Using C++ Templates to Implement Role-Based Designs

Michael VanHilst and David Notkin
Technical Report 95-07-02
Department of Computer Science & Engineering
University of Washington,
Box 352350
Seattle, WA 98195-2350 USA
{vanhilst,notkin}@cs.washington.edu

July 10, 1995

Abstract

Within the object-oriented technology community, much recent work on design reuse has focused on role-based collaborations distributed across multiple objects. Many benefits can be derived by mapping role-based designs directly into implementations, including greater ease in maintaining the connection between designs and implementations under change, and the opportunity for code reuse along with design reuse. Current efforts in role-based designs don't provide these benefits. We provide a method for mapping role-based designs into implementation, preserving the design without unnecessary constraints on the design structures. Roles are represented as parameterized classes, where the parameters represent the types of the participants in the collaboration. Composition of roles is implicit in the binding of parameters to classes in the implementation. The bindings are created at compile time by type definitions that are separate from the role implementations. In this paper we focus on the use of templates and typedef statements in the C++ language as the supporting mechanisms.

1 Introduction

The objective of design reuse is to allow software developers to exploit parts of earlier designs that seem applicable to a new system or application. Successful design reuse is characterized by the ability to construct new applications by composing new parts with reused parts sculpted to fit the new context.

Within the object-oriented technology community, much recent work on design reuse has focused on collaborations of responsibilities distributed across multiple objects [3, 9, 5, 16]. "[N]o object is an island" [3, p. 2]: instead, each object has roles that it plays in collaboration with other objects. Designs are composed from groups of these collaborations, where each collaboration abstracts a task or concern in the design that may be appropriate for reuse. Roles are the elements of a collaboration, with each role encapsulating the responsibilities of a single participating class or object in a given collaboration. While the combination of roles played by a given participant may differ from one application to the

next, the details of a given role are stable and are thus reusable in other designs having the same concerns.

Carrying the benefits of role-based design reuse into implementation promises additional benefits. The two primary benefits of increasing the connection between design and implementation are that changes at the design level are easier to keep consistent with changes at the implementation level, and that it provides the opportunity for reusing sections of code that correspond to the reusable pieces of design.

Current efforts in role-based designs do not provide this benefit. Some approaches encourage implementation structures in which the design-level structures are not visible, weakening the connection between the two levels. Beck and Cunningham [3] and Reenskaug, et al. [17], for example, each allow an object's multiple roles to be combined at the design level, but, once combined, lose the identity of individual roles. Other approaches may maintain a clear association between design and implementation for the roles of one or two collaborations, but are difficult to apply to designs consisting of many collaborations. An example of this is Holland's use of frameworks [10]. In practice, approaches such as frameworks may constrain the designs that are used because developers often restrict themselves to designs that they believe can be implemented in a reasonably straightforward way.

We propose an alternative approach for mapping role-based designs into implementations, preserving the design without unnecessarily constraining the design structures that can be used. Roles are represented as parameterized classes, where the parameters represent the types of the participants in the collaboration. Compositions of roles are implicit in the binding of parameters to classes in the implementation. The bindings are created at compile time by type definitions that are separate from the role implementations. In this paper we focus on the use of templates and typedef statements in the C++ language as the supporting mechanisms.

Section 2 uses an example of Holland's to clarify role-based design. Section 3 uses this example to describe the framework approach to implementing role-based designs.[1] Section 4 introduces our alternative approach, again in the context of Holland's example, describing how we map role-based designs to C++ templates and how we instantiate these templates to create an application. Section 5 discusses related work, while Section 6 presents our discussion. Our conclusion appears in Section 7.

2 Role-Based Design: An Example

Holland's [10] design for graph traversal is a simple but useful example of role-based design. In addition, it provides a foundation for understanding and comparing the implementation structures defined using the framework approach as well as our template-based approach.

Holland's example defines a general depth first traversal over an undirected graph in adjacency list format and three specializations of that traversal.[2] In this case, "general"

[1] All code discussed in this paper is available by anonymous ftp from `ftp.cs.washington.edu`. The file is in the `pub/se/code` directory and is called `TR-95-07-02.tar.gz`

[2] The example represents relatively low-level design. However, it captures both the design-level and the implementation-level issues in a concrete way.

means that hooks are provided for doing various kinds of work as part of the traversal, but the work itself is left to the later refinements. Holland considers three refinements to depth first traversal: *NumberedVertices*, which numbers each vertex of the graph in the order it was visited by the traversal; *ConnectedRegions*, which numbers all the vertices of connected regions equally; and *CycleCheck*, which checks for cycles in the graph.

Considering role-based designs requires us to view them in two ways: one in terms of the participants or types that are involved, and the other in terms of the tasks or concerns of the design. The design of the depth first traversal has three participant types. The *Graph* type represents the overall graph and its interface to clients, the *Vertex* type defines the properties that each vertex must provide, and the *Workspace* type includes the common part of any work that must be passed along during a traversal. The example design can also be viewed as the combination of two separate collaborations: *UndirectedGraph* and *DepthFirstTraversal*. *UndirectedGraph* is concerned both with the representation of a graph in terms of vertices and edges and also with maintaining the graph's invariant properties (for example, that the graph is undirected). *DepthFirstTraversal* is concerned with visiting vertices in a particular order and providing support for doing work as part of the traversal. Each of the three refinements would be an additional collaboration.

As stated earlier, a role is a part played by a participant in a collaboration. Thus a role is a unit of design common to both views. The role of the *Vertex* participant in the *UndirectedGraph* collaboration supports the storage and retrieval of neighbors, while in *DepthFirstTraversal* its role supports being visited and traversing outgoing edges. The *Graph* participant also has a role in *UndirectedGraph*, which includes providing the client interface for adding vertices and edges and maintaining certain invariants. In *DepthFirst-Traversal*, the role of the *Graph* participant includes an interface to start the traversal, and it also ensures that every vertex gets visited.

It is the symbiosis of these two views of design that gives role-based design its power. The participant view captures conventional notions of object-oriented design; without it, connections to implementation would be difficult. The collaborative view captures cross-cutting aspects of designs, such as *DepthFirstTraversal*; without it, the relationships across objects are lost.

3 The Framework Implementation

Holland uses a framework [11] to implement the basic collaborations from the design. The participants in the collaborations are implemented as classes, while roles from a refinement are added to these participants by subclassing.

The base classes of the framework implement the roles from the *UndirectedGraph* and *DepthFirstTraversal* collaborations. `GraphDFT`, the *Graph* participant in *DepthFirstTraversal*, is implemented as a subclass of `GraphUndirected`, the *Graph* participant defined in *UndirectedGraph*. The graph participant in the *ConnectedRegions* refinement, called `GraphRegions`, is then implemented as a subclass of `GraphDFT`.

The hooks to allow refinements to add work to the traversal are implemented by calls to work methods in the traversal code of `GraphDFT`. `GraphDFT` defines a set of default work methods for these calls, and declares them as dynamically bound (`virtual`) so that

they can be overridden as necessary by a refinement subclass of `GraphDFT`. In addition, while `GraphDFT` passes a `Workspace` object as part of the traversal, the framework for *DepthFirstTraversal* leaves the definition of the `Workspace` class to the refinement. In these ways, the framework implementation of *DepthFirstTraversal* is able to support a variety of refinements for traversal work.

The simplicity of the framework implementation is attractive, but assorted problems remain, most of them related to naming. As one example, the names of participant classes appear in the framework code, which implicitly binds the implementation to a particular structure of composition; for instance, `GraphDFT` names `GraphUndirected` as its parent superclass, even though depth first traversal is equally meaningful for a directed graph. Holland's design level description of *DepthFirstTraversal* reflected this specificity by including the details that the graph was undirected and used adjacency lists. As Johnson and Foote point out, "reusing the edifice that ties the components together is usually possible only by copying and editing it" [11, p. 26].

A more serious problem is that it is difficult to apply more than one refinement to a given framework. Since the implementations of both refinements would assume the framework as a base, editing is again required to achieve the required unambiguous combination of common and separate pieces in a single class. This situation also has a naming problem due to the use of dynamic binding, because it always binds to the most specialized definition of a method. For example, suppose an application needs two different traversals and thus two separate `GraphDFT`s on the same *UndirectedGraph*, each doing a different set of work. If both refinements of the *Graph* participant override the same work method, the first traversal's `GraphDFT` will call the work method meant for the second `GraphDFT`.[3]

4 A Template-Based Approach

Our alternative approach uses class templates to represent and implement roles. Participants to these roles are passed as parameters to the templates. A role's template parameterizes the type of every participant to which it refers, including itself. This use of parameterization enables us to overcome the naming problems of the framework approach. In addition, because class names are bound to the parameters of the role implementations by a separate set of type definitions, we treat as separate concerns the implementation of roles and the composition of those roles.

The remainder of this section applies these techniques to Holland's graph example, making the comparison between the approaches easier to understand.

4.1 Role Implementation

Roles are implemented using C++ templates. For example, Figure 1 shows the C++ code to implement the `GraphDFT` role as a class template. The three parameters, `V`, `WS`, and `G` respectively represent the class names `VertexDFT`, `Workspace`, and `GraphUndirected` in

[3]Holland observed this problem and proposed a somewhat complicated "lens" mechanism that required the different refinements to be instantiated between traversals [10].

```
template<class V, class WS, class G>
class GraphDFT : public G {
public:
  bool depthFirst(WS* workspace) {
    V* v;
    for( v = firstVertex(); v; v = nextVertex() )
      v->setNotMarked();
    initWork(workspace);
    for( v = firstVertex(); v; v = nextVertex() ) {
      regionWork(v, workspace);
      v->visitDepthFirst(workspace);
    }
    return finishWork(workspace);
  }
};
```

Figure 1: A template implementation of a role.

the original framework implementation.[4] The template is instantiated to a class by binding the three parameters to the desired participants. The part of our approach that does this binding is explained in the next subsection.

The framework version of `GraphDFT` declared three virtual work functions and provided the default implementations. Dynamic binding was needed to support their refinement by a descendent of the `GraphDFT` class. In the template version, these hooks for refinement are not declared in `GraphDFT`, but are instead provided, together with the default implementations, by a separate class template, called `GraphDefaultWork` which must be included as part of the superclass of the `GraphDFT` class. (The process of refinement by inserting ancestors will become clearer in the discussion of role composition.) The `firstVertex` and `nextVertex` methods are also assumed to be part of the superclass. If the superclass does not provide these methods, an error is reported at compile time.

4.2 Role Composition

Roles are composed at the implementation level to form the classes of the participants. We separate composition from role implementation using a series of type definition statements. These statements are analogous to the type equations used by Batory [2] to specify the composition of classes in his data structure generator. In addition, they form a kind of textual description of the structure of the entire design of the application, analogous to a class dictionary in Demeter [13].

The details of this aspect of the approach depend on some C++-specifics about class

[4] Participant classes are formed, in part, by combining their role implementations through inheritance. In `GraphDFT<V,WS,G>`, the `G` parameter represents its immediate superclass in the *Graph* participant inheritance hierarchy. Although the code fragments in the paper do not show it, there is a default ancestor class.

```
typedef WorkspaceDefault                WS;

typedef VertexAdj                       VGraph;
typedef VertexDefaultWork<WS,VGraph>    VWork;
typedef VertexDFT<WS,VWork>             V;

typedef GraphUndirected<V>              GGraph;
typedef GraphDefaultWork<V,WS,GGraph>   GWork;
typedef GraphDFT<V,WS,GWork>            Graph;
```

Figure 2: Type definitions for the general case.

naming and templates. In C++ a class template is not a class, but rather a specification of a family of classes, all sharing the template's implementation, but having different bindings for the parameters. A template class is a class defined by binding other classes to the parameters of a class template. Thus, if `VertexAdj` is the name of a class and `template<class V> GraphUndirected` is the declaration of a class template, then `GraphUndirected<VertexAdj>` is the name of a template class. A class of that type will be instantiated when the full template class name appears in the type declaration of a variable. For the `GraphDFT` class template of Figure 1, the type names bound to `V`, `WS`, and `G` become part of the name of any class instantiated from the `GraphDFT` class template. Because these types would differ if the role was in a different part of the application, a role can be reused in the same application without causing a class name conflict.

Figure 2 shows a set of typedef statements that defines the *Workspace*, *Vertex*, and *Graph* classes for the general depth first traversal algorithm. Although `WorkspaceDefault` has no parameters, we chose to define `WS` as a short alias for it because it is used in several places later on. We next define the *Vertex* participant type by combining its roles from the *UndirectedGraph* and *DepthFirstTraversal* collaborations. `VertexAdj` is the implementation of the *vertex-with-an-adjacency-list* role in the *UndirectedGraph* collaboration. `VertexDefaultWork` appears next in the order of inheritance for the *Vertex* class, followed immediatedly by `VertexDFT`. As we stated before, we make the `Work` roles ancestors of the `DFT` roles so `DFT` can call the work methods they define using only static binding. The *Graph* participant is similarly constructed.

Binding template classes to template classes may create long names. While typedefs allow us to define shorter aliases, the use of typedefs is not merely a matter of saving keystrokes. Using typedefs to give names to types at intermediate levels and aliasing the type names makes it easier to change the types of participants when modifying the implementation, because the type only has to be changed in the definition of the alias.

Figure 3 shows the set of type definitions for the classes in the *NumberedVertices* refinement of the *DepthFirstTraversal*. *NumberedVertices* defines its own `Workspace` role and an additional role for the *Vertex* participant, but it needs no additional role for the *Graph* participant. In refining *DepthFirstTraversal*, we replaced `WorkspaceDefault` with `WorkspaceNumber`, and inserted the `VertexNumberWork` role between `VertexWork` and

```
typedef WorkspaceNumber                 WS;

typedef VertexAdj                       VGraph;
typedef VertexDefaultWork<WS,VGraph>    VWork;
typedef VertexNumberWork<WS,VWork>      VNum;
typedef VertexDFT<WS,VNum>              V;

typedef GraphUndirected<V>              GGraph;
typedef GraphDefaultWork<V,WS,GGraph>   GWork;
typedef GraphDFT<V,WS,GWork>            Graph;
```

Figure 3: Type definitions for the *NumberedVertices* refinement.

`VertexDFT` in the `Vertex` inheritance hierarchy. In this way, the calls to work routines in `VertexDFT` will call the methods defined in `VertexNumberWork` instead of the ones in `VertexDefaultWork`. Without the same flexibility in ordering inheritance, the framework approach needed dynamic binding to accomplish this.

The type definitions for composing the *ConnectedRegions* and *CycleCheck* refinements are similar to those for *NumberedVertices* in Figure 3. Both also have a role for the `Graph` participant which is inserted between `GraphDefaultWork` and `GraphDFT` in the `Graph` inheritance hierarchy.

How could we adapt this design and associated implementation for an application that needed a directed graph? A *DirectedGraph* collaboration could share the same interfaces as those in *UndirectedGraph*, and even use the same implementation of `VertexAdj`. We could then reuse this implementation for a depth first traversal on *DirectedGraph* simply by switching `GraphUndirected` to `GraphDirected` in the definition of `GGraph`, without touching any template code. Both versions could in fact coexist in the same application with no conflict.

4.3 Composing Multiple Collaborations

Composing roles from one collaboration with roles from different collaborations is an important part of role-based design. Equally important is the ability to compose roles from similar collaborations and from repeated uses of the same collaboration. Our approach permits mapping these design structures directly into the implementation. To demonstrate, we add a second *DepthFirstTraversal*, together with the *CycleCheck* refinement, to the composition in the previous section.

The `typedef` statements for this new composition are shown in Figure 4. This example highlights the importance of specifiable types for these kinds of compositions. For example, `GraphUndirected` must manage vertices that are appropriate for both traversals, but the first traversal must see a *Vertex* that results in the correct method and variable bindings for that traversal. The issue is handled by binding the *Graph* roles of the first traversal to the `V` *Vertex* type, while binding the remaining roles of the *Graph* participant to the more

```
typedef WorkspaceNumber                 WS;
typedef WorkspaceCycle                  WS2;

typedef VertexAdj                       VGraph;
typedef VertexDefaultWork<WS,VGraph>    VWork;
typedef VertexNumber<WS,VWork>          VNumber;
typedef VertexDFT<WS,VNumber>           V;
typedef VertexDefaultWork<WS2,V>        VWork2;
typedef VertexCycle<WS2,VWork2>         VCycle;
typedef VertexDFT<WS2,VCycle>           V2;

typedef GraphUndirected<V2>             GGraph;
typedef GraphDefaultWork<V,WS,GGraph>   GWork;
typedef GraphDFT<V,WS,GWork>            Graph;
typedef GraphDefaultWork<V2,WS2,Graph>  GWork2;
typedef GraphCycle<WS2,GWork2>          GCycle;
typedef GraphDFT<V2,WS2,GCycle>         Graph2;
```

Figure 4: Type definitions to compose multiple refinements and traversals.

refined `V2` *Vertex* type. Although the flexibility is greater than in the framework approach, all references are handled using only static binding.

This composition does not require the editing of any of the role implementations. The client creates a graph instance of type `Graph2`, but casts it to a `Graph` type when it calls the first, *NumberedVertices*, traversal.

5 Related Work

A number of authors have addressed either the composition of parts of objects to form the objects of an application, or the use of type parameters to achieve flexible implementations.

Batory et al. [1, 2] have developed a number of code generators that compose sets of object features, called factors, to form complete classes. Batory's type equations compose the factors into classes in much the same way our typedef statements compose the roles of a participant to form its type. Batory has done much work identifying compositions that may be semantically incorrect, and finding optimizations to apply to the generated code. Batory's approach requires careful domain analysis to identify common interfaces, followed by the development of specialized generators. Our approach views any application and its refinements as its own domain, and requires no special generator, thus avoiding much of Batory's analysis and preparation overhead.

Batory's work has focused on broadly used domains, such as data structures. The general-purpose nature of these domains encourages both careful domain analysis and also the development of specialized generators. With comparatively low overhead, our approach

provides much of the same flexibility for arbitrary applications.

Lieberherr's Demeter [12, 14, 13] is a system that also addresses extensible structures in object-oriented development, and combines design reuse with source code reuse. Like our approach, Demeter minimizes and localizes dependencies on the class structure, and avoids overspecification. Demeter separates the specification of structure from the implementation. But Demeter requires a special development environment and uses annotations in the implementation of each collaboration to specify the composition. We have chosen to restrict ourselves to using the features found in a widely used language and to separate the implementation code from the specification of composition.

Harrison and Ossher [7, 8] have been working on composing fragments of object-oriented programs under the name subject-oriented programming. Their approach uses a special dispatcher that "merges" classes by rerouting method calls. Their system allows a variety of merging semantics. Our approach stays within the semantics of inheritance and aggregation and does not require any special, extra-language mechanisms.

Musser, Stepanov and Lee [15] have recently gained attention in the C++ community for their work on libraries of generic container classes and iterators. Their work has helped push builders of compilers for C++ to provide better support for parameterized types. They do not use parameterized inheritance and have not considered using templates as a structuring mechanism for entire applications. But some of the ways in which they use template programming could be applied alongside those suggested here.

Bracha [6, 4] addressed the use of types with parameterized superclasses, called mixins. His work dealt largely with extending languages to support mixins, but also addressed merge semantics and module composition. We view the parameterized types within the context of a tool for structuring designs, while his focus was at a lower level.

6 Discussion

Implementations produced using our approach are relatively efficient, largely because of our use of static binding as opposed to dynamic binding. We timed closely matched implementations of the graph example in both the framework and the template versions, and found that the template version ran about twice as fast (e.g., 28 seconds vs. 61 seconds CPU time, for 10000 traversals of a graph with 800 edges on a PC using Linux/gcc 2.6.3). The object code produced using templates was comparable in size to the framework implementation (16423 bytes vs. 16988 before stripping, 8192 for both after stripping)

The approach is quite portable as well, primarily because it depends only on the availability of features—templates and typedefs—that are widely available in a commonly available language—C++. To apply the approach, then, requires an understanding of role-based design, an understanding of how to map these designs using C++ templates and typedefs, and the presence of a C++ compiler. The use of C++ and C++ templates and typedefs raises a number of questions that we have not yet answered, including:

- How hard is it to debug templates like these? Our initial experience is that it is not a serious problem, since C++ provides a degree of static typing for template classes. Some compilers do not give as useful error messages as we would like (e.g., gcc), although others are already quite good (e.g., IBM's xlC). Without debugger

support for aliases, the long type names add a degree of complexity to debugging. Although we have found some approaches to address this, the question is open for larger applications.

- Do current compilers provide sufficient support for templates? So far we have observed that the template support provided by C++ compilers varies widely. It is not unusual to find templates that work in some compilers and fail in others. We expect it to take some time before this technology is stable enough to use in our style across essentially all compilers.

- Can our use of C++ templates be applied to other object-oriented languages that have parameterized type mechanisms (e.g., Eiffel generics)? We see no a priori problems, but neither have we tried this yet.

The principal open question we face is, does the approach scale? That is, for larger, more complicated role-based designs, can we construct implementations that are reusable in the implementations of similar (but not identical) role-based designs? One technical problem we face in answering this question is how to manage the composition of large numbers of roles using typedefs, which can be confusing even in small examples like Holland's graph. Determining whether or not the approach indeed enhances design and implementation reuse across large applications is an important, and complicated, question to answer.

7 Conclusion

Our template-based approach to implementing role-based designs provides a relatively strong mapping between the design and the implementation. In particular, roles have an explicit representation, as templates. This contrasts with approaches that transform designs into collections of objects where the roles cannot be easily discerned. Our approach does not, however, capture collaborations as clearly as it does roles. The initial step towards addressing this shortcoming is to define a design-level notation that explicitly represents collaborations. Overall, our mapping from design to implementation is good, although not yet ideal.

Designs consisting of multiple collaborations are implemented as directly as those consisting of one or two collaborations. This contrasts with framework-based implementation approaches, which have natural mappings only to implementations of a single collaboration and one refinement. Developers, then, need not be as constrained in their designs to ensure straightforward implementations.

Finally, the amount of work required to accommodate many changes at the implementation level is proportional to the effort in changing the design level. Conventional object-oriented programming styles, as well as framework approaches, use naming in a way that tends to complicate the way design changes are realized in the implementation. In even simple systems, such as Holland's graph example, the potential benefits of our approach become apparent.

Acknowledgments

We would like to thank George Forman for his helpful feedback.

References

[1] D. S. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4):355–398, October 1992.

[2] Don Batory, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Scalable software libraries. In *Proceedings of the First ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 191–199, 1993.

[3] Kent Beck and Ward Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 1–6, 1989.

[4] Gilad Bracha. *The programming language JIGSAW: Mixins, Modularity and Inheritance*. PhD thesis, University of Utah, 1992.

[5] Eric Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[6] Bracha. Gilad and William Cooke. Mixin-based inheritance. In *Proceedings of the 1990 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 303–311, 1990.

[7] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the 1993 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 411–428, 1993.

[8] William Harrison, Harold Ossher, Randall B. Smith, and David Ungar. Subjectivity in object-ooriented systems workshop summary. In *Addendum to the Proceedings of the 1993 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 131–136, 1994.

[9] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Proceedings of the 1990 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 169–180, 1990.

[10] Ian M. Holland. Specifying reusable components using contracts. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 287–308, 1992.

[11] Ralph Johnson and Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, pages 22–35, June/July 1988.

[12] Karl J. Lieberherr and Arthur J. Riel. Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, pages 8–22, August/September 1988.

[13] Karl J. Lieberherr and Cun Xiao. Minimizing dependency on class structures with adaptive programs. In *Object Technologies for Advanced Software: Proceedings of the First JSSST International Symposium*, pages 424–441, 1993.

[14] Karl J. Lieberherr and Cun Xiao. Object-oriented software evolution. *IEEE Transactions on Software Engineering*, 19(4):313–343, April 1993.

[15] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software Practice and Experience*, 24(7):623–642, July 1994.

[16] Trygve Reenskaug and Egil P. Anderson. System design by composing structures of interacting objects. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 133–152, 1992.

[17] Trygve Reenskaug, Egil P. Anderson, Arne Joggan Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Naess-Ulseth, Gro. Oftedal, Anne Lise. Skaar, and Pal Stenslet. OORASS: Seamless support for the creation and maintenance of object-oriented systems. *Journal of Object-Oriented Programming*, 5(6):27–41, October 1992.