Adaptive Plug-and-Play Components for Evolutionary Software Development

Mira Mezini and Karl Lieberherr
College of Computer Science, Northeastern University, Boston, MA 02115-9959
e-mail: {mira,lieber}@ccs.neu.edu

Abstract

In several works on design methodologies, design patterns, and programming language design, the need for program entities that capture the patterns of collaboration between several classes has been recognized. The idea is that in general the unit of reuse is not a single class, but a slice of behavior affecting a set of collaborating classes. The absence of large-scale components for expressing these collaborations makes object-oriented programs more difficult to maintain and reuse, because functionality is spread over several methods and it becomes difficult to get the "big picture". In this paper, we propose *Adaptive Plug and Play Components* to serve this need. These components are designed such that they not only facilitate the construction of complex software by making the collaborations explicit, but they do so in a manner that supports the evolutionary nature of both structure and behavior.

1 Introduction

This paper describes an approach to generic, composable components that encapsulate collaborations between objects, providing a linguistic construct for expressing design segments found in *collaboration-based (role-based) designs* [8, 28, 24]. The term *collaboration-based design* [1, 9, 23] describes a methodology for decomposing object-oriented applications into a set of classes and a set of *collaborations*. Collaborations express distinct and (relatively independent) aspects of an application which may involve several participants, or roles. Each application class may play different roles in different collaborations, where each role embodies a separate aspect of the class behavior.

In this way, collaboration-based designs represent object-oriented applications in two different ways: in terms of participants or classes that are involved, and in terms of the tasks or concerns of the design, as illustrated in Fig. 1. This twofold representation results in more comprehensive and reusable designs [1, 9, 23]. For the same reason, several other design methodologies support *collaboration diagrams* as one of their most important artifacts which facilitate understanding the overall behavior of high-level system operations resulting from the use-cases [4].

	class K1	class K2	class K3
collaboration C1	role $K_{1,1}$	role $K_{1,2}$	role $K_{1,3}$
collaboration C2	role $K_{2,1}$	role $K_{2,2}$	
collaboration C3		role $K_{3,2}$	role $K_{3,3}$
collaboration C4	role $K_{4,1}$	role $K_{4,2}$	role $K_{4,3}$

Figure 1: Collaboration-Based Decomposition

However, as also indicated in several other works [7, 8, 28, 24], object-oriented languages lack appropriate language constructs for expressing collaboration-based designs. This causes the clarity of the designs to get lost in the control flow of several small methods scattered around the class hierarchy, when mapped into code. The aim of our work is to bridge this gap between design and implementation. We propose a new component construct that:

(a) explicitly captures a slice of behavior (a specific task) that affects several classes,

(b) is complementary to the existing object-oriented

models in the sense that it does not substitute classes but rather complements them, and

(c) supports a decomposition granularity that lies between the granularity supported by classes and package modules.

The motivation behind (a) is that the unit of reuse is generally not the class, but a slice of behavior affecting several classes. The idea is that single methods often only make sense in a larger context, and are difficult to reuse individually. This is actually the core of the object-oriented application framework technology. However, frameworks are described by means of programming languages. As a result, they suffer from problems that are due to the lack of language constructs for clearly expressing patterns of collaborations [9, 12, 19].

This is supported by studies conducted by Wilde et al. [30] which observe that object-oriented technology can be a burden to the maintainer because functionality is often spread over several methods which must all be traced to get the "big picture". Similar results have recently been reported by a series of studies on real-life object-oriented systems by Lauesen [14]. He observes that object-oriented technology has not met its expectations when applied to real business applications and argues that this is partly due to the fact that there is no natural place where to put higher-level system operations which affect several objects. He justifies his claim based partly on the experience of companies which did not succeed with large OO systems until they put control flow and higher-level system operations outside class behavior: "if built into the classes involved, it was impossible to get an overview of the control flow. It was like reading a road map through a soda straw" [14].

The rationale behind (b) is that both classes and modules are essential language constructs for programming software systems [27]. The construct we are proposing does not substitute classes; it is a higher-level construct that complements classes in expressing collaborations rather than isolated behaviors. As stated in [28], it is the symbiosis of the two views, participant-based versus task-based, that gives the *collaboration-based design* its power. The participant view captures conventional notions of object-oriented design – classes are a perfect construct to map the participant view from design into code. The collaboration view captures crosscutting aspects of designs; without it, the relationships across objects are lost.

The requirement stated in (c) follows from the fact that classes are "too small" while package-like modules à la Oberon [31] are "too big" program building blocks. The class construct is a perfect vehicle to implement variations of an abstract data type. What is needed, however, is a mechanism to built higher-level program entities that capture the collaboration of classes. On the other hand, package-like modules are important to master complexity and are often used as a unit for separate compilation and dynamic loading, but they are only syntactic in nature and have no direct significance at run-time.

The new linguistic constructs proposed in this paper are called *Adaptive Plug and Play Components* (APPCs for short). The name stems from two features in their design. First, APPCs are called adaptive because they are designed to express collaborations that can be matched against a whole family of applications (class graphs). Collaborations are written to an abstract class graph that is made a formal parameter of an APPC, called the *interface class graph*. The interface class graph defines the "view" of the collaboration to a "true" class graph. By letting each component define a view (interface) to the "true" class graph, and by programming to that interface, we achieve a big win in reusability; to reuse the component, one needs only wrap the interface around another class graph and the component should work in that setting too. Some properties that must be fulfilled in order for the wrapping of the interface to work have been defined.

Second, APPCs are attributed as "plug and play" components because they have been designed to support flexible white/black-box compositions that allow to incrementally evolve existing collaborations, respectively to create higher-level collaborations out of simpler ones, while keeping the ingredients in the composition loosely coupled for better reuse.

The remainder of the paper is organized as follows. Section 2 justifies the need for the "adaptive" and "plug and play" features in the design of the intended component construct. Section 3 presents APPCs and shows how they actually provide the intended features. Some implementation issues are considered in Section 4. Related proposals are discussed in Section 5. Section 6 concludes the paper and outlines areas of future work.

2 Requirements on the Design of Collaborative Components

In order to outline and justify some requirements on the design of a language construct for explicitly expressing collaborations, we elaborate an example originally presented in a more simple form in Holland's thesis [9]. The example originated from an application system generator developed at IBM for the domain of order processing systems. The goal of this application generator was to encode a generic design for order entry systems that could be subsequently customized to produce applications meeting a customer's specific needs. Customer's

specific requirements were recorded using a questionnaire. The installation guide supplied with the questionnaire described the options and the consequences associated with questions on the questionnaire.

We start the discussion with a simplified version of the pricing component of such an application generator. Fig. 2 and 3 provide a partial specification of the functionality of this component by a partial class diagram and a simplified collaboration diagram for the main operation, price(). Although the design presented in these two diagrams is fairly simple, complexity was still a problem with the pricing component in the application generator [9]; it's functionality was described in nearly twenty pages of the installation guide.

The complexity resulted from numerous (and arbitrary) pricing schemes in use in industry and from the representation of these schemes in the system. The price of an order depends on several factors, such as the type of the customer (government, educational), the type of payment (regular, cash, etc.), the time of the year (high versus low demand season), whether cost-plus, or discounting applies, whether prior negotiated prices are involved, whether extra charges for the items apply such as taxes, deposits or surcharges, etc. For the purpose of this paper, we consider two examples of pricing schemes presented in [9]:

1. *Regular Pricing:* each product has a base price which can be discounted depending on the number of the units ordered.

2. *Negotiated Pricing:* a customer may have negotiated certain prices and discounts for particular items.

Given this short introduction of our running example, let us now consider some requirements on the design of a construct for expressing collaborative behavior that are illustrated by the example.

First, collaborative behavior should, in general, be reusable with a range of concrete applications, i.e., concrete class graphs. Recall that the design for the pricing collaboration specified in Fig. 2 and 3 is part of the design of an application generator. As such, the design is actually not bound to any particular application. The design specifies only minimal requirements on the concrete applications to which this generic collaboration may be attached. Concerning the structural aspects of the application it states that there should be classes in the application that play the roles of the participants in Fig. 2, and these classes should directly or indirectly be connected according to the connection pattern between the participants in the class diagram of Fig. 2. With regard to the behavioral aspects, the design only mentions some operations expected in the interface of the participants and the control flow among these operations to realize the collaboration.

Given an actual application, the generic design of the pricing component in Fig. 2 and 3 needs to be mapped to the concrete class structure. By mapping we mean assigning the responsibility for playing particular roles in the generic collaboration to particular classes in the concrete application and mapping paths in the generic design into concrete association paths in the concrete class diagram. The collaborative behavior in the collaboration diagram in Fig. 3 will carry over.

Second, even within the same application, it should be possible to reuse the same collaboration multiple times with different objects playing different roles. For instance, given a concrete application, the participant-to-class assignment for the pricing collaboration may differ from one pricing scheme to the other. For illustration, assume a hypothetical application for order processing that has a Product and a Customer class. In this case, the role of the PricerParty would be played by both Product and Customer depending on the pricing scheme. Product has the information needed when the regular pricing scheme is applied, while the Customer is the knowledge expert when the negotiated pricing scheme is applied.

The example considered so far is not special in the requirement to write collaborative components that can be applied to a family of class graphs. In general, reusing collaborative components with a class graph is useful for supporting any kind of polytypic function (algorithm) [10] on data structures. These are functions (algorithms) that apply to a range of data structures. A very simple example would be counting all instances of type `x` that are part of the structure representing a type `y`. Even if we consider operations that apply only to a certain data type, there might be several possible representations of the data type, and it is desirable to have algorithms adaptable to concrete representations.

In the remainder of the paper, we refer to the requirement for reusing collaborations with both a family of class graphs and with several class-to-participant mappings for the same class graph, as the requirement for supporting *structure-generic* components.

Let us now state some requirements on the composition mechanisms for collaborative behavior, using the term composition for both white-box and black-box composition. That is, we intend to support both (a) incrementally evolving collaborative behavior, i.e., being able to define new collaborative behavior by incrementally refining the definitions of existing collaborations (white-box composition), and (b) creating higher-level collaborative behavior by making use of existing collaborations (black-box composition). The main goal in the design of these compositions is to maximize reuse.

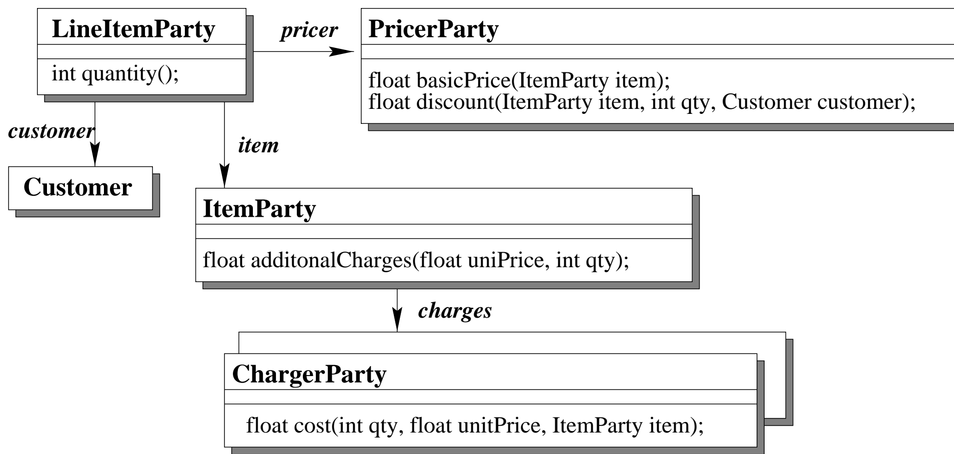

Figure 2: Class Diagram for the Pricing Component

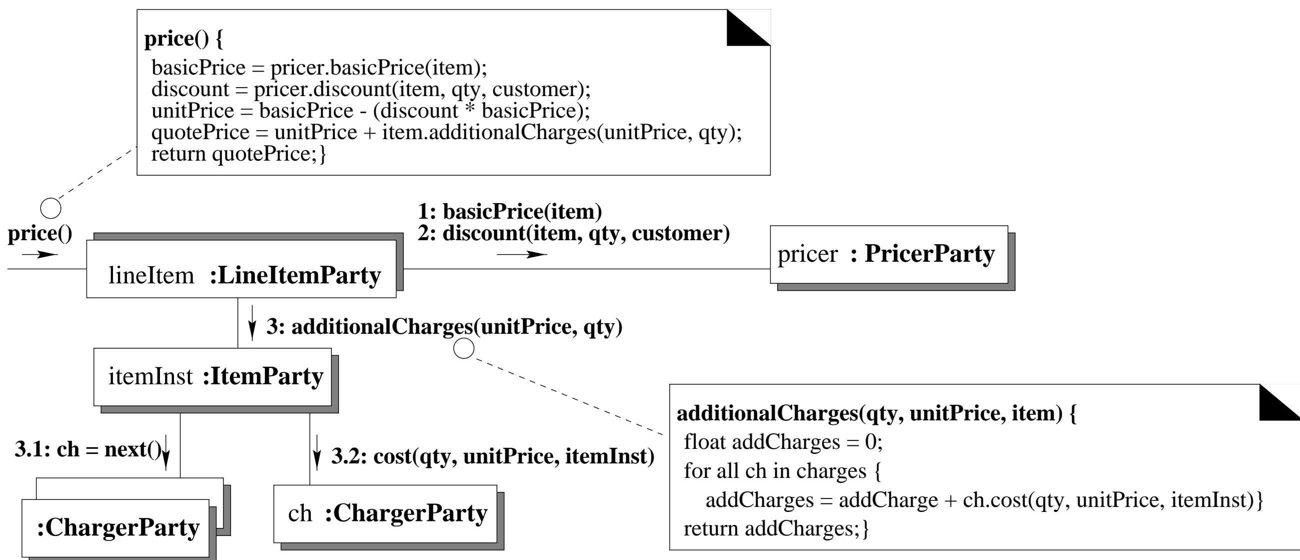

Figure 3: Collaboration Diagram for the Pricing Component

Let us first consider white-box composition. One can envisage several refinements of the pricing functionality considered so far (independent of the pricing schemes in use). Two of them are considered here. A first example, called AgingPricing, is a pricing policy in which certain items get a reduction on their calculated price based on the period of time they have been in stock and on some characteristics of the items, e.g., whether they are of certain brands. Another scenario is to have reduction on certain items for frequent customers that are identified by a frequent customer advantage card. The height of the reduction is determined based on the recorded buying profile of the customer. We call this second pricing policy FrequentCustomerPricing.

Both AgingPricing and FrequentCustomerPricing also embody a collaboration among LineItemParty, ItemParty, PricerParty, ChargerParty, and Customer. For instance, calculating the (reduced) price remains the responsibility of the LineItemParty. This calculation is based on the height of reduction, computing which would be the responsibility of ItemParty and Customer in AgingPricing and FrequentCustomerPricing, respectively. The collab-

orations for AgingPricing and FrequentCustomerPricing represent refinements of Pricing, in the sense that the role of each participant in each of them can be expressed as a refinement of the role of the same participant in Pricing. For instance, calculating the price in the AgingPricing policy is a refinement of the way price is calculated with the standard pricing policy.

The discussion above indicates inheritance-like relationships, now at the level of collaborations between several classes rather than at the level of single classes. These relationships are schematically shown in Fig. 4, with ellipses representing collaborations, and arrows representing refinement relationships between collaborations without any commitment to the composition mechanism that would express these relationships at the implementation level.

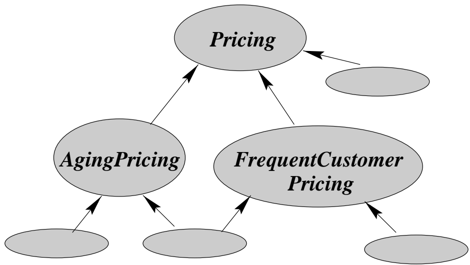

Figure 4: Refinement Collaborations

Several other variations of the basic pricing functionality can be specified. In addition, one can think of further refining AgingPricing and/or FrequentCustomerPricing or combining them into a composed variation, say AgingAndFrequentCustomerPricing, to apply both reductions at the same time. The existence of a variety of other possible variations of the pricing functionality is presented in Fig. 4 by the unnamed ellipses. Concrete applications may not need all available variations of the pricing collaboration. Furthermore, a single application may need several variations of Pricing, e.g., both AgingPricing and FrequentCustomerPricing and/or combinations of variations, e.g., AgingAndFrequentCustomerPricing.

This analogy of the refinement relationships among collaborations and inheritance among classes indicates that it is desirable to support a white-box composition mechanism similar to inheritance for components that would enable to reuse the definition of Pricing in defining the collaborations for AgingPricing and FrequentCustomerPricing. If we consider standard object-oriented mechanisms for behavior variations, we could naturally think of using application frameworks for realizing the refinement relationships between collaborations. The participants in the collaborations would be implemented as classes; roles from are added to these participants by subclassing.

If a framework was used in our example, base classes in the framework would implement the elements of a concrete application and their role in the base Pricing collaboration. For instance, suppose that in an application ConcreteLineItem is the class that will play the LineItemParty role. The contribution of ConcreteLineItem in the Pricing collaboration would then be implemented as a subclass of ConcreteLineItem, say PricingConcreteLineItem. In order to accommodate possible future refinements, the pricing functionality would be implemented as a template method in PricingConcreteLineItem, i.e., it would invoke other methods that are left unimplemented in PricingConcreteLineItem. These abstract methods would then be implemented in subclasses of PricingConcreteLineItem that implement the roles played by ConcreteLineItem in the refinement collaborations, e.g., AgingPricingConcreteLineItem or FrequentCustomerPricingConcreteLineItem.

Holland [8], as well as VanHilst and Notkin [28], investigate several problems with a framework based solution for composing collaborations. The main problem is the implicit commitment to a particular composition structure. For instance, PricingConcreteLineItem must statically bind a concrete implementation of LineItemParty as its parent class (ConcreteLineItem in this case) even though the same Pricing collaboration could equally be reused for other implementations of LineItemParty. As pointed out in [11], reusing the edifice that ties the components together is usually possible only by copying and editing it. Another problem with the framework based composition results from naming issues that might raise when an application combines more than one refinement of the framework (e.g., conflicts may occur in AgingAndFrequentCustomerConcreteLineItem).

In a more general context, several works [18, 20] show that inheritance does not provide an effective composition mechanism when a considerable number of behavior variations exist and can be arranged in several different combinations for creating more complex variations. These works motivate the need for more flexible mechanisms for composing class-defined behavior, using variants of mixin-classes [2]. Mixin-classes also called *abstract subclasses*, define behavior that will eventually inherit from a superclass – the behavior defined in a mixin-class is expressed in terms of a *super* parameter. This parameter, however, is not bound to any concrete superclass when the mixin is defined.

The discussion above indicates that similar techniques are relevant also at the level of whole collaborations. The construct we intend to design for explicitly expressing collaborations should have a *mixin flavor* in

the sense that when defined a collaboration makes as few assumptions as possible about other collaborations it is going to be combined with in an actual application. Furthermore, while we want to support white-box reuse of the collaboration definitions, the composition mechanism should be designed such that, it maintains the "encapsulation" and independence of collaborations when involved in compositions with other components. The aim is to avoid name conflicts and allow simultaneous execution of several collaborations even if these may share a common "parent".

The requirement for loose coupling holds also for the black-box composition. As already mentioned above, is should be possible to create higher-level collaborations that make use of the functionality provided by other collaborations. A simple scenario for illustrating this kind of composition in our running example is the Total collaboration, whose functionality is schematically presented in Fig. 5. Calculating the total of an order involves a simple collaboration between an OrderParty object and the LineItemParty objects contained in it: the total of an order is the sum of the price of each line item contained in it.

Again, the design presented in Fig. 5 is expressed in terms of an abstract class graph. Furthermore, the only assumption it makes about the behavior of the participants is that whatever class is going to play the LineItemParty role in a concrete application, this class is responsible for calculating the price. However, the design does not make any commitment as what concrete pricing scheme, or pricing policy is in use. As a consequence of the loose coupling, the design can be reused with a variety of pricing schemes and policies. By requiring that the mechanism for black-box composition of collaborative components should allow for loose coupling among collaborations, i.e., that one collaboration uses another collaboration without explicitly mentioning it in its implementation, we simply aim at preserving this feature of the design at the implementation level.

In the remainder of the paper, we will refer to the requirements we posed on the omposition mechanisms uniformly as the *decoupled behavioral composition* requirement. After having outlined and justified the requirements in detail, below we briefly summarize their meaning and implications.

- **$\mathbf{R}_1$: structure-generic.**

  Generic specification of the collaboration with respect to the class structure it will be applied to should be enabled. This serves two purposes: (a) allow the same component to be used with a family of class graphs, and (b) allow a collaborative component to be matched against several places in the same class graph, i.e., with different class-to-participant mappings for the same class structure.

- **$\mathbf{R}_2$: decoupled behavioral composition**

  Flexible composition mechanisms are needed to support reusing the definition of existing components to build more complex collaborations. This implies: (a) loose coupling among collaborations in the sense that their definitions do not make explicit commitments to a particular structure of composition, and (b) maintaining the "encapsulation" and independence of collaborations when involved in white-box compositions with other components. The aim is to facilitate reusing the same components with several compositions.

3 Adaptive Plug and Play Components

An *Adaptive Plug and Play Component* is a language construct for expressing collaborative behavior that involves a set of participants (classes) in an object-oriented application domain. It extends the standard object-oriented model, while being orthogonal to it, and is designed to satisfy the requirements ($\mathbf{R}_1$ and $\mathbf{R}_2$) listed above:

- In achieving structural genericity we borrow and further develop the *Adaptive Programming* [16, 17] technology for decoupling structure from behavior.
- In achieving decoupled behavioral composition we partly reuse ideas from RONDO [20] and partly apply the technique of programming to interfaces.

Because of its importance for understanding APPCs, in the following key ideas behind *Adaptive Programming* will be first briefly outlined.

3.1 Adaptive Programming

One of the key elements of adaptive programming is the concept of a *traversal strategy graph*, also called *traversal strategy*, or *strategy* for short [22, 17]. The emergence of traversal strategies is the result of the Law of Demeter (LoD) [15] which says that a method should only talk to its "direct friends": the argument objects, the part objects and the newly created objects. If you do not follow LoD, you get methods that contain too many details about the object structure and are for this reason brittle with regard to structural changes. If you follow LoD, the situation is better but you get many small methods that refer to the details of the object structure. To solve this trade-off and build more maintainable systems, *Adaptive Programming* [16] introduces *traversal strategy graphs* which are use-case based abstractions of class graphs.

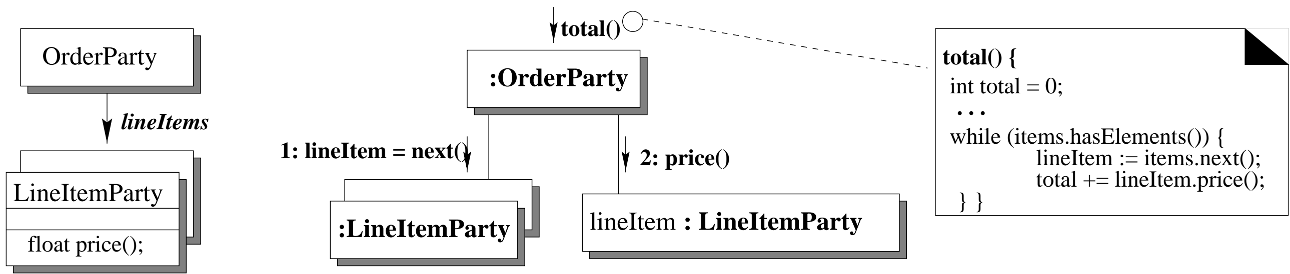

Figure 5: Partial Class Diagram and Collaboration Diagram for the Total Component

Strategies have their origin in automata theory applied to structural architectures. Strategy graphs consist of nodes and edges. They describe the overall topology of a group of collaborating classes. The nodes are the cornerstones of the topology and the edges describe that certain connections must exist. The details of those connections at the class graph level are left unspecified. The edges may also have constraints assigned. An edge `(A,B)` without constraint means `A.any*.B` and an edge `(A,B)` with a constraint `C` means `a.(any, satisfying C)*.B`. A regular expression like `A.any*.B` for a class graph means to take all paths from class `A` to class `B` and to view them as a group of collaborating classes.

Thus, trategies add a third layer of abstraction to the usual two level object model of object graphs and class graphs. A third layer is needed because class graphs serve several purposes, and for each purpose only a part of the class graph is really important; the rest is just noise. With strategies we can focus on the important part and filter out the noise. As mentioned earlier, it has been observed that following the control flow in an object-oriented program is like reading a road map with a soda straw [14]. Strategies help to solve this problem by expressing an important part of the control flow, namely the object navigation part, at a high level of abstraction in a localized manner and not spread through many classes.

The compilation problem for strategies is as follows: given a strategy S and a class graph G, generate a program in some object-oriented language which will perform the correct traversals (specified by S) in the object graphs defined by G. A strategy graph defines a "standard" traversal (depth-first, ordering specified by class graph) of object graphs. While the details of an efficient algorithm are non-trivial [22, 17, 21] the basic idea is to adapt the intersection algorithm for non-deterministic finite automata (NDFA). Both the strategy S and the class graph G can be viewed as NDFAs for which we want to compute the intersection resulting in a third NDFA TG, called a traversal graph. Conceptually, to traverse an object graph O, we need to compute the intersection of an NDFA corresponding to O and the NDFA TG. In the implementation we do something similar to the simulation of an NDFA to avoid the exponential state explosion problem of expansion into a DFA.

How can we write programs using strategies? First, it should be noted that one of the key features of adaptive programming style is the separation of structure from behavior specification. The structural aspect of an application is specified by a textual form of the application's UML class diagram. The specification of the behavior consists of a set of traversal strategies and a set of "tasks" to be performed on the nodes encountered during the traversals. The behavior specified in this way is less brittle with respect to changes in the structure of the application than behavior written in a standard object-oriented style. This is because the behavior specification is based on strategy graphs, rather than detailed class graphs. For this reason adaptive programs are also called *structure-shy*.

Let us illustrate this brief introduction with the simple order processing application for hardware products written in *Demeter/Java*[1] [25, 16] style, presented in Fig. 6. Appl.cd is the textual representation of the application's (simplified) class diagram, a graphical representation of which is also drawn in Fig. 7 (a) for clarity reasons. There is one "equation" in Appl.cd for each class in Fig. 6. The right hand side of the equation for a class C lists all adjacent classes of C in the class diagram, C_a, along with the name of the link connecting C with C_a. Thus, Quote = <prod> HWProduct ... has to be read as, "in the class diagram, there is an association named prod connecting classes Quote and HWProduct".

In the second part of Fig. 6, Appl.beh lays down the behavior of the application. Note that there are no attribute (instance variable) declarations in Appl.beh,

[1] Demeter/Java is the embodiment of *Adaptive Programming* in Java.

Appl.cd
```
Quote = <prod> HWProduct <quantity> int <cust> Customer.
HWProduct = <price> float <salePrice> float <taxes> ListOf(Tax) <discountTable> DiscountTable.
Tax = <percent> float <taxKind> String.
Customer = <name> String.
```

Appl.beh
```
class HWProduct {
  float salePrice() {return salePrice;};
  float saleDiscount(int qty, Customer c) {return 0; };
  float regPrice() {return price;};
  float regDiscount(int qty, Customer c) {return discountTable.lookUp(qty);};
  Vector allTaxes() {
     Vector all;
     to Tax {
        init {all = new Vector(); }
        at Tax {all.addElement((Object)taxKind);}
        return {return all;} } }
}
class Tax {
  float taxCharge(int qty, float unitPrice, HWProduct p) {return unitPrice * percent/100};}
class Quote {
   int quantity() {return quantity; } }
class Customer {
   float negProdPrice(HWProduct p) { ...  };
   float negProdDiscount(HWProduct p, int qty, Customer c) { ...  };
}
class Main {
  static public void main(String args[]) throws Exception {
   Quote aQuote = Quote.parse(System.in);
   float r = aQuote.allTaxes();
   System.out.println(``Taxes'' + r);
}
```

Figure 6: Application Specification

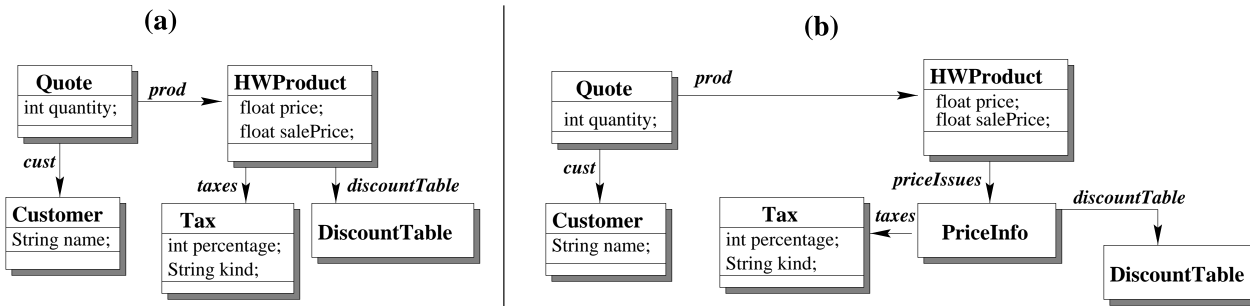

Figure 7: Application Class Diagram

since the specification of the structure is taken out of the classes and defined separately in Appl.cd. Classes in Appl.beh look pretty much like classes in a Java program. The only exception is the method allTaxes() in HWProduct, which returns a collection of all different kinds of taxes that apply for a HWProduct object. allTaxes is an example of the so-called *adaptive methods* in Demeter/Java [16]. The method allTaxes involves traversing part of the object graph(s) that are defined by the class graph represented by Appl.cd. This traversing is specified in a succinct way in Appl.beh by means of the strategy specification "to Tax".

This definition has to be read as "starting from a HWProduct object, follow the links that lead to all its

Tax subparts, and once at a Tax object, add the string describing the kind of the Tax object to the vector all". The definition includes no further details about the intermediate nodes involved and the behavior performed at the intermediate nodes, since they only contribute traversal behavior passing the responsibility down to Tax. As the result of this succinct specification, the implementation of allTaxes can be reused without modifications for the slightly modified class structure presented in Fig 7 (b). This modification is representative for the category of structural changes that might be performed to an application as part of a perfective maintenance process. This illustrates why adaptive programs are called structure-shy.

3.2 Defining Structure-Generic Collaborative Behavior

So far, we explained structure-shy behavior and next we discuss structure-generic behavior. Being structure-shy is only a special case of being structure-generic. Obviously, assumptions about the shape of the application's structure are made in a more robust manner in adaptive programs as compared to standard object-oriented programs, due to using strategy graphs rather than embodying the exact shape of the class graph in the behavior implementation. However, the strategies are still written to a concrete class graph. This damages their adaptability.

For instance, the program in Appl.beh in Fig. 6, cannot be reused with a class graph that names classes in a different way. In general, if an adaptive program contains n strategies s_1, s_2, ..., s_n, written to the class graph C_g, updating of C_g to C'_g may require the updating of n strategies to make them select the right paths in C'_g. This might reveal a lot of representation information.

To remedy this situation the APPC construct uses the traversal strategy graph technology at a higher level of abstraction. An APPC is written to a *interface class graph* rather than to a concrete class graph. An APPC has two parts: a *interface class graph* and a *behavior definition* part. This is reflected in the syntactic structure of APPCs in the grammar of Fig. 8[1]. The interface class graph of an APPC (ICG for short), declares the "type" of class graphs the APPC can be used with. This type declaration consists itself of two sub-parts.

The Structural-Interface declares (a) the participants in the collaboration, and (b) the pattern of their relationships, pretty much the same way the structure of an application is specified in a structure-shy adaptive program. In contrast to the class graph specification of a structure-shy adaptive program whose nodes and edges are concrete classes and link names in an application, the structural interface of an APPC is declared by using formal names for both nodes and edges, called *class-valued* and *link-valued* variables, respectively.

The Behavioral-Interface part of a ICG declares the signatures of operations that are required to be provided by the participants in the collaboration. These operations are used (invoked) in the definition of the collaboration (*Behavior-Definition* part of the APPC). The behavioral interface is organized as a collection of interface declarations, in general one interface for each participant. Note that as with the structural interface, method names in the behavioral interface, are not concrete method names of a particular application – they are formal names, called *method-valued variables.*

Summarizing, the ICG of an APPC can be thought of as the textual specification of the UML class diagram of an abstract application.

The Behavior-Definition part of an APPC resembles an object-oriented program. The main part of it consists of behavior definitions (Participant-Behavioral-Entry in Fig. 8) for all participants that play an active role in the collaboration, in general one for each of them. As it can be noticed from the grammar in Fig. 8, a participant behavior definition resembles a class in an object-oriented program, in that it contains attribute and method definitions. In general, a participant behavior definition is expected to be much more "lightweight" than a class, in the sense that it defines fewer behavior than a class generally does. This is because a participant behavioral entry defines only one role among several possible played by a class. Participant behavior is written to the ICG: methods in a participant definition may invoke messages required in the interface of other participants along links in the ICG.

As with adaptive programming, participant roles may also be defined in terms of traversal strategies graphs and behaviors to be performed at their nodes. Traversal strategies used in an APPC are, however, specified to the interface class graph of the APPC rather than to a concrete class graph. Furthermore, besides the participant behavioral entries, the behavior definition part of an APPC may also define some auxiliary objects that are used in the definitions of the participants (called WorkSpace in Fig. 8).

Similar to a Java program, an APPC also has a distinguished method that serves as an "entry point" for invoking the collaboration defined by the APPC. An APPC is a slice of an adaptive program, implementing only one coherent design segment involving inter-

[1] This is only a partial grammar for the APPCs. A full-fledged grammar is left out of the scope of this paper and can be retrieved from [3]

```
APPCDefinition                        =   Interface-Class-Graph Behavior-Definition
Interface-Class-Graph                 =   Structural-Interface Behavioral-Interface
Structural-Interface                  =   Class-Graph-Specification
Behavioral-Interface                  =   {Participant-Type}*
Participant-Type                      =   Participant-Name { {Method-Signature}* }
Method-Signature                      =   Return-Type Method-Name({Argument-Type Argument }*);
Behavior-Definition                   =   WorkSpace {Participant-Behavioral-Entry}*
WorkSpace                             =   {Class-Definition}*
Participant-Behavioral-Entry          =   {Variable-Definition}* {Method-Definition}*
Method-Definition                     =   Plain-Method-Definition | Inlined-Adaptive-Method-Definition
Inlined-Adaptive-Method-Definition    =   {Variable-Declaration}* Traversal-Behavior-Specification
Traversal-Behavior-Specification      =   [Init-Method] {At-Node-Behavior-Definition}+ [Return-Method]
```

Figure 8: Syntactic Structure of APPCs

object collaborations. APPCs serve as the linguistic counterpart of design level collaboration diagrams for high-level, system operations extracted from use cases. The "main" method of an APPC then corresponds to the method defined by the collaboration diagram. It is part of the definition of one of the participants and lays down the overall flow of control of the collaboration, by eventually initializing the workspace, invoking methods defined in the same or other participant behavioral entries, and returning a value.

For illustration, an APPC that models the pricing component introduced in section 2 is presented in Fig. 9. Comparing Fig. 9 with Fig. 2 and Fig. 3, it becomes clear that design artifacts are mapped to the elements of the APPC in a straightforward way. The interface class graph of the APPC codifies the partial class diagram in Fig. 2. It states that there are five participants in the collaboration: LineItemParty, ItemParty, ChargerParty, PricerParty and Customer. Also, it is assumed that there is a way to get from e.g., LineItemParty to ItemParty denoted by <item> in the scope of the APPC, etc. On the other side, the behavior definition part is a codification of the collaboration diagram in Fig. 3.

The only active participant in the APPC in Fig. 9 is LineItemParty, whose behavioral entry contains the main entry of the collaboration, price(). The other participants contribute passively by providing services to the LineItemParty role. LineItemParty is a client of the expected interface of PricerParty: price() invokes basicPrice() and discount() on its assumed pricer subpart. These operations are declared in the expected interface for PricerParty, which means that in any concrete application to work with the APPC, whatever class will be assigned to play the PricerParty role must provide implementations for these operations.

In addition to the main entry, an auxiliary method, additionalCharges, is defined to calculate additional charges to be added to the computed price. Based on the collaboration diagram in Fig. 3, calculating charges involves the direct participation of ChargerParty with its cost functionality; besides that, calculating the additional charges, involves the indirect participation of ItemParty. ItemParty only contributes traversal behavior by passing the responsibility for calculating additionalCharges to ChargerParty. This flow of control is expressed within the definition of additionalCharges by means of a traversal strategy.

The computation of additionalCharges is itself a (mini) collaboration which is nested within Pricing. This collaboration has a slightly different flavor: it happens during a traversal of some part of the object structure. We call such collaborations, *traversal driven*. They are modeled after a generalized form of adaptive methods of Demeter/Java. While in this particular example, the traversal driven collaboration happens to be inlined in a non-traversal driven collaboration, in general they may exist as first-class APPCs. The participants in a traversal driven APPC and the partial control flow among them are specified by means of a strategy graph – "from LineItemParty via itemInst: ItemParty to ChargerParty".

In addition, a set of tasks to be performed at each node of the strategy graph are specified. Only important tasks are mentioned, i.e., those that contribute non-traversal behavior to the overall task (a single task to be performed when at ChargerParty, in this case). Given a concrete class graph and the strategy, the code for performing the traversal and for executing the specified tasks at each node will be generated. This makes the definition of additionalCharges adaptive to several different structures of concrete paths connecting whatever classes will play the LineItemParty, ItemParty and ChargerParty roles in a concrete application.

As indicated both by the grammar for Inlined-Adaptive-Methods in Fig. 8 and the definition of additionalCharges in Fig. 9, the specification of traversal driven collaborations differs in two ways from the specification of

```
APPC Pricing {

  Interface Class Graph:

    // structural interface

    LineItemParty = <item> ItemParty <pricer> PricerParty <customer> Customer
    ItemParty = <charges> ListOf(ChargerParty)

    // behavioral interface

    LineItemParty { int quantity();}
    PricerParty {
          float basicPrice(ItemParty item);
          float discount(ItemParty item, int qty, Customer customer); }
    ChargerParty { float cost(int qty, float unitP, ItemParty item);}

  Behavior Definition:
      LineItemParty {
          main-entry float price(){
              float basicPrice, unitPrice;
              int discount, qty;
              qty = this.quantity();
              basicPrice = pricer.basicPrice(item);
              discount = pricer.discount(item, qty, customer);
              unitPrice = basicPrice - (discount * basicPrice);
              return (unitPrice + additionalCharges(unitPrice, qty)); }

          private float additionalCharges(float unitP, int qty) {
              float total;
              from LineItemParty via itemInst:  ItemParty to ChargerParty {
                  init {total = 0; }
                  at ChargerParty {total += cost(qty, unitP, itemInst); } }
                  return {total;} }
  } }
```

Figure 9: The Pricing APPC

"norma;" collaborations. First, they have distinguished methods for initializing their working space (init) as well as for returning their result (return). Second, traversal driven collaborations do not have a main entry method. Both these differences are due to the fact that these collaborations are driven by the (generated) traversal code – this is their main entry. Recall that initializing the workspace and returning a value was inlined in the distinguished main entry of the non-traversal driven collaborations. With the traversal driven collaborations, however, the main entry is not explicit. That is why we need to put the code for initializing the workspace and for returning the result of the collaboration in distinguished methods, that are known to the generated traversal code and are called in the appropriate place during the traversal.

Besides control flow, a strategy graph may also specify how data should be propagated during a traversal. An instance name associated with a strategy node is an indication that during the traversal of the object graph, instances of that node should be passed further down as parameters of the traversal method. In the strategy used in additionalCharges in Fig. 9, itemInst associated with ItemParty indicates that each time the traversal reaches an object of type ItemParty, this object will be passed down the traversal of the remaining object structure still to be traversed. In this way, the implementation of the task to be performed at a ChargerParty, cp, can access the ItemParty instance, itemInst, containing cp.

3.3 Instantiating Collaborative Behavior

An APPC specifies an abstract collaboration. In order to turn it into executable code, a concrete application must be provided that implements the interface class graph of the APPC. Binding an application, appl, to the ICG of an APPC, appc, is also referred to as instantiating the APPC with the application and denoted by the operation ::+.

The first issue in instantiating an APPC with an application is to map ingredients in the APPC's interface class graph, i.e., (a) participant names, (b) their assumed relations, and (c) assumed operation names, to

corresponding elements in the application, i.e., classes, paths, and operation names, respectively. While (a) and (c) are realized by simple name maps, (b) requires, in the general case, explicit mapping of edges in the ICG to strategies over the CCG.

Given a participant-to-class name map, N, and assuming that the CCG is a valid implementation of ICG (see the ICG-CCG conformance definition below), there is always a default mapping of edges in ICG to paths in CCG. For any edge $e = (v_1, v_2)$ in ICG, the default mapping is the set of paths in CCG from $N(v_1)$ to $N(v_2)$, denoted as *Paths[CCG](N(*v_1*), N(*v_2*))*. Based on the ICG-CCG conformance definition below, this path set is not empty. There might be, however, cases where the default mapping of ICG edges to CCG paths does not suffice. For instance, in dense class graphs we might want to reduce the cardinality of the default path set, or, in general, we may want some fancy mappings other than those provided per default. In these cases, the programmer can explicitly map ICG edges to concrete traversal strategies over the CCG.

To illustrate the mapping process, in Fig. 10, we apply the Pricing APPC to the concrete application that was presented in Fig. 6 (denoted by HWAppl in Fig. 10), for modeling two pricing schemes, *Regular* and *Negotiated*, discussed in Sec. 2. The regular scheme implies that the role of PricerParty is played by the class that models the products being ordered, HWProduct in our case. On the other side, applying the negotiated scheme means that the PricerParty role is played by the class that models customers, Customer in our case.

```
HWAppl::+ {float regularPrice() = Pricing with {
        LineItemParty = Quote;
        PriceParty = HWProduct
               {basicPrice = regPrice;
               discount = regDiscount};
        ItemParty = HWProduct;
        ChargerParty = Tax {cost = taxCharge};} }

HWAppl::+ {float negotiatedPrice() = Pricing with {
        LineItemParty = Quote;
        PriceParty = Customer
               {basicPrice = negProdPrice;
               discount = negProdDiscount};
        ItemParty = HWProduct;
        ChargerParty = Tax {cost = taxCharge};}}
```

Figure 10: Instantiating the Pricing Collaboration

Consequently, the APPC needs to be applied twice, each time with a different mapping of roles to classes and method-valued variables to actual operation names. Note that in both cases, mapping participant to class names is sufficient for generating paths in the application that correspond to edges in the APPC. No actual strategies are required for realizing this mapping; there is a single mapping from relations between the participants in the generic collaboration to relationships between the corresponding classes in the application.

Once a mapping of an ICG to a CCG is given, it should be checked whether the CCG does actually implement (we also say conform to, or specialize) the ICG. Conformance checking includes (a) checking whether the relation pattern between the participants specified in ICG can be matched by the relations between corresponding classes in CCG, and (b) checking signature compatibility among the assumed operations in the APPC and the corresponding implementation methods in the application. For (a), we require the following to hold[1].

Definition: CCG-ICG conformance

Given node-labeled graphs $G_1=(V_1,E_1)$ *and* $G_2=(V_2,E_2)$ *and a name map* $N\colon V_2 \rightarrow V_1$, G_1 *conforms to* $G_2 \iff \forall\ e = (e_s,\ e_t) \in E_1,\ \exists\ p \in$ *Paths[G2](N(*e_s*), N(*e_t*))*

The intuition is that each edge of the ICG defines a path in the class graph. This means that the interface "matches" the class graph in terms of paths; we can embed the paths in the ICG in the paths in the class graph. We use the conformance concept in two stages. At the adaptive level we check that the ICG conforms to each strategy (graph) specified in the behavior definition part of an APPC. At the instantiation level we check that the concrete class graph conforms to the interface class graph.

Assuming that an application, appl, conforms to the interface of an APPC, appc, conceptually, the result of instantiating appc with appl is an enhanced application gained by adding the participant behavioral definitions of appc onto the corresponding classes in appl. The implementation of the added methods is automatically generated from the behavior definition part of the APPC by (a) replacing participant names with the class names they are mapped to, (b) replacing edges from the APPC with the concrete paths they are expanded by in the class structure, and (c) replacing formal method names in the APPC code with the application's method names they are mapped to. In addition, traversal code as well as code for calling role implementations during traversals will be generated, if there are traversal strategy specifications involved in the implementation of the APPC.

For giving the reader an intuition of what an APPC instantiation produces, pseudo-generated code for the instantiations in Fig. 10 is given in Fig. 11. The pseudo-generated code indicates that the generator engine would map collaborations down to objects; there is one class

[1]The condition can be checked in polynomial time. Further details of this checking are out of the scope of this paper.

generated for each pricing scheme instantiated in Fig. 10, RegularPrice and NegotiatedPrice. Furthermore, a method for each scheme, regularPrice(), respectively negotiatedPrice() is added onto the application class Quote. These methods simply create a "collaboration object" and delegate the responsibility to it. They are inserted in Quote because Quote is mapped to LineItemParty participant where the main method of the collaboration was defined. The generated methods can be invoked by other collaborations that build on top of Pricing, as it will be illustrated in the following section.

```
class Quote {
  // ...  as before ...
  public float regularPrice() {
     RegularPrice rp = new RegularPrice(qty, this);
     return rp.price(); }
  public float negotiatedPrice() { ...  }
  }

class RegularPrice {
  public RegularPrice(int quantity, Quote hst) {
     host = hst;
     qty = quantity;
     prod = pricer = host.getProduct();
     cust = host.getCustomer(); }
  public float price() {
     float basicPrice, quotePrice, unitPrice;
     basicPrice = pricer.regPrice();
     int disc = pricer.regDiscount(prod, qty, cust);
     unitPrice = basicPrice - (disc * basicPrice);
     return (unitPrice +
              additionalCharges(unitPrice, qty);
  }
  private float
       additionalCharges(float unitPrice, int qty) {
     float total = 0;
     Enumeration taxes = pricer.getTaxes();
     while (taxes.hasElements()) {
        Tax tax = taxes.nextElement();
        total += tax.taxCharge(unitPrice, qty); }
   }
  private Quote host;
  private int qty;
  private HWProduct pricer;
  private HWProduct prod;
  private Cust customer;
}

class NegotiatedPrice { ...  }
```

Figure 11: Pseudo-Code Generated for Regular Pricing

This implementation is rather naive. First, it assumes that the application is available in source code and that recompilation can be performed at any time, both unrealistic and undesirable assumptions in many real situations. Second, putting collaborations in a single class in the generated code is against the object-oriented spirit. However, while this is the approach currently taken, it should be noticed that the way APPCs are mapped down to object-oriented code is an implementation detail that has nothing to do with the concept. We are already working on a better generator engine in Java organizing the collaborations in separate Java packages.

Apart of the few implementation details discussed above, at the conceptual level, the sample instantiations above demonstrates how the APPC construct allows us to write collaborative software that (a) adapts itself to the concrete shape of particular applications, and that (b) can be reused with different participant-to-class mappings over the same application. In other words, it illustrates that APPCs satisfy R_1.

3.4 Composing Collaborative Behavior

As indicated in Sec. 2, we give APPCs a mixin [2] flavor, in order to allow for more flexible white-box composition. A mixin-flavored APPC defines collaborative behavior in terms of a *super* component which is, however, not bound at component definition time. The super-component parameter will be bound later, at component composition time.

Consider, for instance, how both special pricing policies, AgingPricing and FrequentCustomerPricing discussed in Sec. 2, can be defined as deltas to the Pricing APPC. Both policies share the same refinement pattern: first the price is calculated and then a reduction on that price is computed based on a certain criteria that is specific for each policy. This common pattern is factored out in the SpecialPricing APPC in Fig. 12 which is specified to be a modification of Pricing. Both AgingPricing and FrequentCustomerPricing further modify SpecialPricing, each implementing reducedPrice in a specific way.

The *modifies* relationship between APPCs resembles the inheritance relationship between classes. A modification APPC inherits the interface and behavior definition of the APPC it modifies, while refining the behavior definition by making super calls. So far, the super parameter of a modification APPC resembles the super parameter of Smalltalk/Java subclasses. There is, however, an important difference: in contrast to inheritance, the *modifies* relationship does not imply any composition structure between APPC.

Declaring that SpecialPricing *modifies* Pricing does not mean that the super parameter in the definition of SpecialPricing is fixed to Pricing. The super parameter will be rather bound at composition time. That is, the *modifies* relation does not compose the APPCs. The composition is a separate operation that is explicitly applied to the APPCs after they have been defined. In the following, this operation is denoted by +. It is non-commutative, since the order of applying the operation establishes the bindings of the super parameters of the APPCs involved. The super parameter of the compo-

```
APPC SpecialPricing modifies Pricing {
  Behavior-Definition:
    LineItemParty {
      public float price() {
         float calcPrice = super.price();
         return reducedPrice(calcPrice); }
      protected float reducedPrice(float calcPrice); }
}
APPC AgingPricing modifies SpecialPricing {
  Interface-Class-Graph:  //more behavioral interface
    ItemParty {Time stockTime();
               Time stockTimeLimit(); }
   Behavior Definition:
    LineItemParty {
      protected float reducedPrice(float calcPrice) {
        float newPrice = calcPrice;
        if (item.stockTime()>item.stockTimeLimit()) {
          newPrice = newPrice - (newPrice * 0.1); }
        return newPrice; } }
}
APPC FrequentCustomerPricing modifies SpecialPricing {
  Interface-Class-Graph:  //more behavioral interface
    Customer {boolean frequent();
        BuyHistory getBuyHistory();}
   Behavior Definition:
    LineItemParty {
      protected float reducedPrice(float calcPrice) {
        float newPrice = calcPrice;
        if (customer.frequent()) {
          BuyHistory hist = customer.getBuyHistory();
          float freqRed = item.frequentReduct(hist);
          newPrice = newPrice * freqRed; }
        return newPrice; } }

    ItemParty {
      float frequentReduct(BuyHistory hist) {
         // given hist calculate reduction } }
  }
```

Figure 12: Refining the Pricing Collaboration

nent on the left hand side of the operation is bound to the component on the right hand side. Dispite this difference in the semantics, we have preferred to use the Java/Smalltalk keyword, super, in the definition of modification APPCs, for the intuition related to it.

For illustration, two different compositions of the pricing APPCs from Fig. 12, AgingPolicy and AgingAndFrequentCustomerPolicy are given in Fig. 13. Both compositions contain an AgingDelta = AgingPricing + SpecialPricing. However, the meaning of the super-call in the SpecialPricing part of AgingDelta is different in each case: in AgingPolicy it denotes Pricing, while in AgingAndFrequentCustomerPolicy it denotes the SpecialPricing part of FrequentCustomerDelta. With standard inheritance, the meaning of super in SpecialPricing would have been fixed to Pricing at definition time. With composition time binding, we are given more flexibility in reusing APPCs with different compositions.

This composition time binding of super has the flavor of super binding with the mixin-based inheritance. However, the semantics of composing mixin-flavored APPCs differs from that of composing mixins, as well. While not binding their super parameter at definition time, mixin-flavored APPCs declare their expected specialization interface at that time, via the *modifies* relation, a feature that mixins do not have. This declaration makes it possible to write the collaboration in the refinement APPC to the interface of the base APPC (and possible extensions of it). More importantly, the declaration of the specialization interface controls the scope of the definitions from individual APPCs within a composition, as outlined in the rules below. Let A be an APPC and A_1 and A_2 be modifications of it. Let $A_c = A_1 + A_2 + A$. The *modifies* relation imposes the following:

1. Let m be a method implemented in A, A_1, and A_2. The implementations of m in both A_1 and A_2 refine the implementation in A (by making super calls). The definition of m in A_c consists then of the chain of definitions in A_1, A_2, and A, i.e., when m is invoked all three definitions are executed in the order mentioned above. This is because m is in the specialization interface of A, the common parent component for both A_1 and A_2. Thus, it is natural to require that both refinements are valid and should be executed.

2. Let m be implemented in A_1 and A_2, but not in A. Since A_1 and A_2 assume the specialization interface of A and not that of each other, and since A does not have m in its specialization interface, both definitions of m in A_1 and A_2 should be invisible to the other component. A_c has two definitions of m and the invocation of m on A_c returns, in general, a wrapper object containing both results.

To illustrate the meaning of these rules and their role in maintaining the encapsulation of individual APPCs involved in a white-box composition, assume A_c = AgingAndFrequentCustomerPolicy, i.e., A_1 = AgingDelta, A_2 = FrequentCustomerDelta, and A = Pricing. Assume that after the application has been enhanced with the AgingAndFrequentCustomerPolicy collaboration price() gets invoked on a LineItemParty object. the chain of method executions resulting from this invocation will be as follows.

Based on rule 1, all three implementations of price get successively executed since price is in the specialization interface of Pricing and is refined in both AgingDelta and FrequentCustomerDelta. The implementation of price in AgingDelta will be executed first. The super call in AgingDelta::price() invokes FrequentCustomerDelta::price. The latter will invoke the implemen-

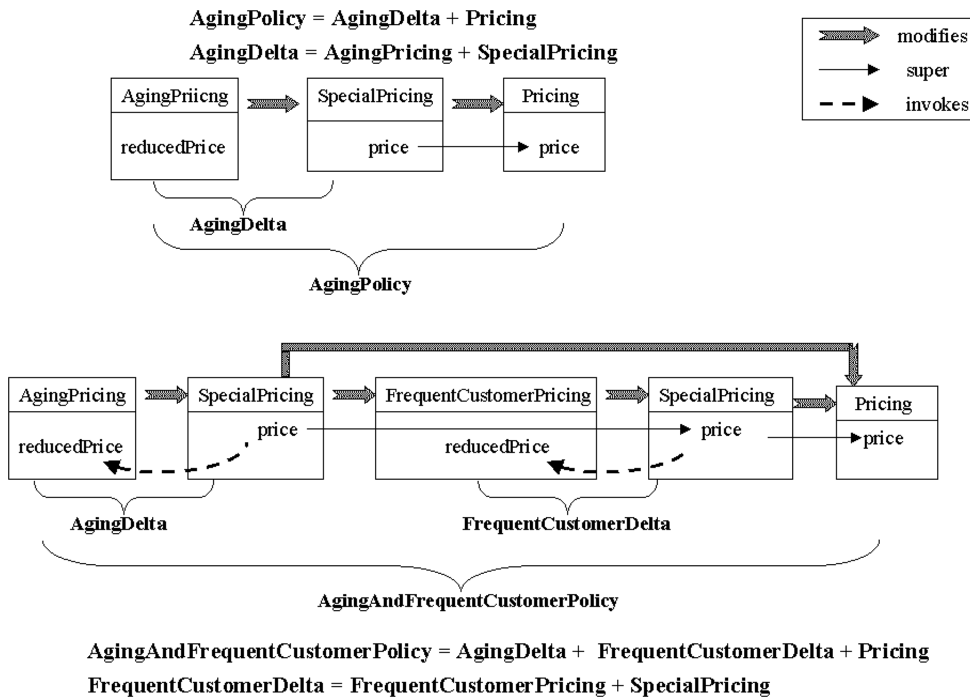

Figure 13: White-Box Composition of APPCs

tation of price in Pricing. After Pricing::price() returns, FrequentCustomerDelta::price() invokes reducedPrice.

Although, there are two implementations of reducedPrice in the definition of the receiver, the implementation of FrequentCustomerDelta has visibility only for its own definition (rule 2), i.e., only the frequent customer reduction will be executed. After the execution of FrequentCustomerDelta::price() returns, the execution of AgingDelta::price() resumes by executing its own implementation of reducedPrice. Although present within the definition of the same object, the definitions of reducedPrice do not collide due to their well-defined scopes established by the *modifies* relationships between AgingDelta, FrequentCustomerDelta, and Pricing. The same definitions would collide in a framework-based implementation, since there are no scoping boundaries between the definitions of different classes involved in the definition of an object.

Thus, the white-box composition supported by APPCs satisfies the R_2 requirement we posed in Sec. 2. Composition-time binding of super keeps the coupling of APPCs loose: there are no commitments to a particular composition structure in the implementation of a modification APPC. This is in contrast to the framework based composition we discussed in Sec. 2. While borrowing this feature from mixins, APPCs are more disciplined with regard to scope issues, due to their definition-time declarations of the modification relationships. This maintains the encapsulation of the APPCs when they get involved in a white-box composition and avoids name conflicts problems we indicated for the framework based approach.

Let us now consider how APPCs are composed into higher-level collaborations in a black-box manner without being strongly coupled to each other. Loosely coupled black-box composition is automatically supported in the proposed design. This is because APPCs define collaborative behavior to an interface, rather than to a concrete class graph. As already discussed, the behavior definition part of an APPC uses operations declared in the behavioral part of its interface without committing to a particular implementation of these operations. The actual implementation, which is bound to the interface operations at APPC instantiation time, may be one generated by instantiating another APPC. It is at this point that the APPC that uses the operation gets composed with the APPC that generates the implementation; the composition is uncoupled since the implementation of the client APPC does not explicitly mention the supplier APPC.

Consider for illustration the Total APPC in Fig. 14 which is defined as a collaboration between two partic-

ipants: an OrderParty and a set of LineItemParty contained in it. The collaboration happens during traversing from an OrderParty to all LineItemParty objects contained in it (collaboration driven APPC). Once at a LineItemParty, the result of invoking price is added to the total calculated so far.

```
APPC Total {
  Interface-Class-Graph:
    OrderParty = <customer> Customer <lineItems>
                        SetOf(LineItemParty)
    LineItemParty { float price(); }
  Behavior-Definition:
      float total;
      from OrderParty to LineItemParty {
          init {total = 0 };
          at LineItemParty {total += price();}
          return { total }; } }
```

Figure 14: Defining the Total Collaboration

Total merely assumes that LineItemParty provides an price operation without committing to any particular implementation of it. Even the name of the actual method that eventually will serve as the implementation of price in whatever class will play the LineItemParty role in a concrete application may be different. Given the definition of Total in Fig. 14, we can now create collaborations for computing the total regular, negotiated, aging, or frequent customer price, by simply "plugging in" the method generated by the respective component as the implementation for price in the interface of Total. For illustration, instantiating the Total APPC for a regular pricing scheme is given in Fig. 15. We assume that the example application we have used so far (Fig. 6) has an additional class called Order for modeling orders of hardware products. In Fig. 15, the regularPrice method which was added to Quote by the instantiation of Pricing in Fig. 10 is bound to the operation price assumed in the interface of LineItemParty. In this way, the example illustrates how APPCs can serve as building blocks for higher-level collaborations.

```
HWAppl ::+ {float totalReg = Total with {
    OrderParty = Order;
    LineItemParty = Quote {price = regularPrice}; } }
```

Figure 15: Instantiating Total

4 Implementation Issues

So far, for the most part, APPCs are simulated in Demeter/Java; a code generator translates high-level descriptions of collaborative behavior from APPCs into standard object-oriented implementations based on visitor objects [5]. In abstract terms, the workings of the generator for the case where no white-box compositions of APPCs are involved is described by the following steps:

- Create a visitor class for the component(s) being instantiated. Different visitor classes will be created for different instantiations of the same component with different mappings of participants to concrete classes. For instance, there will be different visitor classes created for different pricing schemes of the running example.
- If link-valued paths are used in the definition of a component, replace them with the corresponding concrete paths in the graph resulting from applying the concrete strategies to the concrete class structure. Generate the needed code to traverse the paths in all nodes involved in the paths.
- Generate code in the application class structure for the main entry method of the APPCs. This is responsible for initiating the collaboration by creating and passing the responsibility to the appropriate visitors (recall negotiatedPrice and regularPrice generated in Quote in the sample generated code in Fig. 11). Furthermore, code for traversing the object structure, and delegating the responsibility to the created visitors during the traversal is also generated in the application class structure for traversal driven APPCs.

As far as the above functionality is concerned, the existing generator technology of *Demeter/Java* [25, 16] can be reused with some small modifications to arrange for the fact that APPCs are more generic than *Demeter/Java* software (visitors) as well as for syntactic modifications associated with the introduction of APPCs. As mentioned in conjunction with the presentation of strategies, polynomial compilation algorithms exist for *Adaptive Programming*. They are at the core of *Demeter/Java* and are thus simply reused in the implementation of APPCs. Details about compilation of strategies can be found in [22, 17, 21].

The question remains how to realize the flexible component composition mechanism discussed above. As mentioned earlier, the mixin-like composition of APPCs is inspired by a similar composition of classes in the RONDO model [20]. Unfortunately, there are no such mechanisms in *Java* that could be immediately applied to compose the visitor classes generated from the

individual APPCs. We have worked around the problem within the same visitor-based framework described above. When compositions are involved, issues related to maintaining the control flow among the visitors generated by the individual APPCs need to be considered.

We assign the responsibility of maintaining the control flow among the individual APPCs in a composition to a *visitor-composer object*. This is an instance of the predefined class Visitor-Composer and it is created and initialized with the visitors to be composed. A Visitor-Composer maintains a list of visitors to be executed one after the other as well as their relations. The composer object plays the role of a script object for connecting the elementary components. The structure of the script is derived from the composition statements. In other words, the composer object in the case of our example will encode the relationships illustrated in Fig. 13. While composer objects are created individually for each composition statement, the functionality for "interpreting" the script they encapsulate is implemented in the predefined class VisitorComposer. It should be noticed that composers do not have themselves any application functionality. They simply know how to control the flow among other visitors, given the structure of their composition.

The implementation issues discussed in this section can be summarized as follows. Instead of letting the programmer figure out how to use the visitor pattern for expressing collaborative behavior and then encode the pattern several times (e.g., for several pricing schemes), we let the generator do the work. The primary motivation behind this is actually not necessarily to ease the work of the programmer. The main goal is rather the resulting increase of adaptability, reusability, understandability, and the decrease of maintenance costs.

Note that mapping APPCs down to visitor objects is an implementation detail. It is rather a compromise that allows us an easy reuse of the existing Demeter/Java technology. Visitor objects has been preferred over parameterized classes as in some related approaches [28, 24], even though visitor objects might be less efficient than inlining code. This is because by mapping collaborations down to visitor objects, instead of inlining code, they can be shared by several parts of a class structure. More importantly, the same visitor classes can be shared to create different compositions. The advantages are twofold: both code explosion and name conflicts, characteristic for template based approaches, are avoided.

As already indicated, the implementation sketched here is rather naive. It assumes that the source code of the application is available and lack an object-oriented organization of the generated code. However, this is only a first implementation and we are currently working on an enhanced implementation translating APPCs to Java packages. In this implementation, the application and the code generated for the APPCs will be put in different packages with the latter importing from the former. In this implementation, the mini-language for instantiating APPCs plays the role of a module linking language.

5 Related Work

The Visitor pattern was proposed in [5] as an idiom for expressing collaborative behavior in standard object-oriented models. The essence of the pattern consists in encapsulating a "task" to be performed on an object structure within a visitor object. The interface of the visitor object consists of a set of operations – one for each class in the "basic" class structure to be visited. In order to allow for future visits, classes in the basic structure implement an *accept* operation. The accept operation of a class C, expects a visitor object as a parameter and sends a message to this visitor to invoke the visitor's method for C, generally named visitC. There are several problems with applying the visitor pattern for coding collaborative-based designs:

1. As noted in [5, 25, 20], the pattern is applicable only when the base class structure to be visited does not change and when there are not very many additions of new concrete element classes.
2. Visitors make strong commitments to the particular class structure they are supposed to visit. This unnecessarily hinders the reuse of collaborative behaviors that might be generic enough to be applied to a range of different class structures.
3. Variations in visiting behavior are not properly supported, because the mechanism for supporting variations is inheritance, which performs rather poorly when a great number of variations of a basic functionality (or combinations of those) are to be supported [20].

Other approaches, such as the work by VanHilst and Notkin [28], and the work by Smaragdakis and Batory [24], similarly recognize the need for constructs to support collaboration-based design, and show how to use existing language mechanisms such as template classes for this purpose. In [28], roles of classes are implemented by means of template classes, where the superclass is made a parameter of the role, as illustrated below:

```
template <class Super> class Role:public
   Super{... role implementation... };
```

In addition to the superclass, a role (template class) is also parameterized with the other roles, played by other classes within the same collaboration. For instance, the role of the class K_2 in the collaboration C_4 in Fig. 1 would be expressed as follows:

```
template <class RoleSuper, class K1,4, class
K3,4> class K2,4:  public RoleSuper {...  role
  implementation using K1,4 and  K3,4 ..  };
```

This approach has a scalability problem [24]. Composing the template classes results in long and complicated template instantiation statements even for relatively small examples. There is no syntax for the collaboration entity. This is implicitly encoded in the parameter-relationships between the roles of the classes contributing to the same collaboration. The programmer has to explicitly keep track of the collaborations in which a class participates. For instance, the $K_{1,4}$ in Fig. 1 must be explicitly parameterized with $K_{1,2}$ – there is no way the programmer can ignore the fact that K_1 is not involved in C_3. As indicated in [24], the length of parameterization expressions increases exponentially with the number of different roles for a class.

In the *mixin-layers* approach, these problems are taken into account by implementing collaborations as mixins (outer mixins) that encapsulate other mixins (inner mixins). An outer mixin is called a *mixin layer*. The super-parameter is specified at the level of a mixin-layer (collaboration). By explicitly representing collaborations as mixin layers and by defining the super-parameter at the level of collaborations, this approach provides for a much better organization of the code and addresses the scalability problems of the work by Van-Hilst and Notkin. In [24], a possible realization of the general concept of a mixin layer by using C++ parameterized inheritance and nested classes is presented.

A mixin-layer is conceptually similar to a visitor class. However, mixin-layers are reusable and interchangeable, which is not true for visitor classes. A single layer can be used in several different compositions and is, to an extent, isolated from other layers. However, more complex combinations of collaborations that contain more than one refinement of the same base collaboration are not considered in [24]. As stated by Holland [9], neither inheritance nor parameterization fully support the desired features of a composition mechanism listed in Sec. 2. Furthermore, *mixin-layers* are not structure-generic.

Both the solution based on the visitor pattern and the solutions based on template classes are only idioms to be used to be used to resolve certain non-functional forces. However, patterns and idioms are cover-ups and not principal solutions to the problems they cover. The principal solution implies turning idioms into language features. Concerning the expression of collaborative behavior, the principal solution is the design of large-scale components, such as APPCs, that directly support coding the collaborations captured by *use cases*, *collaboration diagrams* and other similar artifacts available during analysis and design in many design methodologies.

Holland also proposes a language construct for expressing collaborations called *Contracts* [9]. Both Contracts and APPCs aim at making the collaboration patterns between classes involved in an application explicit by means of higher-level constructs that go beyond classes. APPCs and Contracts are themselves the building blocks of an application. Given a partly implemented application, with a given structure and some low-level behavior for accessing this structure, Contracts and APPCs fill it with additional behavior. Like APPCs, contracts make explicit (a) classes involved in a collaboration, and (b) the subset of instance variables, method interfaces and implementations provided by each class for this collaboration.

However, APPCs are superior as compared to Contracts. First, due to the use of interface class graphs, APPCs provide an elegant and effective way for specifying the collaboration pattern of a task, which is missing in contracts. As a result, the definition of a collaboration as well as the instantiation of it for concrete applications is more elegant and succinct. More importantly, in contrast to contracts APPCs are structure-shy. Second, the composition mechanism used with APPCs is more powerful. Holland uses frameworks as the basic composition mechanism combined with *lenses* to avoid name conflicts when conflicting collaborations are involved in the same application. Lenses are meta-objects that keep track of the current active collaboration among conflicting collaborations. Before each collaboration is executed, an appropriately initialized lense-object needs to be enabled. After a collaboration is executed, the current lense-object must be disabled to allow the execution of other collaborations. In this way, only one collaboration can execute at a time. More importantly, as indicated in [28], using frameworks makes contracts less reusable.

Related to APPCs is the work on *Subject-Oriented Programming* (SOP) [6]. A subject is a collection of class fragments whose class graph models its domain in its own subjective way. Subject composition combines subjects to produce new higher-level subjects. A subject has an affinity to APPCs. While a subject deals with class fragments, an APPC deals with class-valued variables which are mapped later onto class fragments automatically generating the necessary glue code. While a subject has to deal with all involved class fragments explicitly, an APPC only talks about the important class-valued variables. After the mapping to

classes, code will be generated automatically for the less important classes based on the information in the traversal strategies and the class graph. We believe that traversal strategies simplify the composition rules for SOP and that composition of APPCs can benefit from the composition ideas already developed for SOP.

Furthermore, the work presented here is related to the ongoing research on *Aspect-Oriented Programming* (AOP) [13]. Aspect-oriented programs are specified by collaborating building blocks, each one addressing a different concern of the application. The main goal is to minimize dependencies between the building blocks, so that modifications in one building block has a minimum impact on the other building blocks. APPCs seem to be useful building blocks for aspect-oriented programming within the object-oriented paradigm, since they are designed to support minimizing tangling between class collaborations and between class collaborations and class graphs.

The importance of supporting collaboration diagrams beyond the design phase has been recognized also by companies specializing in software development tools. Structure Builder from Tendril Inc. (www.tendril.com), supports turning interaction diagrams into executable code, however in a non-adaptive way. Structure Builder also facilitates object transportation in a similar way as APPCs.

Finally, component technologies such as Corba, COM and JavaBeans offer good facilities to describe components but there is not much help in making the collaborations explicit or for making independent components adaptable to a range of applications and for composing them together in complex ways.

6 Conclusions

In this paper, we proposed components for expressing collaborative behavior in object-oriented programs, called *Adaptive Plug and Play Components* (APPC). Adaptability is achieved by making the class graph of applications a formal parameter of the APPCs. The collaborations are written to this formal parameter rather than to a concrete application. On the other hand, APPCs are components with a mixin flavor, in that they define collaborative behavior in terms of "parent" components that are left unbound at component definition time. In this way, APPCs do not make any implicit commitment to a particular structure of composition, resulting in better reuse.

We demonstrated the benefits of APPCs by means of a simple example. However, we are confident that similar results can be reported for more complex applications. Our confidence is supported by the fact that APPCs were born within the well-developed *Demeter/Java* technology. The results with Demeter/Java have been very encouraging in several commercial projects (see [3]). We expect that the enhancements provided by APPCs will lead to even larger productivity.

The design of the APPCs is only the first step in a research path we intend to follow. First, as already indicated, we are working on separate compilation of APPCs, which requires some considerations on how to organize code generation for APPCs. In a more general context, design tradeoffs in the implementation of an adaptive software engineering system will be an area of future work. It is of interest to investigate the design of a package facility for APPCs collecting several APPCs, e.g., encoding collaboration diagrams from the same use case. These APPC packages can be mapped down to modules of the underlying languages (e.g., Java Packages). Another interesting area of future work is to develop a methodology for translating use cases from the analysis to APPC packages and evaluate the adaptive software engineering system in real applications.

Acknowledgements

Thanks go to the Demeter Seminar participants, especially M. Wand, D. Orleans, J. Ovlinger, G. Hulten and L. Blando for being a sounding board for the ideas presented in the paper, and to Y. Smaragdakis for his feedback both on strategies and his work with D. Batory on mixin layers and their relationship to APPCs. Especially, D. Orleans and J. Ovlinger provided valuable ideas that went into this paper.

This work has been partially supported by the Defense Advanced Projects Agency (DARPA), and Rome Laboratory, under agreement number F30602-96-2-0239. The views and conclusions herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

References

[1] K. Beck and W. Cunningham. A Laboratory for Teaching Object-Oriented Thinking. In *Proceedings of OOPSLA '89*, ACM SIGPLAN Notices, Vol. 24, No. 10, pp. 1–6, 1989.

[2] G. Bracha and W. Cook. Mixin-Based Inheritance. In *Proceedings of OOPSLA-ECOOP '90*, ACM SIGPLAN Notices, Vol. 25, No. 10, pp. 303–311, 1990.

[3] Demeter Research Group. Online Material on Adaptive Programming, Demeter/Java, and APPCs. http://www.ccs.neu.edu/research/demeter/

[4] M. Fowler. UML distilled. Prentice Hall, 1997

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994.

[6] W. Harrison and H. Ossher Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of OOPSLA '93*, ACM SIGPLAN Notices, Vol. 28, No. 10, pp. 411–428, 1993.

[7] R. Helm, I. Holland, and D. Gangopadhyay Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of OOPSLA '90*, ACM SIGPLAN Notices, Vol. 25, No. 10, pp. 303–311, 1990.

[8] I. Holland. Specifying Reusable Components Using Contracts. In *Proceedings of ECOOP '93*, LNCS 615, pp. 287–308, 1992.

[9] I. Holland. The Design and Representation of Object-Oriented Components. PhD Thesis, Northeastern University, 1993.

[10] P. Jansson and J. Jeuring. PolyP - a Polytypic Programming Language Extension. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 470–482, Jan. 1997.

[11] R. Johnson and B. Foote. Designing Reusable Classes. In *Journal of Object-Oriented Programming*, 1(2), pp. 22-35, June/July 1988.

[12] R. Johnson. Frameworks = (Components + Patterns). In *Communications of ACM*, Vol. 40, No. 10. 1997

[13] Kiczales G., Lamping J., Mendhekar A, Maeda C., Lopes C. V., Loingtier J. M., Irwin J. *Aspect-Oriented Programming.* Invited Talk. In *Proceedings of ECOOP '97*, LNCS 1241, pp. 220–243, 1997.

[14] S. Lauesen. Real-Life Object-Oriented Systems. *IEEE Software*, pages 76–83, March/April 1998.

[15] K. J. Lieberherr and I. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, pages 38–48, September 1989.

[16] K. J. Lieberherr and D. Orleans. Preventive Program Maintenance in Demeter/Java (Research Demonstration) In *Proceedings of the ICSE*, 1997, pp. 604–605, ACM

[17] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient implementation. TR NU-CCS-97-15, College of Computer Science, Northeastern University, 1997.

[18] M. Mezini. Dynamic Object Evolution Without Name Collisions. In *Proceedings of ECOOP '97*, LNCS 1241, pp. 190–219, 1997.

[19] M. Mezini. Maintaining the Consistency of Class Libraries During their Evolution. In *Proceedings of OOPSLA '97*, Sigplan Notices Vol. 29, No. 10, pp.1–22, 1997.

[20] M. Mezini. Variation-Oriented Programming Beyond Classes and Inheritance PhD Thesis, University of Siegen, Germany, 1997.

[21] J. Palsberg, B. Patt-Shamir, and K. Lieberherr. A New Approach to Compiling Adaptive Programs. *Science of Computer Programming*, 29(3):303–326, 1997.

[22] J. Palsberg, C. Xiao, and K. Lieberherr. Efficient Implementation of Adaptive Software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, Mar. 1995.

[23] T. Reenskaug et al. OORASS: Seamless Support for the Creation and Maintenance of Object Oriented Systems. In *Journal of Object-Oriented Programming*, Oct. 1992.

[24] Y. Smaragdakis and D. Batory. Implementing Layered Designs with Mixin-Layers. In *Proceedings of ECOOP '98*. To appear.

[25] L. M. Seiter. *Design Patterns for Managing Evolution.* Ph.D. Thesis, Northeastern University, 1996.

[26] P. Steyaert, W. Codenie, T D'Hondt, K. De Hondt, C. Lucas, and M. Van Limberghen. Nested Mixin-Methods in Agora. In *Proceedings of ECOOP '93*, LNCS 707, pp. 197–219, Springer-Verlag, 1993.

[27] C. A. Szyperski. Import is not Inheritance – Why We Need Both: Modules and Classes. In *Proceedings of ECOOP '92*, LNCS 615, pp. 19–32, Springer-Verlag, 1992.

[28] M. VanHilst and D. Notkin Using Role Iomponents to Implement Collaboration-Based Designs. In *Proceedings of OOPSLA '96*, ACM SIGPLAN Notices, Vol. 28, No. 10, 1996.

[29] M. VanHilst. Role-Oriented Programming for Software Evolution. PhD thesis, University of Washington, 1997.

[30] N. Wilde, P. Matthews and R. Huitt. Maintaining Object-Oriented Software. In *IEEE Software*, 10(1), pp. 75-80, Jan. 1993.

[31] N. Wirth and J. Gutknecht. The Oberon System. In *Software–Practice and Experience*, Vol. 19, No. 9, Sept. 1989.