Utilizing Behavioral Abstractions to Facilitate Maintenance During Class Evolution

Linda M. Keszenheimer

Northeastern University, College of Computer Science
Cullinane Hall, 360 Huntington Avenue, Boston MA, USA 02115
seiter@ccs.neu.edu

Abstract. Software maintenance can be a difficult and time consuming process. To facilitate this process, application development must produce software that is designed to continually evolve. While object oriented methodologies address some of the maintenance issues that have troubled traditional functional techniques, object oriented development must overcome the problems involved in maintaining existing object behavior when the underlying class structure evolves. Utilizing adaptable abstraction models for defining class structure and behavior can facilitate software maintenance during class evolution. This paper describes adaptive software development using the Demeter Model for defining object structure and behavior. The maintenance impact of class evolution on existing behavioral implementations is detailed.

1 Introduction

Evolution can be a difficult force to manage during the software development process. The natural tendency of application domains to change and expand can quickly turn software development into a primarily maintenance-oriented process. Object oriented programming facilitates many aspects of software development and maintenance through the features of class reuse, information hiding and delegation. However, an object oriented program is typically implemented based on a particular class structure, and it can be difficult to maintain existing behavioral implementations when the class structure evolves. Class evolution occurs often throughout the software life cycle, due to both a continual improvement in the understanding of the application domain as software development proceeds, as well as the tendency for application domains to evolve to support changing business needs. While class transformations have been studied previously, [1, 18, 19, 7, 4, 6, 14], the research has been primarily based on structural transformations and the consequences of maintaining existing object structure, not addressing the maintenance of existing object behavior.

This paper describes an adaptive software development technique for implementing applications. Since class structures tend to evolve frequently, it is desirable to implement behavior in a flexible manner that can adapt to a changing class organization. Object behavior is described using an abstraction technique

called Propagation Patterns [17, 15, 10], which are specifications of class collaborations from which C++ code is generated. The ways that class structures evolve are detailed, and strategies for maintaining propagation patterns based on those transformations are given. While there exist other models for specifying inter-object behavior and patterns [3, 11, 5, 8], Propagation Patterns provide formal language support for the specification and implementation of behavior, with emphasis on supporting class evolution.

The ideas presented here are based on experience gained while using the Demeter CASE tools to implement systems for Citibank and Merrill Lynch. The systems were developed using class models that were under constant flux, requiring propagation patterns to be maintained to support the changing class structures.

1.1 Describing Class Structure with Class Dictionary Graphs

The examples presented in this paper are described using the Demeter data model for defining class structure and behavior [17]. Class structure can be represented graphically in a *Class Dictionary Graph* defined as $\Gamma = (V, \Lambda, E)$. V is a set of vertices in the graph which represent classes. E is a set of edges between vertices which represent relations among classes. Λ is a set of labels which represent the names of the relations.

The Demeter class model defines several kinds of classes, drawn as different types of vertices in the class dictionary graph. A *Construction* class denotes a concrete definition of some entity, and is drawn as a rectangular vertex. An *Alternation* class is an abstraction of the common attributes and behavior found among a group of objects, and is drawn as a hexagon. A *Repetition* class is a container class used to aggregate multiple instances of a class, and is drawn as a hexagon containing a rectangle. A *Terminal* class represents a basic data type, such as a *Number*, *Ident*, or *Real*, and is drawn as a rectangle. Construction, terminal and repetition classes are instantiable, while alternation classes are used purely for inheritance [9].

The Demeter model defines several types of relations among classes. A *Construction* edge represents the *uses* or *part-of* relation, and is drawn as a single line arrow. An *Alternation* edge represents the *isa* relation, and is drawn from superclass to subclass as a double line arrow. The inverse of the alternation relation is the inheritance relation, which is represented by an *Inheritance* edge, and is drawn from subclass to superclass as a double dashed line arrow. It is not necessary for the model to include both *inheritance* and *alternation* edges, since existence of one will imply the other in an inverse direction. Allowing the existence of both edges can simplify the visual depiction of paths in the graph.

A *Repetition* edge indicates the relation between a Repetition class and the class that it aggregates. Finally, a *Behavioral* edge indicates a relation between a source and target class which results from the source class executing a behavior which creates a relation to the target class, and is drawn as a single dashed line arrow. A model for extending the class dictionary graph to include behavior as a relation was initially presented in [16].

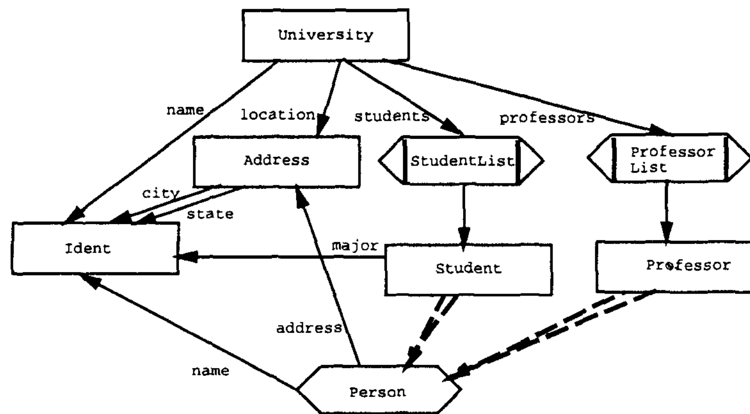

Fig. 1. A Class Dictionary Graph Describing the University Domain.

Figure 1 contains a class dictionary graph that describes the organization of classes in the university domain. The instantiable construction classes are *University*, *Address*, *Professor* and *Student*. *Person* is an alternation class. *Student* and *Professor* are the alternatives, or subclasses of *Person*, that inherit the common attributes and behavior of the *Person* class, a *name* and *home* address. *ProfessorList* and *StudentList* are repetition vertices, each representing a class that collects multiple instances of the associated class, *Professor* or *Student*. *Ident* is a terminal class, a predefined class that will instantiate identifier objects. The class dictionary graph describes only the structural relationships of the classes; behavior has not yet been added to the graph.

Given a class dictionary graph, the Demeter Tool generates C++ code to represent the organization of classes. It is easy to evolve the class structure by simply reorganizing the class dictionary graph. C++ code will be generated to correspond to the new class structure.

2 Defining Object Behavior

An object has certain responsibility and can make requests of other objects to help it accomplish a task. One task in the university domain might be to determine which professors have a long commute to work, which may be relevant in trying to schedule early morning classes. In an object oriented program, functionality is implemented by attaching responsibilities to classes, abstracting common behavior to a superclass, and using message passing protocols to disperse responsibility among many classes to accomplish a task.

It is often the case when implementing a task that it is necessary to involve several classes, each responsible for requesting some behavior of another class which may contain data relevant to the task at hand. Often behavior consists of propagating a request along a path of relations in the class dictionary graph,

with certain classes along the path performing work in addition to message propagation. Following the guidelines of encapsulation and delegation to implement behavior can facilitate the maintenance of code if each class minimizes its' dependency on knowledge of the structure of other classes. Well written programs that follow the Law of Demeter [13] may be more easily maintained when class structures evolve, due to a coding style which minimizes reliance on a particular class structure. While this style of coding improves maintenance in some aspects, it forces one to write many small methods which primarily consist of code to propagate a message along a path of relations. There exists the issue of maintaining these methods when class structures change and new message paths must be found. Propagation Patterns facilitate this process.

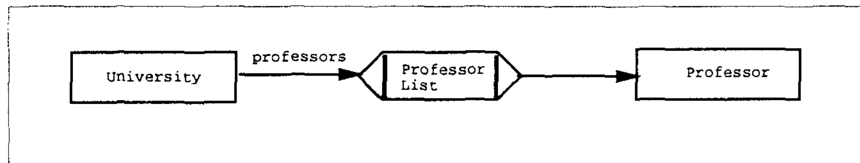

Fig. 2. Propagation Graph Describing A Path From University To Professor

For instance, to collect a list of commuting professors in a given university, the *University* class would need to request a list of commuting professors using the *professors* relation. The *professors* relation produces an instance of the *ProfessorList* class, which would iterate through each of the *Professor* objects that it aggregates. Each *Professor* class instance would be responsible for supplying behavior to provide its' *address* for comparison. The implementation of this task consists of several small methods being implemented, with reliance on the current class structure and relationships. The subgraph of the Class Dictionary Graph involved in implementing this task is shown in Figure 2. This is called the *Propagation Graph* and describes the classes and relations used in accomplishing the task.

2.1 Describing Behavior with Propagation Patterns

Code designed to propagate a message request along a path of class relations can be time consuming to produce and difficult to maintain. Certain communication paths among classes tend to be reused often, with different tasks attaching behavior to different classes along those communication paths. It is useful to define paths among classes, and augment those paths with task specific code to implement a particular behavior.

A *Propagation Pattern* [17, 15, 10] defines behavior by specifying the class collaborations which occur to implement a task, while avoiding writing traversal code that is overly dependent on the class structure. They provide a necessary level of abstraction when implementing object behavior which can facilitate class

evolution. The propagation pattern to accomplish the task of determining commuting professors is shown in Figure 3. While propagation patterns have been implemented based on the Demeter class model, most of the concepts presented in this paper could easily apply to other data models.

```
*operation* ProfessorList* commuters()
*init* (@ new ProfessorList() @)

*dir* allProfsPath = *from* University *to* Professor

*traverse* allProfsPath

*carry* Address* univAddress
 *along* allProfsPath
*at* University univAddress = (@ location @)

*wrapper* Professor *prefix*
(@
   if (univAddress->farFrom(get_address()))
      return_val->append(this);
@)
```

Fig. 3. A Propagation Pattern For Obtaining The List Of Commuting Professors.

The structure of this propagation pattern contains an interface statement with the method name *commuters* and return type *ProfessorList**. A message is sent along a path from the *University* class to the *Professor* class, with the University's *location* being transported along the path as an object that is accessible to other classes. The *Professor* class has the responsibility shown in the wrapper code which adds the current *Professor* object to the resulting list if the Professor's address is far from the University's address.

The behavior implemented by this propagation pattern effectively modifies the class structure defined in the original class dictionary graph to add a behavioral edge to the *University* class. The propagation pattern adds a new relation called *commuters* between the *University* class and the *ProfessorList* class. This new edge can now be used by other propagation patterns in defining paths.

Definition 1. The definition of a *Propagation Pattern* for a given class dictionary graph $\Gamma = (V, \Lambda, E)$ is a tuple of the form (S, M, PD, TP, CF).

- S is the *Signature* of the behavior being defined, and is of the form $(rettype, fname, init)$ where:
  - *rettype* is the return type of the behavior, $rettype \in V$
  - *fname* is the name of the behavior
  - *init* is an optional expression which initializes the result of the behavior.

The Signature S can be described in a textual form as:

```
*operation* rettype fname()  [ *init* (@ expression @) ]
```

There is a special variable named *returnval* of type *rettype*, which holds the return value of the behavior being implemented. It can be initialized using the **init** expression, and modified by any class that has the *fname* message propagated to it.

– M is a *Meta Declaration*, which is used to define constraints in the propagation pattern. While there may be several types of constraints, this paper will only describe a *Meta Graph Directive* constraint, which is used to specify a subgraph of the class dictionary graph. A Meta Graph Directive has the form $(GDName, GD)$, where *GDName* is a variable name used to represent the graph directive GD. A propagation pattern can define several graph directives.
A Meta Graph Directive has a textual form:

```
*dir* GDName = GD
```

GD is a *Graph Directive* which specifies a subgraph of a class dictionary graph. A GD has the form (F, I, X, V, T) where:

- F is a non-empty set of vertices in the class dictionary graph specifying the starting or source vertices in the subgraph, or **from** classes.
- T is a set of vertices in the class dictionary graph specifying the ending or target vertices in the subgraph, or **to** classes.
- V is a set of vertices in the class dictionary graph specifying vertices which the subgraph must contain as intermediary vertices along the subgraph, these are the **via** classes.
- I is a set of edges in the class dictionary graph which the subgraph must include. These are the **through** edges.
- X is a set of edges in the class dictionary graph which the subgraph must exclude. These are the **bypassing** edges.

A graph directive GD has the textual form:

```
*from* class
[ *through* edge-patterns ]
[ *bypassing* edge-patterns ]
[ *via* class-set ]
[ *to* class-set ]
```

A *class-set* refers to a comma-separated list of class names, and an *edge-pattern* has one of the following textual forms:

```
-> class, label, class    (a construction or behavioral edge)
=> class, class           (an alternation edge)
<= class, class           (an inheritance edge)
~> class, class           (a repetition edge)
```

Given a class dictionary graph Γ and Graph Directive GD, the corresponding subgraph is abstracted. A vertex v or edge e is included in the subgraph if it is located along the path defined between the **from** vertices and the **to** vertices. The path must contain all vertices in the **via** clause and edges in the **through** clause, while excluding all edges given in the **bypassing** clause.

- *PD* is a *Propagation Directive* which specifies a *Propagation Graph*, a subgraph of a class dictionary graph which collaborates in the implementation of the behavior. A *PD* is defined using a graph directive.
  The Propagation Directive *PD* can be described in a textual form as:

```
*traverse* GDName
```

  GDName must be a defined meta graph directive variable.
  A propagation directive will define a subgraph of the class dictionary graph, the *Propagation Graph*. Each class along the propagation graph will have a C++ member function generated that will propagate the behavior to any outgoing edges for that vertex that are in the propagation graph.
- *TP* is a *Transportation Pattern* which specifies how to transport objects along portions of the propagation graph. A propagation pattern can define several Transportation Patterns. Transportation allows classes along a subgraph of the propagation graph to transport information for use by other classes. A *TP* has the form (TT, TN, TD, TA) where:
  - TT is the type of the object being transported, $TT \in V$.
  - TN is the name of the object being transported.
  - TD is the *Transportation Directive* which defines a *Transportation Graph*, a subgraph of the propagation graph along which the object is transported. A transportation directive is defined using a graph directive.
  - TA specifies the value assignment of the transported object at a particular class along the transportation graph. It is of the form (v, e) where v specifies a vertex, and e specifies the expression that the object is being assigned in the method generated for vertex v.

  The Transportation Pattern *TP* can be described in a textual form as :

```
*carry*
  vartype varname,
   *along* GDName
 *at* class-set
 varname = (@ expression @)
```

  Transportation indicates an additional argument added to the signature of the method for each class along the Transportation Graph, which has the name and type of the transported object. This allows classes along the Transportation Graph to access or modify the transported object.
- *CF* is a *Code Fragment*, which has the form (t, v, cf). A propagation pattern can define many code fragments. Code fragments define behavior for class v in addition to the traversal behavior that is defined by the Propagation Graph.

- t specifies the type of code fragment, which is either prefix or suffix. Prefix code fragments contain behavior which should be executed before traversal behavior for the class v. Suffix code fragments contain behavior that should be executed after traversal behavior.
- v is a vertex in the propagation graph.
- cf is a code fragment describing the prefix or suffix behavior for vertex v.

Code fragments are represented in a textual form as:

```
*wrapper* class
*prefix*
(@ C++ statements @)
*suffix*
(@ C++ statements @)
```

Demeter implements the propagation pattern functionality by generating code to perform the behavior defined by the propagation pattern. For each vertex in the propagation graph, a C++ member function is created for the corresponding class that the vertex represents. The member function will contain traversal code to propagate the message along each outgoing edge contained in the propagation graph. In addition to this traversal code, any prefix or suffix code fragments that were defined for the vertex will be added into the C++ member function. The signature of the member function is extended for any class along the transportation graph to include an argument for the transported object.

A propagation pattern, as defined for a particular class dictionary graph, must satisfy several constraints in order to be considered *Legal* [17]. A propagation pattern is legal for a particular class dictionary graph if the propagation and transportation directives define valid paths in the class dictionary graph. There must exist at least one path in the class dictionary graph between the source or **from** vertices and the target or **to** vertices, including all **through** edges and **via** vertices, while avoiding any **bypassing** edges. The propagation pattern must also satisfy legality constraints involving code fragments. Each code fragment specified must be for a vertex that is defined in the propagation graph. The propagation pattern must satisfy legality constraints for the transportation pattern, which include specifying a legal transportation graph, as well as ensuring that assignments occur only at vertices defined in the graph.

3 Propagation Patterns Facilitating Class Evolution

Behavior is often implemented based on the hope of a sturdy initial class design, yet it is usually the case that the class design must continually adapt as the application domain evolves. This can be difficult to do once there exists a large body of methods which rely on a particular class structure.

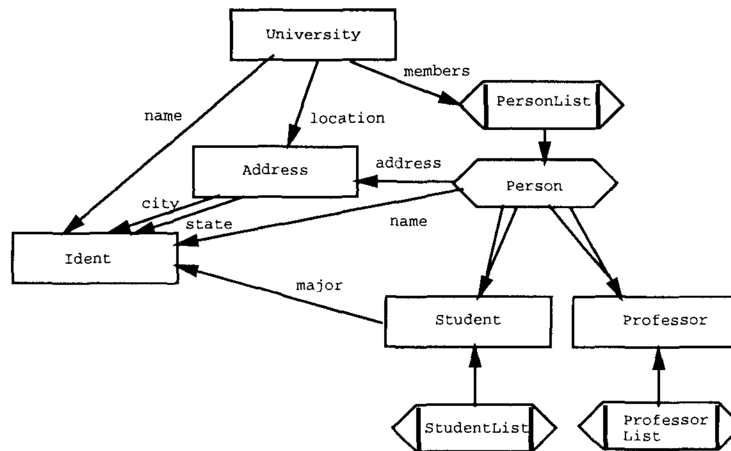

Fig. 4. A Different Class Dictionary Graph For The University Domain

Propagation patterns ease this maintenance issue by providing a more abstract manner to implement behavior than reliance on hand written C++ coding. If the class dictionary graph in Figure 4 is used instead of the original, the propagation pattern in Figure 3 does not need to be modified to maintain behavior, whereas hand-written C++ code would have to be modified to use the new class relations. Class transformations have been studied in [1, 7, 4, 6, 14], most of which discuss structural transformations, but do not address the impact of class evolution on existing behavioral implementations. In [2] the maintenance of $C++$ and CLOS code during class evolution is compared. In [12] the effects of class evolution on propagation patterns are introduced, however the model for propagation patterns and class dictionary graphs did not include behavioral edges or transportation directives. In this section, class transformations are presented, and the potential impact and maintenance requirements are detailed.

3.1 Maintaining or Extending Object Behavior

Given an existing class dictionary graph G and propagation pattern P, a transformation is applied to G which will result in a new class dictionary graph G' and may potentially require adaptation of P to remain legal.

When adapting a class structure, there are primarily two approaches one can take concerning the maintenance of existing behavior. One approach is to strictly maintain the original behavior, such that any object that can be described by both the original and the transformed class dictionary graph should behave in essentially the same manner, while excluding behavior from new objects. This approach may require the propagation pattern to be modified to define new propagation and transportation directives in an attempt to preserve the original propagation and transportation graphs, as well as ensure that the propagation pattern is still legal.

The other approach taken during class evolution is to allow the propagation pattern to remain essentially the same, checking only that it is still legal and evolving it minimally to ensure this. In this second approach, the propagation pattern may now extend or decrease the behavior of objects from their original behavior. For instance if a new relation is added to the class dictionary graph which adds an additional path to the propagation graph, objects will now communicate using this new relation, as well as the other relations which existed in the old propagation graph. Under the first approach, the new relation would automatically be added to a **bypassing** clause in order to strictly maintain the original behavior.

3.2 Class Transformations

In this section primitive and compound transformations are described, with the resulting maintenance requirements for propagation patterns detailed. Modifications which occur along a portion of the class dictionary graph that fall within the bounds defined by the propagation graph or transportation graph can automatically be maintained when the propagation pattern is regenerated.

The maintenance requirements for Primitive Class Transformations are now presented.

- Addition of a new vertex v to the existing set of vertices. This basic transformation alone, without the addition of edges which include v, will have no effect on existing propagation patterns, since there does not yet exist a path which could include it.
- Deletion of a vertex v. It is assumed that all edges which include a vertex v must first be deleted before v can be deleted. If v is used in a propagation pattern, then most issues which occur in deleting v will have already been covered during deletion of edges which utilize v. The only remaining requirement is that v not exist in a propagation or transportation directive, signature return type, or code fragment. If v is found in the signature return type, the propagation pattern is no longer legal, since the result of the propagation pattern is no longer an existent class. If v has a prefix or suffix code fragment attached to it, the propagation pattern is not legal since behavior is being defined for a nonexistent class. If v is found in a directive, it must be removed in order for the directive to be legal. Removal of v from the directive may cause it to no longer define a legal path, in which case the propagation pattern is no longer legal. This may imply that the behavior can no longer be implemented.
- Rename a vertex v to v'. Vertex v must be replaced with v' wherever v is referred to in the propagation pattern signature, propagation directive, transportation directive or code fragment. While this may require manual intervention, there is no work required when other vertices contained within the graph defined by a directive are renamed, since the code will be regenerated using the correct name.

- Addition of a new edge. To maintain the original behavior, automatically add the edge to the **bypassing** set (X) for any Graph Directive which will otherwise include the edge in its' defined subgraph. To extend or refine behavior, allow the new edge to potentially be used in the original directives to define new propagation and transportation graphs. This second approach may drastically change the behavior being defined by the propagation and transportation directives.
- Deletion of an edge. If the edge is defined in the **bypassing** set (X), remove the edge from the set to maintain a legal propagation pattern. If the edge is included in the **through** set (I), the propagation pattern will no longer be legal. The edge can be removed from (I) and another path will be used, if one exists. If deletion of the edge causes the propagation or transportation graph to become disconnected, the propagation pattern is no longer legal. If the edge is used in a code fragment, or initialization expression, the propagation pattern is no longer legal.
- Rename an edge e to e'. If the edge e is referred to in the **bypassing** or **through** sets (X, I), a code fragment or initialization expression, it must be updated to e'. Renaming of an edge which is not explicitly defined in a graph directive, but contained in a propagation or transportation graph, does not require manual maintenance since the correct code will be regenerated.

Single primitive transformations by themselves are not typically the way a class dictionary graph evolves. Experience based on transforming class dictionary graphs during application development shows that compound class transformations are often performed to a class dictionary graph and the maintenance of propagation patterns should support these higher level transformations. The maintenance requirements for compound class transformations are presented.

- Transform a construction edge to a behavioral edge. This transforms a relation from being stored (a construction edge) to being derived (a behavioral edge), and is a common transformation. From a modeling viewpoint, these two edge types should be interchangeable, since it is usually a design or performance decision to either store or calculate an attribute. This transformation should require no maintenance of the Propagation or Transportation Directives. It may be necessary to add code to handle storage issues involving the allocation and deallocation of objects, but this is dependent on garbage collection tactics [16]. Code fragments which refer to the original relation may need to be adapted to add argument parenthesis () after the label, which could be automated.
- Transform a behavioral edge to a construction edge. Opposite requirement of previous transformation.
- Abstract a relation l up the inheritance hierarchy to a superclass, indicating the deletion of a construction edge (v, l, w) and the addition of a construction edge (v', l, w), where v' is a superclass of v. This transformation occurs when a relation found in a subclass is deemed appropriate to be inherited from a superclass. If there is a subclass u of v' which did not originally have the

relation l, to maintain original behavior an inheritance edge (w, v') and an alternation edge (v', w) must be added to the **bypassing** set X. If the original edge (v, l, w) was contained in a bypassing (X) or through (I) edge directive, it must be replaced either with the new edge (v', l, w) to affect all of the subclasses which now inherit the relation, or replaced with the inheritance edge (v', v) or alternation edge (v, v') to maintain the directive. Using meta-characters for edge specifications can avoid this problem, such as specifying a **bypassing** edge as $(*, l, *)$ instead of specifying the source and target class names of the relation l.

- Distribute a construction edge down the inheritance hierarchy. Again if the edge is specified in a graph directive, the new edge(s) will have to be specified. If the edge is not specified directly, the propagation and transportation graphs will be correctly computed and the C++ code regenerated. If the relation represented by the construction edge was used in a code fragment, initialization or transportation assignment attached to the superclass, this would have to be modified to attach the code fragment or assignment to each subclass which now contains the relation.
- Replace a direct relation or edge between two vertices with a sequence of edges. This transformation often occurs when additional partitioning of objects is needed. Two objects must go through a longer sequence of objects to communicate a message. If the original edge was used in a directive, it must be replaced with enough of the new path to distinguish the new path from any other paths, potentially replacing it with the entire new path. If the old edge was not directly contained in a directive, but was contained in either the propagation or transportation graph, the new path should also be contained in the graphs, and therefore the correct code will be regenerated to utilize the new path.
- Replace a sequence of edges between two vertices with a direct edge. This transformation occurs when it is decided to simplify the object structure. The maintenance requirements are similar to the previous case.
- Generalize the domain of a relation. In this case a construction edge (v, l, w) is replaced by a construction edge (v, l, u) where u is a superclass of w. To maintain the original behavior, it is necessary to exclude any of the new objects which might have behavior propagated to them, namely the subclasses of u other than w. Therefore an alternation edge (w, w') and an inheritance edge (w', w) are added to the **bypassing** set (X) for each subclass (minus w) of u.

  If the original edge (v, l, w) was contained in a propagation or transportation directive, it must be replaced with the new generalized edge (v, l, u) for the directives to remain legal. Any code fragments which utilized the old edge should still hold correct. This maintenance effort would not be necessary if meta-characters are used in the edge specification.
- Specialize the domain of a relation. Here construction edge (v, l, w) is replaced by construction edge (v, l, u), where w is a superclass of u. The original behavior can not be maintained, since objects which received a message

request given the original class dictionary graph will not be contained in the new propagation graph.
The behavior can only be refined, with substitution of the old edge for the new in any graph directives to ensure the propagation pattern is still legal.

4 Conclusion

As many of the transformations show, there may be manual effort required to maintain a propagation pattern when the class dictionary evolves. However there are many transformations that require minimal effort in comparison to maintaining hand-written C++ code. Using graph directives to specify traversal paths can greatly facilitate the maintenance of object oriented programs, which are highly reliant on class structures when implementing behavior. Defining a graph directive in a meta declaration and reusing it in many propagation directives and carry directives also facilitates maintenance since the communication paths among objects in the form of a graph directive need only be specified and maintained in one place. The benefit of propagation patterns is the ability to minimize hard-coding the class structures into C++ code, so that evolution is supported.

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 graph directives were consistent with the new class structure and no change was required, the code was simply regenerated to fit the new structure.

Utilizing high-level abstractions like class dictionary graphs and propagation patterns can further expand the benefits of object oriented technology by minimizing the maintenance effort required when application domains change. The ability to support and encourage change is a necessary part of any software development model.

Acknowledgements

I would like to thank Karl Lieberherr, Greg Sullivan and Cun Xiao for providing many useful ideas about propagation patterns and modeling behavior. Cun Xiao has produced a powerful propagation pattern tool and has patiently implemented many enhancement requests.

References

1. 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).
2. Paul L. Bergstein and Walter L. Hürsch. Maintaining behavioral consistency during schema evolution. pages 176–193, Kanazawa, Japan, November 1993. JSSST.

3. Grady Booch. *Object-Oriented Design With Applications.* Benjamin/Cummings Publishing Company, Inc., 1991.
4. Eduardo Casais. An incremental class reorganization approach. In *European Conference on Object-Oriented Programming.* Springer Verlag, 1992.
5. Peter Coad. Object oriented patterns. *Communications of the ACM*, 35(9):153–159, September 1992.
6. 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.
7. 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.
8. Ian M. Holland. The design and representation of object-oriented components. Technical report, Northeastern University, 1993. Ph.D. thesis.
9. Walter L. Hürsch. Should Superclasses be Abstract? In *European Conference on Object-Oriented Programming*, Bologna, Italy, July 1994. Springer Verlag, Lecture Notes in Computer Science. To appear.
10. Walter L. Hürsch, Linda M. Seiter, and Cun Xiao. In any case: Demeter. *The American Programmer*, 4(10):46–56, October 1991.
11. Ralph E. Johnson. Documenting frameworks using patterns. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, Vancouver, Canada, 1992. ACM Press.
12. Linda Keszenheimer. Specifying and adapting object behavior during system evolution. In *Proceedings of the 8th International Conference on Software Maintenance*, pages 254–261. IEEE Computer Society, 1993.
13. Karl J. Lieberherr and Ian Holland. Assuring good style for object-oriented programs. *IEEE Software*, pages 38–48, September 1989.
14. Karl J. Lieberherr, Walter L. Hürsch, and Cun Xiao. Object-extending class transformations. *Formal Aspects of Computing, the International Journal of Formal Methods*, 1993. Accepted for publication, also available as Technical Report NU-CCS-91-8, Northeastern University.
15. Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, May 1994. Accepted for publication.
16. Karl J. Lieberherr and Greg T. Sullivan. Procedural extensions of class dictionary graphs. Technical Report Demeter-9, Northeastern University, March 1992.
17. Karl J. Lieberherr and Cun Xiao. Object-oriented software evolution. *IEEE Software*, pages 313–343, April 1993.
18. P. Poncelet and L. Lakhal. Consistent structural updates for object database design. In *Proceedings of the Conference on Advanced Information Systems Engineering.* Springer-Verlag, 1993.
19. Christiaan Thieme and Arno Siebes. Schema integration in object-oriented databases. *Proceedings of the Conference on Advanced Information Systems Engineering*, 1993.