This paper identifies a tension between the benefits of adaptive programming and those of encapsulation. It begins by pointing out several common manifestations of the tension, then it proposes a source of the tension. Finally, it present a couple potential solutions and suggests a direction for future development that seems most promising.

1. Introduction

The goal of adaptive programming has long been to specify programs in the most 'structure shy' manner possible. This is accomplished mainly by using traversal strategies to treat behavior at a higher level of abstraction, separating it from the details of a particular class structure. Later, as part of the compilation process, this higher level behavior would be combined with the details of a specific class structure to create a concrete program. The idea is that this separation allows behavior to be more robust with respect to a large group of evolutions in the class structure[ref all the papers that talk about this; lieber's book, chun's thesis?, greg's paper?].

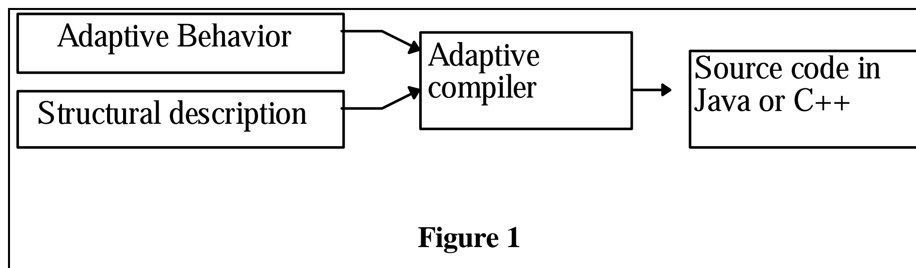

Figure 1

This adaptive method of program specification works exceptionally well for many types of evolutions, however it is still vulnerable to certain types of evolutions. This paper focuses on one of these vunlerabilities, a change in representation. Informally, the vulnerability is present because there is no way to talk about information hiding in the structural description language. This means that representation details have to be scattered through a program's adaptive behavior. As change of representation is such a common evolution, it seems that this is a problem that must be addressed.

This paper's purpose is not to present adaptive programming. If you are unfamiliar with adaptive programming concepts you will probably need to see [ref lieber's book, lieber/boaz's new paper?, others?] before getting the most out of this paper.

OUTLINE THE REST OF THE TECH REPORT

2. The Problem

To some extent, programmers are forced to choose between the benefits gained from encapsulation and those gained from adaptiveness. For example, look at the the class graph snippet shown in the Figure 2. This simple conceptual picture is not representable with current adaptive methods. The problem is that when a part of the program's structure becomes visible to some adaptive behavior it becomes visible to all of the adaptive behavior. There are many possible work around, but all of them have the disadvantages that behavior specifies more details about the program's structure than absolutely needed, and the resulting software is more ridgid.

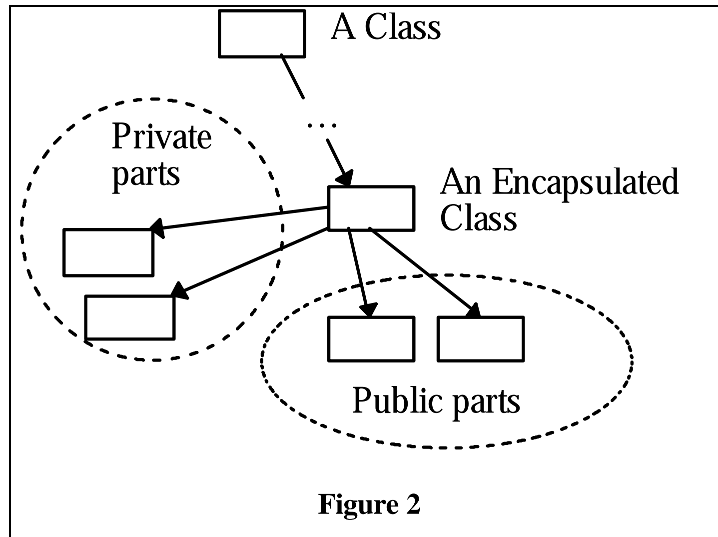

Figure 2

2.1 Complex Example

Adaptive behavior must mention implementation details found in a program's class graph. For example, an application that deals with complex numbers will need to pick a representation(polar or rectangular), and specify it's strategies and behavior in terms of that representation. Later, if the representation is changed, the strategies and behaviors will need to be updated. Further, it may be difficult to tell exactly which behavior are effected by the change in representation.

The figure below (REF IT) shows an example of how adaptive methods help in evolutions. One of the assumptions of adaptive programming has been that behavior can be specified in terms of a few major structural participants, only woring vaguely about the raltionships among them. Later, this higher level behavior would have a clear (one to one) mapping onto a large group of concrete class structures.

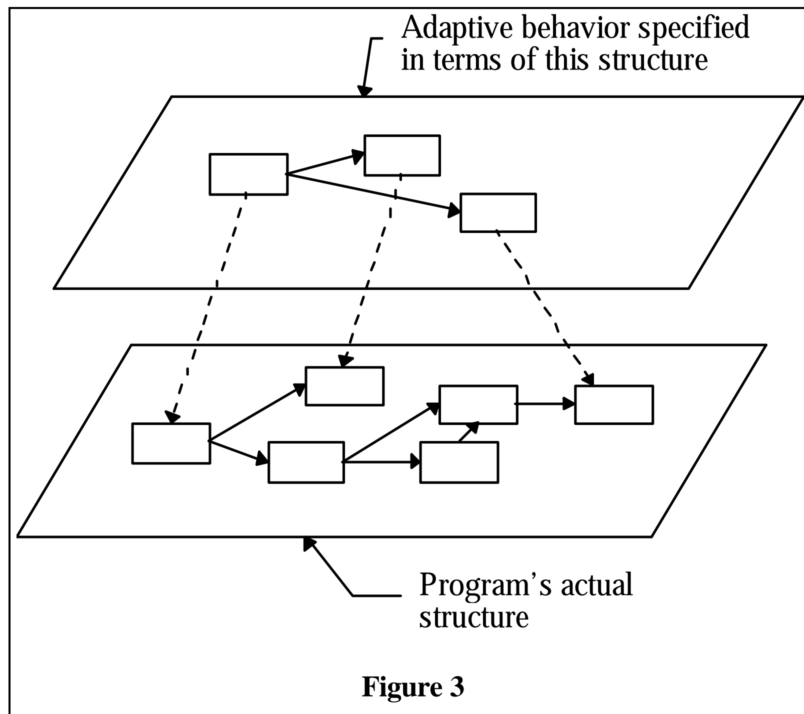

Figure 3

This works out very well for many types of evolutions (adding or deleting edges, changing labels on parts or edges, adding or deleting vertices, and others [REFERENCE SOMETHING]) but not all of them. Consider the next two figure. The first shows an abstract view of an adaptive behavior for complex numbers. It is trivial to see how this behavior will work on a structure shown in REF FIGURE.

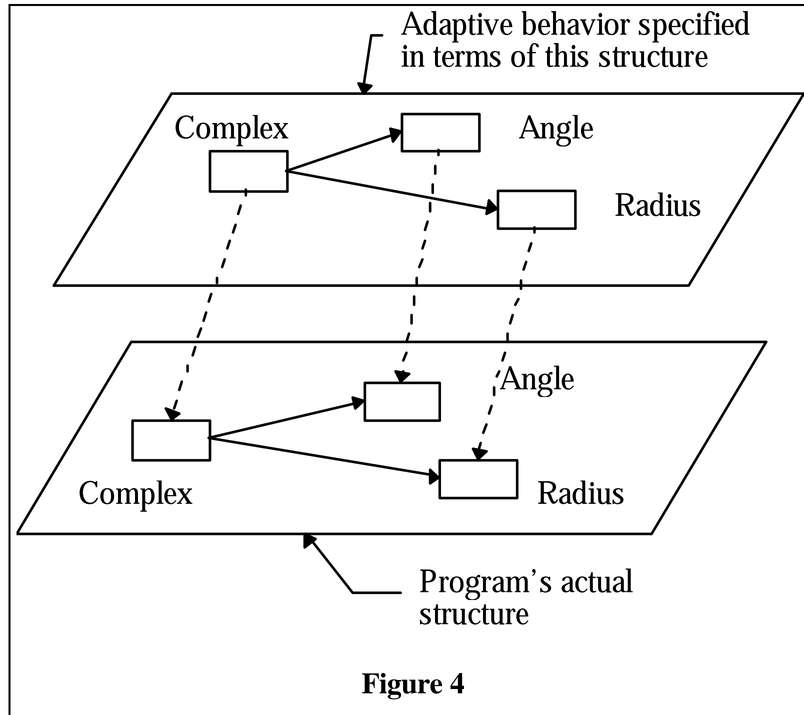

Figure 4

REF NEXT FIGURE shows what happens when there is a change in the structure's representation. There is no longer a clear mapping from the major players specified in the adaptive behavior onto the program's structure. In fact, all adaptive behavior specified in terms of the old representation will need to be modified to work with the new representation.

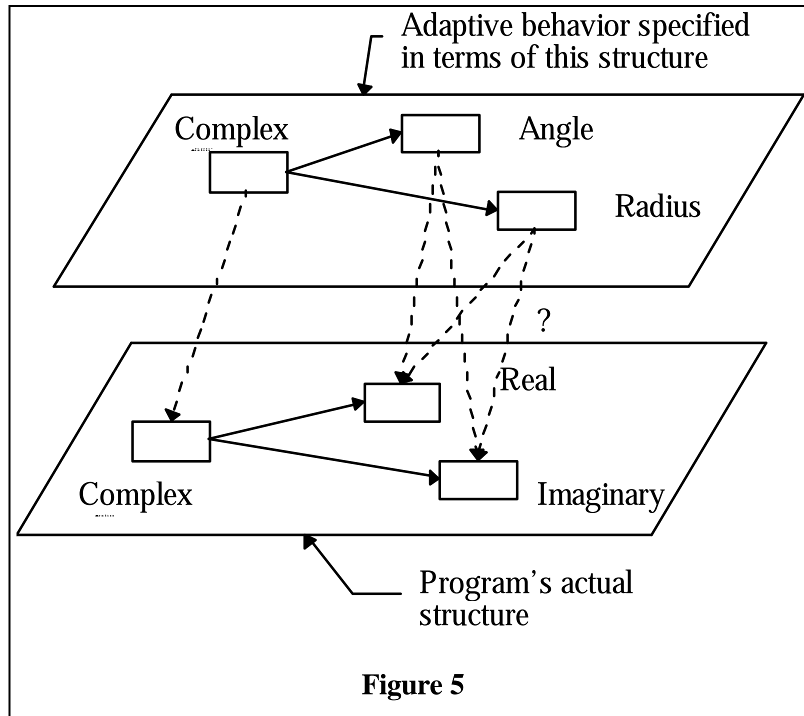

Figure 5

Informally, this vulnerability can be worked around by taking classes with potentially volitile implementations out of the adaptive part of a program. So, instead of using traversal strategies directly to get access to the parts of these volitile classes, stratigies must use wrapper code at the volitile class to access the classes public parts. Perhaps starting another traversal on the retrived part. This work-around forces behavior to mention more details about a program's structure than absolutely needed, it also forces programmers to manually write sections of the program's traversal code. So programs must weigh one form of flexibility for another.

2.2 Cache Example

All of a program's strategies must be manually checked for inadvertant changes every time the program's class graph changes. This is not true for every class graph evolution, but unfortunately holds for some common changes of representation. For example, imagine an adaptive program with a strategy S that targets all objects of class Point (see REF NEXT FIGURE).

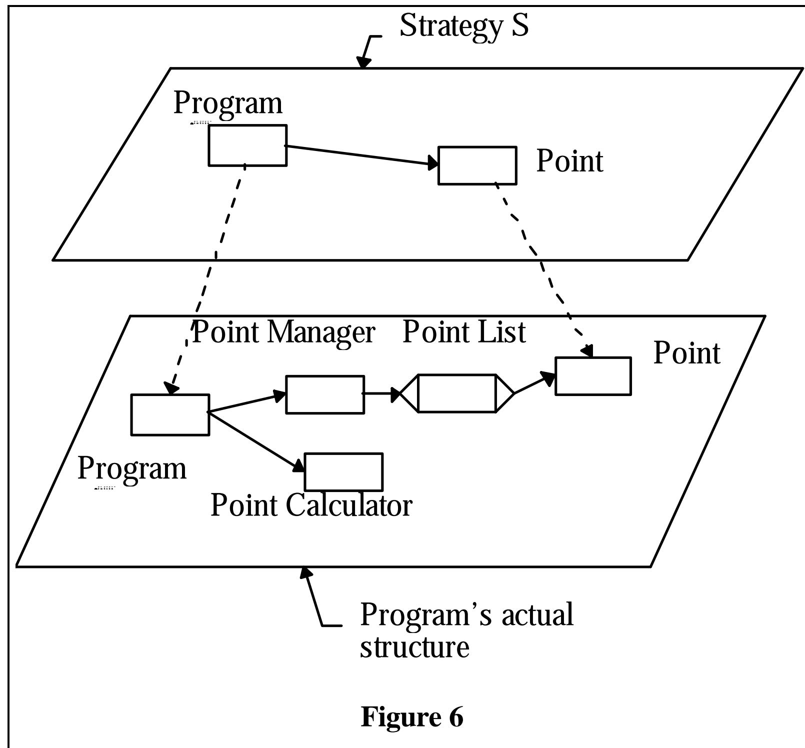

Figure 6

Later a runtime cache of Points is added to the system for efficiency reasons, see REF FIGURE BLEOW. S will begin reaching the Points in the cache, and no longer work as expeted (Imagine S is used with a visitor to count the number of points in the system). Informally, there are two obvious work-arounds for this problem; take the cache out of the adaptive level (see REF COMPLEX EXAMPLE SECTION), or update S (and all other strategies like it) to explicitly bypass the cache. In any case, the tedious and error prone problem of identifying places where such work-arounds are needed remains. There seems to be a real need for some mechinism that buffers strategies from these types of evolutions.

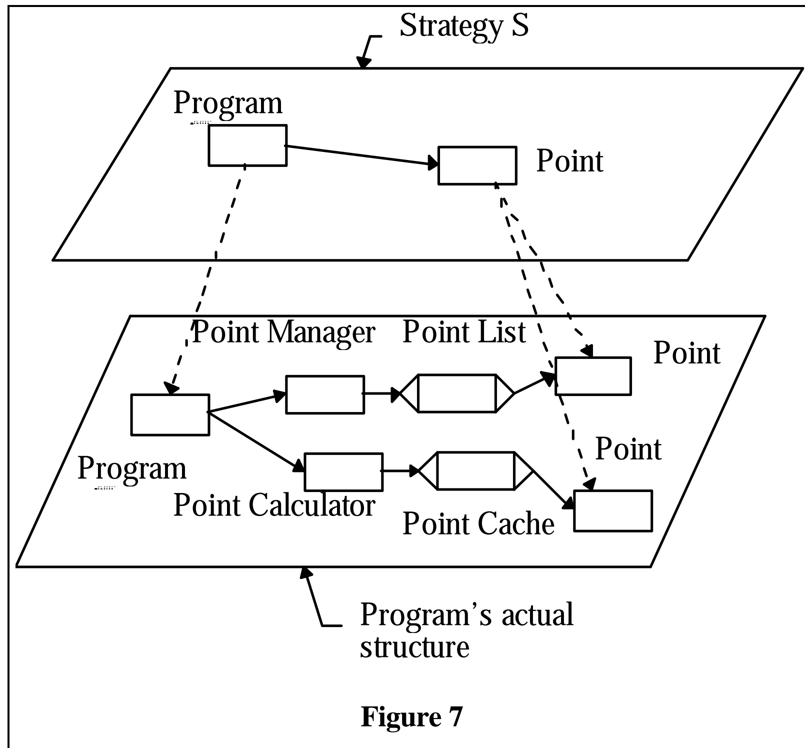

Figure 7

2.3 Higher Level View

The previous two examples presented the problem on a class based scale, however module level encapsulation is is also. At the class level, an interface is a set of public parts on the class, and a representation is a set of private parts. . On the module level, an interface is a set of collaborating classes and their inter-relationships, and an implementation is another set of collaborating classes and their inter-relationships. REF NEXT TWO FIGURES illustrate some of the issues involved in this higher level view. The REF NEXT FIGURE shows the current method of adaptive programming where behavior is specified directly in terms of the representation. When a piece of the program's representation changes, potentially all of the behaviors will need to be modified.

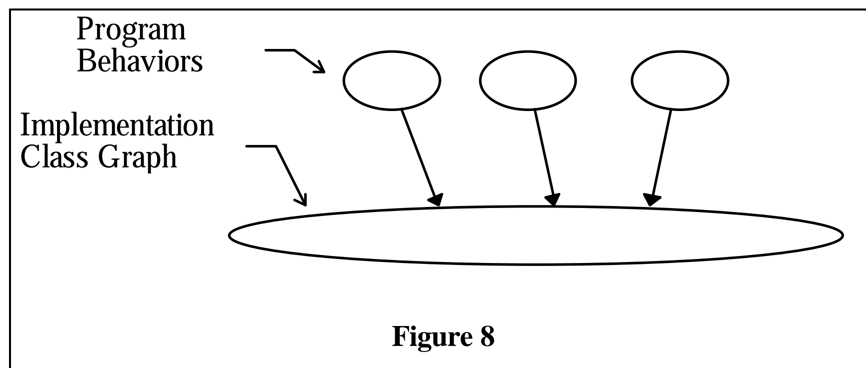

Figure 8

REF NEXT FIGURE shows an updated view of adaptive programming. A group of mappings buffer the program's behavior from the details of it's class structure. When a piece of a program's representation changes under this view, the only thing that needs to be modified is the specific mapping that is associated with the representation. This makes adaptive programs potentially robust with respect to a much larger class of common program evolutions.

An interesting thing to note about REF NEXT FIGURE is that the Representation Class Graphs can themselves contain Encapsulated modules. This seems to lead to a natural way of expressing layered adaptive software systems.

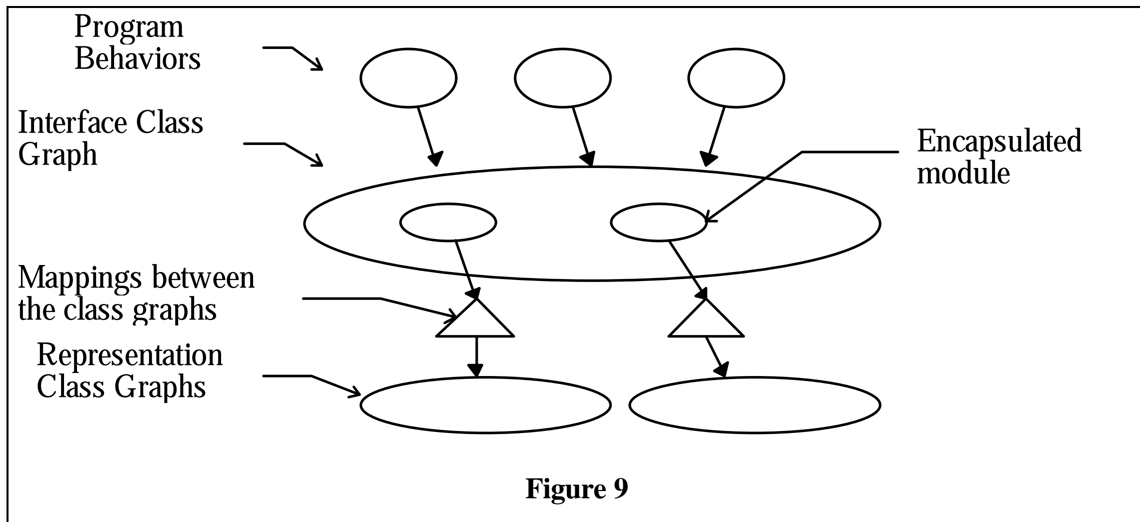

Figure 9

3. Potential solutions

The flavor of all of these proposed solutions is that they support one or more layers of indirection between a program's structure and the adaptive behavior that works on it. This indirection allows the representation of the structure more freedom to evolve, while still allowing a program to take advantage of the flexibilities offered by adaptiveness.

3.1 Derived and Hidden edges

The idea of this solution is that it allows a program to contain parts that don't directly exist in its structure and to hide parts that do exist in its structure. Using these techniques, it is possible to present an interface to a program's structure that is very different from the structure's representation.

3.1.1 Define derived edges

Derived edges are very much like the functional edges presented in REF LIEBER AND SULLIVAN PAPER. The main difference between Sullivan and Lieberherr's functional edges and the derived edges used here is that their edges could have multiple sources while these can appear anywhere a construction edge can appear.

Here is a brief definition. Any construction edge in a class dectionary can be labled as a derived edge. Instead of indicating that class u has a particular instance variable, a derived edge *u -> l v* indicates that class *u* has a function named *get_l()* that returns a value of class *v*. It also indicates that class *u* has a function named *set_l()* that takes one argument of class *v* and returns nothing.

3.1.2 Hidden edges

I beleive that the layout presented in **Error! Reference source not found.** can be achieved in terms of derived edges (as presented by lieberherr and sullivan) and hidden edges (which I will present

here. Briefly, the semantics of a hidden edge is that all traversal stratigies implicitly bypass the edge unless they explicitly mention it). Informally, you make all of the classes representation hidden, and then construct the interface to the class as a bunch of derived edges that explicitly use the classes representation.

3.1.3 How this helps

SHOW SOME MORE FIGURES OF THE COMPLEX/CACHE EXAMPLE AND HOW THIS DERIVED/HIDDEN STUFF HELPS

3.2 Implementation Edges

3.3 Module System

4. Conclusion

Derived and hidden edges are useful on their own rights, but may make for a clunky solution to our problem.

Implementation edges are nice, but seem to be subsumed by the module system idea.

Module system has many advantages, but is also kind of subtle and needs a lot more work.