# Implementing Reusable Object-Oriented Components

Yannis Smaragdakis and Don Batory
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
{smaragd, dsb}@cs.utexas.edu

**Abstract**[1]

*Object-oriented (OO) classes are generally not reusable because they are not meaningful in isolation; most classes only have meaning as members of cooperating suites of classes (e.g., design patterns). These suites usually arise in designs, but rarely exist as encapsulated entities in OO implementations. In this paper we present a method for directly mapping cooperating suites of classes into encapsulated C++ implementations. Our method is an improvement over the VanHilst and Notkin approach for implementing collaboration-based designs and constitutes a step towards more reusable (object-oriented) components.*

## 1 Introduction

The reuse benefits of object-oriented programming are often limited by the small scale of components (object classes). Larger scale (i.e., multi-class) components have appeared mainly as design entities (e.g., design patterns [8]) that lead to high levels of design reuse and maintainability. Ideally, we would like to obtain the same benefits at the implementation level. This is, in general, a hard goal. A way to achieve it is through straightforward mapping of design-level concepts into distinct implementation entities. Then the original modularity of a design is preserved and the implementation is easily maintainable under design changes.

Our work examines and relates two design approaches: Object-oriented *collaboration-based* designs (e.g., [6], [10], [11], [17]) and the *GenVoca* model [1]. Both emphasize viewing application components as a collection of responsibilities for various objects. Components are self-contained, thus yielding high degrees of modularization. Different aspects of an application can be abstracted into individual design entities, called *collaborations* (in collaboration-based designs) or *layers* (in GenVoca). These satisfy the *black box* property: access to their functionality can be obtained without reviewing or changing their internal implementation.

In this paper we present a method for mapping such design entities into encapsulated OO implementations. In this way the reusability benefits of high-level design translate directly to an implementation. Our implementation requires only standard programming language features (inheritance and parameterization). In effect, we are saying that certain component-based design methodologies are ideally suited for implementation in languages that support both object-orientation and generics (e.g., templates).

The usual way to map large scale OO designs into implementations is through a framework [12]. The work of VanHilst and Notkin [17][18][19] showed that frameworks are not flexible enough for the task, and suggested a templates implementation instead. Our approach follows a similar path. It yields, however, simpler implementations with significantly better reuse and maintenance properties. This is a result of clearly capturing collaborations and layers into black-box components (parameterized nested classes). Our research, therefore, directly addresses the need for future work stated in [17].

## 2 Collaboration-Based Designs and GenVoca

### 2.1 Collaboration-Based Designs

In an object-oriented design, objects are encapsulated entities that are rarely self-sufficient. Although an object is fully responsible for maintaining the data it encapsulates, it needs to cooperate with other objects to complete a task. An interesting way to encode object interdependencies is

1. We gratefully acknowledge the sponsorship of Microsoft Research, the Defense Advanced Research Projects Agency (Cooperative Agreement F30602-96-2-0226), and the University of Texas at Austin Applied Research Laboratories.

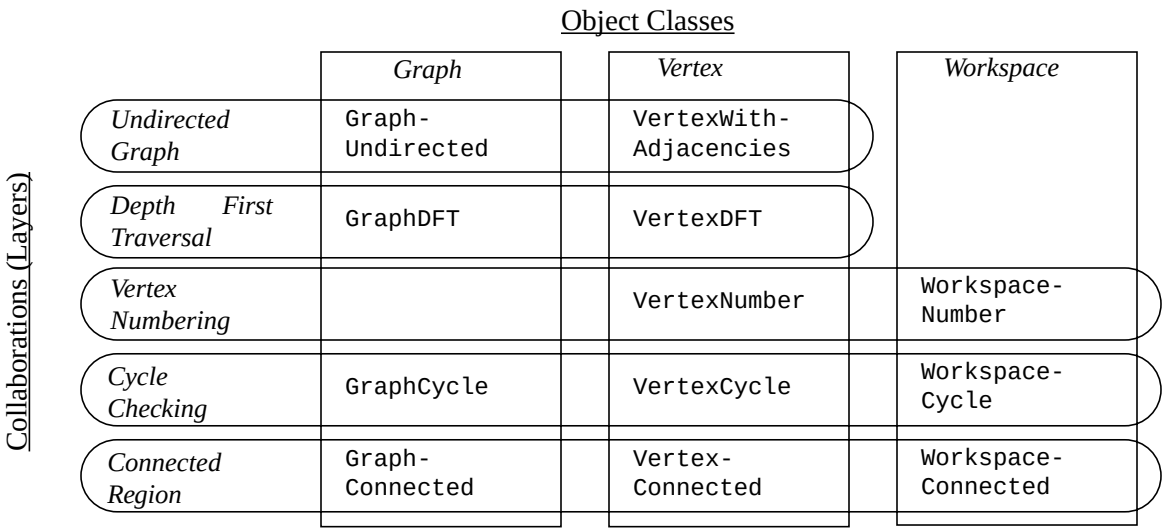


Figure 1: Collaboration decomposition of the example application: A depth-first traversal of an undirected graph is specialized to yield three different graph operations. Ovals represent collaborations (GenVoca layers), rectangles represent classes.

through collaborations. A *collaboration* is a set of objects and a protocol (set of allowed behaviors) that determines how these objects interact. The part of an object enforcing the protocol that a collaboration prescribes is called the object's *role* in the collaboration. In general, an object of an application may participate in several collaborations simultaneously and, thus, may encode several distinct roles. This means that collaborations commonly represent (mostly) independent aspects of object interaction.

In *collaboration-based design* we try to express an application as a composition of independently-definable collaborations. In this way, each object of an application will be a collection of roles describing actions on common data. Each collaboration, in turn, will be a collection of roles, and will represent relationships across its corresponding objects.

## 2.2 An Example

We will consider the graph traversal application that was examined initially by Holland [11] and subsequently by VanHilst and Notkin [17]. This application defines three different operations on an undirected graph, all implemented using a depth-first traversal: *VertexNumbering* numbers all nodes in the graph in depth-first order, *CycleChecking* examines whether the graph is cyclic, and *ConnectedRegions* classifies graph nodes into connected graph regions. The application has three distinct classes: *Graph*, *Vertex*, and *Workspace*. The *Graph* class describes a container of nodes with the usual graph properties. Each node is an instance of the *Vertex* class. Finally, the *Workspace* class includes the application part that is specific to each graph operation. For the *VertexNumbering* operation, for instance, a *Workspace* object holds the value of the last number assigned to a vertex as well as the methods to update this number.

A decomposition of this application into collaborations is relatively straightforward. One collaboration expresses properties of an undirected graph. Another collaboration encodes the specifics of depth-first traversals and provides a clean interface for extending traversals. That is, at appropriate moments during a traversal (the first time a node is visited, when an edge is followed, and when a subtree rooted at a node is completely processed) control is transferred to specialization methods that can obtain information from the traversal collaboration and supply information to it. For instance, to implement the *VertexNumbering* operation, we have to specialize the action performed the first time a node is visited. The action will assign a number to the node and increase the count of visited nodes.

Using this approach, each of the three graph operations can be seen as a refinement of a depth-first traversal and each can be expressed by a single collaboration. Figure 1 is reproduced from [17] and presents the collaborations and classes of our example application. The intersection of a class and a collaboration in Figure 1 represents the role prescribed for that class by the collaboration. A role encodes the part of an object that is relevant to a collaboration. For instance, the role of a *Graph* object in the "*Undirected Graph*" collaboration supports storing and retrieving a set of vertices. The role of the same object in the "*Depth First Traversal*" collaboration implements a part of the actual depth-first traversal algorithm.

Note that the design of Figure 1 does not define any particular composition of collaborations in an application.

It is really just a decomposition of a restricted software domain into its fundamental collaborations. Actual applications may not need all three graph operations. Additionally, a single application may need more than one operation applied to the same graph. This is accomplished by having multiple copies of the "*Depth First Traversal*" collaboration in the same design (each traversal will require its own private variables and traversal methods). We will later see examples where composing instances of the collaborations of Figure 1 will yield an actual application design.

The goal of a collaboration-based design is to encapsulate within a collaboration all dependencies between classes. In this way, collaborations themselves have no outside dependencies and can be reused in a variety of circumstances. The "*Undirected Graph*" collaboration, for instance, encodes all the properties of an undirected graph (pertaining to the *Graph* and *Vertex* classes, as well as the interactions between objects of the two). Thus, it can be reused in any application that deals with undirected graphs. Ideally, we should also be able to easily replace one collaboration with another that exports the same interface. For instance, it would be straightforward to replace the "*Undirected Graph*" collaboration with one representing a directed graph. Of course, simple interface conformance will not guarantee composition correctness — the application writer must ensure that the algorithms used (for example, the depth-first traversal) are still applicable after the change. The algorithms presented in [11] for this example are, in fact, general enough to be applicable to a directed graph. If, however, a more efficient, specialized-for-undirected-graphs algorithm was used (as is, for instance, possible for the *CycleChecking* operation) the change would yield incorrect results. See Section 4 for a more detailed discussion of the dangers involved in swapping reusable software components.

### 2.3 The GenVoca Model

Collaborations have received significant attention in the OO literature (e.g., [6], [10], [11], [13], [18]). Interestingly, the collaboration-based approach to software design is very closely related to the GenVoca model. GenVoca [1] is a model for constructing hierarchical software systems from reusable components. Each GenVoca component encapsulates a consistent refinement of multiple classes. Components are composable because they export and import standardized interfaces; this feature gives them "lego-like" qualities. Different systems of a domain are composed from these components. Although the GenVoca design model is fundamentally object-oriented, existing implementations do not rely on object-oriented techniques (inheritance). Instead, powerful parameterization (e.g., [14]) and software generators (e.g., [3], [15]) have been employed to produce target applications from components.

The spectrum of GenVoca implementations varies along two axes [4]: components may be either *compositional* or *transformational*, and either *dynamic* or *static*. Compositional components define the source code that an application will execute; transformational components define code that, when executed, will generate the source code that an application will execute. The dynamic/static attribute refers to the time of component composition. When components are composed at application run-time, they are dynamic. When composed at compile-time, they are static. The choice of how components should be implemented and when they should be combined is largely dependent upon the applications that are to be constructed.

The main concepts of GenVoca and collaboration-based designs are identical: Object classes are of secondary importance and components interrelate many classes. To build even one application class, however, many components need to be combined. The terminology is slightly different (for instance, GenVoca layers correspond to collaborations, GenVoca has no name for roles). The similarities led us to observe that static-compositional GenVoca components (e.g., P++ [14]) are a special case of collaboration-based designs. In particular, they correspond to collaboration-based designs where instances of the same role are never played by two different implementation classes. In essence, GenVoca fixes the names of concrete classes that can play a role. Such collaboration-based designs and very common and we will mainly focus our attention on them.

Consider again the example of Figure 1. Every collaboration entity (oval) can be viewed as a GenVoca layer. Composing layers is as simple as stacking them on top of one another. The object classes of the final application are determined by the refinements specified in each of the layers. Our goal is to find a concrete representation of layers (collaborations) in C++ and a method to compose them. Then, application classes become simple by-products of layer composition.

## 3 Implementing Collaboration-Based Designs

We now consider how collaboration-based designs are mapped to implementations. This has also been the research focus of VanHilst and Notkin [17][18][19]. An important point about our implementation approach is that it is applicable only to *static* compositional GenVoca designs (see Section 2.3). In other words, the way collaborations (GenVoca layers) are composed to define an application must *not* change at application run-time. This is

also true of the method of VanHilst and Notkin, although not explicitly stated.

### 3.1 Inheritance of Nested Classes

Our implementation of layers/collaborations uses C++ inheritance of nested classes to express layer composition. In C++, class declarations can be nested inside other class declarations. Nested classes behave in most respects (e.g., access control, scoping) just like regular members of a class. Interestingly enough, nested classes can also be inherited. Consider the following example:

```
class OuterParent
{  class Inner { ... }; };

class OuterChild: public OuterParent
{ };
```

In this case, `OuterChild` is a subclass of `OuterParent` in an inheritance hierarchy. Although no `OuterChild::Inner` class is explicitly defined, such a class does, in fact, exist (it is inherited from `OuterParent`). We will use this capability, combined with parameterization (C++ templates), to map collaborations into implementation components.

### 3.2 Mapping Primitives

In our method, each collaboration corresponds to a single parameterized class (we will subsequently use the term "collaboration" for the implementation of a layer/collaboration when no confusion can result). This class will contain nested classes that correspond to roles. More specifically, the general form of a collaboration representation is:

```
template <class NextCollab>
class ThisCollab : public NextCollab
{
public:
   class RoleForObject1 :
    public NextCollab::RoleForObject1
   { ... };

   class RoleForObject2 :
    public NextCollab::RoleForObject2
   { ... };
   ...
};                                              (1)
```

We will call the above template a *collaboration-component* and the implementation of a role a *role-member*. Thus, in (1), `ThisCollab` is a collaboration-component and `ThisCollab::RoleForObject1` is a role-member. Note the two mechanisms that we exploit: parameterized inheritance (parameterize a class with respect to its superclass) and type inheritance (inherit static entities such as nested classes).

Collaborations are composed by instantiating one collaboration-component with another as its parameter. The two classes are then linked as a parent-child pair in the inheritance hierarchy. The final product of a collaboration composition is a class `T` with the general form:

```
typedef Collab1 < Collab2 < Collab3 < ...
            < FinalCollab > ... >  T        (2)
```

That is, `Collab1`, `Collab2`, …, `FinalCollab` are collaboration components, "`<...>`" is the C++ operator for template instantiation, and `T` is the name given to the class that is produced by this composition[2]. The individual classes that the original design describes are members (nested classes) of the above components. Thus, `T::RoleForObject1` defines the application class `RoleForObject1`, etc.

Composition (2) has a direct counterpart in GenVoca. Applications are defined in GenVoca models as compositions of components called *type equations*. (2) has the exact form used in the GenVoca literature for type equations, except for syntax ("`[...]`" replaces "`<...>`"). Thus, (2) corresponds to type equation (3):

```
T = Collab1 [ Collab2 [ Collab3 [ ...
            [ FinalCollab ] ... ]           (3)
```

where `Collab1`, … are GenVoca components. This unification of collaboration-based designs and GenVoca offers important insights into both approaches. What GenVoca lacks is a way to recognize and express the concept of a layer in terms of OO designs — collaboration-based designs naturally reveal GenVoca layers and explain their encapsulations. On the other hand, collaboration-based designs have focused on the identification and design of the pieces of a collaboration (i.e. role classes), rather than on a broader, architectural framework where collaborations are primitive building blocks and applications are valid compositions of these blocks. It is this architectural perspective — that of building scalable families of applications from components, validating compositions, etc. — that GenVoca brings to collaboration-based designs.

---

2. We will use (without distinction) two C++ idioms for creating synonyms of complex instantiations. The first is using `typedefs`: `typedef A < B < C > >  D`. The second is using inheritance: `class D : public A < B < C > >`. The two forms are not equivalent: the first has the advantage of preserving constructors of component `A` in the synonym `D` (constructor methods are not inherited in C++). The second idiom is more cleanly integrated into the language (e.g., can be templatized, compilers create short link names for the synonym, etc.).

```
typedef DFT < NUMBER < DEFAULTW < UGRAPH > > > NumberC;
```

Figure 2a: A composition implementing the vertex numbering operation

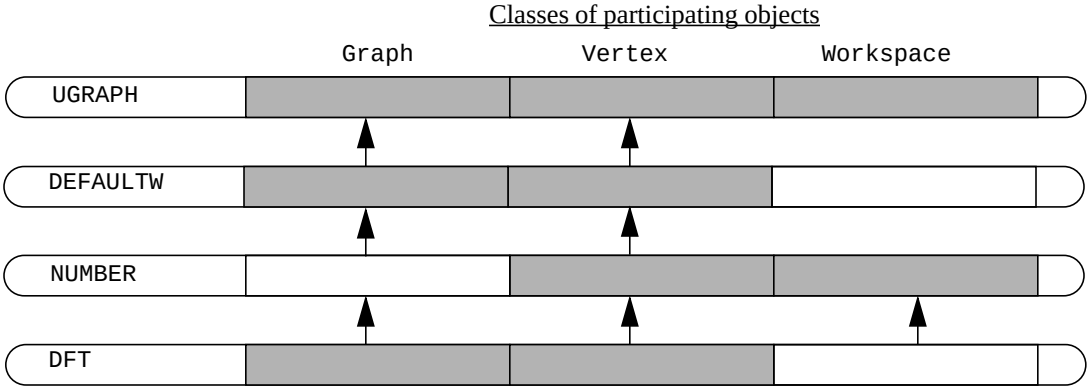


Figure 2b: Collaboration-components (ovals) and role-members (rectangles inside ovals) in the composition. Every component inherits from the one above it. Shaded role-members are those contained in the collaboration, unshaded are inherited. Arrows show inter-collaboration control flow (see Section 3.6 for their significance).

### 3.3 A Concrete Example

Consider again the graph traversal application of Section 2.2. Each collaboration will be represented as a collaboration component. *Vertex Numbering*, for example, prescribes roles for objects of two different classes: *Vertex* and *Workspace*. Its implementation has the form:

```
template <class NextCollab>
class NUMBER : public NextCollab
{
public:
  class Workspace :
     public NextCollab::Workspace {
  ... // Workspace role methods
  };

  class Vertex :
     public NextCollab::Vertex {
  ... // Vertex role methods
  };
};                                          (4)
```

Note how the actual application classes are nested inside the collaboration component. For instance, the roles for the *Vertex* and *Workspace* classes of Figure 1 correspond to `NUMBER::Vertex` and `NUMBER::Workspace` respectively. Since roles are encapsulated, there is no possibility of name conflict. Moreover, our approach relies on a standardization of role names. In this example the names `Workspace`, `Vertex`, and `Graph` are used for roles in all collaborations. Note how this is used in (4): Any class generated by this template defines roles that inherit from classes `Workspace` and `Vertex` in its superclass (`NextCollab`).

Other collaborations of our Section 2.2 design are similarly represented as collaboration components. Thus, we have a `DFT` and a `UGRAPH` component that capture the *Depth-First Traversal* and *Undirected Graph* collaborations respectively. To implement default work methods for the depth-first traversal we introduced an extra collaboration component, called `DEFAULTW`[3].

Consider now a simple composition — for instance, that producing the vertex numbering operation. The resulting application is obtained from the composition of Figure 2a. We will soon explain what this composition means but first let us see how the different classes are related. The final implementation classes are members of the product of the composition, `NumberC` (e.g., `NumberC::Graph` is the concrete graph class). Figure 2b shows the collaboration-components and their role-members as they are actually composed. Each component inherits from the one above it. That is, `DFT` inherits role-members from `NUMBER`, which inherits from `DEFAULTW`, which inherits from `UGRAPH`. At the same time, `DFT::Graph` inherits methods and variables from `NUMBER::Graph`, which inherits from `DEFAULTW::Graph`, which inherits from `UGRAPH::Graph`. It is this parameterized inheritance of nested classes that makes our approach so powerful. Note, for instance, that, even though `NUMBER` does not specify a `Graph` member, it inherits one from `DEFAULTW`. The simplicity that this design affords will be made apparent in Section 3.4, where we compare it with alternatives.

3. The introduction of `DEFAULTW` is an implementation detail, borrowed from the VanHilst and Notkin implementation [17]. Its purpose is to avoid dynamic binding by changing the order of composition. We discuss C++ specifics (such as, why composition order matters) in Section 3.6.

The interpretation of the composition in Figure 2 is straightforward: Every component is implemented in terms of the ones above it. For instance, the DFT component is implemented in terms of methods supplied by NUMBER, DEFAULTW, and UGRAPH. An actual code fragment from a method implementation in DFT::Vertex is the following:

```
for ( v = (Vertex*)firstNeighbor();
      v != NULL;
      v = (Vertex*)nextNeighbor() )
 {  edgeWork(v, workspace);
    v->visitDepthFirst(workspace);  }     (5)
```

The firstNeighbor, nextNeighbor, and edgeWork methods are not implemented by the DFT component. Instead they are inherited from components above it in the composition. firstNeighbor and nextNeighbor are implemented in the UGRAPH component (as they encode the iteration over nodes of a graph). edgeWork is a traversal refinement and (in this case) is implemented by the NUMBER component.

We can now more easily see how collaboration components are in fact both reusable and interchangeable. The DFT component of Figure 2 is oblivious to the components above it. Thus, the code of (5) represents a skeleton expressed in terms of abstract operations firstNeighbor, nextNeighbor, and edgeWork. Changing the implementation of these operations merely requires the swapping of collaboration components. For instance, we can create an application (CycleC) that checks for cycles in a graph by replacing the NUMBER component with CYCLE:

```
typedef DFT < CYCLE < DEFAULTW <
           UGRAPH > > > CycleC;
```

Note that (unlike other approaches — e.g., frameworks) no direct editing of the component is necessary and multiple copies of the same component can co-exist in the same composition.

As another example, the design may change to incorporate a different collaboration. For instance, operations could now be performed on directed graphs. The corresponding update (DGRAPH replaces UGRAPH) to the composition is straightforward (assuming that the algorithms are still valid for directed graphs as is the case in [11]):

```
typedef DFT < CYCLE < DEFAULTW <
           DGRAPH > > > NumberC;
```

Again, note that the interchangeability property is a result of the independence of collaborations. A single UGRAPH collaboration completely incorporates all parts of an application that relate to maintaining an undirected graph (although these parts span several different classes). The collaboration communicates with the rest of the application through a well-defined and usually narrow interface. It is exactly this same notion of component interchangeability that is the hallmark of the scalability of GenVoca designs [2].

### 3.4 The VanHilst and Notkin Approach

The original implementation of this example by Holland [11] used an application framework [12]. VanHilst and Notkin presented a parameterized implementation [17] but concentrated on representing roles — not collaborations. Further, they compared their technique to a framework implementation (which did not use parameterization). Their conclusions also apply to our work. Namely, compared to the framework implementation, both our approach and that of VanHilst and Notkin are more flexible (parent classes are not fixed), more efficient (avoid dynamic binding when not required), and allow multiple refinements in a given composition. In the rest of this section, we focus on the differences between our work and that of VanHilst and Notkin.

The VanHilst and Notkin approach concentrates on modeling roles as parameterized classes (C++ class templates). Role implementations are composed using standard C++ template parameterization to yield objects. Collaborations, however, are only implicitly represented in the final implementation. Compared to our method, this complicates the resulting code (roles are finer-grain components than collaborations and need to be composed explicitly). It also puts a burden on the programmer; roles have to be composed consistently when collaborations are introduced. Additionally, local design changes cannot easily be isolated, since collaborations are not explicitly represented as components.

We can see these benefits with an example. VanHilst and Notkin discussed in [17] a composition applying two of the graph refinements of Section 3.3 on the same graph. In particular, the graph class supports both the *CycleChecking* and the *VertexNumbering* operation. We select which of the two is to be performed on a certain graph object by casting an object pointer to the appropriate type and using it to call the depth-first traversal method. (An alternative would be to qualify method names directly, e.g., g->NumberC::Graph::Traverse()). The ability to compose more than one refinement (or multiple copies of the same refinement) is an advantage of the templates approach (both ours and the VanHilst and Notkin method) over frameworks implementations.

Our implementation of this example uses components very similar to those used by VanHilst and Notkin. Due to the compact representation of collaborations as nested class templates, however, our source code is much

```
class NumberC    : public DFT < NUMBER < DEFAULTW < UGRAPH > > > {};
class CycleC     : public DFT < CYCLE < NumberC > >              {};
```

Figure 3a: Our implementation of a multiple-collaboration composition. The individual classes are members of `NumberC`, `CycleC` (e.g., `NumberC::Vertex`, `CycleC::Graph`, etc.).

```
class Empty {};
class WS         : public WorkspaceNumber                {};
class WS2        : public WorkspaceCycle                 {};
class VGraph     : public VertexAdj<Empty>               {};
class VWork      : public VertexDefaultWork<WS,VGraph>   {};
class VNumber    : public VertexNumber<WS,VWork>         {};
class V          : public VertexDFT<WS,VNumber>          {};
class VWork2     : public VertexDefaultWork<WS2,V>       {};
class VCycle     : public VertexCycle<WS2,VWork2>        {};
class V2         : public VertexDFT<WS2,VCycle>          {};
class GGraph     : public GraphUndirected<V2>            {};
class GWork      : public GraphDefaultWork<V,WS,GGraph>  {};
class Graph      : public GraphDFT<V,WS,GWork>           {};
class GWork2     : public GraphDefaultWork<V2,WS2,Graph> {};
class GCycle     : public GraphCycle<WS2,GWork2>         {};
class Graph2     : public GraphDFT<V2,WS2,GCycle>        {};
```

Figure 3b: Same implementation using the VanHilst/Notkin approach. `V` corresponds to our `NumberC::Vertex`, `Graph` to `NumberC::Graph`, `WS` to `NumberC::Workspace`, etc.

shorter[4]. Our specification is shown in Figure 3a. A compact representation of a Van Hilst and Notkin specification is shown in Figure 3b. (A more readable version of the same code included in [17] is even lengthier).

Figure 3b makes apparent the complications of the VanHilst/Notkin approach. Each role-component can have an arbitrary number of parameters and can instantiate a parameter of other role-components. In this way, parameterization expressions of exponential (to the number of collaborations) length can result. To avoid this problem, the programmer has to explicitly introduce intermediate types to encode common sub-expressions. For instance, `V` is an intermediate type in Figure 3b. Its only purpose is to avoid introducing the sub-expression `VertexDFT<WS,VNumber>` three different times (wherever `V` is used). Of course, `VNumber` itself is also just a shorthand for `VertexNumber<WS,VWork>`. `VWork`, in turn, stands for `VertexDefaultWork<WS,VGraph>`, and so on[5]. In the case of a composition of *n* collaborations, each with *m* roles, the VanHilst and Notkin method can yield a parameterization expression of length $m^n$ (although this worst case does not exhibit itself in this example). Additional complications arise when specifying a composition: users must know the number and position of each parameter of a role-component. Both of the above requirements significantly complicate the implementation and make it error-prone.

Using our method, the exponential blowup of parameterization expressions is avoided. Every collaboration-component only has a single parameter (the collaboration above it). By parameterizing a collaboration A by B, A becomes implicitly parameterized by all the roles of B. Furthermore, if B does not contain a role for an object that A expects, it will inherit one from above it (as discussed in Section 3.3). This is the benefit of making the collaborations themselves be classes: they can extend their interface using inheritance.

Another practical advantage of our approach is that it encourages consistent naming for roles. As mentioned in Section 3.3, no name conflicts are possible among different collaborations: roles are encapsulated. Hence, instead of explicitly giving unique names to role-members, we have standard names and only distinguish instances by their enclosing collaboration-components. In this way, `VertexDFT`, `GraphDFT`, and `VertexNumber` become `DFT::Vertex`, `DFT::Graph` and `NUMBER::Vertex`, respectively.

In [17], VanHilst and Notkin questioned the scalability of their method. One of their concerns was that the composition of large numbers of roles "can be confusing even in small examples..." The observations above (length of parameterization expressions, number of components, consistent naming) show that our approach addresses this problem and does scale gracefully.

4. The object code is, as expected, of almost identical size.
5. Some compilers (e.g., MS VC++, g++) internally expand template expressions, even though the user has explicitly introduced intermediate types. This caused page-long error messages for incorrect compositions when we experimented with the VanHilst and Notkin method, rendering debugging impossible.

```
typedef DFT < REGION < DEFAULTW < UGRAPH > > > RegionC;
```

Figure 4a: A composition implementing the connected regions operation

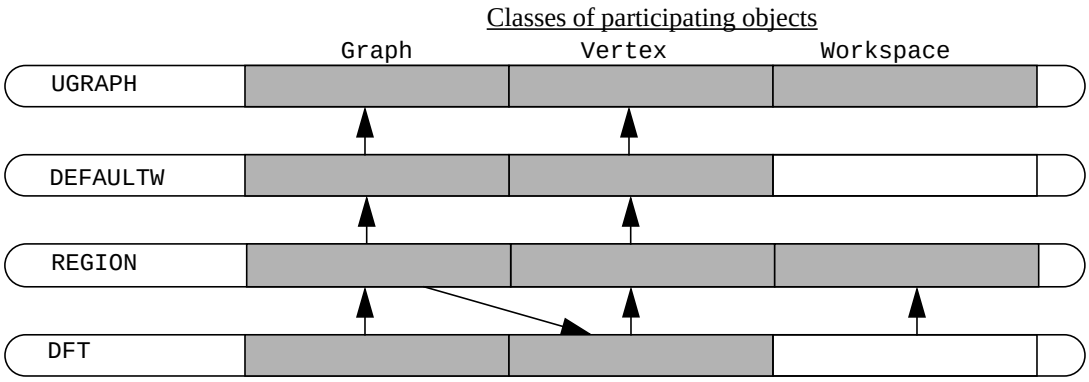


Figure 4b: Components in the *ConnectedRegions* composition. Arrows show inter-collaboration control flow.

### 3.5 Non-GenVoca Designs

As mentioned in Section 3.3, static compositional GenVoca designs are really a special case of collaboration-based designs. An important aspect of our implementation method is the fixed naming of role-members. This makes our method mostly applicable to GenVoca designs (i.e., collaboration-based designs where instances of the same role are always played by objects of the same class). To implement unrestricted collaboration-based designs, the super class of each role member must be specified via a unique parameter. This is precisely how VanHilst and Notkin implement their collaborations. An alternative technique, which can be used with our method, consists of introducing "renaming" components so that classes can play any role, regardless of their actual names. Although this solves the problem, the result may be cumbersome in practice.

Another way to solve the problem would be to employ a hybrid of our method and the VanHilst/Notkin approach (in a single application): The two kinds of components can be combined since our collaboration-components contain role-members analogous to the role-components of VanHilst and Notkin. Hence, when it is convenient to treat many roles as a single collaboration-component we can do so and gain in simplicity. When, however, we need to express role-components directly (so that they can be parts of different classes), we can employ the VanHilst and Notkin method.

The fully general form of collaboration-based designs is not often needed: all collaboration-based designs we have encountered (including that of Section 2.2 and all other experiments by VanHilst and Notkin) are, in fact, GenVoca designs. This suggests that the need for the above techniques may be rare in practice.

### 3.6 Discussion — C++ Specifics

There is a possible disadvantage in our approach of grouping role-components together. This has to do with the C++ binding policy, rather than the concepts that we have proposed. In C++ it is more costly to have a superclass invoke a method of its subclass than vice versa. The only way for a superclass to transfer control to subclass code is through a `virtual` method — which is bound dynamically. Thus, the ordering of collaboration-components in a composition is important. For best performance, it must be done in such a way that inter-collaboration method calls are made only to methods of classes "above" (see Figure 2b). Most of the time there is a natural ordering of collaborations present in the design that we can exploit.

In fact, similar concerns apply to the VanHilst and Notkin method. The `DEFAULTW` component of Section 3.3 is the counterpart of two role-components (`GraphWork` and `VertexRole`) introduced in [17]. These components encoded parts of the *Depth-First Search* collaboration that need to be called by the refinement code (e.g., the *Vertex Numbering* collaboration). Making this code a separate component enabled its placement above the refinement components (`NUMBER` in Figure 2b). This made all control-flow arrows (appearing in Figure 2b) point upwards. As a consequence, no dynamic binding was required for any methods in the application.

The essential difference of our approach is that we have to treat all roles in a collaboration as a unit. That is, we cannot decide to place one role "below" another individually but only together with all other roles in its collaboration. In this way conflicting role ordering requirements that were handled by the VanHilst and Notkin method may not be as straightforward in ours. Consider the example of Figure 4. It shows the components implementing the *ConnectedRegions* graph operation. The `DFT` component calls

methods in the REGION component to refine the traversal. This is represented in Figure 4b by the three "up"-arrows originating from roles in DFT. The REGION component, however, also needs to call a method in DFT (to learn if a node is marked during the traversal, so that the region count can be increased). This is shown as a "down"-arrow between REGION::Graph and DFT::Vertex in Figure 4b. This arrow creates a cycle in the collaboration-components control flow graph (REGION—DFT—REGION). No such cycle exists in the role-members graph: We can parameterize DFT::Vertex with REGION::Vertex and REGION::Graph with DFT::Vertex. This parameterization is possible in the VanHilst and Notkin method. With our approach, however, REGION has to be a superclass of DFT and the only way for REGION::Graph to transfer control to DFT::Vertex is through a virtual method declared in REGION::Vertex.

There are two observations to be made regarding this problem. First, if dynamic binding does turn out to be costly in an application, we can again resort to a hybrid of our method and the VanHilst/Notkin method (see Section 3.5). Second, in our examples we have found the simplicity advantages of our approach to far outweigh the cost of an occasional dynamic method binding. Note that in the example of Section 3.3, only a single virtual method needed to be added and only for the implementation of the *ConnectedRegions* operation. Our *VertexNumbering* and *CycleChecking* operations were exactly as efficient as their implementation in [17] (since all the differences concern compile-time parameterization issues).

This problem is entirely an artifact of C++ binding and parameterization. One can easily imagine a mechanism that supports static binding for method invocation from a superclass. Such a feature does not break the separate compilation abstraction and would seem to fit nicely in the C++ framework. Similarly more powerful parameterization facilities would also solve the problem. In fact, the parameterization mechanism of the P++ language [14] addresses this problem directly.

## 4 Related Work

There is a wealth of research work on design and code reuse through components. Instead of citing an enormous list of references, we chose to only position our approach relative to a few fundamental pieces of work and discuss others that we feel have influenced us significantly.

Biggerstaff [7] has pointed out that concrete component technologies are not suited for high levels of reuse. According to his *vertical/horizontal scaling dilemma*, components should incorporate large parts of an application's functionality (to obtain high levels of reuse) but when this happens they become more specific and, hence, less reusable. On the other hand, if components have large interfaces in an attempt to become more reusable, their performance deteriorates or the number of versions needed to support specific combinations of features increases exponentially. Our approach follows the *factored library* paradigm [7] and, as such, it partly addresses this concern. Components can be composed in exponentially many ways to adapt to many uses (scale horizontally) and form large parts of an application (scale vertically). Features that are not required don't participate in the composition and impose no overhead.

Goguen's distinction between vertical and horizontal parameterization of components (e.g., see [9]) is a particularly important one. We have used template arguments to express both kinds of parameters in our experiments. Our technique relies on passing all horizontal parameters as vertical ones to the bottom-most layer and letting them propagate upwards. Hence, our approach does not distinguish between the two kinds of parameterization, but we believe it is as good as could be hoped for, under the restrictions of commonly used programming languages. With extra linguistic support, horizontal parameterization could be straightforwardly expressible (as is, for instance, in P++).

Actually, the P++ language [14] is relevant to our work in more than one way. P++ extends C++ with syntactic features especially suited for the development of static compositional GenVoca components. Compared to the P++ approach, ours obtains the benefits of its reusable components without employing any special purpose tools. P++, however, has other advantages, such as a type system for composition parameters. We saw in Section 2.2 how composing different concrete components (for instance, DFT with a directed graph) may lead to incorrect programs. This is indicative of the dangers involved in reusing components. To reduce the potential for error we would like to have some method for expressing properties of components and checking them at parameterization time. C++ does not offer any mechanism to validate a template parameterization. The P++ type system for components, on the other hand, allows a parameterization to be constrained to only certain types of component parameters. In this way many erroneous compositions can be detected at parameterization time.

Type checking alone is insufficient. Additional properties often need to be represented to validate constraints and more expressive mechanisms may not be automated. For instance, the *contracts* formalism, (e.g., [10]) can be used to make invariants explicit at the design level. Errors can still be introduced in the specification of design invariants or in mapping them to an implementation. They will, however, be genuine mistakes — not the result of insufficient information about the component specifications.

## 5 Conclusions

Software development in OO languages will be ubiquitous in the foreseeable future. Nevertheless, basic OO languages mechanisms are not sufficient to obtain high degrees of reuse: The reuse of individual classes is minimal, because most classes have no meaning in isolation. It has been observed [2][6][11][18] that software building blocks are suites of reusable classes. It is these suites that we want to encapsulate both as design and as implementation building blocks of applications. The work of VanHilst and Notkin has showed how collaboration-based designs can be the basis for reusable OO software component technologies. Their implementation of OO components, however, had serious shortcomings which raised questions about the practicality and scalability of their approach. We have presented an alternative implementation method based on similar ideas and showed that it is practical and does scale. We rely on a sophisticated use of nested classes, parameterization, and inheritance to achieve encapsulations of "large-scale" OO components. We have also shown how collaboration-based designs relate to GenVoca, a scalable model of component-based software construction.

We are already conducting further work based on these ideas. In another paper [16] we examine the general OO language principles behind our technique and we look at ways to apply it to languages other than C++. We implemented a medium-size software project in Java using our component approach: The Bali compiler-precompiler generator [5] encapsulates language extensions as collaboration-components. The components are of substantial size (some with over a hundred classes) and they form the core of the Bali language extension mechanism. In all, we feel that our technique yields significant modularity and reusability benefits in an elegant fashion.

**Acknowledgments**

We would like to thank Michael VanHilst for his very helpful suggestions and for supplying us with his source code.

## References

[1] D. Batory and S. O'Malley, "The Design and Implementation of Hierarchical Software Systems with Reusable Components", *ACM TOSEM*, October 1992.

[2] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, "Scalable Software Libraries", *ACM SIGSOFT* 1993.

[3] D. Batory and J. Thomas, "P2: A Lightweight DBMS Generator". To appear in the *Journal of Intelligent Information Systems*.

[4] D. Batory, "Intelligent Components and Software Generators", *Software Quality Institute Symposium on Software Reliability*, Austin, Texas, April 1997.

[5] D. Batory, B. Lofaso, and Y. Smaragdakis, "JTS: Tools for Implementing Domain-Specific Languages", *International Conference on Software Reuse*, 1998.

[6] K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking", *OOPSLA 1989*, 1-6.

[7] T. Biggerstaff, "The Library Scaling Problem and the Limits of Concrete Component Reuse", *International Conference on Software Reuse*, 1994.

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

[9] J. Goguen, "Reusing and Interconnecting Software Components", *Computer*, Feb. 1986.

[10] R. Helm, I. Holland, and D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-Oriented Systems". *OOPSLA 1990*, 169-180.

[11] I. Holland, "Specifying Reusable Components Using Contracts", *ECOOP 1992*, 287-308.

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

[13] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet, "OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems", Journal of Object-Oriented Programming, October 1992, 5(6):27-41.

[14] V. Singhal, "A Programming Language for Writing Domain-Specific Software System Generators". Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, August 1996.

[15] Y. Smaragdakis and D. Batory, "DiSTiL: a Transformation Library for Data Structures", *USENIX Conference on Domain-Specific Languages (DSL 97)*.

[16] Y. Smaragdakis and D. Batory, "Implementing Layered Designs with Mixin Layers", to appear in *ECOOP 1998*.

[17] M. VanHilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs". *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

[18] M. VanHilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs". *OOPSLA 1996*.

[19] M. VanHilst and D. Notkin, "Decoupling Change From Design", *ACM SIGSOFT* 1996.