

# Class Graph Views

Johan Ovlinger  
johan

Karl Lieberherr  
lieber

College of Computer Science  
Northeastern University  
360 Huntington Av.  
Boston, MA 02115, USA  
+1 617 373 2077  
@ccs.neu.edu

## ABSTRACT

An application typically has many conceptually separate pieces of behavior written over a common class graph. Each behavior makes different assumptions about the form of the class graph and the methods provided by the classes. The behaviors' overlapping assumptions make it difficult to modify the class graph in mid-development; each modification can break assumptions that must be tracked down and mended.

We propose a flexible system for providing separate views of the class graph to each behavior. Each view specifies only the details that the particular behavior makes assumptions about. By programming to the simplified view of the class graph, each behavior becomes more robust to modifications to the class graph. Inevitably, modifications to the class graph will require that broken assumptions be patched up, but these modifications need only be performed in the view specification, simplifying maintenance. A view can be applied simultaneously to several parts of the class graph, promoting code reuse.

A key innovation is that the same view is exported regardless of how it is applied. This is vital to being able to transparently layer views and thus minimize maintenance costs. A common organization is to have a general purpose lower level view provide a stable interface to a changing class graph and to build high level views tailored to specific behavior on top of that. This approach offers minimized maintenance costs (by virtue of all high-level views sharing the provided low-level interface), and simplified behavior (as each behavior sees a class graph tailored to its needs).

## Keywords

Evolution, Maintenance, Code Reuse, Roles, UML, Adapter Pattern

*Draft of paper submitted to ICSE 99.*

## 1 INTRODUCTION

A major problem in the evolution of software is a lack of robustness w.r.t. changes in the class graph. During the evolution of an application, more and more behaviors are written over the class graph of the application. Each behavior makes some assumptions and requirements about the class graph it is written over. Whenever the class graph is modified—something we expect to happen often in an evolving system— all the behaviors need to be reexamined to assure that all their requirements are fulfilled and assumptions unbroken. Fixing broken assumptions is often easy but tracking them down can be time consuming, and has been found to be a major factor in software maintenance workloads [17].

Code reuse is one predicted benefit of OO technology that has turned out to be more difficult to realize than foreseen. One reason for this failure is that it is difficult to separate behavior (that we want to reuse) from information about the class graph where it was developed (which we will need to modify) [10]. The upshot is that in order to reuse behavior, the programmer needs a deep understanding of how it fits into the host class graph. Thus, code reuse is easily dominated by identifying all of its assumptions. Accordingly, any system that makes identifying or organizing assumptions easy will promote code reuse.

Roles have been proposed by the OO community to model behavioral variations in objects, and views have been initially proposed by the OO-DB community to impose homomorphic schemas onto heteromorphic databases. We combine the two approaches, applying a set of cooperating roles as a unit to a class graph. This unit is separated from the class graph to a much greater extent than roles typically are; there is no way to directly reference the host object for a role, for example. This separation, combined with a flexible coupling to underlying class graphs, allows us to provide a stable interface to changing host class graphs.

## Terms and Concepts

A *class graph* is related to UML class diagrams [14]. Each node (class) of the graph is connected to the other nodes by edges representing their various static relation-

ships.

In our nomenclature, a role is a *view class* (as we reify the roles are separate classes), and a set of cooperating roles is a *view class graph*, or just a *view* for short. The purpose of a view is to present a stable interface to some host class graph. A view provides access to the behavior and structure of the host class graph, but does so in a stable way, abstracting away from such details as what class names are used in the host class graph, and what the host class graph structure is. In addition, a view can provide additional behavior, based on the behavior exported from the host class graph.

Views are connected to their host class graphs by *maps*. Maps insulate the behavior of the view from the details of the host class graph. The interface provided by a map to its view is static, as is the interface provided by the view to its users. Thus, a view may be compiled separately from the host it is mapped to. Mapping is performed at runtime, by telling the view what host class and concrete map to use. The concrete map is the implementation that provides the interface the view expects on top of the host class graph. Concrete maps are statically compiled, to gain static type checks, but applied at runtime, to gain flexibility.

### Contributions

The main contribution of this paper is to argue that code robustness and reuse rely on providing a static interface to behavior, regardless of changes to the host class graph, and that to do this, the behavior must be separated from the host class graph. Earlier approaches have focussed on inserting a view's structure and behavior – by automatic rewriting – into a host class graph, rather than providing a stable interface to it. The insertive approach thus cannot be as dynamic as ours; we can decide at runtime whether or how to apply a view to a host class graph, while inserted behavior is a static property. In both cases, behavior in the view is programmed to the class structure of the view, rather than the host class structure.

By having each view present an interface that is constant, regardless of the host class graph it is used with, views can be easily layered on top of each other without regard for underlying class graphs, and without having to recompile the views or the mappings used. This is in contrast with code-insertion approaches where the low-level view cannot provide a stable interface to high-level views. Since the exact interface of each layer in the insertive approach relies on the previous, all layers must be recompiled should the lowest level change.

Not needing to recompile layers of views is key to our claims of reducing code-maintenance. The benefit comes from the fact that the only aspect that makes assumptions about the host class graph is the is the map

between it and the first layer. Hence only that map needs to be maintained to fix any broken assumptions that a change to the host class graph might induce.

We foresee a three layer approach as being the most common. The bottom layer is the host class graph, which is analogous to the interior of a module, and is the layer that is presumed to be in flux. The first layer presents a static view of the changing host class graph, and is analogous to the interface presented by a module. The many behaviors that are to be added to the host class graph are layered on top of this layer. There will be a large number of such behaviors, and thus potentially a large number of views.

Dynamic information can play a part in deciding which implementation of behavior to use. A very small example<sup>1</sup> is a `Set`; if we expect to do many inserts and few extractions, then a `LinkedList` would be a good choice for a host class, while a `Hashtable` would be preferable if we will be performing many extractions and know how many elements are going to be inserted. Allowing these decisions to be made at runtime gains us significant flexibility.

### Organization and Presentation

The ideas of this paper are made concrete by the introduction of specification languages for views and maps. We use the concrete languages to demonstrate how views can be applied to robustness and reuse. Finally, we sketch the areas where more research is needed.

## 2 RELATED WORK

The authors of [15] focus on roles played by an object. They discuss classes offering different behavior through separate interfaces. The caller is able to decide which interface to use, which in turn influences behavior. This is related to our views, but only offers role-dependent functionality for one object. Their views are not types, and do not offer any of the reuse or robustness support we are looking for.

A view and host are related to multi-class versions of *control* and *entity* objects [5]. As behavior is moved out of the host class graph and into views, the host class graph is used mainly to model the data. The behavior is specified separately from the class structure, and mapped to the data at a later time. In general, behavior crosscuts class boundaries.

This is exactly the split suggested by Jacobson. Entity objects model structure and control objects model multi-class behavior. Single class behavior stays in the entity object. The analogy becomes a little strained when we try to add another layer of views into the pic-

<sup>1</sup>In this case, the behavior can be implemented using inheritance directly, but it serves as an indication of what kind of flexibility we expect to gain.

ture. This is because views can abstract both behavior and structure at one time.

### Subject-Oriented Programming

Subject-Oriented Programming's subjects [3, 11] are the opposite of views. SOP proposes that programs should be built from several private class graphs and their behaviors, combined to generate an executable program. Views, on the other hand, seek to separate out predefined subjects from an evolving program, using extracted subjects as the base for new behavior for the program.

By being able to generate new classes from the combination of subjects, SOP improves on naïve code-insertion's abilities to compose layers of subjects. However, the lack of declared interfaces for the result of combination seems to rule out runtime specification. Rather, subjects are combined at compile time using a rich composition language that can be statically extended with new composition primitives [12].

The lack of interfaces for subjects seems also to imply that a change to an underlying subject will necessitate all dependent subjects to be recomposed, or at least semantically re-checked. Views need to be recompiled to modify their behavior, but not to modify how they are applied, and a change in the host of a view will require only a modification in how that view is mapped to the modified host.

Subject-acquired behavior is accessed transparently, while view-acquired behavior needs to be accessed via the view classes, which in turn are managed by a maps. In practice this adds one level of delegation to calls to view-acquired methods. On the other hand, maps are first-class values in our approach, and as such can be passed like all other arguments to parameterize behavior.

The obvious way to avoid name clashes between subjects is to use alpha renaming. While transparent inside the subject, renaming implies that if two subjects wish to export methods with the same name, one of them must be renamed to something else. Separate views, by virtue of the separate name spaces provided by keeping each view in a separate package, side step all name clashes.

### Contracts

Ian Holland describes contracts and lenses in his thesis [4]. Contracts and lens objects are even more closely related to our views than SOP, being analogous to view class graphs and map objects, respectively. Holland's work is aimed at managing interfaces between separate pieces of behavior (contracts).

Holland chooses to use the insertive approach sketched in the introduction, rewriting contract specifications automatically to fit into the classes they are mapped to.

Thus, contracts do not provide stable interfaces to different host class graphs. Consequently, composition (layering) of contracts is not a trivial process like it is with views or SOP, and Holland introduces a special purpose operator to allow the system to support composition.

A second side-effect of renaming is that name-clashes between different contracts must be managed. Holland introduces Lenses to map between the names of methods and classes as known to the contract, and as the names that are in use in the host. Like maps, lenses are runtime objects. Lenses are less flexible than maps because no two overlapping lenses can be active for a single object at one time. This makes composition of behavior troublesome. To provide behavior from one lens to another on the same host object, an outside mediator must intervene in the general case; delegating the request and enabling and disabling the lenses on the host. Multi-threaded behavior is even more troublesome.

### Adaptive Programming and Plug-and-Play Components

Adaptive Programming (AP) [6] is another approach to writing robust software. It proposes that we write software using a version of the Visitor pattern [2] that is controlled by an external path description. AP tries to be as loosely coupled to the class graph as it can be without a translation layer, and is effective by virtue of the visitor only making local assumptions about the structure of the class graph. The result is a system that can automatically adapt to many changes in the class graph.

Three weaknesses make the approach unsatisfactory. Most importantly is that the system is only likely to automatically adapt to changes in the class graph. It is difficult to predict how and when it will fail to adapt. Secondly, while AP makes fewer assumptions about the structure of the class graph, those assumptions that it does make are scattered all over the program. Reusing adaptive programs thus requires that fewer assumptions be tracked down, but for large programs, this can still be a considerable chore. Lastly, since the visitor by design does not know how it has arrived at an object, implementing role dependent behavior is very awkward.

The concept of providing views was proposed in reaction to these weaknesses, and also to be a first step towards a module system for AP. In all three cases, it was deemed useful to be able to provide a stable view of a changing class graphs, tailored to the needs of the behavior.

Combining contracts [4] with Adaptive Programming [6], Adaptive Plug and Play Components (APPCs) [8] avoid the need for lenses by having each contract (component) export only one method. Like contracts, APPCs have their own inheritance hierarchy, managed with

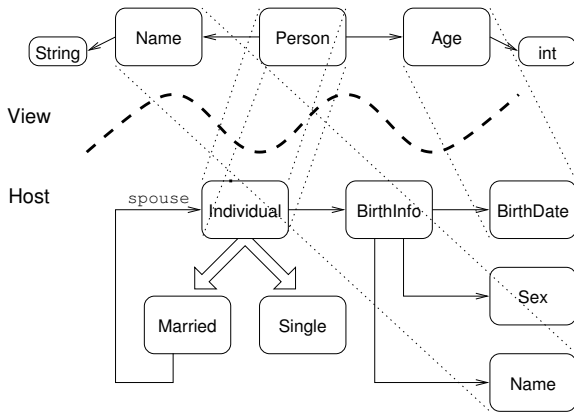


Figure 1: Views can hide the complexity of class graphs. Each behavior sees only the information it needs, in the way it wants to see it. In this case, the view is tailored providing simple views of a person’s Name and Age. The age is computed from today’s date and the BirthDate.

purpose-specific directives.

APPCs can be seen as a special case of views where the view classes are not exported outside the view. By not exporting the interface they apply to a host class graph, APPCs are restricted in how they may be composed. In order to use one APPC on the interface seen by another, that interface must be visible to the used APPC. The only way to manage that is by specifying the used APPC inside the APPC providing the interface. The used APPC cannot be exported.

The benefit to code-maintenance mentioned in section 1 depends on several behaviors sharing a lower-level view that insulated them from changes to the host class graph. APPCs cannot use this approach, and any change to the host class graph must be reconciled with each APPC separately.

### 3 VIEWS

We propose to provide each behavior with a view of the class graph that is tailored to the needs of the behavior. Views encode the requirements and assumptions that a behavior makes on the host class graph. Behavior and structure that is needed from the host class graph is clearly delineated from code that is provided by the behavior. A view provides a stable foundation for behavior; the view remains constant while the host class graph can change.

Behavior seldomly needs access to the full class graph (see figure 1). By providing a subset of the full class graph to work over, issues of navigation through the class graph are removed from the core behavior. This clarifies the code, and decouples it from the vagaries of the host class graph.

Views may be composed; each view provides a class graph that in turn can have views applied to it. If a number of behaviors make the same requirements of the class graph, they can all be layered on top of a common view that guarantees those requirements. Further layers can specialize the view to each behavior. By sharing—rather than duplicating—assumptions, layers of views simplify maintenance.

Each view is connected to the host class graph by a map. A view object can require behavior from the map; the map in turn can require behavior from a host object. The view and host class graph are completely unaware of each other; all interaction is mediated by the map. All assumptions about the host class graph are localized to the map; changes to the host class graph will often require changes to the map, but never to the view. Of course, if the behavior to be provided by the view changes, it will need to be updated.

Behavior in the view only mentions globally constant classes (like `java.lang.Vector`) or locally defined view classes (like `Person` in figure 1). We like to think of views defining sets of peer classes (domains). The domain of the view is conceptually separate from the domain of the host class graph, in that the view classes do not know about the host classes, and can make no assumptions about them.

This separation is maintained by translating objects from one domain to the other. If the map calls a method on a host object that returns an object from the host domain, the object will be translated to the domain of the view before it is returned to the caller (in the view domain). Thus, the view never refers to the host class graph.

The methods to perform the translation are automatically generated from correspondence information in the map. These methods are called automatically in many cases, although in some cases (black-box container objects like `java.util.Vector` for example), the translation methods must be manually invoked on each element. Future research will determine if the system can be made completely automatic, or if there are pathological cases that must always be dealt with manually.

A view may be applied (mapped) to a class graph several times, in different ways. Each map is separate from the others, even if it uses the same host objects. The layer of indirection and the translation of domains offered by the map makes this possible.

Behaviors can be reused simply by generating a suitable map object. This is substantially simpler than rewriting the behavior as the map class is concerned only with the translation between the two realms, and it does not have this behavior interspersed with methods that implement

behavior.

In the special case of a view with just one class being applied to a class graph, views act like mixins [9, 1]; They add some behavior to a class based on a well defined set of requirements from the superclass.

### What is Provided by a View

On a high level, a view provides a class structure and methods on classes in that structure. The classes have access to the full features of the core language. A view has a name, and this name is used to collectively group the classes it provides<sup>2</sup>.

A class in the view can require behavior from the host class that it is a view of. All behavioral requirements are mediated by the map, so it is more correct to say that the view class delegates behavior to the map. The map in turn can delegate to the host class to provide the required behavior, or provide the behavior itself. The map can only provide methods, not instance variables. However, as good OO style proscribes using getters and setters methods to access instance variables, the restriction is not problematic in practice.

All parts of a view class are accessed by getters and setters. By delegating getters and setters (via the map) to the host class graph, the object structure of the view mimics that of the host. This is how a view of a class graph is provided.

The view can remain static in the face of a changing host class graph by having the map compensate for changes to the host class graph. By finessing the code that performs the delegation in the map, the interface to the view stays the same while the host class structure can change. Of course, some state can be purely local (attached to the view class itself), but how to specify that is beyond the scope of this paper.

Thus, the view consists of three parts:

- The name of the view. A map will need to specify which view it is mapping to a class graph.
- The class graph, which consists of the classes in the view and their methods and parts. These classes and methods implement the bulk of the provided interface.
- The signatures of the methods that a view class requires from the map make up the rest of the provided interface (unless marked private, that is). The most common example of such methods are getter and setter methods for the parts of a view class. By having required methods be re-provided by default, views can be easily layered.

The syntax of the view is given in figure 2. Some aspects

<sup>2</sup>As we use Java as the host language, we use the view's name to determine the *package* of the classes of the view.

```
View      := ‘‘view’’ ViewName ClassGraph.
ClassGraph := ClassDef*.
ClassDef  := Part* ‘‘.’’ Methods.
Methods   := ‘‘{’’ Method* ‘‘}’’.
Method    := BehaviorMeth
           | MapMeth.
MapMeth   := ‘‘mapmethod’’ MethodSig.
BehaviorMeth := MethodSig MethodBody.
```

Figure 2: Syntax of view specifications.

have been ellided to save space. We build on the class graph handling facilities of Demeter/Java [7], borrowing and extending its textual format for class graphs. The name of the view and the structure of its class graph are readily identified. The behavior provided by the view is the combination of the `BehaviorMeths` and `MapMeths`. `BehaviorMeths` are implemented in the view while `MapMeths` are delegated to the map.

Figure 3 is a simple view. It offers the behavior of sorting a list of elements in place. The view, called `Sort`, implements most of the behavior locally. However, six methods are required of the map (and in turn, probably from the host class graph). These methods are declared as required with the `mapmethod` keyword.

### The Map Interface

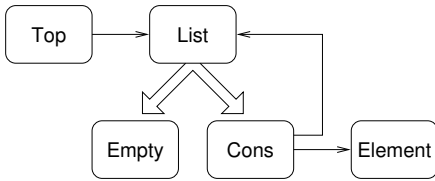
Each view specification makes requirements from a map class. The signatures of the methods required from the map are static, although we expect their implementations to be heavily dependent on the host class graph. To allow the view classes to be compiled independently of the map and the host class graph, an interface for map classes is automatically generated from the view specification.

Into the interface are put all the `MapMeth` signatures from the view specification, suitably modified to avoid name collision and to accept some extra arguments. In particular, the name of the class it is defined on is appended to the method name, and the host and view objects are prepended to the arguments. The interface generated by the `Sort` view is given in figure 4. Any map class to be used with `Sort` must implement `Sort_Map`.

### Layering Views

We can identify two forms of composition; a second view can be layered on top of the class graph of a first view, or a second view can make requirements that are fulfilled by the host object via behavior gotten from a first view.

Both forms are trivial, in that the system needs not offer support for them. The result of a view is a class with some behavior on it. We can use the class as a host for another view (thus building on top of the first), or we can have our host export the added behavior by



```

view Sort
Top = <list> List.
  mapmethod List get_list();
  mapmethod void set_list(List ls);
  void sort()
    { this.set_list(this.get_list().sorted()); }
List : Empty | Cons.
  abstract List sorted();
  abstract List insert(Element e);
  mapmethod List addToFront(Element e);
Empty = .
  List sorted() { return this;}
  List insert(Element e)
    { return this.addToFront(e); }
Cons = <elem> Element <tail> List.
  mapmethod Element get_elem();
  mapmethod List get_tail();
  List sorted()
    { List rest = this.get_tail().sorted();
      return rest.insert(this.get_elem()); }
  List insert(Element e)
    { Element head = this.get_elem();
      List rest = this.get_tail();
      if (e.comesbefore(head))
        return this.addToFront(e);
      else
        return rest.insert(e).addToFront(head); }
Element = .
  mapmethod bool comesbefore(Element other);
  
```

Figure 3: A simple view that can sort lists of elements.

writing a method to call the view object (thus letting this method be provided to another view).

#### 4 CONNECTIONS: CONCRETE MAPS

Views define unchanging class structures and behavior written to those structures. The host class graph is potentially in turmoil. It is up to the map to keep a view connected to its host class graph.

Maps are specified by implementing an interface generated from a view specification. The syntax is given in figure 5. We use a simple syntax rather than plain java to emphasize the fact that the system will generate some of the code for us.

Maps consist of a list of class correspondences, and bodies for all the methods required by the view. Each correspondence maps (hence the name) a class in the view (on the left) to one in the host class graph (on the right).

```

interface Sort_Map {
  List get_list_Top(Object host,Top view);
  void set_list_Top
    (Object host,Top view, List ls);
  List addToFront_List
    (Object host, Top view, Element e)
  Element get_elem_Cons(Object host,Cons view);
  List get_tail_Cons(Object host,Cons view);
  bool comesbefore_Element
    (Object host, Element view, Element e);
}
  
```

Figure 4: The map interface generated for the Sort view.

```

Map      :=  ‘‘map’’ MapName
          ‘‘of’’ ViewName ClassMap* .
ClassMap :=  ClassName [‘‘==’’ ClassName]
          TranslMeth*.
TranslMeth :=  MethodSig MethodBody.
  
```

Figure 5: Syntax for maps.

If a view class has no correspondence entry, its instances will not play a role for any host object. A correspondence `Foo == Bar` will generate two methods on the map class: `Foo makeFoo(Bar b)` and `Bar makeBar(Foo f)`. The two are used to translate between the two domains, instantiating new objects if necessary.

These translation methods are called automatically on the arguments and results of `TranslMeths`. This allows many maps to be specified in a very natural manner; however, sometimes the translation methods will need to be called explicitly. For example, a `Vector` of view objects will need to be translated manually by a map method to a corresponding vector of host objects before being passed on. Were this not done, a type error would occur when the host object tried to extract and cast an element from the vector. Due partially to a lack of type parametricity, this error can be caught only at runtime.

Somewhat misleadingly, the definitions of `TranslMeth` and `BehaviorMeth` look the same. Instead, it is with `MapMeth` from the view specification that `TranslMeth` is related. `TranslMeths` are the implementations of the `MapMeths` of the view the map is extending. They are interesting in that their `MethodSig` is given in the domain of the view, and the `MethodBody` in the domain of the host class graph.

Since the map consists of fully executable code, the map can perform arbitrary computation. We expect that commonly, map methods will just access variables and methods on the host object structure. However, since the map is written by the maintainer of the host class

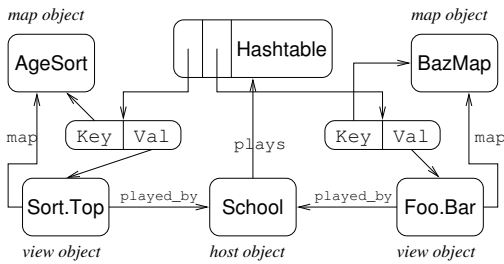


Figure 6: The object layout of a host object that plays two roles.

graph, and the view is potentially written by persons unknown, a map could be used to implement a simple minded security system<sup>3</sup>, or be used to provide mappings that vary depending on the state of the host or view.

The map specification language presented here is not the only way to specify maps. All maps will need to encode the class correspondence and provide the required behavior, but any of a number of techniques can be used. The one static requirement is that the class produced from the specification implements the interface of the view it is extending.

### Translation Between Domains

Only classes mentioned in the correspondence list are translated. This is mainly because we need the information from the correspondence to set up translation functions. However, this also allows instances of global classes (like `java.lang.String`) and primitives (like `int`) pass unchanged across the boundary.

Translating from the view to the real class graph is trivial; the `played_by` relationship [16] is implemented as an instance variable on the view object. To find the host object of a view object, one needs merely to look there.

Translating back is harder. We require that each host class that wants to participate in role playing must be primed to do so. Priming is primarily a performance optimization, and if the host class must remain inviolate, it is possible to mimic – transparently – the same behavior by storing information in the map. Each host class needs to be primed once only. View classes are primed automatically.

The priming includes adding a variable holding a **Hashtable** and implementing an interface to allow access to this variable. Indexed by map objects, the **Hashtable** holds all the view objects that a host object is associated with.

An example of the object linkage is illustrated in figure

<sup>3</sup>This mechanism will in no way be sufficient for any real security, but it is a neat idea.

6. The host, a **School** object, plays two roles; it is a **Sort.Top** and a **Foo.Bar**. Both roles are held in the table, indexed by their maps. The **School** object could play another **Sort.Top**, but as long as a different instance of **AgeSort** were used, it wouldn't interfere with the first<sup>4</sup>. Both view objects point directly to the host and the map; these are passed to the view object's constructor, and never changed.

An unfortunate side effect of the use of **Hashtables** is that view objects never get garbage collected before the host object. It is hoped that weak pointers and guardian objects will enable garbage collection of view objects when the map object is discarded.

### Using a Behavior from a View

The point of a view is to add behavior to a host class graph. So far, we have seen how to map behavior from the host to the view, but to be able to use the view-provided behavior in the host, the behavior must flow in the other direction.

Since the host class graph knows all about the view, this is considerably simpler than providing host behavior to the view. A large part of the complication of views is maintaining a studied ignorance of the host.

The name of a view class is derived by taking the name of the view as the package name of the class. Knowing the name of the class in the view (the role) that provides the method to be invoked, all the host needs to do is set up the role-playing relation (by providing a map object) and call the method. Since the host is the importer of the view, there is no loss of robustness by hard-coding the name of the view or map.

Figure 7 illustrates applying the **Sort** view's behavior to a hypothetical school class graph. We would like to sort the list of students. There are several possible sorting criteria: age, name, even the number of friends a student has can be used.

We decide to sort on the age, as this requires the least amount of coding. A map needs to be constructed to provide the needed behavior the view. We need to provide bodies for all the `mapmethod` signatures from the view specification (see figure 3). Any type that has a correspondence in figure 8 is automatically translated when passing into or out of the method body. For example, `addToFront` takes an **Element** as an argument, but the body of the method is passed a **student**. This is fortunate, as the constructor for **SCons** (which is not show, but takes a **Student** and a **SList**) would have balked at an **Element**.

The map specification is translated into a java class **Age-**

<sup>4</sup>The figure is of course simplified. A host object can play several roles in the same view, so in addition to the map object, the key to the table also includes the role name.

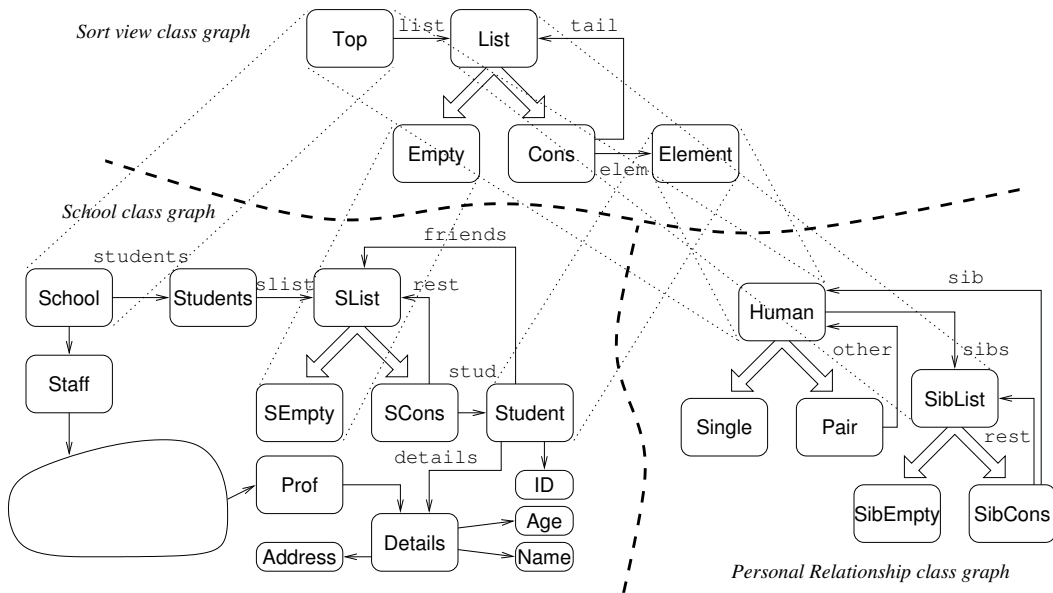


Figure 7: Two class graphs being mapped to the same view. Light dotted lines hint which classes play which roles (not all are shown).

```

map AgeSort of Sort
  Top == School
  List get_list()
  { return host.get_students().get_slist(); }
  void set_list(List ls)
  { host.get_students().set_slist(ls); }
  List == SList
  List addToFront(Element e)
  { return new SCons(e,host); }
  Empty == SEmpty
  Cons == SCons
  Element get_elem()
  { return host.get_stud(); }
  List get_tail()
  { return host.get_rest(); }
  Element == Student
  bool comesbefore(Element e)
  { return host.get_details().get_age()
    <
      e.get_details().get_age(); }

```

Figure 8: Mapping the Sort view onto a host class graph.

Sort. To use the sorting behavior, we first instantiate the map class. The map class provides factory methods that create view objects from host objects; this is called with the School object. Once the view has been set up, all we need do is call the `sort` method.

```

School sc = ....
AgeSort map = new AgeSort();
Sort.Top top = map.makeTop(sc);
top.sort();

```

## 5 ROBUSTNESS, REUSE, and ROLES: THREE Rs OF VIEWS

The paper thus far has presented concrete view and map specification languages. In this section we demonstrate the applicability of the approach to several toy examples. As is the case with many software methodologies, small examples are not as convincing as real world examples. The interested reader is invited to visit the home page [13] for some larger examples.

### Robustness and Flexibility

Let us change the organization of the host class graph and track the necessary changes to the map. These changes won't be any easier than they would be for normal OO programming, but they need only be performed once, and all the behavior provided by the classes of the view can remain unchanged.

Let us assume that the Age of a Prof is no longer relevant to the school. Thus, the `age` part has been moved off the Prof and to the Student.

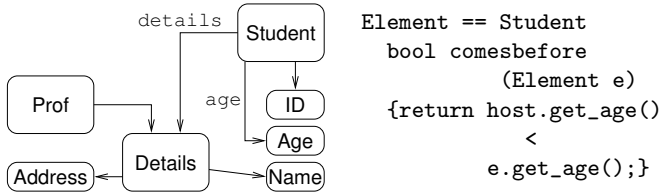
```

view SortSib of Sort
Top == Human
List == SibList
Cons == SibCons
Empty == SibEmpty
Element == Human
bool comesbefore
  (Element e)
{return
  e instanceof Pair;}

\\use view in a method
class Human {
void sortsibs()
{ SortSib map =
  new SortSib();
Sort.Top top =
  map.makeTop(this);
top.sort();}
}

```

Figure 9: Sorting some siblings



The changes that need to be made to the map are small; `comesbefore` now accesses `age` directly from a `Student`, rather than going via `Details`. The change may seem trivial, but we have found that a large component of the software maintenance burden of normal OO programs [17] is fixing small errors like the above.

The indirection through the map object offers a high degree of flexibility. Following the strategy pattern [2], we can pass different map objects to methods to vary their behavior. To sort the students by number of friends, all we need to do is to make a map class that is like `AgeSort`, but instead of looking at `age`, `comesbefore` would compare the length of the friends list of each student.

### Reuse

Because of the separation of the host and view class graphs, the only difference between reusing code and maintaining it is that for reuse, a map is written from scratch, rather than being modified.

To illustrate this, we apply the `Sort` view to class graph that models relationships between `Humans` and their `Siblings` (see figure 7). This is interesting as `Humans` play two roles in the view; they are both `Top` and `Element`.

A partial map is shown in figure 9. Of the four required methods we only specify the method `comesbefore`. In this case, we sort unmarried siblings to the front of the list. Hopefully, the example is close enough to the earlier ones that the reader will be to fill in the blanks. We don't show the use of the maps; these maps are used like the `AgeSort`; by instantiating the map, asking it to convert the host object to a `Top`, and then calling `sort`.

### Roles vs Classes: The Inlaws Example

A weakness of the visitor pattern [2] is that it has diffi-

culty dealing with role-dependent behavior. The typical dispatch mechanism uses the runtime type of the host to decide which visit method to invoke. However, it is not common for OO languages to support dynamic changes in an object's runtime type. Rather, role information is often passed as an argument to any role-dependent behavior, to be manually dispatched on. Passing arbitrary arguments is not supported by many implementations of the visitor pattern. Furthermore, the tangling of the explicit maintenance of the role relationships with behavior dependent on those relationships can easily lead to unclear programs.

We suggest that a better way to deal with roles is to provide a view where each role has been rendered as a distinct class. This separates maintenance of the roles from maintenance of behavior dependent on the roles. Furthermore, we are able to apply the whole range of OO programming styles to the roles, rather than the rather stilted one that tends to be adopted when mixing dispatch and behavior.

Figure 10 shows the simple host class graph for the role based inlaw implementation and the larger view class graph that is provided to the visitor. The view and map specification are lengthy, due the large number of classes in the view, and have been ellided. The interested reader is invited to peruse them online [13].

However, to give the flavor of the specification, we include a short snippet in figure 10. The snippet illustrates how the class correspondence automatically maintains the correct roles for hosts, based on the view's class graph. The map specifies that a the `get_spouse` method on `Married` is mapped to `get_other` on `Pair`. Using the return type of `get_spouse`, the system knows to translate the `Human` result from `get_other` to a `Spouse`. Thus, the roles are maintained automatically by the types specified in the map.

## 6 FUTURE WORK

There are still a number of areas that require more work before the system can be regarded as usable.

- We would like to be smarter about the boundary between classes that correspond to some host class and those that don't. Currently, it is up to the user to insert the correct translation method calls in the map to make sure the two class graphs stay separate.

An example is the class `java.util.Vector`. Being a global class, it is passed through the map w/o any change by the system. This is of course not good, as now the view has a `Vector` full of host objects, while the view class expects a vector of view objects. The only current solution is for the user to translate a vector of `Humans` to a new vector of `Siblings` manually.

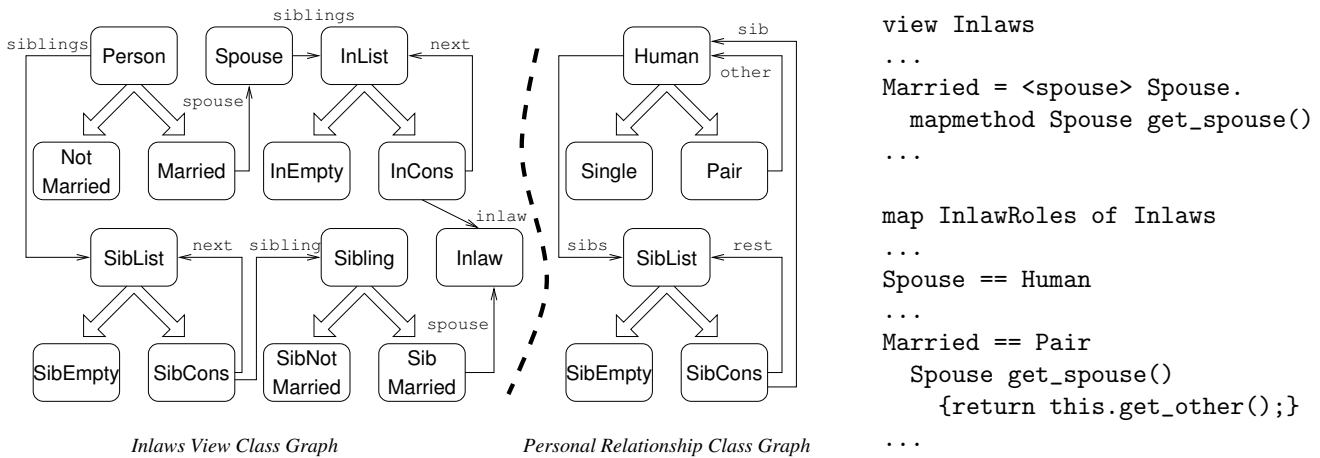


Figure 10: Roles are automatically maintained by the correspondance relationship. Here we specify that the `Human` pointed to by an `other` part plays the `Spouse` role.

In general, all container classes would need their own translation class. Future research will evaluate if it is possible to describe the behavior of container classes at a high level and automatically derive a translation function from that.

- We believe that that a map will never fail to translate an object, as long as the object is of a translatable class. A translatable class for a map is a class that occurs on the left or right side of a `==` in the map. A proof of this claim would be nice.
- We expect maps to display recurring patterns. We would like to improve on the specification language presented here to better capture common usage patterns.
- A view class can only correspond to a single host type (per map). If we want a view class to correspond to several host classes, the host classes must be made to implement a common interface, thus gaining a common type. If this turns out to be a common usage, this restriction should be lifted.
- Adaptive Programming [6] proposes techniques that would allow maps to automatically adapt to a wide set of evolutionary changes to a host class graph. Borrowing these techniques would make maps quite painless to use. Reuse would still have to be specified manually.

The system presented here is under development, and several areas need more research before the system becomes practical. However, it is our belief that the problems will prove surmountable.

## 7 CONCLUSION

This paper presents Class Graph Views as a way of providing stable interfaces to varying host class graphs.

Views improve on earlier approaches in that the high-level class graph is statically compiled and only changes

in response to modifications in the behavior it provides. Since the class graph offered by the view is static regardless of what it is applied to, it offers a stable interface to a wide variety of host class graphs.

Providing a stable interface (a view) to a host class graph is a win because the effort in maintaining a large number of behaviors written over this interface is proportional only to the change in the host class graph. By having two layers of views, with many high-level views layered on top of a low level view, all high-level views will benefit from any maintenance performed to the low-level view.

In a system that maps all behaviors down (through renaming) to the host class graph, no static interface is provided, so it is difficult to use the two layer organization described. Each layer can add behavior, but not influence the exported class structure. A change to the host class graph will need to be reconciled with map of the low-level view, and also all with all of the high-level views. The effort is proportional to the product of the number of views in the system and the change to the host class graph.

The paper suggests that adaptive programming might be better suited for use in the maps than in application behavior. Behavior will be written to tailored high level class graph views<sup>5</sup>, rendering adaptive programming useless, as it is aimed at writing code that is robust to changes in the class graph. However, this is exactly what is needed for maps. By using adaptive programming for maps, we expect that many changes to the host class graph can be dealt with automatically.

<sup>5</sup>There is no contradiction with the implication that we expect large numbers of behaviors to be written over the same view. We envision a base view that provides a stable class graph, and many specialized views layered on top of that to provide behavior specific views.

## REFERENCES

- [1] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and mixins. In *POPL '98*. ACM Press, 1998.
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [3] W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 411–428, Oct. 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.
- [4] I. M. Holland. *The Design and Representation of Object-Oriented Components*. PhD thesis, Northeastern University, 1993.
- [5] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.
- [6] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. ISBN 0-534-94602-X, entire book at [www.ccs.neu.edu/research/demeter](http://www.ccs.neu.edu/research/demeter).
- [7] K. J. Lieberherr and D. Orleans. Preventive program maintenance in Demeter/Java (research demonstration). In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997. ACM Press.
- [8] M. Mezini and K. Lieberherr. Adaptive plug-and-play components for evolutionary software development. Technical Report NU-CCS-98-3, Northeastern University, April 1998. To appear in OOPSLA '98.
- [9] D. A. Moon. Object-Oriented Programming with Flavors. In *Object-Oriented Programming Systems, Languages and Applications Conference*, in *Special Issue of SIGPLAN Notices*, pages 1–8, Portland, OR, 1986.
- [10] H. Ossher. Subject-oriented programming. Technical report, IBM Research, 1998. <http://www.research.ibm.com/sop/>.
- [11] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, pages 235–250, Oct. 1995. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 30, number 10.
- [12] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.
- [13] J. Ovlinger. Views homepage. <http://www.ccs.neu.edu/home/johan/views/index.html>.
- [14] Rational. The rational objectory process 4.1. <http://www.rational.com/support/techpapers/RationalObjectoryProcess4.1>.
- [15] J. Shilling and P. Sweeney. Three steps to views: Extending the object-oriented paradigm. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, volume 26, pages 353–361, 1989.
- [16] R. Wieringa, W. de Jonge, and P. Spruit. Using dynamic classes and role classes to model object migration. *Theory and Practice of Object Systems*, 1(1):61–83, 1995.
- [17] N. Wilde and R. Huitt. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering*, 18(12):1038–1044, December 1992.