheritance hierarchy) which makes the programs much less dependent on the specific details of the hierarchy. Hence, the class structure is a less constraining structure for the evolution of the programs.

Propagation patterns are motivated by the key idea behind the Law of Demeter [LH89]. The Law of Demeter essentially says that when writing a method, one should not hardwire the details of the class structure into that method. Propagation patterns take this idea one step further by keeping class structure details out of entire programs as much as possible.

In the database field, our work builds on techniques for data navigation in the hierarchical database model. Tsichritzis and Lochovsky write in [TL82]: "The restrictions on the connections between records in a hierarchical data model permits the selection of a set of records according to hierarchical paths."

Propagation patterns promote the description of behavior at a high-level of abstraction. There are constructs related to propagation patterns which specify behavior at a larger grain size than the class level. These include object groups [Yon90] and contracts [HHG90]. Contracts offer a different way of structuring object-oriented programs: Instead of defining the complete interfaces for the classes, the interface information is described in contracts and then used in the classes. An object group is a collection of objects which forms a natural unit for performing collective tasks. In ABCL they were introduced for debugging.

Foote and Johnson [JF88] introduce a recursion introduction rule as a means to define standard protocols to design reusable classes. We believe this rule to be an important factor in achieving greater adaptiveness. Propagation patterns, by specifying the signature of an operation singly for a collection of collaborating classes, directly implement standard protocols by enforcing recursion introduction.

An experience report on the use of propagation patterns appears in [LHSLX92]. Further work on propagation patterns is discussed in [Lie92].

## 6 Conclusion

This paper introduces adaptive programming as a new modeling technique for describing complex systems. According to the Encyclopedia of Computer Science and Engineering [Ral83], a complete model frequently includes both a structural model and a process model. The structural model describes the organization of the system and the process model describes the operation or behavior of the system. In this context, an adaptive program is analogous to a process model parameterized by graph constraints which define compatible customizers of the behavior. Adaptive program customizers define structural models inducing a traditional model as the result of being applied to an adaptive program. The innovative feature of adaptive programs is that they use graph constraints to specify possible customizers.

Adaptive programming, realized by the use of propagation patterns, extends the object-oriented paradigm by lifting programming to a higher level of abstraction. In their simplest form, which also turns out to be the worst in terms of adaptiveness, adaptive programs are nothing more than conventional object-oriented programs, where no traversal is used and where every class gets a method explicitly. But, for a large number of applications, represented by related customizers, nothing has to be done to an adaptive program to select the conventional object-oriented program corresponding to any of the customizers. Moreover, when changes to an adaptive program are indeed necessary, they are considerably easier to incorporate given the ability that adaptive programs offer to specify only those elements that are essential and to specify them in a way that allows them to adapt to new environments. This means that the flexibility of object-oriented programs can be significantly improved by expressing them as adaptive programs, which specify them by minimizing their dependency on their class structures.

The following advantages stem from the use of adaptive programs.

- Adaptive programs focus on the essence of a problem to be solved and are therefore simpler and shorter than conventional object-oriented programs.
- Adaptive programs promote re-use. Many behaviors require the same customization and thus customizers are effectively re-used. More importantly, every time an adaptive program is customized re-use is taking place.
- There is no run-time performance penalty over object-oriented programs. By using appropriate inlining techniques, traversal methods can be optimized, eliminating apparent performance penalties.

**Acknowledgements.** This work is supported in part by IBM Yorktown and IBM Mid-Hudson Valley, Mettler-Toledo and the National Science Foundation under grants CCR-9102578 (Software Engineering) and CDA-9015692 (Research Instrumentation). We would like to thank Mitchell Wand, Paul Bergstein, Walter Hürsch, Ian Holland, Linda Keszenheimer and Greg Sullivan for their feedback and support.

## References


[Ber93] Edward V. Berard. *Essays in Object-Oriented Software Engineering*, volume I. Prentice-Hall, 1993.

[GTC+90] Simon Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz, and Xavier Pintado. Class management for software communities. *Communications of the ACM*, 33(9):90–103, September 1990.

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

[JF88] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

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

[LH89] Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.

[LHSLX92] Karl J. Lieberherr, Walter Hürsch, Ignacio Silva-Lepe, and Cun Xiao. Experience with a graph-based propagation pattern programming tool. In *International Workshop on CASE*, pages 114–119, Montréal, Canada, 1992. IEEE Computer Society.

[Lie92] Karl J. Lieberherr. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In J. van Leeuwen, editor, *Information Processing '92, 12th World Computer Congress*, pages 179–185, Madrid, Spain, 1992. Elsevier.

[LX93a] Karl Lieberherr and Cun Xiao. Object-Oriented Software Evolution. *IEEE Transactions on Software Engineering*, 19(4):313–343, April 1993.

[LX93b] Karl J. Lieberherr and Cun Xiao. Formal Foundations for Object-Oriented Data Modeling. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):462–478, June 1993.

[LXSL93] Karl J. Lieberherr, Cun Xiao, and Ignacio Silva-Lepe. An object-oriented implementation of adaptive software. *IEEE Transactions on Software Engineering*, 1993. Accepted for publication.

[Pol49] George Polya. *How to solve it.* Princeton University Press, 1949.

[Ral83] Anthony Ralston. *Encyclopedia of Computer Science and Engineering.* Van Nostrand Reinhold Company, Inc., 1983. Second edition.

[Rum88] James Rumbaugh. Controlling propagation of operations using attributes on relations. In *OOPSLA Conference, Special Issue of SIGPLAN Notices*, pages 285–297, San Diego, 1988. ACM.

[TL82] Dennis Tsichritzis and Frederick Lochovsky. *Data Models.* Software Series. Prentice-Hall, 1982.

[WH91] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. In *Conference on Software Maintenance*, pages 162–170, Sorrento, Italy, 1991. IEEE Press.

[Yon90] Akinori Yonezawa. *ABCL: An Object-Oriented Concurrent System.* Computer Systems Series. The MIT Press, 1990.