Specifying and Adapting Object Behavior during System Evolution

Abstract

Object-Oriented software engineering must address new issues during system evolution, namely the effects of class evolution on existing class methods. A Class Dictionary Graph describes the class structure and relationships of a given domain. A Propagation Pattern defines object behavior by describing responsibilities among a group of collaborating classes. The behavior described in a Propagation Pattern is mapped onto the class structure defined in a Class Dictionary Graph, and the appropriate C++ code is generated. Class structures evolve in many ways. This paper details the effects of class evolution on the object behavior defined in existing Propagation Patterns, and describes the requirements for adapting Propagation Patterns during class evolution, as compared to the efforts of maintaining C++ code.

1 Introduction

Evolution of business processes and organization is a major force to contend with during the software life-cycle. Object-oriented software development addresses the demand for continual change. By modeling the structure and behavior of objects found in the enterprise, and incorporating the object model into class structures, systems developed using object-oriented methods are more flexibly implemented and thus better support change. While the use of object-oriented programming languages may help build better structured systems, evolving these systems continues to be a challenge. The effects of schema evolution on existing classes and objects have been well studied and described, ([1],[13], [14], [9]) while the issues relevant to maintaining object behavior have only begun to be addressed. ([5], [2], [6]).
Utilizing a high-level descriptive language to define

class structure and behavior enables development of systems which can support evolution, as well as maintain consistency between documentation and code. Use of a CASE tool to define the system and generate the corresponding code should minimize the amount of effort required to support change. This paper describes some of the approaches utilized in the Demeter Method, [7] and demonstrates its ability to support evolution during the software life-cycle. Demeter is an adaptive software development method which encompasses both an object-oriented software engineering model as well as a CASE toolset to support the model. A Class Dictionary Graph is created to describe class structure. A Propagation Pattern [10] is developed to describe the collaboration within and between classes to accomplish a task. System evolution is supported through the use of class dictionary graphs and propagation patterns, which are high-level specifications of classes and behavior, from which code can be generated.

The ideas presented in this paper on adapting propagation patterns during class evolution are based upon experiences gained while working

for CitiCorp with the Global Finance Application Architecture Team (GFAAT), within the Technology Planning Division. The goal of the team is to provide an architecture for future system development, taking advantage of object-oriented technology in redefining their corporate business model. This paper discusses the evolution experiences obtained while building a tool to simulate the GFAAT business model. Class dictionary graphs and propagation patterns were developed and continuously evolved during the production of the simulation tool. The requirements for maintaining propagation patterns during system evolution are detailed.

1.1 Class notation and class dictionary graphs

There exist many object-oriented data modeling notations. ([3], [4], [8]). This paper uses the Deme-

ter class model and notation [10] which defines several kinds of classes:

Construction class
A construction class denotes a concrete definition of some entity, such as a car, person or chair. A construction class consists of zero or more component classes, or parts. Construction classes can be instantiated, but can not be used for inheritance.

Alternation class
An alternation class is a superclass of other classes. It is an abstraction of the common attributes and behavior found among a group of classes. An example would be an automobile, which could either be a car or a truck. Both car and truck inherit attributes and behaviors from automobile, while adding or refining some within their own classes.

Repetition class
A repetition class is a container used to aggregate multiple instances of another class. A repetition class is analogous to a List structure.

Terminal class
A terminal class is a building block from which other classes are constructed. Terminal classes encapsulate the basic data types. Standard terminal classes are: Ident , String , Number , Real , Text .

The relationships among a group of classes can be visually described in a Class Dictionary Graph. A class dictionary graph is a directed graph whose vertices represent the classes of the domain being modeled, and whose edges represent the isa, part-of and uses relationships among the classes. The construction and terminal classes are represented as rectangles, the alternation classes are hexagons, and the repetition classes are hexagons containing rectangles. A single-line edge between two vertices represents a partof or uses relation, while a double-line edge represents an isa or inheritance relation.

Figure 1 contains a class dictionary graph which details the

organization of several classes. It describes a Company class, which has a name and departments. Each Department class has a name and employees. The Employee class is an abstract class which is either FullTime or PartTime. The Employee class has a name and salary . These attributes are inherited by the FullTime and PartTime classes.

The formal definition of a class dictionary graph is described in [12]. A class dictionary graph ? is a

NumberDepartmentemployeesIdentnameIdentEmployeeListIdentnameCompanyDepartmentListdepartmentsEmployeeFullTimePartTimesalaryname

Figure 1: A Class dictionary graph defining a class organization for companies.

directed graph defined as:
? = (V C; V A; V R; V T; ?; EC;EA;ER) where

VC is a finite set of construction vertices VA is a finite set of alternation vertices
VR is a finite set of repetition vertices
VT is a finite set of terminal vertices

? is a finite set of edge labels.

EC is a relation on (V C[V A)?V C?? describing the labeled construction edges in the graph. For each (v; w; l) 2 EC; there exists a construction edge from v to w labeled l . Class v then has a part, or relationship, named l of class w .

EA is a relation on V A?(V C[VA) describing the alternation edges. For each (v; w) 2 EA; there exists an alternation edge from v to w . Class v then has an alternative w , and class w inherits attributes and behavior from class v . Only alternation vertices can have outgoing alternation edges.

ER is a relation on V R? (V C [V A[VR[V T) describing repetition edges. For each (v; w) 2 ER there exists a repetition edge from v to w . Class v is a container class of instances of class w .

Figure 2 shows the components of the class dictionary graph for the Company class structure. Given a class dictionary graph, Demeter generates C++ code to represent the organization of classes. It is easy to evolve the class structure by simply reorganizing the class dictionary graph. The corresponding C++ code will then be regenerated to adapt to the new class structure.

2 Describing object behavior

An object such as an employee has certain behavior and responsibilities, and can make requests of other objects. One task that could be requested of a company object is to sum the salaries of

its employees. To accomplish this, the Company class would have a method named sumSal to implement the behavior of traversing through its departments to determine its payroll obligations. The Department class would be responsible for determining the salary total of its employees. The Employee class would be responsible for adding its salary value to the total.

When developing code to implement a task, it is important to follow the guidelines of information hiding and delegation so existing code will be easy to evolve when class structures change. Well written programs which follow the Law of Demeter [11] will contain many small methods each having the purpose of propagating a message request from one class to another using existing class relationships. A request is passed along a path of relationships until the class possessing the responsibility and attributes necessary to accomplish the task receives the message.

To sum the salaries of employees in a company, a sumSal request is propagated along a path or subgraph of the class dictionary graph contained between the Company and Employee class. This path is shown in Figure 3. Edges in the class dictionary graph that do not lead to the Employee class are not shown in this Propagation Graph since paths along those edges will not contribute to the desired behavior. The responsibility each class has along the propagation graph involves passing the request from one class to another, with only the Employee class possessing the responsibility of incrementing the salary total variable.

When writing code, the programmer should concentrate only on the important classes involved in accomplishing a task. One wants to concentrate on the classes which have the major responsibilities involved in the task, namely the classes which possess the data relevant to the task, while avoiding writing the trivial message passing code which propagates a request from one class to another along the class hierarchy.[15]

The implementation details of writing methods for each class responsible for propagating a message along a path can be time consuming to write and difficult to maintain. Given several classes, it is desirable to abstract out of an existing class dictionary the communication path between these classes. This is done by utilizing the parts and relationships among the classes. If there exist several relationships or paths through which the initial sender of a message can access the final receiver of the message, the path can be restricted by specifying which parts and relationships to utilize

or avoid to accomplish the task.

2.1 Propagation patterns

A Propagation Pattern [10] is used to alleviate the need to write trivial message passing code, allowing one to concentrate on the important aspects of the task. Propagation patterns support a style of

coding which focuses on the important classes involved in accomplishing a task, with emphasis on minimizing reprogramming effort when the class structure changes. They provide a necessary level of abstraction when implementing object behavior which can accommodate class evolution. The propagation pattern to accomplish the salary summation task is shown in Figure 4.

The structure of this propagation pattern contains an interface statement with the method name sumSal , its return type void , and the parameters to the method (Number* totalSal).

An example of some the C++ code that is generated by the propagation pattern to accomplish this task is shown in Figure 5. The sumSal request is sent from the Company class to the Employee class. Each class on the path between Company and Employee will have the responsibility of passing the request to any parts which can reach the Employee class. The Company class has two parts, name and departments. The sumSal method for Company should send a sumSal message to its departments part but not to the name part. This is because there is a path following the departments edge in the class dictionary graph to the Employee class, but there is not one following the name edge. Construction, alternation, inheritance and repetition edges are utilized in determining if one class is reachable from another.

The Employee class will have special behavior for its sumSal method. This behavior is found in the *primary* Employee clause, which states that it should add its salary value to the variable totalSal . At runtime, Employee objects will be instances of either the FullTime or PartTime class, both of which inherit the

Employee class code of adding the salary to the total salary variable.

The definition of a Propagation Pattern for a given class dictionary graph ? is a tuple of the form (M; PD;CF).

M is a method interface of the form (r; n; s) where

{ r is the return type of the method,

{ n is the name of the method,

{ s is the signature of the method of the form (t1v1; : : : ; tnvn) where

? t is the type of the variable argument

? v is the name of the variable argument

PD is a propagation directive of the form (F; I; X; T) where

{ F is a non-empty set of vertices in the class dictionary graph

specifying the starting vertices in the propagation graph, or *from* classes.

{ T is a set of vertices in the class dictionary graph specifying the ending vertices in the propagation graph, or *to* classes.

{ I is a set of edges in the class dictionary graph which the propagation graph must include. These are the *through* edges.

{ X is a set of edges in the class dictionary graph which the propagation graph must not include. These are the *bypassing* edges.

CF is a Code Fragment, which is of the form (v; cf), where v is a vertex in the propagation graph and cf is a string describing the behavior for vertex v .

Propagation Patterns have a textual form, like the example in Figure 4.

The method interface M can be described as: *interface* M
The code fragments (v; cf) are described as: *primary* v (@ cf @)
A propagation directive can be described using the following textual form:

Each fi is a vertex in F , the from vertices; ti is a vertex in T , the to vertices; ii is an edge in I which the message must pass through and xi is an edge in X which must be bypassed.

Given a class dictionary graph ? and propagation directive PD , the corresponding Propagation Graph PG is abstracted. A vertex v or edge e is included in the propagation graph if it is located along the path defined between the *from* vertices to the *to* vertices, avoiding edges given in the *bypassing* clause and following edges in the *through* clause.

Demeter implements the propagation pattern functionality by generating the appropriate code to perform the behavior defined for the classes involved in the propagation. For each vertex in the propagation graph, a C++ member function is created which has the method interface defined in the propagation pattern, and which contains a message request for each edge of the vertex contained in the propagation graph. In the example, Department is contained within the propagation graph defined by the propagation directive, since it is located along the path from Company to Employee. It will have a sumSal method generated for it, which contains code to send a sumSal message to each of its parts which can reach Employee. There will be a message sent to its employees part with the

appropriate arguments passed, since this part corresponds to an edge that reaches Employee. The method for Department will not send a message to its name part, since there is no path from that part that could reach Employee.

If a vertex v is contained in a code fragment pair (v; cf), then the code generated for class v is not the standard message propagation code, but rather is the code described in the code fragment cf . In the example, the sumSal method for class Employee will not be the generated message passing code, it will be the code described in the *primary* clause for Employee, which increments the salary total variable.

2.2 Propagation patterns supporting system evolution

The true strength of propagation patterns becomes evident when the class structure evolves during the software life cycle. Because propagation patterns allow the developer to describe high-level collaborations among classes to accomplish behavior, when the class structure defined by the class dictionary graph evolves, the amount of work needed to restructure a propagation pattern is often minimal as compared to maintaining actual C++ code.

As an example, assume the class dictionary graph is modified to define companies that contain divisions, which are made up of departments, as described in Figure 6. The propagation pattern which sums the salaries of employees in a company does not need to be modified to support this new class structure. A new propagation graph is abstracted from the propagation directive, and the corresponding C++ code is generated to support this new class structure.

There are several situations that occur when evolving a class dictionary graph which may require changes to an existing propagation pattern. When the class structure changes, the propagation graph may change and it may be necessary to evolve the propagation pattern to support the new class structure. Observing the changes in the class dictionary graph, and the affect on the corresponding propagation graph, helps identify the potential ways to restructure the propagation pattern to ensure that the behavior remains consistent. Evolution of the GFAAT Simulation tool identified the following types of transformations as commonly occurring:

In the example of partitioning companies into divisions, the class dictionary graph is evolved to elongate the path between two vertices, implying a single edge between two vertices is replaced

by a sequence of edges. This transformation occurs either when it is desirable to further partition the objects in the domain or when a direct relationship between objects is transformed into a sequence of relationships. In either case the source vertex must go through a longer sequence of edges, or relationships, to reach the target vertex. This type of change in a class dictionary graph causes the depth of the corresponding propagation graph to increase. This is the case in the example that was just presented by adding the Division and DivisionList classes into the class dictionary graph. This increase in depth of the propagation graph is exactly what is desired and the propagation pattern does not need to be modified to accommodate the new objects, but is simply regenerated to produce the correct C++ code.

Sometimes it is decided that the class structure has been overdefined, or there exist performance requirements which require a collapse of object structure to a simpler structure with less detail. A decrease in depth of the propagation graph is due to a collapse of a group of edges into a single edge. This occurs either if there existed too much partitioning and the class structure is being simplified, or when it is desirable for a vertex to have a more direct communication path with another vertex, rather than going through a sequence of edges. As long as none of the intermediate edges being replaced are referenced in a *through* or *bypassing* clause, there is no additional work to be done.

It is decided that a relationship between two classes should not exist, and this may cause a deletion of an edge from the class dictionary graph that is used in a *bypassing* or *through* clause. This can be easily detected, and the corresponding clause removed so that the propagation graph does not become disconnected, implying other relationships are to be used if they are available. However, the semantics of the propagation pattern may no longer be correct. The propagation directive should be looked at to determine if the behavior is properly implemented using alternative relationships.

A particular relationship between two classes no longer exists, requiring a change in the class dictionary graph which removes an edge between vertices. The propagation graph may then become disconnected. As in the example of removing an edge that is found in a *bypassing* or

through clause, the propagation pattern may need to be rebuilt to try to use other relationships or class structures to accomplish the task, or the behavior it was supporting may simply no longer be valid in which case it should be discarded.

DepartmentemployeesIdentnameEmployeeListIdentnameCompanyDepartmentList departmentsIdentnamedivisionsDivisionListDivisionNumberIdentEmployeeFullTimePartTimesalaryname

Figure 6: Class Dictionary Graph for Companies consisting of Divisions
.

An additional relationship is established between two classes found in the propagation graph, meaning a construction edge was added between the vertices in the class dictionary graph. For example, the Company class is modified to have an additional part named president which causes a direct link from the Company class to the Employee class, in addition to the previously existing link through the employees part. The change in the class dictionary graph causes the width of the propagation graph to increase, where the source vertex, Company , has an additional branch that

it previously did not have which can reach the target vertex, Employee. The propagation graph now has an additional path. This may or may not require a change in the propagation pattern, depending on

the semantics of the task being implemented. If it is acceptable to allow multiple access paths between objects, then no modification to the propagation pattern is required. If, however, there should be only a single path, then it is simple enough to add a *bypassing* clause for the new edge or a *through* clause for the old edge and the propagation graph would remain as it previously was defined.

If the company class dictionary graph were modified to describe conglomerates, which contain companies, then there exists a path from the conglomerate class to the company class. The existing propagation directive defines a propagation graph *from* Company *to* Employee. The propagation directive potentially should be modified to indicate that propagation should start at the conglomerate class rather than the company class by changing the propagation directive to specify *from* Conglomerate *to* Employee.

The change in the class dictionary graph occurs in a portion of the graph outside of that specified by the propagation directive and thus does not appear to directly affect the propagation graph. However, the propagation pattern may need to be modified to expand the behavior to include more objects. If an edge, or sequence of edges, is added to the class dictionary graph which allows some vertex v to reach a vertex w contained in the original propagation graph, then the propagation pattern might need to evolve to include behavioral definitions for instances of v . The *from* clause should possibly be changed to begin propagation at v .

The other situation which may occur is that edges are added which allow some vertex w in the propagation graph to reach a vertex v outside of the propagation graph. It may be necessary to evolve the propagation pattern to utilize v in the implementation of the behavior, by modifying the *to* clause to reach v .

A class renaming of vertices or edge labels in the class dictionary graph occurs. This is easily automated to rename occurrences of the vertices and edges in the propagation patterns. Notice in the simple situation of class renaming, if the class

were used in the method interface as a parameter, only the interface statement would need to be updated. All methods defined by the propagation pattern would be regenerated with the new class name in the interface. This is certainly less effort than manually having to change a class name in many existing method interfaces if they are written by hand.

An additional subclass is added to an superclass, thus adding an additional alternation edge to the graph. If the superclass were contained in the original propagation graph, then the new subclass that has been added is contained. This is very handy since it is easy to add subclasses and automatically have behavior propagated to them without reprogramming effort. If, however, the new subclass should not have behavior propagated to it, then a *bypassing* clause is added to prevent the new subclass from inheriting the behavior.

As many of these cases indicate, there may be work to be done to existing propagation patterns when the class dictionary graph evolves, but the work required to evolve a propagation pattern is often minimal and easier to identify than implementing straight C++ code. Because class dictionary graphs and propagation patterns work as high level descriptions to implement the system intent, it is easier to work with them to evolve a system. The effects of class evolution on existing behavior are more easily identified by viewing the effect of evolution on existing propagation graphs.

3 Conclusion

Propagation Patterns have been used in industry in situations where the class structure was under continuous change. The effort required to maintain existing propagation patterns was minimal as compared to maintenance of C++ code. In many cases, the propagation directive was consistent with the new class structure and no change was required, the code was simply regenerated to fit the new structure. The situations which required change to the propagation directive involved the need to restrict a new class from the propagation graph, and this is easily accomplished by adding a *bypassing* clause.

Adaptive system development using object-oriented techniques provides a foundation for building flexible systems. Attention is paid to the entities of the domain being modeled. Their similarities and differences are scrutinized in order to capitalize upon reuse poten-

tial. Utilizing high-level abstractions like class dictionary graphs and propagation patterns further expands the benefits of object-oriented technology by minimizing the amount of effort required when systems need to change. The ability to support and encourage change is a necessary part of any software development model.

Acknowledgements

I would like to thank Ted Seiter, Karl Lieberherr and Karl Frank for providing valuable input during the production of this paper.

References

[1] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented data bases, San Francisco, California. In Proceedings of ACM/SIGMOD Annual Conference on Management of Data, May 1987.

[2] Gilles Barbedette. Schema modifications in the lispo2 persistent object-oriented language. In European Conference on Object-Oriented Programming ECOOP'91. Springer-Verlag, 1991.

[3] Grady Booch. Object-Oriented Design With Applications. Benjamin/Cummings Publishing Company, Inc., 1991.

[4] Peter Coad and Edward Yourdon. ObjectOriented Analysis. Yourdon Press, 1990. second edition.

[5] Christine Delcourt and Roberto Zicari. The design of an integrity consistency checker (ICC) for an object-oriented database system. In European Conference on Object-Oriented Programming, pages 377{396, Geneva, Switzerland, 1991. Springer Verlag.

[6] Mohammed Erradi, Gregor Bochmann, and Rachida Dssouli. A framework for dynamic evolution of object-oriented specifications. In Proceedings of the Conference on Software Maintenance. IEEE Computer Society, 1992.

[7] Walter L. H?ursch, Linda M. Seiter, and Cun Xiao. In any CASE: Demeter. The American Programmer, 4(10):46{56, October 1991.

[8] Ivar Jacobsen, Mangus Christerson, Patrik Jonsson, and Gunnar Overgaard. Object-Oriented

Software Engineering - A Use Case Driven Approach. ACM Press, Addison-Wesley, 1992.

[9] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices, 25(10):67{76, October 1990.

[10] Karl J. Lieberherr. The Art of Growing Adaptive Object-Oriented Software. PWS Publishing Company, a Division of Wadsworth, Inc., 1994.

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

[12] 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.

[13] D. Jason Penney and Jacob Stein. Class modification in the GemStone object-oriented DBMS. In Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices, pages 111{125, Orlando, Florida, 1987. ACM Press.

[14] Markus Tresch. A framework for schema evolution by meta object manipulation. In Proceedings of the 3rd International Workshop on Foundations of Models and Languages for Data and Objects, Aigen, Austria, September 1991.

[15] Norman Wilde and Ross Huitt. Maintenance support for object-oriented programs. IEEE Transactions on Software Engineering, 18(12), 1992.