Evolution of Object Behavior using Context Relations

Linda M. Seiter*
Computer Science Dept.
Boston College
Chestnut Hill, MA 02167
email: seiter@cs.bc.edu

Jens Palsberg
Laboratory for Computer Science
MIT
NE43-340
545 Technology Square
Cambridge, MA 02139
email: palsberg@theory.lcs.mit.edu

Karl J. Lieberherr
College of Computer Science
Northeastern University
360 Huntington Avenue
Boston, MA 02155
email: lieber@ccs.neu.edu

Abstract

A collection of design patterns was described by Gamma, Helm, Johnson, and Vlissides in 1994. Recognizing that designs change, each pattern ensures that a certain system aspect can vary over time such as the operations that can be applied to an object or the algorithm of a method. The patterns are described by constructs such as the inheritance and reference relations, attempting to emulate more dynamic relationships. As a result, the design patterns demonstrate how awkward it is to program natural concepts of behavioral evolution when using a traditional object-oriented language.

In this paper we present a new relation between classes: the context relation. It directly supports behavioral evolution, and it is meaningful at the analysis, design, and implementation level. At the design level we picture a context relation as a new form of arrow between classes. At the implementation level we use a small extension of C++. The basic idea is that if class C is context-related to a base class B, then B-objects can get their functionality dynamically altered by C-objects. Our language construct for doing this is a generalization of the method update in Abadi and Cardelli's imperative object calculus. A C-object may be explicitly attached to a B-object, or it may be implicitly attached to a group of B-objects for the duration of a method invocation. We demonstrate how the context relation can be used to easily model and program the Adapter, Bridge, Chain of Responsibility, Decorator, Iterator, Observer, State, Strategy, and Visitor patterns.

Keywords: Reusable design, a new class relation, behavior composition.

1 Introduction

1.1 Background

. The production of highly reusable software components is a fundamental goal of software engineering. While code reuse has long been acknowledged as a valuable engineering practice, design reuse has received less attention. The software engineering community has only recently been provided with a clear notion of reusable design constructs, in the form of *design patterns* as presented by Gamma, Helm, Johnson, and Vlissides [6]. The patterns have been fundamental in helping the software engineering community recognize forms of evolution for objects in an application domain. Each design pattern identifies an aspect of a system that may vary, and proposes a way of writing programs such that the variation is possible. Table 1 contains a partial list of design patterns from [6], along with the variation each is intended to support.

Pattern	Aspect(s) that vary
Adapter	object interface
Bridge	object implementation
Chain of Responsibility	object that handles a request
Decorator	object responsibilities
Iterator	composite object traversal
Observer	inter-object dependency
State	value-dependent object behavior
Strategy	algorithm implementation
Visitor	class responsibilities

Table 1: Variations defined by design patterns

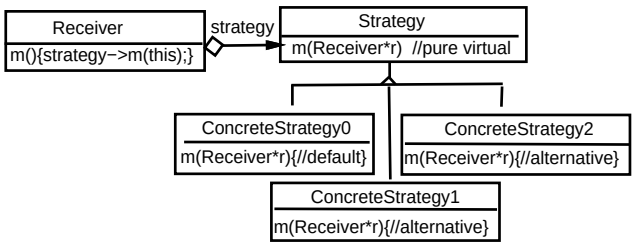

Figure 1: Strategy Pattern

As an example, the *Strategy* pattern allows an algorithm to be encapsulated as a construct, and supports a mechanism for varying parts of the algorithm. Thus, a strategy allows multiple implementations to be defined for a single interface. Figure 1 describes the strategy pattern using the aggregation and inheritance relations. A *Receiver* class has a method m, which we will assume has a void type. A *Strategy* class hierarchy is defined to provide alternative implementations of the method m, with each concrete strategy subclass representing a particular variation. An instance of

*Research conducted while at Northeastern University

the receiver class will reference a strategy object. The implementation of the m method for the receiver class simply delegates the request to its strategy, passing its *self* reference along as an argument. The m method defined in the concrete strategy is executed to perform the correct variation of the algorithm for the receiver object.

There are several reasons for using the aggregation relation to link the receiver and strategy classes rather than simply using subclasses of the receiver class to implement the method m. First, there may be many algorithms of the receiver class that should be varied, and it would be undesirable to define subclasses to cover all permutations. Moreover, it may be necessary to vary dynamically which strategy is used for a receiver class instance. The static inheritance relation can not support such dynamic variation.

An alternative would be to provide methods with different names for each implementation, or to have a method that takes a parameter to indicate which implementation should be executed. This approach however requires the client, which is the object that will make the call, to understand enough of the details of the implementation to be able to indicate which implementation they would like. This violates class encapsulation by requiring the client class to know details about the implementation of the receiver class.

Like the Strategy pattern, all of the design patterns in [6] are described using constructs such as the static inheritance and aggregation relations, with examples of each pattern given in C++ [5]. The code unintentionally demonstrates how awkward it is to program natural concepts of behavioral evolution in a static, class-based language. We will now examine the nature of and reasons for this awkwardness.

1.2 Issues Involved in Supporting Behavioral Evolution

The strategy pattern emphasizes that trying to support dynamic behavior in a static, class-based model results in the following:

1. Explicit delegation along aggregation relations,
2. Violation of class encapsulation, and
3. Creation of auxiliary structures when using inheritance.

Representing the relation between the receiver and strategy as aggregation (reference or association) requires explicit delegation to the strategy object, with the receiver object being passed as an argument. The strategy object then executes the appropriate behavior for the receiver object. Conceptually, the implementation of m given in the *Strategy* class should be executed upon the receiver of the message, namely the *Receiver* class instance. This corresponds to the self-reference principle supported with the implicit delegation relation [16]. In a language such as C++ that does not support implicit delegation, the alternative technique of explicit delegation will usually require the strategy class to be made a friend of the receiver (a *friend* is a class that is given access to the private members of another class), thus violating class encapsulation.

Far more significant than the issues associated with explicit delegation are those that arise from using the inheritance relation to group the alternative behavior implementations. While the inheritance relation may be useful for defining alternative implementations of a method in subclasses (method overriding), it is not adequate for defining multiple implementations of a method for a single object. The strategy pattern tries to get around this by combining aggregation and inheritance, requiring an abstract strategy superclass to represent the common interface to the concrete subclasses that define the actual algorithm variations. As there is nothing for the concrete subclasses to inherit, the strategy superclass serves no purpose other than to define the common interface of its subclasses, which is often a duplication of the interface of the receiver class. Relying on the inheritance relation to group the alternative method implementations for the strategy will always require this extraneous class and interface duplication. What is needed is a mechanism to allow the concrete strategy classes to be linked directly to the receiver class, as they provide alternative implementations of a method of the receiver class.

An alternative to modeling behavioral evolution with static aggregation and inheritance is the use of dynamic inheritance, as is found in prototype-based languages such as Self [25] and Lieberman [16]. Dynamic inheritance, or delegation, allows the behavior of an object to evolve dynamically by modifying its *parent* relation to other objects. If an object does not have an implementation for a message it receives, the message is implicitly forwarded to its parent who either executes an implementation of the message on the receiver object, or continues forwarding of the message along the parent chain. An object may dynamically change its parent; thus it may portray different behavior for the same message. While dynamic inheritance avoids some of the problems associated with using static aggregation and inheritance to model behavioral evolution, prototype-based languages present other problems:

1. Type checking,
2. Change impact, and
3. Performance.

While dynamic inheritance allows an object's behavior to change by varying its parent, it also allows the possibility of a message being received that can not be resolved. An additional problem associated with prototype-based languages arises from the inability to distinguish between treating an object as simply an object as opposed to a prototype (or class). Dynamically changing a property of an object may impact its children, which may or may not be the desired result. Another issue involves incremental versus overriding inheritance. Adding a parent to a base object extends the behavior of the base object, but the behavior defined in the parent does not override or hide that defined in the base object. Dynamic inheritance may be used to add behavior to an object, but not to override existing behavior defined for the object. Finally, there are the performance issues associated with dynamic inheritance, in that the parent chain must be searched at run-time to find the implementation of a message.

In summary, the basic problem with supporting behavioral evolution in static class-based languages like C++ is that there does not exist direct language support for allowing multiple implementations of a method for an object. While it is easier to support behavioral evolution in prototype-based languages using dynamic inheritance, the inability to ensure safety, the inability to override existing behavior in a controlled manner, as well as the negative performance impact, offset the benefit of their dynamic properties.

1.3 Safely and Efficiently Supporting Behavioral Evolution

To achieve dynamic behavior, it is necessary to take more than just the static class of an object into account when invoking a method. There are many factors that can influence

the behavior of an object, such as the current state of the object, the task being performed, or even the platform on which the program is executing. In this paper, we focus on two forms of behavior evolution:

1. dynamically altering a method implementation for a single object, and
2. dynamically altering a method implementation of a class (affecting all class instances) during the execution of some task.

To achieve behavioral evolution we introduce a new relation called the *context* relation to the object-oriented model. The relation exists between classes and is intended to support behavioral evolution while maintaining the safety and performance benefits available with strongly typed, class-based languages. The context relation produces class hierarchies orthogonal to inheritance hierarchies, in many cases replacing inheritance. The relation is meaningful at the analysis, design, and implementation level. The basic idea is that if class C is context-related to a base class B, then B-objects can get their functionality dynamically altered by the presence of C-objects. Our language construct for doing this is a generalization of the method update in Abadi and Cardelli's imperative object calculus [1]. A C-object may be explicitly attached to a B-object, or it may be implicitly attached to a group of B-objects for the duration of a method invocation. This supports reuse of the class B because like inheritance, the class B does not know about class C. Unlike dynamic inheritance, methods in the context class C override methods in the base class B for B-objects. The relation supports dynamic behavioral evolution because the C-object related to a B-object can vary at run-time.

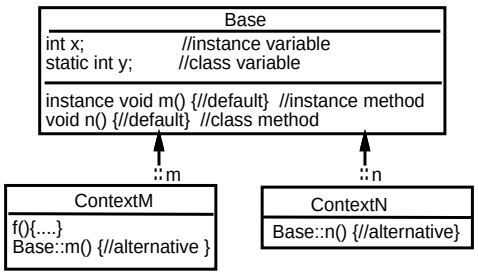

Figure 2: Static class definitions

Figure 2 contains an object model that will be used to demonstrate the two forms of behavioral evolution. The figure shows the relation between a base class called *Base* and two context classes *ContextM* and *ContextN*. The context relation is drawn as an arrow between classes, using the C++ scope operator :: to emphasize its role of altering the base class implementation. A context class does not inherit methods from a base class, and is not considered a subclass of a base class. Rather, a context class can define methods for its own class instances, such as the *f* method of the *ContextM* class, as well as define method implementations for base class instances, such as the *m* method of the *Base* class that is redefined in the *ContextM* class. A context class may have multiple direct base classes. Likewise, a base class may have multiple context-related classes, each altering one or more methods of the base class. Several context classes may redefine the same method of a base class.

Recall that we want to support two forms of behavioral evolution: (1) dynamically altering a method implementation for a single object, and (2) dynamically altering a method implementation of a class (affecting all class instances) during the execution of some task. Languages such as C++ differentiate the attributes (data members) defined for a class as being either instance variables or class (static) variables. To achieve the two forms of behavioral evolution, we differentiate between methods whose implementation should vary on a per object basis (instance methods), and methods that will vary for the class as a whole.

The *Base* class in Figure 2 defines two attributes x and y. The x attribute is an instance variable, implying each *Base* class instance will have memory allocated to store the attribute. The y attribute is a static class variable, implying all class instances refer to the same memory location for this attribute. The *Base* class also has two methods, m and n. The m method is an *instance* method, implying each *Base* class instance will have memory allocated to store the address of the m implementation. This allows the implementation of m to vary per object. The n method is a class method whose implementation may vary dynamically, however all class instances will refer to the same implementation since the implementation is stored with the class rather than individual class instances.

In static languages like C++, the method implementation invoked when an object receives a message is either (1) the implementation defined for the class of the variable used to invoke the message, or it is (2) the implementation defined for the class of the object receiving the message, as in the case of a virtual function and pointer reference. In the simple case of non-virtual functions, instantiating a class and sending messages to its instances will cause the method implementations defined in class to be invoked (or those inherited from a superclass). Even with virtual functions, each class still has only one implementation per method interface, thus the virtual function tables are static. Each object conceptually has one context defining its run-time behavior, namely its static class implementation.

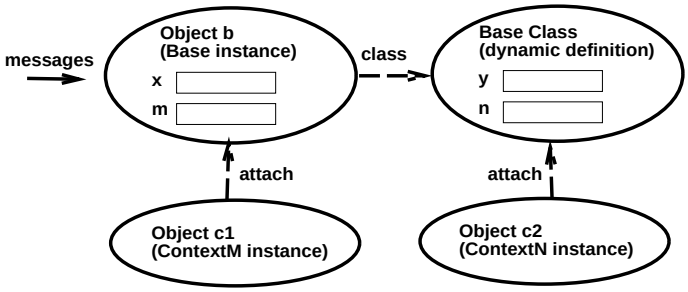

Figure 3: Dynamic method lookup

However, we want to support more dynamic forms of behavior than the traditional static class model allows. This is easily achieved using the context relation between classes. Figure 3 shows object b, an instance of the *Base* class from Figure 2, receiving messages. An m message will invoke the m implementation that is stored with the object. An n message will invoke the implementation stored with the dynamic class *Base*. The classes *ContextM* and *ContextN* are context-related to class *Base*, and define alternative implementations for the *Base* class methods. Instantiating class *ContextM* produces a context object, which may be used to alter instances of class *Base* dynamically. For example, attaching the *c1* context object to the *Base* object b will alter

the m method implementation to be that which is defined in the *ContextM* class. Instantiating the *ContextN* class and attaching the *c2* context object to a method invocation will alter the *Base* class implementation of method n for the duration of the call.

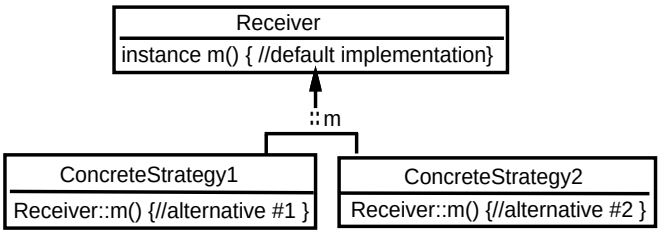

Figure 4: Strategy pattern using context relations

Figure 4 describes how to model the Strategy pattern using context relations. The context relation may be labeled with the name of the method being redefined, to facilitate diagram understanding. Notice there no longer exists the abstract class *Strategy*, as was required in Figure 1.

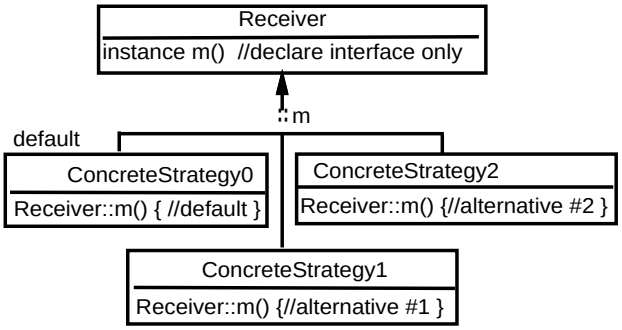

Figure 5: Alternative strategy pattern using context relations

Alternatively, the strategy pattern could be modeled as in Figure 5. The *Receiver* class describes the method interfaces for its instances, yet the implementations are given separately in the strategy hierarchy defined by the context relation. Thus, the *Receiver* will define the structure and interface of its instances, namely that there is a method m, while the alternative implementations of m are given in the strategy classes, one of which is designated as a default implementation. When the *Receiver* class is instantiated, the object will have the default m method implementation initially defined for it. Separating the implementations of m from the *Receiver* class allows dynamic variation of the method. Ensuring that one of the implementations is declared as a default will protect against a "message not understood" error.

There have been various proposals for separating an interface from an implementation. For example the language Java supports this separation [19]. An *Interface* defines a set of method signatures, while a *Class* defines the structure and behavior of its instances. A class may match several interfaces, if it implements the methods defined by each interface. Classes are instantiable, while interfaces are not. In contrast, we propose a different separation of interface and implementation, based on the properties supported by the context relation. A base class defines the structure and interface (method signatures) of its instances, and may provide implementations for the interface as well. If an implementation of a method is not provided directly in the base class, the implementation may be separately given in one or more classes that are context-related to the base class. In contrast to the notion of *Interface* in Java, a base class that defines only interfaces *may still be instantiated*. To prevent a method resolution error at run-time, we require an implementation of the method to either exist directly in the base class, as in Figure 4, or to exist in some context-related class that is declared to be the default implementation, as in Figure 5.

C++ extension	Purpose
`o ::+ c`	context c attached to object o
`m{c}()`	context c attached to method call m
`{+c}`	incremental composition (vs. overriding)
`context`	metavariable reference current context
`default`	default method implementation
`instance`	instance method (vs. class method)
`Class C { ... B::m(){...} }`	Class C redefines B's m method

Table 2: C++ extensions

1.4 Paper organization

Design patterns have been categorized as creational, structural, and behavioral [6]. Creational patterns abstract the instantiation process, structural patterns abstract object composition, and behavioral patterns abstract object communication and responsibilities. The structural and behavioral patterns that benefit from the context relation can be categorized into two groups:

1. those based on dynamically altering the behavior of a single object, such as adapter, bridge, chain of responsibility, decorator, observer, state, and strategy; and
2. those based on dynamically altering a class definition (affecting all instances) for the duration of some task, such as iterator and visitor.

In the following sections, we discuss several of these design patterns. We first describe them using only standard object-oriented modeling constructs, and then using also the *context* relation. To enable the direct expression of the context relation in C++, we introduce the constructs depicted in Table 2. We show implementations of the patterns both in C++ and in our extended version of C++. Our claim is that by adding context relations to the design and implementation vocabulary, design patterns are more easily modeled and implemented.

2 Attaching a context to an object

Several design patterns propose techniques for supporting the dynamic modification of a method or interface. In this section, we present the notion of dynamically altering the behavior of an object by attaching contexts to it. An object may have many contexts attached to it, with each context altering one or more method implementations for the object.

2.1 Strategy Design Pattern

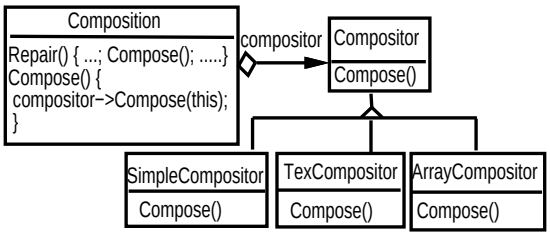

Figure 6: Object model for line breaking strategy

The *strategy* pattern allows a family of interchangeable algorithms to be defined and encapsulated. This allows the algorithm to vary independently of the clients that use it [6]. Figure 6 contains a partial object model for representing the relation between a *Composition* class used to maintain and update the line-breaks of text displayed in a text viewer, and a *Compositor* class which represents different variations of the line-breaking algorithm. For instance, *SimpleCompositor* implements a simple default line-breaking technique, *TexCompositor* implements a complex line-breaking technique for a TEX viewer, while *ArrayCompositor* provides a fixed number of elements per line. The aggregation relation is used to link the *Compositor* and *Composition* classes, while the inheritance relation is used to group the alternative implementations of the line-breaking strategy. Note the *Compositor* abstract class serves only to provide a common interface for the concrete line-breaking strategy alternatives.

```
class Composition {
public:
 Composition(Compositor* _comp){compositor=_comp;}
 void Repair() {...; Compose(); .... }
 void Compose() {compositor->Compose(this); }
 ...
private:
 friend class Compositor;
 Compositor* compositor;
};
class Compositor {
public:
 virtual void Compose(Composition* c) { }
 ...
};
class SimpleCompositor: public Compositor {
public:
 void Compose(Composition* c){//default}
};
class TexCompositor: public Compositor {
public:
 void Compose(Composition* c){//alternative}
};
```

Figure 7: Composition Class

Figure 7 contains C++ code for implementing the line-breaking strategies. The *Composition* constructor initializes the *compositor* reference, which may be altered dynamically. The *Repair* method for the *Composition* class invokes its *Compose* method, which delegates the request along the *compositor* reference to the appropriate strategy object, passing its self reference as an argument to allow the strategy to work on the *Composition* object. Note that the implementations defined in the *Compositor* hierarchy are actually intended to perform actions on the *Composition* object that receives a *Repair* message. The essence of the strategy pattern is to allow multiple variations of an algorithm. As there does not exist a direct mechanism in a traditional class-based language or model for supporting multiple implementations of a method for an object, the translation of the pattern into a corresponding class-based model, as in Figure 6, turns the relation into aggregation and inheritance.

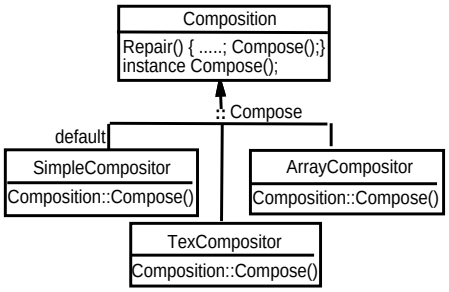

Figure 8: Updated object model for Composition

The context relation has neither an analogous abstraction in existing object models, nor an analogous program construct. Thus, it is implemented using existing relations and constructs, with the essence of the strategy pattern lost in both the object design and the translation to code. To remedy this, the *context* relation is used to link the *Composition* class and its alternative method implementations, as shown in Figure 8.

```
class Composition {
public:
 Composition(Composition:: _comp) {this ::+ _comp;}
 void Repair() {...; Compose(); .... }
 instance void Compose(); //Interface only.
 ...
};
class SimpleCompositor {
 default void Composition::Compose(){//default implementation }
};
class TexCompositor {
 void Composition::Compose(){//alternative implementation }
};
class ArrayCompositor {
 void Composition::Compose(){//alternative implementation }
};
```

Figure 9: Composition Class Using Context Relation

Programming the context relation requires language support for (1) indicating whether a method is an instance method or a class method, (2) allowing a context class to define method implementations of its base class, termed *method update*, and (3) dynamically altering a method implementation, through *context attachment*. The context relation itself may be inferred from the class body. The C++ code from Figure 7 has been rewritten in Figure 9, using new program constructs for implementing the context relation. The *Compositor* class and its associated inheritance hierarchy are no longer necessary.

To achieve *method update*, we extend the standard class construct. A class defines data members and methods, and may redefine methods for its base class. The *Composition*

base class defines methods *Repair* and *Compose*. The *Compose* method is defined as an *instance* method, indicating each *Composition* class instance may vary the implementation of this method. The *Repair* method is a class method, indicating the method implementation is stored with the class rather than with each individual class instance. The *SimpleCompositor, TextCompositor,* and *ArrayCompositor* classes redefine the *Compose* method of the *Composition* class. The implementation invoked when a *Composition* object receives a *Compose* message depends on whether an instance of the *SimpleCompositor*, *TexCompositor*, or *ArrayCompositor* class has been attached. Note that a *Compose* message may be sent only to *Composition* objects.

To achieve *context attachment*, we propose the operator `::+`. When an object is first instantiated, its behavior is defined by its static class definition. This may be dynamically altered by attaching contexts. The context relation introduces a new type that is inferred from the class model. We use *Composition*:: to describe the set of classes that are context-related to base class *Composition*. The context classes update the *Compose* method of the *Composition* class, with the implementation given in the *SimpleCompositor* class defined as the default. The *Composition* constructor will attach a context object passed in as an argument to the newly created *Composition* object. The expression `this ::+ _comp;` alters the *Composition* object by updating the *Compose* method implementation to be that defined in the context class.

In C++, a dynamic method lookup is required when a *virtual* method is invoked through a pointer. In a language such as Java [19], all methods are assumed virtual (unless declared final) and all objects are referenced through pointers. Each method invocation will require a dynamic lookup. The context relation allows method update, not just for subclasses but for the base class itself. Dynamic lookup occurs regardless of whether a method is invoked through a pointer or variable. We propose the use of context tables, a dynamic version of static virtual function tables, to avoid pointer chasing through lists of context objects. The use of the tables will support fast method lookup. For simplicity, we assume all methods to be context-updatable, although it would be possible to add a keyword to limit dynamic lookup such as the *final* keyword in Java.

Context attachment may alter a method implementation, which may refer to attributes and methods of either the base class or the context class itself. Thus, when a message is sent to an object, the method implementation should be executed on the receiver object as well as the context object. The self reference (*this* pointer in C++) refers to the receiver object. Additionally, it is useful to have a reference to the context object defining the method implementation. We call such a reference *context*. Executing the method on both the receiver and context will provide a means for simulating dynamic extension of the data members of the receiver class. As will be shown with the Visitor pattern, executing the method on the receiver and context also allows sharing of state among collaborating objects.

2.2 Other Patterns For Varying Object Behavior

The *State* pattern allows an object to change its behavior based on its internal state, where state represents a partitioning of data member values. An object in different states may react differently to the same message. Figure 10 contains a partial dynamic model for representing a TCP network connection, based on the example presented in [6]. A TCP connection can be in one of several states such as closed, established, or listening, and may receive messages such as open, close, send, etc. Figure 10 also contains a partial object model, with *TCPConnection* linked to *TCPState* through an aggregation relation labeled *state*. The essence of the state pattern is to define a relation between the attribute values of an object and its behavior. This is modeled through the conceptual link between the object model and the dynamic model [21]. However, the translation of the dynamic model into a corresponding class-based model usually turns the relation into aggregation and inheritance. As with the strategy pattern, the state pattern is poorly modeled using aggregation and inheritance. To remedy this, the *context* relation is used to link a class and its alternative method implementations.

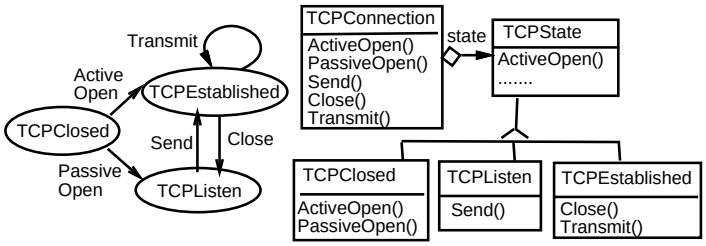

Figure 10: Example dynamic and object model for TCP connection

The *Adapter*, *Bridge*, *Chain of Responsibility Decorator* and *Observer* patterns can likewise be improved by using the context relation.

3 Attaching a context to a method invocation

In the previous section, the behavior of a single object was modified at run-time. In this section, we demonstrate how to alter the behavior of a group of collaborating objects during execution of some task by attaching a context to a method invocation. We present a technique for altering the implementation of a class method to affect all class instances, rather than an instance method which affects a single class instance.

3.1 Iterator and Traversal Pattern

The *iterator* pattern is used to allow the elements of a composite or aggregate structure to be accessed without exposing the details of how the composite is implemented (array, linked list, etc.). For example, a list contains a collection of elements. The responsibility for traversing the list is given to an iterator rather than polluting the list class with numerous iteration operations.

A more flexible pattern of object traversal allows navigation through composite objects of any form, not just homogeneous list structures. A *composite* is a structural pattern that represents a whole-part hierarchy [6]. For example, the class structure in Figure 11 describes a containment hierarchy for a computer consisting of equipment, which may contain other equipment as parts. Iterators at each level in the composite hierarchy could be written to traverse a computer object, performing some task during the traversal.

A *traversal* pattern defines paths through a class structure, along which object navigation should occur. Existing

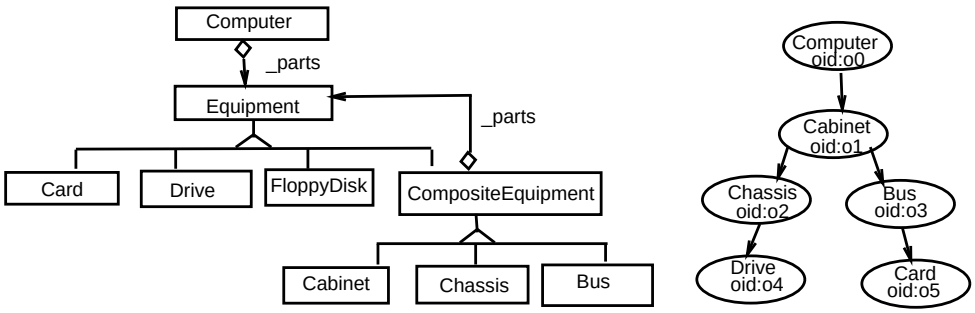

Figure 11: Computer equipment containment hierarchy and example object

languages require traversal patterns to be hard-coded, with specific reference relations between classes used for traversal. Modification of the class design requires maintenance of the traversal-oriented code. A *traversal specification* may be used to define paths within class structures adaptively, implying details of the actual class structure are not hard-coded [14]. A traversal specification supplies the minimal information required for abstracting a path from a class structure.

```
void Computer::allEquip() {  traverse to Equipment; }
```

Figure 12: Traversal Expression

Traversing the parts of a computer requires navigation along paths that lead to pieces of equipment, thus the target of traversal is the equipment class. Figure 12 shows example code to accomplish this task, using a traversal expression style proposed by Lopes and Lieberherr [17]. The keyword *traverse* is used along with a path description, for example, indicating paths that lead *to* equipment objects. The keywords *to*, *via* and *bypassing* can be used to constrain paths through a class structure.

```
void Computer::allEquip() { traverse(); }

void Computer::traverse() {
  before();
  for ( ListIterator<Equipment*> i(_parts);
        !i.IsDone(); i.Next())
             {i.CurrentItem()->traverse();}
  after();
}

void Equipment::traverse() { before(); after();}

void CompositeEquipment::traverse() {
  before();
  for (  ListIterator<Equipment*> i(_parts);
         !i.IsDone(); i.Next())
              {i.CurrentItem()->traverse();}
  after();
}
```

Figure 13: Equivalent C++ code

Executing a traversal expression will trigger the navigation of the method's host object. The simple traversal behavior will be composed with task-specific actions to perform as objects are encountered. The basic traversal algorithm consists of the following three steps: (1) an action to perform upon encountering an object, (2) traversal of the object, and (3) an action to perform after object traversal. The variable part of the algorithm is the action surrounding the object traversal. Figure 13 contains example C++ code conceptually equivalent to the traversal expression in Figure 12, assuming the class structure from Figure 11. The code contains template methods, or a skeleton, for defining the traversal algorithm. The variable part of the algorithm is the actual *before* and *after* behavior. The *allEquip* method should be executed within some context that defines implementations of the *before* and *after* class methods. We assume empty default *before* and *after* method implementations for each class, and that the methods are *class* methods rather than *instance* methods. One or more contexts will be used to override the default empty behavior, providing task-specific implementations.

3.2 Visitor Pattern

The *visitor* pattern is used to specify the operations to perform during traversal [6]. A visitor allows behavior to be added to a composite structure, without changing the existing class definitions. Visitors reduce the number of operations directly embedded within a class, thus preventing class definitions from becoming cluttered. Without a visitor, the implementation of a collaborative task may be spread over methods in different classes. Visitors group related operations performed on multiple classes (which need *not* be related through inheritance) together into one program component, supporting a task-based style of programming.

For example, two operations to be performed on the equipment class structure include computing inventory of materials and calculating total cost of equipment [6]. Inventory simply accumulates a list of equipment objects. Equipment cost is calculated by computing the net price of simple equipment and the discount price of composite equipment. As the purpose of a visitor is to add task-specific behavior to collaborating classes, the visitor defines the meaning of the *before* and *after* methods that are executed during a traversal. Figure 14 depicts the relation between the visitors and the *Equipment* class hierarchy.

Figure 15 contains the pricing and inventory visitors. The inventory visitor updates the *before* method for *Equipment*, which is inherited by its subclasses. The pricing visitor updates the *before* method for *Equipment*, as well as the *before* method for *CompositeEquipment*. Note that since the *before* and *after* methods are class methods rather than instance methods, there is only one implementation of each method, which is stored with the dynamic class definitions. Context attachment will update the dynamic class method

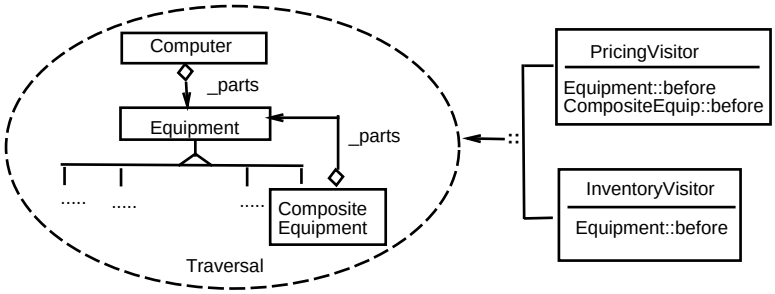

Figure 14: Context relation between visitor and equipment

```
Class PricingVisitor {
public:
 Currency _total;
 PricingVisitor():_total(0) {}
 Currency total() {return _total;}
 void Equipment::before(){_total+= net();}
 void CompositeEquipment::before(){_total+=discount();}
};
Class InventoryVisitor {
public:
 Inventory _inventory;
 InventoryVisitor():_inventory() {}
 Inventory inventory() {return _inventory;}
 void Equipment::before(){_inventory.accumulate(this);}
};
```

Figure 15: Pricing and Inventory visitors

implementations, thus affecting *all* class instances.

Notice the *before* method that is updated in *PricingVisitor* refers to the *_total* data member of the *PricingVisitor* class, as well as the *net* method of the *Equipment* class. An updated method is similar to a multimethod, in that it can refer to the receiver object (instance of the Equipment class, referenced by *this*) as well as the context object (instance of the *PricingVisitor* class, referenced by *context*). Thus, the self reference is maintained to the original receiver of the message, yet the context object may partake in the behavior as well. This serves two purposes:

1. The context object may simulate dynamic structural extension for an object, and
2. The context object may serve as a shared repository among collaborating objects.

```
int main() {
 Computer *c=new Computer(new Equipment....);
 InventoryVisitor *inv = new InventoryVisitor();
 PricingVisitor *price = new PricingVisitor();
 c->allEquip{inv}();   //execute call within context
 c->allEquip{price}(); //execute call within context
 cout<<"Inventory"<<inv->inventory()<<endl<<
       "Price"<<price->total()<<endl;
}
```

Figure 16: Pricing and Inventory visitors

The main program in Figure 16 instantiates the two visitors, and invokes the *allEquip* method for the *Computer* object. The expression `c->allEquip{inv}();` indicates the *allEquip* method should be executed within the context of the inventory object *inv*. Attaching the visitor to the call causes the dynamic *Equipment* class definition to be updated based on the method update in the inventory visitor. The visitor thus updates the *before* methods of the *Equipment* class hierarchy, for the duration of the *allEquip* method invocation.

Figure 17 depicts the relation between the method invocation and the inventory visitor. While the *allEquip* method is executing, which simply traverses an Equipment object, each *before* and *after* message will be evaluated within the context surrounding the *allEquip* invocation. Unlike the state and strategy patterns, where a context was attached to a specific object, the visitor pattern attaches a context to a traversal. This allows the context to affect messages sent to all objects for the duration of the traversal method.

If a composite object is large, it might be preferable to compute several visitors during a single traversal. As an object may have its run-time behavior affected by many context objects, a method invocation may as well. Both the inventory and pricing visitors define implementations of the *before* method for the *Equipment* hierarchy. If we want to compute both visitors during a single traversal, the call should be executed within the context of each.

Method updates may be composed in an incremental or overriding manner. This does not refer to the inheritance relations of the *Equipment* hierarchy, rather it refers to the dynamic composition of a class method when updated by multiple contexts. The composition determines whether multiple implementations of the method will be executed per message. By default, contexts are attached in an *overriding* mode, in that a context serves to override or hide the current method implementation. However, incremental attachment will allow multiple method implementations to be executed for a single message, as is the case when executing several visitors per traversal. Of course, there are the standard compositional constraints involving return type and arguments. Incremental mode allows a behavior to be dynamically extended rather than overridden, which is in fact the goal of the *Decorator* pattern. Figure 18 shows the main program rewritten with incremental context object attachment to compute both visitors during one traversal, simply by using the + operator.

4 Related Work

Holland summarizes extensions to the object model that have been proposed for varying the interface and behavior of an object [10], such as *perspectives* [7], *views* [22], and *contexts* [2]. He notes that the extensions can be characterized in terms of:

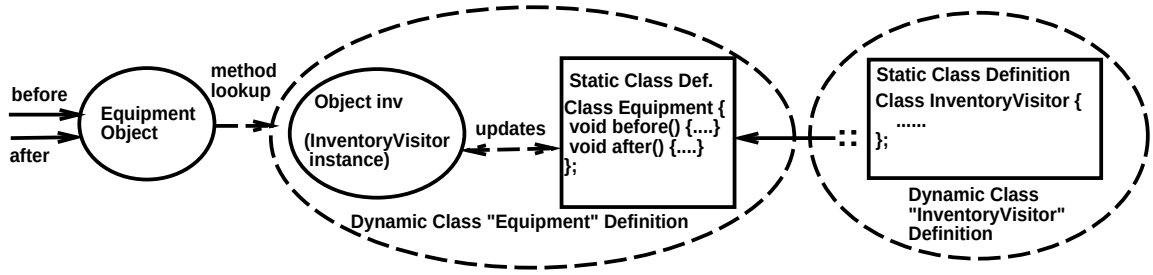

Figure 17: Executing a traversal within the context of a visitor

```
int main() {
 Computer *c=new Computer(new Equipment....);
 InventoryVisitor *inv = new InventoryVisitor();
 PricingVisitor *price = new PricingVisitor();
 c->allEquip{+inv,+price}();  //incremental composition
 cout<<"Inventory"<<inv->inventory()<<endl<<
       "Price"<<price->total()<<endl;
}
```

Figure 18: Pricing and Inventory visitors rewritten with incremental composition

- Object structure: are interfaces embedded in the structure of objects or layered on top of objects?
- Visibility of interface: are clients of an object restricted to using a specific interface or are all interfaces visible?
- Ability to define multiple implementations per interface: can an interface have different implementations or is there only one implementation?
- Effect on the call site: is the interface selected explicitly at the call site, or implicitly?
- Scope of the interface: is the interface defined over many object types or just one?

In the remainder of this section, we compare other proposals for evolving object behavior.

4.1 Evolving object interfaces and behavior

Perspectives [7] were among the earliest proposals for varying the interface of an object. Software components are represented as a network of *nodes*. The attributes of a node are grouped into *perspectives*, each providing a different view of the entity. To access a particular attribute of an object, one must specify the perspective along with the object. *Views* [22] present another approach to partitioning an object interface. As with perspectives, the required view must be explicitly paired with an object to access an attribute or invoke a method of the object. Both perspectives and views require clients to know about the interface subset partitioning, thus violating encapsulation. Arapis [2] proposed the notion of *contexts* to represent the roles of an object. As with *perspectives* and *views*, the client of an object must specifically state the method of a particular context to be invoked. The *mode* concept of Taivalsaari partitions the implementations of an interface, rather than the interfaces of an object [24]. When an object receives a message, the implementation triggered depends on the *mode* or state of the object. As opposed to roles and views, clients need not be aware of the current mode at the call site.

Each of these proposals is based on changing behavior for a single object.

4.2 Evolving collaborative behavior

Many tasks require the collaboration of a group of objects. Researchers have acknowledged the need to lift abstraction above the class level, and several approaches have been proposed for modeling collaborative, task-based behavior, including Frameworks [11], Clusters [18], Mechanisms [4], Contracts [9, 10], Propagation Patterns [26, 14], and Subjects [20].

As an example, a *Contract* [9, 10] models a set of object interactions. The *participants* of the contract have certain *obligations* (attributes and behaviors) that they must support. A contract defines a template of participants and obligations of a task, and is reused by instantiating and enabling the contract, attaching each participant role to an object and obligations to specific properties of the object. When an object is sent a message, its active contracts determine its behavior. While an object may participate in several contracts simultaneously, it may not participate in multiple contracts that define the same method. Additionally, the method implementation defined in a class hides that defined in a contract. Thus, it is not possible to dynamically alter an existing class method. With contracts, one must specifically attach (activate, enable) a participant role to a particular object in order to modify its run-time behavior. A context class may be related to several base classes, thus serving to define a collaborative, task-based style of programming, as was presented with the visitor pattern. With contexts, it is possible to implicitly alter many objects by call attachment.

An additional difference arises from the composition mechanism, which is the process of creating new components from existing ones. Contracts support composition through *refinement* (inheritance) and *inclusion*, which involves injecting the code of one component into another. Propagation patterns [14] provide an adaptive approach to collaborative behavior, in that behavior is defined in a manner that can be reused with many class structures. A propagation pattern consists of (1) a signature, (2) a *traversal directive*, which is an adaptive specification of object navigation similar to the traversal expression presented in this paper, and (3) a set of *code wrappers*, which describe task-specific behavior to be executed during the object navigation. A code wrapper maps code to classes to be executed during traversal. As an example, Figure 19 contains the *Pricing* and *Inventory* visitors rewritten as propagation patterns. A propagation pattern combines a traversal function with the behavior

that should be executed as class instances are encountered during the traversal. Notice that the propagation pattern is not a class definition, rather it should be thought of as a set of functions or methods. Thus, any calculated values such as the *_total* value must be passed along during traversal in the form of an argument parameter. This requires the client to initialize the value before the traversal is initiated. The main program calls each propagation pattern by invoking a method of the same name for the computer object.

```
*operation* void PricingVisitor(Currency &_total)
*traverse* *from* Computer *to* Equipment;
*before* Equipment{_total+=net();}
*before* CompositeEquipment{ _total+=discount();}

*operation* void InventoryVisitor(Inventory &_inv)
*traverse* *from* Computer *to* Equipment;
*before* Equipment{_inv.accumulate(this);}

int main() {
 Computer *c=new Computer(new Equipment....);
 Currency price;
 Inventory inv;
 c->PricingVisitor(price);
 c->InventoryVisitor(inv);
 cout<<"Inventory"<<inv<<endl<<"Price"<<price<<endl;
}
```

Figure 19: Pricing and inventory visitors as propagation patterns

Notice that two separate traversals must occur to compute the inventory and pricing visitors, since the traversal directive is a fixed part of each propagation pattern. A propagation pattern directly pairs an adaptive traversal specification with a particular task. In this paper, we separate the two, allowing adaptive traversal specifications to be reused with many tasks.

To execute both visitors during a single traversal, the traversal directive would have to be removed from one of the propagation patterns, which would be converted into a *transportation pattern* [15]. The two visitors could then be composed by injecting the code wrappers from the transportation pattern into the propagation pattern. The composition mechanism used with propagation patterns, namely transportation pattern inclusion, suffer from the same problem found with many approaches, including contracts. Specifically, when component inclusion is implemented by simply copying code, naming issues arise when a component is included into another more than once or if two included components overlap in named elements. While not a problem with the example pricing and inventory visitors, a problem would arise if they have both chosen to use the same name for their formal argument, rather than *_total* and *_inv*.

As the context construct is an extended class definition, the inheritance and aggregation relations may be used for class-level composition. At the dynamic object level, both incremental and overriding method update are possible. As a context object scopes its attributes and methods, incremental composition will not result in naming issues.

4.3 Meta-level approaches

A metaobject protocol (MOP) is an interface that allows a programmer to customize properties of the programming language, such as adding persistence and concurrency [3, 12]. A *reflective* programming language is one which supports such customizations, allowing the program to reason about its own execution state and alter behavior accordingly [23]. The context relation and context objects can be implemented using reflection.

Open Implementation (OI) principles advocate reflection to provide the programmer with more control, to avoid the inefficient work-arounds often caused from black-box languages and systems [13]. The context relation supports OI principles, providing the programmer with a better tool for controlling dynamic behavior.

5 Summary

To model and implement the evolution of behavior, we propose the *context* relation as an extension to the existing object-oriented model. Several design patterns were presented to demonstrate the use of the relation for facilitating evolution, evolving either the behavior of a single object or the behavior of collaborating objects.

As discussed, the context relation is meaningful at the analysis, design and implementation levels. It may also impact software testing. As inheritance and dynamic binding modified traditional program flow and subsequent testing models, the context relation may as well. If a base class is well tested, and a context class is added to alter its methods, how much retesting is required? Harrold, McGregor and Fitzpatrick present a hierarchical incremental testing (HIT) technique for reusing base class tests with derived classes[8]. The algorithm can be customized for testing the context relation.

Software engineering principles have long supported a black-box or strongly encapsulated form of programming, with a strict boundary formed around a software component, namely its interface. This is founded on the principle that hiding implementation details will facilitate evolution by reducing the impact of change. Such encapsulation is advocated in object-oriented programming by providing a boundary around a class, with access control mechanisms. It has been widely acknowledged that inheritance breaks encapsulation, and it is clear that the context relation may do so in the same manner as inheritance. If a derived class is allowed to access members inherited from a base class, changes in the base class may require maintenance of the derived class as well. The same maintenance relation exists between a base class and its context classes. As C++ tries to manage the maintenance effort by providing access control, the same may apply to the context relation.

It is clear that the shortcomings of the static class construct, namely that there is one implementation of an interface per object, often require elaborate work-arounds as demonstrated with the strategy pattern. The work-arounds themselves usually violate encapsulation, for example the *Strategy* class must be made a friend of the *Receiver* class. While a static class definition may be considered the encapsulation boundary, it can be argued that a more extended view of the class should be the boundary, namely the base class along with its contexts, representing the dynamic class definition.

An implementation of the visitor pattern using context attachment to calls has been developed in Java and STKLOS by Doug Orleans at Northeastern University.

Acknowledgments. We thank Walter Hürsch, Doug Orleans and Salil Pradhan for fruitful discussions. This work has been partially supported by the National Science Foundation under grant numbers CDA-9015692 (Research Instrumentation) and CCR-9402486 (Software Engineering), and by DARPA.

References

[1] Martín Abadi and Luca Cardelli. An imperative object calculus. In *Proc. TAPSOFT'95, Theory and Practice of Software Development*, pages 471–485. Springer-Verlag (*LNCS* 915), Aarhus, Denmark, May 1995.

[2] Constantin Arapis. Specifying Object Life-Cycles. *Object Management*, 1990.

[3] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification:X3J13 document 88-002R. volume 23. ACM Press, September 1988. Special Issue of SIGPLAN Notices.

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

[5] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, MA, 1990. ISBN 0-201-51459-1.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, Reading, MA, October 1994. ISBN 0-201-63361-2.

[7] Ira P. Goldstein and Dainel G. Bobrow. Extending Object-Oriented Programming in Smalltalk. In *Proceedings of the Lisp Conference*, 1980. Stanford, CA.

[8] Mary Jean Harrold, John D. McGregor, and Kevin J. Fitzpatrick. Incremental Testing of Object-Oriented Class Structures. In *Proceedings of the 14th International Conference on Software Engineering*, pages 68–80. IEEE Computer Society, 1992.

[9] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 169–180, Ottawa, 1990. ACM Press. Joint conference ECOOP/OOPSLA.

[10] Ian M. Holland. *The design and representation of object-oriented components*. PhD thesis, Northeastern University, 1993. http://www.ccs.neu.edu/home/lieber/theses-index.html.

[11] Ralph E. Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[12] G. Kiczales, J. des Rivieres, and D.G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[13] Gregor Kiczales, editor. internet publication (http://www.parc.xerox.com/PARC/spl/eca/oi/workshop-94).

[14] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94620-X.

[15] Karl J. Lieberherr and Cun Xiao. Minimizing Dependency on Class Structures with Adaptive Programs. In S. Nishio and A. Yonezawa, editors, *International Symposium on Object Technologies for Advanced Software*, pages 424–441, Kanazawa, Japan, November 1993. JSSST, Springer Verlag.

[16] Henry Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, number 11, pages 214–223. ACM, 1986.

[17] Cristina Videira Lopes and Karl Lieberherr. AP/S++: case-study of a MOP for purposes of software evolution. In *Reflection '96*, S. Francisco, CA, April 1996.

[18] Betrand Meyer. Tools for the new culture: Lessons from the design of the Eiffel libraries. *Communications of the ACM*, 33(9):68–88, September 1990.

[19] Sun Microsystems. *The Java Language: A White Paper*. http://www.javasoft.com.

[20] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-oriented composition rules. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 235–250, Austin, Texas, October 1995. ACM Press.

[21] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[22] J.J. Schilling and P.F. Sweeny. Three steps to logical views: Extending the object-oriented paradigm. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 353–361, New Orleans, LA, 1989. ACM Press.

[23] B.C. Smith. Reflection and Semantics in Lisp. In *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages*, pages 23–35, 1984.

[24] Antero Taivalsaari. Object Oriented programming with modes. Technical report, University of Jyvaskyla, Finland, 1991.

[25] David Ungar and Randall B. Smith. Self: The power of Simplicity. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, number 12, pages 227–242. ACM, 1987.

[26] Cun Xiao. *Adaptive Software: Automatic Navigation Through Partially Specified Data Structures.* PhD thesis, Northeastern University, 1994. http://www.ccs.neu.edu/home/lieber/theses-index.html.