

Using Role Components to Implement Collaboration-Based Designs

Michael VanHilst and David Notkin
Department of Computer Science and Engineering
University of Washington
PO Box 352350
Seattle, Washington 98195-2350 USA
{vanhilst,notkin}@cs.washington.edu

Abstract

In this paper we present a method of code implementation that works in conjunction with collaboration and responsibility based analysis modeling techniques to achieve better code reuse and resilience to change. Our approach maintains a closer mapping from responsibilities in the analysis model to entities in the implementation. In so doing, it leverages the features of flexible design and design reuse found in collaboration-based design models to provide similar adaptability and reuse in the implementation. Our approach requires no special development tools and uses only standard features available in the C++ language. In an earlier paper we described the basic mechanisms used by our approach and discussed its advantages in comparison to the framework approach. In this paper we show how our approach combines code and design reuse, describing specific techniques that can be used in the development of larger applications.

1 Introduction

The notion of collaborations is well accepted in object-oriented design. In collaborations, groups of objects cooperate to perform a task or to maintain an invariant. In the collaboration view, a *role* is the part of an object that fulfills its responsibilities in the collaboration. In most design methodologies, roles are an ephemeral concept, existing

briefly, if at all, between the description of collaborations and the specification of classes. They do not generally exist as identifiable components of the implementation.

In our approach to object-oriented implementation, roles are an important key to code reuse and adaptation. Compared to classes, roles encapsulate fewer decisions and are thus more stable with respect to evolution. We provide a method to implement roles as source code entities—specifically, class templates defined in a stylized way—and compose them into classes using separate specification statements—classes defined in terms of instantiations of those class templates. Our approach improves code reuse and adaptability, and overcomes a number of other limitations found in more traditional approaches to implementation. In this paper we demonstrate how a design that originates with a collaboration-based methodology can be implemented directly using role components.

In the second section we describe collaborations and roles more fully and discuss their significance in object-oriented design. In the third section, we present the basic details of our method of implementing roles as source code components.¹ In the fourth section we describe the design of a container recycling machine similar to the one presented in Jacobson, et al. [11]. In the fifth section, we show how to implement that design using role components. The sixth section discusses some of the differences between our implementation and

¹A lengthier discussion of the details is available in an earlier paper, in which we compared our approach to the use of frameworks [17].

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

the implementation presented by Jacobson, et al., with respect to possible changes. We also describe some of our experiences with a much larger application, a telescope imaging system. The seventh section contains a discussion of related work, while the last section contains a concluding summary.

2 Collaborations, Roles, and Collaboration-based Designs

A collaboration is a set of objects together with obligations on and relationships among those objects. Collaborations are often used to model sequences of message passing and state changes derived from use-case-like scenarios in the requirements analysis. In so doing, collaborations offer a view that complements the static view of class and inheritance structures. Collaborations are often informal [2, 3, 11, 20], but they have also been formalized in contracts [10], given a notation [14, 15], and associated with framework implementations [7, 13].

A collaboration can address only the parts of objects needed to participate in a particular task, concern, or pattern. In this view of collaborations, a role specifies a part of a single object that participates in a particular collaboration. Collaborations may be seen, then, not so much as collections of objects, but as collections of roles.² An object participates in a collaboration by having a role in that collaboration.

An object that participates in several collaborations may have several roles, one from each of the collaborations. The relationship between objects and roles has a number of significant features. Just as a collaboration may be viewed as a collection of roles, an object, or its class, may also be viewed as a collection of roles. So, a role is a unit of design common to both views.

Any object that has the proper role may *play* that role in a collaboration. Thus the same spec-

²Some methodologies make a distinction between collaborations of objects and collaborations of roles. Where that distinction might be significant, we will always mean the latter.

ification of a collaboration may apply to different sets of objects, provided the corresponding objects play the same roles. In this generality we can see the value of roles in reuse. If we can reuse a collaboration, we can reuse its roles, even if the classes of the objects that contain them are different. Viewed from an evolutionary standpoint, if we wish to replace an existing object in a collaboration, the new object may belong to a different class, but it must still play the same role. Conversely, if we change or replace a collaboration, the roles of its participant objects in other collaborations do not change.

A few approaches to object-oriented development already use collaborations and role-like decompositions to achieve better design reuse [7, 13]. Our strategy is to extend this kind of reuse to code by implementing the roles in the design directly as encapsulated source code components. Classes are then literally compositions of role components.

Because we want to implement roles directly, we need a design methodology in which roles are still identifiable late in the design process. We have found this to be possible with a number of existing collaboration-based design methodologies, including use-cases [11], responsibilities [19], and role models [13].

3 A Method for Implementing Roles

To implement the roles from a collaboration-based design, we need a mechanism that gives role components the same properties as roles in the design. In a design, when a role is added to an existing class, it refines or extends the interface of that class. Thus inheritance is a logical glue for composing roles into classes. But roles can be composed with a variety of other roles in a variety of classes, so the inheritance in the implementation must be delayed until composition takes place. Similarly, since roles in the design place no restrictions on the types of objects with which they collaborate other than that they play

the appropriate role, bindings to types for all collaborator references must also be delayed.

We have found that we can satisfy all of these properties using parameterized types in general and C++ class templates in particular. There are two key language structures that we use to explicitly define and implement roles and then to compose those roles into classes.

For each role, we define a separate class template that is parameterized by each of the collaborators. For example, a father's role in a two parent household might be defined in part as:

```
template <class ChildType,
          class MotherType,
          class SuperType>
class FatherRole : public SuperType {
    ChildType *child;
    MotherType *mother
    ...
};
```

The `ChildType` and `MotherType` parameters indicate that the father role will collaborate with a child role and a mother role that are played by objects of yet-unknown types. The `SuperType` parameter is used in every role definition in our approach, since every role is part of some yet-unknown class. A husband's role in a marriage collaboration might be similarly defined, in this case collaborating only with a wife role, as shown here:

```
template <class WifeType, class SuperType>
class HusbandRole : public SuperType {
    WifeType *wife;
    ...
};
```

We compose roles into classes by instantiating templates like these, binding the template parameters to the specific classes that play the roles. An instantiation of the `FatherRole` might, for example, appear as:

```
class Father2Class : public FatherRole
    <ChildClass, MotherClass, Father1Class>{};
```

This says that the `Father2Class` includes the `FatherRole`. It also says that specific classes, `ChildClass` and `MotherClass`, play the child and mother roles in the collaboration with the `FatherRole`. `Father1Class` is the class that we extended to get `Father2Class`. The `Father1Class` might be defined in terms of `HusbandRole` and its instantiation:

```
class Father1Class : public HusbandRole
    <MotherClass, emptyClass> {};
```

This statement defines `Father1Class` as a collaborator with `MotherClass` playing the wife role in a marriage collaboration. The `emptyClass` (essentially a default base class) parameter simply indicates that the `Father1Class` is a base class.

This somewhat abstract description of our implementation method will be clarified by example in Sections 5 and 6; more details also appear elsewhere [17].

By writing different template instantiations, we can define different combinations of roles without modifying the role definitions themselves. This allows us to handle various orders of inheritance, to include the same role twice, and to add additional roles before, after, or in between the roles currently defined. Section 6 discusses the ways in which these kinds of flexibility support our strategy for application development and evolution.

4 The Recycling Machine Design

To demonstrate our implementation approach, we begin with a modified version of Jacobson, et al.'s collaboration-based design for a container recycling machine [11]. This example defines a vending machine that takes empty beverage containers and issues a receipt for the deposit value of the containers. The front of the machine has three slots (one each for cans, bottles, and crates), a button to request a receipt, a slot to issue a receipt, and a lighted panel marked "NOT VALID."

The interaction between the recycling machine and a customer combines two separate activities, *Adding Item* and *Print Receipt*. In the *Adding*

Item scenario, when a customer inserts an empty beverage container into one of the slots, a customer total and a daily total for that item type are both incremented by the system. In the Print Receipt scenario, when the customer presses the receipt button, the customer total is calculated and the following information for each item type is printed on a receipt: name, number deposited, unit deposit value, and total deposit value. Finally the sum of the deposit values is printed, and the receipt is issued through the slot. The customer totals are then cleared, and the machine is ready for a new customer.

An analysis of the recycling machine's detailed requirements yields two extensions to the basic Adding Item scenario, *Validate Item* and *Item Stuck*. These extensions provide additional or alternative sequences of events to the original base scenario. In the Validate Item scenario, when an item is inserted, it is measured by the system. The measurements are used to determine if the container should be accepted for a deposit refund. If it is not accepted, no totals are incremented, and the NOT VALID sign is highlighted. In the Item Stuck scenario, before incrementing any counts, the system checks to see if the item has become stuck. If the item is stuck, the system sounds an alarm and no totals are incremented.

As in other collaboration-based approaches, objects are identified at this point in the design process so we can assign roles to system parts. We identified the following objects needed to support the recycling machine's scenarios: *CustomerPanel*, *DepositReceiver*, *DepositItem*, *ReceiptBasis*, and *InsertedItem*. Figure 1 shows a diagram of the object structure. *DepositItem* is the abstraction for each item type. Its responsibilities are to validate items of its type and to maintain a daily total. It must also know its name and deposit value. *ReceiptBasis* is the abstraction for a single customer session. Its responsibilities are to keep the list of customer totals by item type and to print the appropriate information on a customer receipt. To fulfill its responsibilities, the *ReceiptBasis* maintains a list of *InsertedItem* objects. Each *InsertedItem* keeps the customer's

total for a specific container type and has a reference to that type's *DepositItem* object. The *CustomerPanel* interfaces to the devices and displays of the front panel, while the *DepositReceiver* is the main control object for the system's interactions.

Once the objects are known, the scenarios can be restated as collaborations among participant objects. For example, the Adding Item scenario can be restated as: When a customer inserts an empty beverage container into a slot, the *CustomerPanel* signals the *DepositReceiver* with the slot type. The *DepositReceiver* identifies the corresponding *DepositItem* and signals the *ReceiptBasis*. The *ReceiptBasis* adds an *InsertedItem* to its list if one of that type does not already exist. The *ReceiptBasis* tells the *InsertedItem* to increment the customer's total. The *DepositReceiver* then tells the *DepositItem* to increment the daily total. The interaction diagram for this collaboration is shown in Fig. 2.

5 The Recycling Machine Role Implementation

Our strategy for implementing applications is to create a source code component for each role in the design. The goal is to make it easier to reuse code among related applications, and to support a wider range of future adaptations with less impact on the existing implementation code. In this section we describe the process of implementing the recycling machine design using the method described briefly in Section 3. Our intent is to illuminate where this process and the resulting implementation differ from more traditional approaches.

In the collaboration-based design from the previous section, identifying roles is straightforward. The interaction diagram used to describe the Adding Item collaboration shows the operations that each object needs to fulfill its role in that collaboration. We simply collect the operations for a particular object and determine which attributes those operations use. CRC or class cards [2, 20]

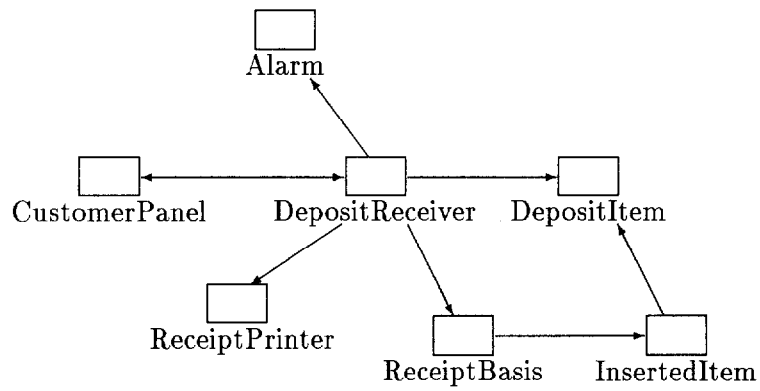


Figure 1: Block diagram of objects in the recycling machine design. (Arrows indicate calls relations.)

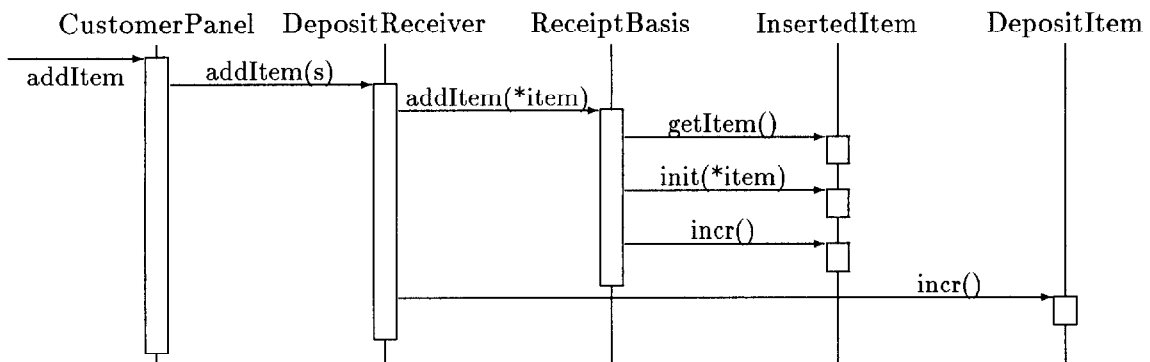


Figure 2: Interaction diagram for Adding Item collaboration.

	CustomerPanel	DepositReceiver	ReceiptBasis	InsertedItem	DepositItem
Linked List			<i>list</i>	<i>*next</i> setNext(*next) getNext()	
Adding Item	addItem(s)	<i>item[N]</i> , <i>*receipt</i> addItem(s)	addItem(*item)	<i>number</i> , <i>*item</i> getItem() init(*item) incr()	<i>total</i> incr()
Validate Item	invalid() addItem(s,l,w,h)	addItem(s,l,w,h)			<i>l,w,h</i> accept(l,w,h)
Item Stuck	isStuck()	addItem(s)			

Figure 3: Roles/responsibilities matrix for part of the recycling machine. (Names followed by parentheses are method names. Names in *italics* are attribute variable names.)

```

class DepositReceiver1Class : public
    DRAddingItemRole<ReceiptBasisClass,DepositItemClass,emptyClass> {};
class DepositReceiver2Class : public
    DRItemStuckRole<CustomerPanelClass,AlarmClass,DepositReceiver1Class> {};
class DepositReceiver3Class : public
    DRValidateItemRole<CustomerPanelClass,DepositItemClass,DepositReceiver2Class> {};
typedef DepositReceiver3Class DepositReceiverClass;

```

Figure 4: Definition statements to compose the `DepositReceiver` class.

can be used in a similar manner, if the responsibilities are annotated with the names of the collaborations to which they belong.

But we need more information to construct our application, especially when trying to compose several roles to form a single class. First, the same operation, or attribute, may be defined for more than one role. If so, we must determine if the duplicate operation is to be shared, repeated, or overridden. Depending on the results of this analysis, some roles may need to be subdivided. Second, where one collaboration extends another, we need to identify the calls between roles within the same class. These may be implied, but not shown, in interaction diagrams. Third, the use of data structures may also be implied, but not shown. The `ReceiptBasis` class in the recycling machine example uses a linked list, in which `InsertedItems` might function as nodes. Finally, we need to determine the order in which to compose the roles.

To aid in the process of answering these questions, we have found the roles/responsibilities matrix, adapted from business management [6], to be a useful tool. Figure 3 shows a roles/responsibilities matrix for part of the recycling machine involving the Adding Item, Item Stuck, and Validate Item collaborations.³ In the matrix, rows represent collaborations, while columns represent classes. The internal cells of the matrix represent roles.

In the column for the `CustomerPanel` class, there are two distinct `addItem()` methods. In the

³To save space, the *Alarm* and *Printer* columns have been left out.

design, the `addItem()` method from the Validate Item extension replaced the one from the Adding Item collaboration. We will implement the three roles with the methods that are shown in the matrix. In the composition of the `CustomerPanel` class, the role from Validate Item must be in a more derived position than the role from Adding Item so that methods in the extension can override those in the base.

The column for the `DepositReceiver` class has three `addItem()` methods, each having a different responsibility. The method from Adding Item adds an item to the receipt basis and increments the counts. The method from Validate Item checks the container's dimensions and calls the original `addItem()` method only if they are valid. The method from Item Stuck checks for a stuck container and calls the original only if no container is stuck. The implementations of the latter two include a call to the original encoded as a call to the superclass: `SuperType::addItem(s)`. Thus their roles must be more derived than the Adding Item role. From the requirements we determine that the Validate Item role's `addItem()` method must be called first, thus it must be the most derived. By inserting the Stuck Item role between the Validate Item role and the Adding Item role, when Validate Item's `addItem()` method calls `SuperType::addItem(s)`, Stuck Item's `addItem()` will be called next. With the composition ordered in this way, the resulting class supports the behavior intended. The class definition statements to compose these roles into a class are shown in Fig. 4.

In the composition for the `DepositReceiver` class, Item Stuck may be thought of as inter-

	CustomerPanel	DepositReceiver	ReceiptBasis	InsertedItem	DepositItem
Print Receipt	printReceipt()	*receipt printReceipt()	printReceipt()	number, *item getNumber() getItem()	name, value getName() getValue()

Figure 5: The roles/responsibilities matrix row for the Print Receipt collaboration.

CPAddingItem	DRAddingItem	HeadLinkedList	IIAddingItem	DIAddingItem
CPItemStuck	DRItemStuck	RBAddingItem	IIPrintReceipt	DIValidateItem
CPValidateItem	DRValidateItem	RBPrintReceipt	NodeLinkedList	DIPrintReceipt
CPPrintReceipt	DRPrintReceipt			
CustomerPanel	DepositReceiver	ReceiptBasis	InsertedItem	DepositItem

Figure 6: Order of inheritance for role composition by class (more derived toward bottom).

cepting the addItem() call from Validate Item to Adding Item to add additional behavior, or, as in this case, additional conditions on the existing behavior. This is an example of what we call specialization by inserting ancestors [17]. It may be helpful to model the flow of control among collaborations as well as within collaborations to analyze the complete behavior. The control flow can be modeled informally by drawing on a copy of the roles/responsibilities matrix, or more formally using state-transition diagrams or petri nets, as shown by Aliee and Warboys [1].

The Adding Item role in the ReceiptBasis class uses a linked list. We separated the *Linked List* collaboration from that of Adding Item in order to reuse an existing linked list implementation. Because Adding Item's role in the ReceiptBasis class uses the list, the Linked List role must be less derived in the ReceiptBasis inheritance hierarchy. When ReceiptBasis calls getNext() on the list, it wants an object of a type that includes the InsertedItem class's Adding Item role. The returned value required an unsafe type cast in the implementation in Jacobson, et al. [11]. We avoid the type cast by making the Linked List's role in the InsertedItem class more derived than Adding Item's role. This ordering is not a problem for

the InsertedItem class inheritance hierarchy, since the Adding Item role does not access anything from the Linked List role. As we explain elsewhere [17], we often treat data structures as collaborations and make the nodes derived classes of the data. This treatment differs from the traditional approach to reusable data structures, where the common data structure parts are base classes and nodes have pointers to data.

Figure 5 shows part of the roles/responsibilities matrix for the *Print Receipt* collaboration. Compared with the row from the earlier matrix for the Adding Item collaboration, the two collaborations share many attributes in common. In this situation, we will want to split one or both collaborations into two parts. The common part will contain most of the attributes, while the specialized part contains mostly methods. This issue is the same as that of *abstract use cases* described in Jacobson, et al. [11].

The order of inheritance for role composition in the five classes discussed is shown in Fig. 6. From this graphical representation, we can generate the type definition statements to compose the roles and form the application's classes. A small amount of additional code is needed to instantiate objects and initialize the application.

6 Discussion

Choosing a structure that supports anticipated change is one of the challenges of object-oriented design. Simple subclassing is limited in the adaptations it can support. Often the designer must choose between evolutionary paths, supporting one set of changes while foregoing the opportunity to make others. Suppose we designed our recycling machine with base classes to measure and count containers as in the original and subclasses to print the receipt. If some stores wanted machines that gave change instead of printing a receipt, we could replace the receipt printing subclasses while reusing the original base. But suppose, instead, that for some states we needed to change the validation criteria from measuring cans to reading their bar codes. How much code could we reuse then? With traditional approaches, we would have to copy and edit the receipt printing subclasses for use with the bar code reading base classes. With our approach, in either case we just recompile with the new roles specified in the appropriate composition statements. If we later want to upgrade the receipt format of the machines—say, to include a machine readable bar code of its own—there will only be one version of receipt printing code to upgrade.

One of the strategies for supporting change is to encapsulate the behaviors of change in separate objects. But objects created to encapsulate change distort designs based on *is-a* and *has-a* relationships. The recycling machine design in Jacobson, et al., had an *alarmist* object intended to encapsulate the Item Stuck extension to the DepositReceiver class. Even then, the ideal of completely encapsulating change could not be realized: “Unfortunately, we cannot accomplish this with today’s programming languages” [11, p.250]. Code in the DepositReceiver class had to be modified to support the Alarmist’s extension. Our approach doesn’t have the same problem because we don’t encapsulate change in separate objects. We encapsulate it in roles that can become an integral part of the original object’s class. The other parts of our DepositReceiver class did not have to be modified for the Item Stuck extension.

What about support for structural change? In the original design, the Item Stuck role of the DepositReceiver class sends an `isStuck()` message to the CustomerPanel object to find out if any containers are stuck. What if a new design uses a ContainerFeed object for that function? In our approach, binding to collaborator types is delayed. If the DepositReceiver’s Item Stuck role used its own reference to the CustomerPanel object, we could change the binding for the type of that collaborator to the ContainerFeed object’s class and initialize the reference to point at the ContainerFeed object. If the DepositReceiver’s Item Stuck role was using the reference to CustomerPanel from its superclass, we could create a new role with its own reference, and insert it between the Item Stuck role and its immediate superclass.⁴ In our designs, we almost always use separate roles for external references. Not only does this approach defer the decision about which roles should share a common reference, but it gives us the opportunity to choose among roles that satisfy an obligation locally, roles that reference other objects in the same address space, and roles that satisfy obligations by accessing a remote server. All of these architectures can be supported without changing the implementations of our roles.

In subdividing roles, as in the above example, we use the term role more loosely: some of the resulting components might more appropriately be called role fragments. The distinction is not significant to how they are handled in the implementation. (In another paper we discuss the value for maintenance and reuse of capturing decisions in separate submodules similar to those discussed here [16].)

Of course, we can’t support every change. By mapping units of design directly into units of implementation, our goal is to make it proportionally easy to change the implementation as it is to change the design. Changes that are difficult to make in the design will probably be difficult to

⁴We left the location of the CustomerPanel reference pointer out of the roles/responsibilities matrix to simplify the earlier discussion.

make in the implementation, as well.

With our approach, the runtime cost of configurability is low. Because we use static inheritance to compose the roles, there are no extra levels of indirection. Composition occurs at compile time, so we can also inline method calls to remove the function call overhead commonly associated with decomposing operations into smaller steps. With so low overhead, our approach encourages a more aggressive strategy toward decomposing applications into smaller pieces that encapsulate fewer decisions. This bias towards decomposition is, in turn, reflected in our designs.

To test the scalability of our approach to large applications, we have undertaken the development of an image display and manipulation application. Our intention is to duplicate an astronomy application that was originally written in 30,000 lines of C code. This display program poses a number of challenges. One of its challenges is a complicated structure involving many-to-many relationships among viewports, images, and coordinate systems. Another is our desire to provide different versions of the program for use in many different contexts. We hope to reuse component code in similar and dissimilar objects to support a variety of configurations tailored to specific uses. We are already using the code written thus far to explore and experiment with different architectures as we try to address the structural issues of the design.

As we had expected, the order of evolution of the image display application does not at all follow the order of inheritance in the class compositions. Added features tend to go in the middle of the class hierarchy where they can interact with existing features in a natural order, while the more derived roles tend to be concerned with event detection and object initialization. Yet, at each step in the evolution we have a workable application.

By being able to reason about the flow of control within and among objects, we are able to structure the flow of control implicitly through the specifications of composition rather than using an encapsulated central dispatcher. This ap-

proach lets us use local decision making at runtime while still maintaining a global overview to determine the order of responding to events such as mouse movements and reconfigured windows.

Our inheritance hierarchies tend to be much deeper than those found in other object-oriented applications. Each of the major classes is composed of 10 to 20 roles, and we are still adding roles. We have found graphical representations of class compositions and role dependencies to be indispensable in managing our designs. By inspecting the role hierarchies, we can easily identify common superclasses. In some cases, by changing the order of role composition, we were able to reduce the number of classes needed by the application. The repeated use of roles within a class is common, especially the generic handle roles used for inter-object referencing.

We have had some difficulties with template instantiation, especially when trying to support separate compilation of source code files. We have also had some difficulty with debugger support for template generated names. Our templates can get nested very deeply. Many of these problems appear to be compiler-specific, so a general solution may not yet be available. We suspect that similar problems are encountered by other users of templates. The fact that some compilers seem to work much better than others gives us hope some of the issues will be ironed out.

We are often asked why we don't use multiple inheritance for composing roles. Roles can be incomplete types that extend other roles and must be able to interact within the composition—something not permitted among multiply inherited classes in C++. Because the same role can appear twice in the same class, we use type names for intermediate levels of composition to disambiguate references to names in the interface, a problem even for CLOS-style multiple inheritance. Finally, we don't need multiple inheritance.

7 Related Work

The OOram methodology has many similarities with the work presented here, but differs in emphasis [13]. Although both approaches start with roles and collaborations, the OOram approach combines or “synthesizes” roles in the design process to improve design comprehension, while our approach subdivides roles in the design process to improve code maintainability and reuse. Code reuse in the OOram methodology is based on object libraries and frameworks of base classes. (A comparison of our approach with frameworks is the focus of our earlier work [17].)

The goal of our work is similar to that of subject-oriented programming in that both approaches address the issue of composing different views on a common set of objects [9]. In subject-oriented programming, separate applications are merged by combining common objects. The mechanism of Harrison and Ossher requires a runtime dispatcher and special compiler tools. By comparison, ours is a lightweight approach intended for building a single application structure.

The decorator pattern from the book by Gamma, et al., is also used to add responsibilities to an object [7]. The decorator has an advantage of being able to support dynamic change. But an object’s decorators must conform to a fixed interface by subclassing from a common base class. Calls to that base class must be explicitly encoded in the object being decorated. Compared to our approach, the decorator mechanism places more restrictions on the location and type of extension that can be applied and does not address the issue of unanticipated change.

The notion of role has a counterpart in object-oriented databases [8, 18]. The issue arises, for example, when an employee object may play the role of trainee at one time and manager at another, or possibly even the same, time. While both uses of role address objects playing roles in different contexts, the database usage is more concrete. In our usage, if an object satisfies the requirements of a role, it can play that role. In the object-oriented database sense, an object must have a role of that

name. For database roles, the main issue is the ability of objects to dynamically change roles.

Bracha and Cooke demonstrated delayed inheritance using type parameters, calling the resulting components mixins [5]. The term roughly corresponds to the use of multiply inherited classes in CLOS. Unfortunately the meaning of the term mixin is often confused with the different semantics of multiply inherited base classes in C++. Bracha’s dissertation focused on semantics and language issues and did not present mixins in the context of a design methodology [4].

The C++ Standard Template Library of Stepanov and Musser uses templates extensively [12]. But STL uses templates for genericity, not composition, and it does not use inheritance. Roles can use STL data structures, but roles can also be used to implement data structures. For instance, we have produced role-based implementations of the list and binary tree data structures and shown how the two could be composed to form a multiply threaded list with a binary tree find() operation. That combination would not be possible with the equivalent data structures in the STL.

8 Conclusion

We have demonstrated a new approach for implementing object-oriented programs using source code role components. We showed how to derive the roles from a collaboration-based design and how to compose the role implementations at compile time to form the classes of the application. Our approach supports more flexibility for change and adaptive reuse than traditional approaches to implementation, while requiring fewer deviations in the design and less runtime overhead than common approaches for supporting change. Our approach requires no special tools and uses only the features associated with class templates in C++. While all of our experience to date has been with small applications, recent experience working on a larger application has been very promising.

References

- [1] F.S. Aliee and B.C. Warboys. Roles represent patterns. In *Proceedings of the Workshop on Pattern Languages of Object-Oriented Programs at ECOOP'95*, 1995.
- [2] K. Beck and W. 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.
- [3] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.
- [4] G. Bracha. *The programming language JIGSAW: mixins, modularity and inheritance*. PhD thesis, University of Utah, 1992.
- [5] G. Bracha and W. Cooke. Mixin-based inheritance. In *Proceedings of the 1990 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 303–311, 1990.
- [6] A.M. Burkett. Clarifying roles and responsibilities. *CMA: the Management Accounting Magazine*, 69(2):26–28, March 1995.
- [7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [8] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), July 1996.
- [9] W. Harrison and H. 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.
- [10] R. Helm, I.M. Holland, and D. 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.
- [11] I. Jacobson, M. Christenson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 2nd edition, 1992.
- [12] D.R. Musser and A.A. Stepanov. Algorithm-oriented generic libraries. *Software Practice and Experience*, 24(7):623–642, July 1994.
- [13] T. Reenskaug. *Working With Objects: The OOram Software Engineering Method*. Manning, 1995.
- [14] T. Reenskaug, E.P. Anderson, A.J. Berre, A. Hurlen, A. Landmark, O.A. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A.L. Skaar, and P. Stenslet. OORASS: Seamless support for the creation and maintenance of object-oriented systems. *Journal of Object-Oriented Programming*, 5(6):27–41, October 1992.
- [15] D. Riehle. Describing and composing patterns using role diagrams. In H. Steffen, editor, *WOON '96, Conference Proceedings*, St. Petersburg, Russia, June 1996.
- [16] M. VanHilst and D. Notkin. Decoupling change from design. In *Proceedings of SIGSOFT'96 Foundations of Software Engineering*. ACM Press, 1996.
- [17] M. VanHilst and D. Notkin. Using C++ templates to implement role-based designs. In *Proceedings of the 2nd JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer-Verlag, 1996.
- [18] R.J. Wieringa, W. de Jong, and P. Sprint. Roles and dynamic subclasses: a modal logic approach. In *Proceedings of the 1993 European Conference on Object-Oriented Programming*, pages 32–59, 1994.
- [19] R. Wirfs-Brock and B. Wilkerson. Object-oriented design: A responsibility-driven approach. In *Proceedings of the 1989 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 71–76, 1989.
- [20] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.