**Adaptive Object-Oriented Programming using Graph-Based Customization**

# 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, Eifel, 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 objects being able to receive messages, different classes of objects responding differently to a given message, and the ability to de ne standard protocols. On the darker side, by explicitly attaching every single method to a specific c 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 c 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? 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. 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 c 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 are specified using propagation patterns, which express program constraints.

# 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.

**Infinitely many class structures**

**Adaptive Program**



**satisfies**

**denotes**

**selects**

**.**

**. . .**

**selects**

**selects**

**. . .**

**. . . .**

**Family of programs**

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.