Role Oriented Programming for Software Evolution

by

Michael VanHilst

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

1997

Approved by ______________________________

(Chairperson of Supervisory Committee)

Program Authorized
to Offer Degree ______________________________

Date ______________________________

University of Washington

Abstract

Role Oriented Programming for Software Evolution

by Michael VanHilst

Chairperson of Supervisory Committee
Professor David Notkin
Computer Science and Engineering

This thesis addresses the problem of changing requirements in software evolution. It presents a method of development and change based on roles, where a role, in object oriented development, is a part of an object that addresses a particular concern or requirment. The concept of a role was originally used in design analysis by Trygve Reenskaug. In this research, we extend it's use into implementation.

The contributions of the research include a development process that takes a set of use case requirements and produces an implementation composed of role components, a set of implementation idioms that separate functional concerns from structural and compositional concerns, and several diagrams to bridge the gap between abstract design and concrete implementation. The feasability of the approach is demonstrated with an efficiant method of implementation using C++ templates.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to thank the many people without whom this thesis would never have been written. Professor David Notkin provided patient support and encouragement. My wife, Luz Angela, and son, Marco, provided continuous motivation. My sister and brother-in-law, Anke and Terry Gray, provided support especially in the most difficult times, and my parents, Ruth and Lucas VanHilst, provided encouragement and financial assistance.

Many other friends and colleagues provided important assistance. The members of my committee provided guidance in expressing my ideas. My fellow students Kingsum Chow, Erica Lan, and Gail Murphy provided support and advice along with many others whom I hope will forgive me for not listing all their names. My friends, Greg Madejski, Edwin Lee, Don Wong, and again many others helped keep me sane. Researchers at other institutions, including Harold Ossher, Don Batory, Dirk Riehle, and Thomas Kuehne, played important roles in helping me form my ideas.

I hope those who I have forgotten to mention in my rush to finish the writing will forgive me. While many have helped make this document more than it would otherwise have been, the errors and weaknesses in this document are my own responsibility.

Chapter 1

INTRODUCTION

Writing good software is hard, requiring domain expertise and skill in design. Writing good software that is also adaptable is even harder. Not only does the software have to meet the present requirements, but it must also isolate the types of decisions that could change to meet future requirements.

Object oriented approaches map entities from the problem space to objects in the solution. But entities are the most stable part of a problem; it is the behaviors that are most likely to change. Popular methodologies for object oriented development lack both the models and the mechanisms needed to map behaviors from the problem domain to changeable components in the implementation. New design methodologies emerging in Europe map behavioral requirements to the initial stages of design, but not to implementation.

In this thesis, we present a way of developing object oriented software that separates the concern of designing for future change from the concern of designing for present requirements. Our approach combines a method of implementation that isolates decisions in fine grained components with a method of design that maps those components from, and to, requirements. The methodology touches on a broad range of issues affecting change, including initial design, reuse, configurability and adaptation, the handling of anticipated and unanticipated change, requirements traceability, and design consistency.

1.1 Supporting Change

Requirements change. As Brooks observed "the software product is embedded in a cultural matrix of applications, users, laws, and machine vehicles. These all change continually, and their changes inexorably force change upon the software product" [15].

In this thesis we describe an approach to software development that supports the process of changing the software product when requirements change. Ours is not the first approach, nor will it be the last, to make this claim. What does it mean to support change? We begin by presenting a list of properties that are important for supporting change.

1. Flexibility addresses the amount of new code needed to add or alter a feature. Ideally, the cost of making a particular change should be proportional to the cost of making that change in the analysis model of the design—small changes should require correspondingly small amounts of new code.

2. Modularity addresses the extent to which concerns are isolated so that changing one concern does not require changes to, or even an understanding of, the implementation of others. Ideally, the concerns of a change will have their own module or modules, and no other modules will be affected. While flexibility helps the maintenance programmer say, "yes, I can add that feature," modularity helps her say, "and yes, everything else will still work as it should."

3. Durability implies the ability to apply change in a way that does not degrade the ability to make further change. Durability addresses the increasing entropy observed by Lehman and Belady—as programs evolve, successive changes becomes harder to apply [36]. The ideal, zero entropy result of applying a change would be a system that was indistinguishable from the system that would have resulted had the new requirements been included in the original requirements.

4. Wholeness implies support for change across all phases and artifacts of software development. An approach that addresses change only in the implementation lacks wholeness. The same can also be said of an approach that focuses on design change, with little guidance for changing the implementation. Support should cover propagating changes from the requirements through the design and into implementation, and be reflected in the artifacts of each. Two stronger properties also apply to wholeness.

   (a) Proportionality suggests that the effort needed to change the implementation should be proportional to the scope of change in the design.

   (b) Traceability suggests the existence and maintenance of mappings between artifacts of successive development phases.

5. Clarity refers to the effort needed to figure out how to make a change, including the adjustments needed to counteract any impact on other concerns. If a program is too complicated for the maintenance programmer to understand, it may be cheaper, and safer, to build a new application than to figure out how to change the existing one.

6. Breadth refers to the range of changes that are supported. Using a particular approach may make some changes easier. But other changes may be just as hard or even harder when compared with other approaches.

7. Depth refers to the degree to which changes can be finely tuned in important ways. It may be easy to adapt a program to be close to what the customer wants, but difficult to get exactly what she wants. A component kit, for example, may make it easy to add dial interactors to an application, but hard to label those dials for a user's logarithmic data.[1]

[1]While scalability refers to an ability to apply something for small or large scale uses, depth emphasizes usage for both small *and* large scales, at the same time.

8. Balance considers the net cost of supporting change. Ideally, there should be no other cost—ceteris paribus. But flexibility often comes at the cost of other desirable qualities, such as development effort and runtime performance.

1.2 Change in Software Evolution

In software evolution change is driven by user requirements. Such requirements typically involve system level behaviors that a user can observe or initiate. In a requirements analysis, these behavioral requirements are described in use cases—sequences of behavior described from the users point of view.[2] In this thesis, the changes we address are changes presented as use case requirements. More significantly, they are more likely to involve system behavior than the format of data.

Changes during software evolution are often small, but rarely trivial. Use cases commonly describe sequences of behavior that involve the participation of several entities in the problem domain. The behavior originates in one entity, typically some part of the user interface, is processed by other entities as they change or interact with system state, and produces results that present themselves through actuating devices or state changes in other parts of the user interface.

Support for software evolution must allow changes to interact with existing state without having to resemble existing behavior. Users often request new kinds of behavior to be added to existing systems. These new behaviors manipulate existing data and introduce new data. Changing the details of an existing behavior is not sufficient.

Changes in software evolution are often hard to predict. Users unfamiliar with a new technology describe their requirements based on the systems they know. After gaining experience with a new system they are likely to discover possibilities that would not have occurred to them earlier. Changes in the environment may also be hard to predict, including regulatory changes, changes due to competition, and

---

[2] A more extensive discussion of use cases, with examples, is provided in Chapter 2.

changes discovered when a product enters new markets. Even for changes that could be predicted, developers aren't perfect. To paraphrase Dave Thomas, who would have expected the year 2000?

In this thesis, we focus only on changes that can be applied at compile time or earlier. There are domains where change must be applied to running systems. But in many domains, changes can be applied by replacing the executable image. Microsoft Word, for example, doesn't have to change from Version 6 to Version 7 as it is running. Applications shouldn't have to pay the price of runtime support for changes that can easily be rerun through the compiler, especially if the penalty is reduced flexibility.

Finally, we see no compelling reason to prevent programmers from seeing the implementations of modules. Software evolution is largely an activity carried out within the walls of the proprietor of the software. In this context, the protection of trade secrets is not an issue. Programmers struggling to satisfy current and future customers can and should have access to the code of any module they are trying to reuse. Similarly, in the argument between black box and white box implementations, while change should be addressed in a systematic way, we strongly believe that providing better paths to success is preferable to adding obstacles.

We focused our research on software evolution in object oriented approaches because that is where we can find our largest audience. Although problematic, we believe that models based on entities and abstract data types provide a better foundation for robust design than earlier functional models. Object oriented technology has demonstrated enough advantages over competing technologies to have earned a following in the software development community and among those interested in change. We feel that this thesis makes a contribution in an area where current practice can still be improved.

1.3 Object Oriented Development

The idea that today's object oriented approaches solve the problems of software evolution is a myth. Object oriented development has been over sold as a solution to software change. Proponents point to the robustness of designs based on entities and the changes supported by encapsulation and inheritance. But the examples of change they point to are always data type changes, with little or no change in behavior.

Current approaches to object oriented design assume that changes can be isolated in separate objects. The argument goes something like this: With support for changing data types, an object can be replaced with another object of a related type having the same interface, but different semantics. Given the right abstractions, code that draws rectangles can be made to draw circles. Code that stores data in linked lists can be made to store the same data in hash tables. Each of these changes is made by replacing a single object. The problem with this argument is that the changes it assumes are few and localized. In software evolution, systems undergo changes that are more varied and less easily isolated. The types of behaviors found in use case requirements criss-cross one another and span large parts of the system.

1.3.1 Design

In object oriented programming, the basic unit of encapsulation is the object. Thus it is not surprising that current design models address change, assuming they address it at all, by attempting to encapsulate it within a single object. The more naive approach assumes that changes will naturally happen only within entities where they can be addressed by the mechanisms of encapsulation and inheritance provided by the implementation. This design approach models entities in the problem domain with little or no attention for the issue of change; "Instead of an indirect mapping from problem domain to function/sub-function or problem domain flows and bubbles, the mapping is direct, from the problem domain to the model" [18, p.32].

Unfortunately for the types of behavior changes discussed above, neither part of the naive assumption holds. Behaviors of interest to the user often do not fall within a single entity of the problem domain. For those behaviors that can be contained, the mechanisms of implementation often break down, as we explain below. The reality is that many changes are not supported.

The less naive approach to object oriented design acknowledges that some changes cannot be addressed by normal entity models. Special efforts must be applied to isolate these more difficult changes. These approaches deviate from the entity model to capture multi-entity change, or add structures of non-entity objects to address weaknesses of the implementation. Behavior and entity models are intermingled in the design—some objects model entities while other objects model behavior.

Design approaches that mix entity objects with behavior objects complicate designs and make them more difficult to maintain over time. New behaviors may be added as behavior objects, but changes to existing concerns, whether behaviors or entities, become progressively more difficult. Concerns become dispersed over both kinds of objects. Mixed designs no longer reflects the logic and structure of the problem domain. The model is also hard to realize—in practice, adding new behavior still requires changing existing objects, either at the time of change or beforehand, to create the points of attachment.

Some in the frameworks and patterns communities have argued that problems of change can be resolved if only we find the right abstractions [25, 33]. There are two problems with this argument. The first problem is that, even if an abstraction can be found for each requirement, leaving the door open for any of a large number of requirements to change, including several at once, requires more than a single abstraction for each change. Requirements overlap and commingle [35]. The other problem is that much of the effort in finding abstractions must be directed at overcoming limitations in the method of implementation. We describe some of those implementation problems, below. Thus, even if the ideal of finding an abstraction for every change could be found, it is still worth considering other approaches if the overall effort can be reduced.

1.3.2 Implementation

The common mechanisms of encapsulation and inheritance have numerous limitations and weaknesses. We briefly list the problems here, discussing them at greater length in Chapter 3. The consequences of changing an interface limits the kinds of changes that can be supported, particularly behavioral changes. The order in which changes are applied is reflected in the inheritance hierarchy and affects the ability to apply future changes. A simple change to a single type can propagate as a chain reaction through structures of inheritance and client relationships. Hierarchical structures can't support the combinatorial variations of features needed to tailor families of products for varied customers. Common type systems only support changes at a single level in the aggregation hierarchy—details of component objects cannot be changed without causing the chain reaction mentioned above. In terms of our initial set of criteria, these problems affect breadth, durability, modularity, and depth.

Interesting structures can be applied to improve abstraction in some places where the existing mechanisms would otherwise break down. Solutions of this kind can be documented for reuse as frameworks or patterns. This approach, however, is also limited in the support it can provide. The changes must be anticipated since the structures themselves are hard to add to an existing design. The supported change must be isolated in a single object, often with a fixed interface.

Some object oriented languages provide additional mechanisms to support change. The weaknesses or advantages of these alternatives depends on the details of each language's implementation. We discuss the limitations of multiple inheritance and interface types in Chapter 3. Some additional discussion of Smalltalk, ML, and BETA appears in Chapter 10.

Ultimately, the design for an object oriented program must be implemented.

Whatever design model of change we choose, we will need mechanisms that support it, and hopefully avoid some of the kinds of problems described above. The approach we will take involves factoring objects into smaller fragments, called roles. We present an implementation based on template parameterization in Chapter 8. Alternative mechanismss that might also work are discussed in Chapter 10.

1.4 A Role Oriented Approach

A new approach emerging in Europe, called role modeling, augments the basic object model with the additional abstractions of roles and collaborations [55, 56, 57]. This new approach is capable of capturing both entities from the problem domain and use case-like scenarios from the requirements analysis.

A collaboration is a group of objects that work together to address a particular concern, such as performing a task or maintaining an invariant. A role describes the responsibilities of a particular object for its part in the collaboration. In Reenskaug's OORAM (Object Oriented Role Analysis and Modeling) methodology, collaborations, called role models, are used in the design analysis, to model the behaviors of use case-like scenarios in an object structure [55].

Figure 1.1 shows graphically how both the entity and use case views are overlayed on the same structure. Objects model entities. Collaborations model behavior. Roles capture the intersections. Not every object participates in every collaboration, thus not every intersection defines a role.

Roles provide an abstraction that allows an object's participation in a use case concern to be modeled without knowing the details of the rest of the object. In fact, a role is not tied to a particular object. Any object that fulfills the responsibilities of a role can play that role in the collaboration.

Collaborations can be subdivided to manage complexity and exploit reuse. The smaller collaborations address smaller concerns with fewer and/or smaller roles. Smal-

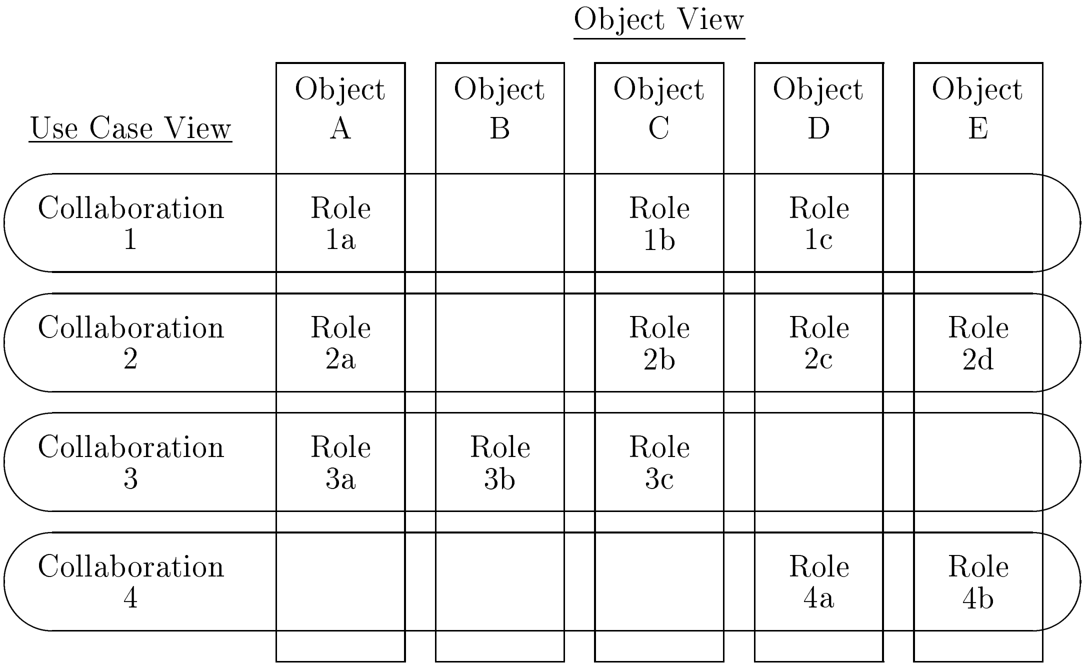

Figure 1.1: Object and use case views overlaying the same design.

ler collaborations can also capture generally useful concerns such as maintaining the relationships in tree or linked list data structures.

In the OORAM methodology, roles exist as an abstraction only in the design analysis. Roles abstractions are combined to define all the responsibilities of an object in a process called role synthesis. The designer then designs and implements classes for each object.

1.5 Roles and Change

The role is a powerful abstraction for evolution. Ignore implementation, for the moment, and consider evolution as applied only in the OORAM analysis model. Figure 1.2 shows the addition of a new use case to part of the model from Fig. 1.1. Collaboration 5 introduces three new roles for the existing entities A, C, and D, and

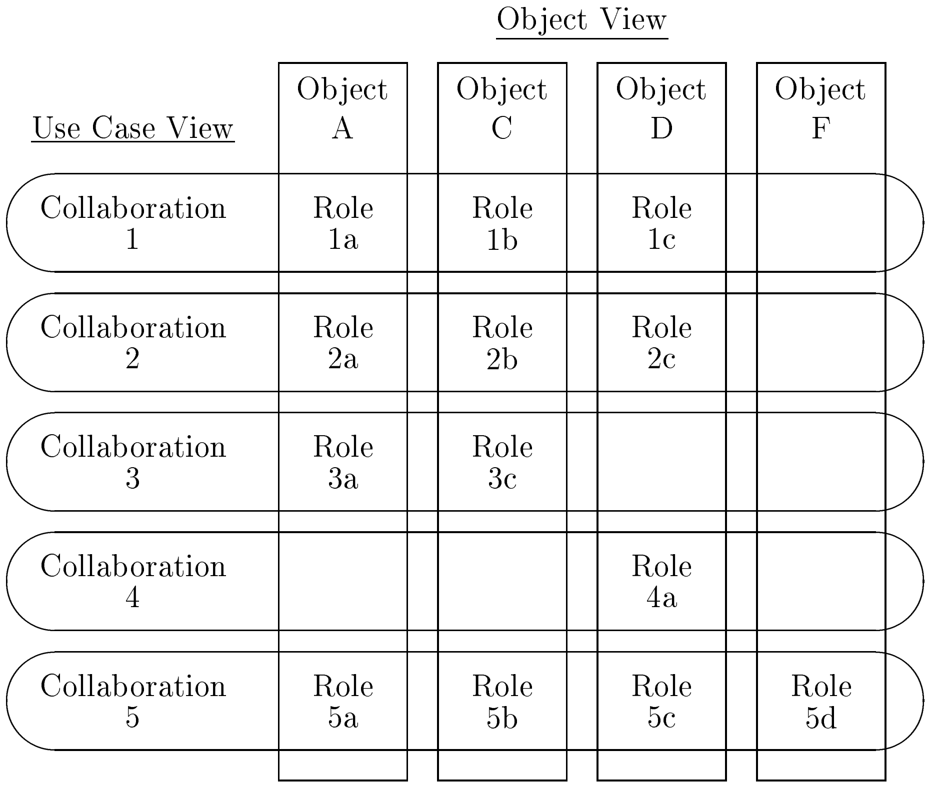

Figure 1.2: Adding a fifth use case with roles for four objects.

one role for a new object, F.

In the analysis model of Fig. 1.2, Collaboration 5 is treated like any other collaboration—as if the design has always had these five collaborations. Each of the objects must still play the same roles as before, for collaborations 1 through 4. But some must also play a new role in collaboration 5. In this model of evolution, roles define both increments of change and units of continuity.

Unfortunately, implementation can't be ignored forever. The role approach to evolution, so compelling in the design analysis model, is poorly supported in implementation. In OORAM, synthesis is applied to each of the affected objects, and a new implementation is created. The entire process from the synthesis step on, is repeated. The designer can refer back to the original implementations for ideas, but there is no actual method for reuse. Because these changes are applied with the normal approaches to implementation, as mentioned earlier, it is difficult to apply the changes as simple extensions to existing code and no protection from the effects rippling through class and aggregation structures.

1.6 Our Approach

In this thesis, we extend the role abstraction into implementation. The increments of change and units of continuity, modeled by roles, become actual source code components. We thus defer the synthesis step, mentioned above, through the remainder of the design process, to be performed at compile time. An abbreviated description of our approach appears elsewhere [75].

In our approach, we take a compositional view. Object implementations are created by composing them from smaller components, as shown in Fig. 1.3. Abstractly, each component handles the responsibilities of a role in a collaboration. Thus in a broader sense, we create applications by composing collaborations. Evolution is applied by adding and replacing sets of components defined by collaborations.

Our approach addresses the issues needed to make roles both implementable and composable. As an example, use cases often overlap—they operate on common state and duplicate pieces of each other's behavior. In OORAM such issues are left to the object implementor to figure out in the synthesis step. Chapter 7 of this thesis defines a development process with steps that address these issues explicitly.

In composing collaborations, the touch points occur within objects rather than between objects. Some of the steps in our process, described in Chapter 7, subdivide roles to separate the concern of addressing the responsibilities in a collaboration from the details that address intra-object composition and sharing. In Chapter 6 we describe a variety of simple idioms to handle issues involved in composition. Separating collaboration concerns from the more context dependent composition concerns frees

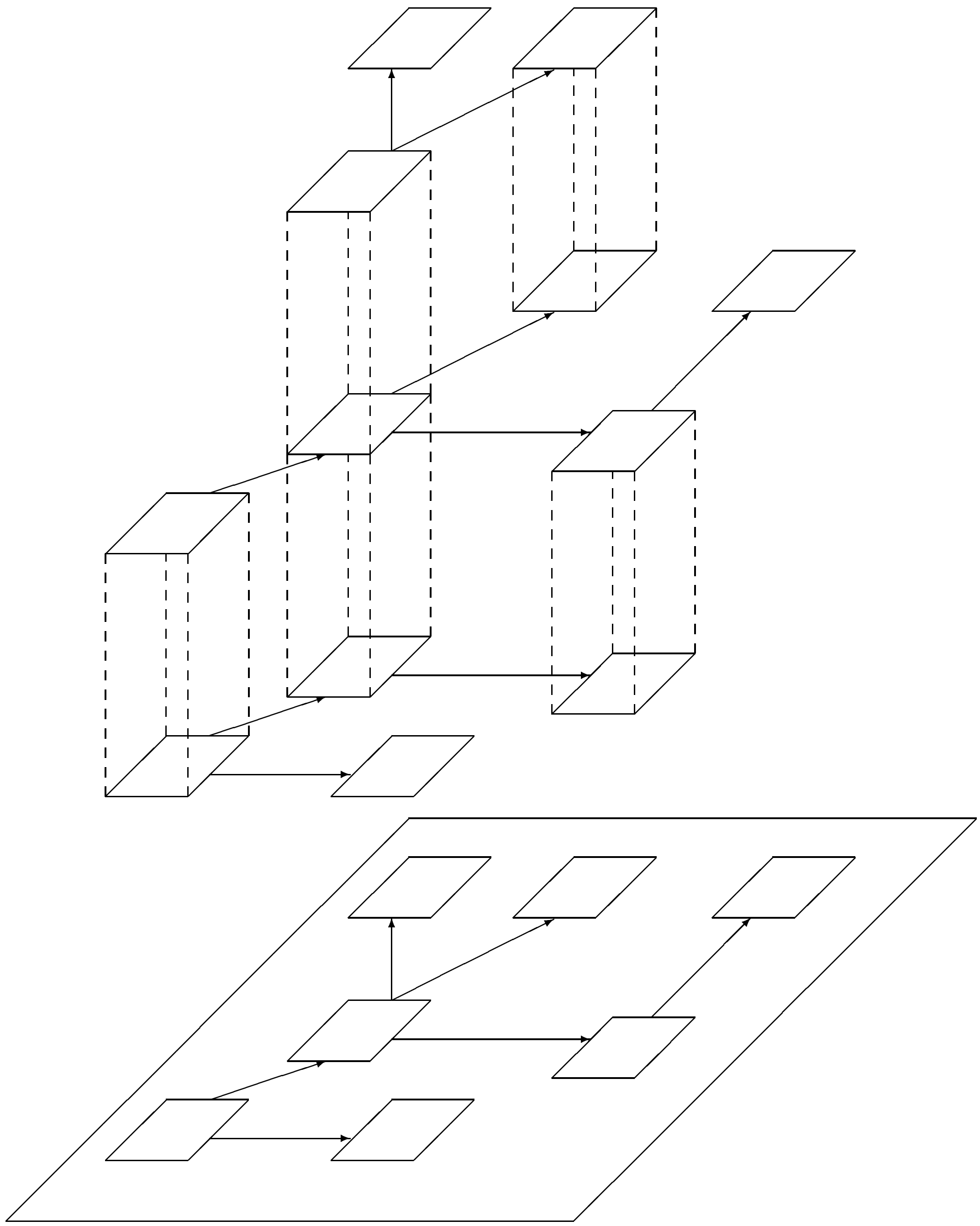

Figure 1.3: Composing roles from different collaborations to form objects.

the components that address collaboration concerns to be reused in different compositions without modification.

Our handling of interactions and interfaces between objects differs from other object oriented approaches. In our approach, interactions between objects are defined within collaborations. Role components handle the parts of an object's interface specific to each collaboration. An object can participate in a collaboration, not just by being able to play its role, but, more concretely, by having the appropriate role component in its composition.

The compositional approach to implementation allows us to better address software evolution. Just as collaborations can be subdivided to reduce complexity, we subdivide collaborations and roles to better isolate concerns that might change in an application's evolution from concerns that don't [73]. In Chapter 8, we describe the process of adding use cases to change or augment the behavior of an existing application. Some of the compositional issues are revisited, but, largely, it is a process of adding new components to existing compositions.

1.7 An Implementation

Role composition is analogous to inheritance. Adding a role component to an existing object extends the existing interface, but does not replace it. Within an object, components of the composition interact through internal interfaces, not necessarily visible to an object's clients. The normal inheritance semantics of object oriented programming provide especially good support for our style of composition because the interaction with other components does not require separate references for each component—their methods and attributes are all accessible in the inherited interface.

The extensive use of subdividing in our design approach would come at a heavy price if the implementation exacted an overhead for connecting each component or the resulting type system became unmanageable. Fortunately, we have found a method of

```
template <class TwoCType, class TwoDType, class SuperType>
class TwoBRole : public SuperType {
   TwoCType* c_object;
public:
   void init(TwoCType* c) { c_object = c; }
   void doSomething(TwoDType* d) {
       d->doThis();
       c_object->doThat(d);
   }
};
```

Figure 1.4: C++ template implementation of hypothetical 2b role.

implementation that, in most cases, exacts no overhead for connections within objects, and computes the type system at compile time each time the system is compiled.

Each role component is implemented as a class template in C++. The name of the parent super class is parameterized, as are the classes of the other objects in its collaboration. Fig. 1.4 shows an implementation of the hypothetical 2b role in Figs. 1.1 and 1.2.

Components are composed by binding concrete class names to the template parameters in template instantiations. Figure 1.5 shows a series of statements to create the implementation of object C in Fig. 1.1, while Fig. 1.6 shows the corresponding statements for object C as defined in Fig. 1.2. Note, in the second definition of the CClass, that the new role component does not have to be added in the most derived position.

When templates are composed, as in the above example, methods are statically bound to their calling site. By using static binding instead of dynamic binding, the methods can be inlined, saving not only the dispatch indirection, but the function call overhead as well. In this way, there is no runtime cost for breaking operations into smaller pieces and later composing them through inheritance.

A common use for dynamic binding, allowing methods to be overridden in more

```
// forward declarations for classes not yet defined
class DClass;
class EClass;
// definition of CClass with OneBRole in base position
class CClass1 : OneBRole            <emptyClass> {};
class CClass2 : TwoBRole <DClass,EClass,CClass1> {};
class CClass  : ThreeCRole             <CClass2> {};
```

Figure 1.5: Template instantiations in class definitions to create CClass.

```
// forward declarations for classes not yet defined
class DClass;
class EClass;
// definition of CClass with FiveBRole inserted in 2nd position
class CClass1 : OneBRole            <emptyClass> {};
class CClass2 : FiveBRole              <CClass1> {};
class CClass3 : TwoBRole <DClass,EClass,CClass2> {};
class CClass  : ThreeCRole             <CClass3> {};
```

Figure 1.6: Template instantiations, including additional FiveBRole, to create new version of CClass.

derived specializations, is not needed for our approach. The same affect is achieved by inserting the specializing role in a less derived position, as demonstrated by the FiveBRole in Fig. 1.6. Chapter 7 discusses the use of composition ordering in more detail. A comparison of this approach with more traditional framework implementations appears elsewhere [74].

Because all classes are parameterized, implementations are not dependent on the existing type structure. The limitations of extension and the problem of propagating changes, described above, don't apply. The type structure is computed by the compiler when it instantiates the classes. The implementation is still fully type checked at compile time.

We present the C++ template approach to implementation for the purpose of demonstrating that our approach to development and evolution is realizable. Other mechanisms for implementing roles may also exist. We discuss a number of alternatives that may work, and a few that don't, in Chapter 10. A set of criteria for viable implementations appears in the appendix.

1.8 Our Contribution

In this thesis we take a promising approach for describing change in the analysis phase of design and extend it all the way into implementation. We present an efficient method of implementation for this approach. We provide a detailed description of the development process, and of the process of applying change. Finally, we present a set of implementation idioms that facilitate our style of implementation. In Chapter 9, we describe our experience in using our approach to develop a medium sized application. Chapter 11 relates our approach to supporting change back to the criteria described above.

Many pieces of the material presented here are not original. The concept of a role originates with Reenskaug, et al. [57]. The use of parameterized inheritance was described by Bracha and Cook [14] and has been used with C++ templates, for example, in the implementation of the IBM Montana compiler.[3] Some of the idioms presented in Chapter 6 have appeared in other contexts [25, 54]. However, by combining these ideas in a comprehensive way, we open up opportunities that have long existed but never fully been exploited.

There are no published reports of other attempts to implement roles from the analysis as source code components, let alone to compose them as anything other than abstractions. Some of the idioms are original—most notably our novel approach to applying data structures. Parameterized inheritance has never been exploited for more than isolated classes in an application. Three of the five diagrams we introduce to aid in program analysis and understanding are new. Finally, our contribution is unique in presenting a complete process for supporting change in software evolution using a common implementation language in a non-domain dependent way.

Many books on software development claim that their approach makes changes easier to apply. Far fewer actually explain how to apply change, and none in a systematic approach that covers software evolution. How much and which parts of the original development effort must be repeated when a requirement changes? Is the original approach reused at all, or does change require a different approach? To what extent are details missing and left up to the individual developer? In this thesis we describe an approach to supporting change that address each of these questions.

[3] Private conversation with the developers.

Chapter 2

A USE CASE EXAMPLE

In this chapter, we present an example application to provide focus and to motivate discussion throughout the remainder of the thesis. The example illustrates use case requirements, an object oriented design to address those requirements, and subsequent changes that a user might request. The application is a container recycling machine—a vending machine that takes empty beverage containers and issues a receipt for their deposit value.

2.1 Use Cases

> Use cases are more dynamic than architecture. I can easily add use cases without changing the architecture. We do it all the time; in fact, that's the most frequent mechanism of system extension.[1]

In software evolution, programs change to satisfy new requirements. It is now commonly accepted that user level requirements can be modeled with use case scenarios [60]. A use case is a description of an observable sequence of behavior from the user's perspective. When refined, a use case describes a flow of control and information among, and changes of state within, entities in the problem domain.

Changes often extend or replace part of the behavior in an original use case. Rather than edit the original use case, such changes are modeled as extension use cases, and stated in terms specific to the earlier use case. "What extension actually

[1] James Coplien, posting to the patterns-discussion mailing list under the subject title, "Re: RE:A Very Disturbing Talk," 21 Nov 1996. Quoted with the author's permission.

means is that behavior b1 will be inserted into another behavior b2. We want (the description of) behavior b2 to be completely independent and have no knowledge of behavior b1" [32, p.212].

The initial requirements for an application are described by a set of use cases. When changes are requested, they will also be described by use cases, or extension use cases. Thus, support for software evolution should be discussed in terms of the addition or replacement of use cases.

2.2 The Container Recycling Machine

To motivate the discussion here, and in later chapters, we present a concrete example adapted from the book, *Object Oriented Software Engineering*, by Jacobson, et al. [32]. The example is the design of the container recycling machine—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 cartons), a button to request a receipt, and a slot to issue a receipt. An illustration of what the machine might look like is shown in Fig. 2.1.

The requirements for the container recycling machine are described by two use cases, called Adding Item and Print Receipt. The Adding Item use case describes the machine's response to a customer inserting an empty beverage container. The Print Receipt use case describes the system's response to the customer requesting a receipt. The two use cases are printed in Fig. 2.2. In our example, we will build the system for these two use cases and then, after the system has been built, apply two changes. The first change is described by the Validate Item extension use case. The second change is described by an extension use case called Item Stuck. The changes are applied, not at the same time, but one after the other to better illustrate issues of software evolution over time.

In an object oriented design of the container recycling machine, the objects model

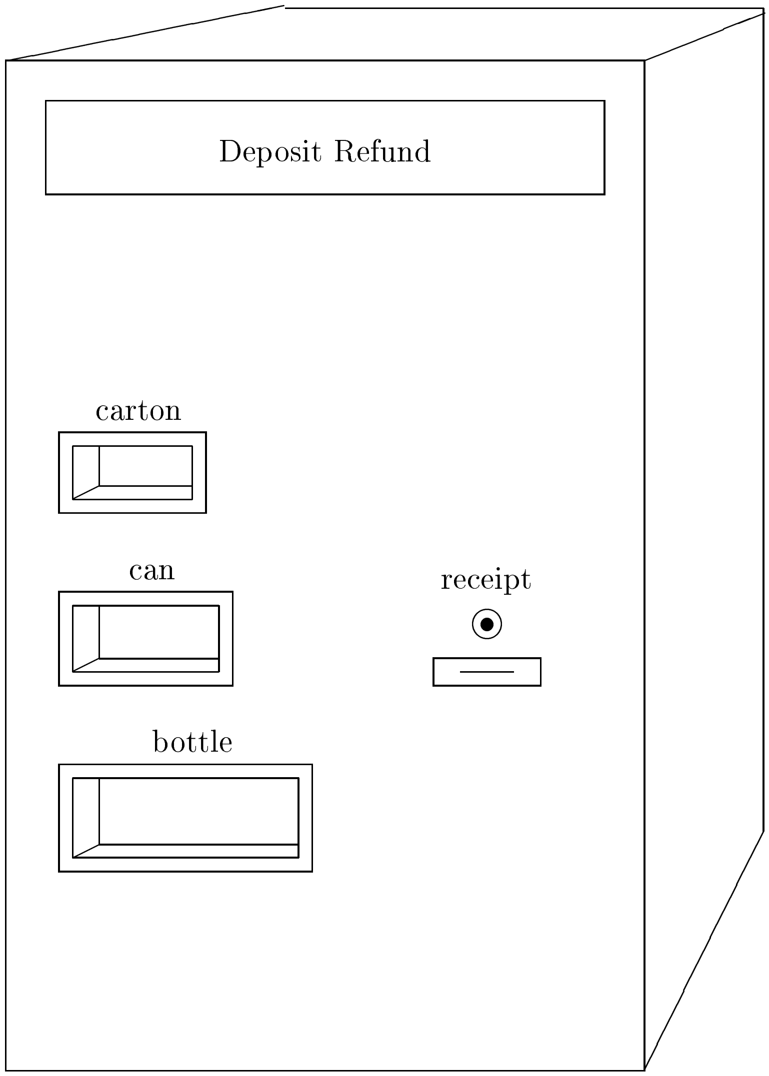

Figure 2.1: The container recycling machine.

When a customer inserts a can, bottle, or carton into the appropriate slot, the system collects and crushes the container, and then increments both a customer total and a daily total for that container type.

(a) Adding Item

When the customer presses the receipt button, the following information for each container type is printed on a receipt: name, number deposited, unit deposit value, and total deposit value. Then the sum of the deposit values is printed, and the receipt is issued through the slot. Finally, the customer totals are cleared, and the machine is ready for a new customer.

(b) Print Receipt

Figure 2.2: Use case requirements for the initial container recycling machine

entities in the problem domain. In the design for the container recycling machine, the two use cases described above are supported by the following object types: CustomerPanel, DepositReceiver, DepositItem, ReceiptBasis, InsertedItem, and ReceiptPrinter. Figure 2.3 shows a diagram of the object structure. The arrows indicate communication associations. The container recycling machine has one object each for the CustomerPanel, DepositReceiver, ReceiptBasis, and ReceiptPrinter. There are separate DepositItem and InsertedItem objects for each of the three container types—can, bottle, and carton.

Each of the objects in the design abstracts an entity in the problem description. The CustomerPanel encapsulates the devices in the front panel. DepositItem is the abstract entity for each item type. It provides information about items of its type, such as name and deposit value, and maintains a daily total. ReceiptBasis represents a customer session. It is responsible for keeping the list of customer totals by item type and assembling the contents of the customer receipt. To fulfill its responsibilities,

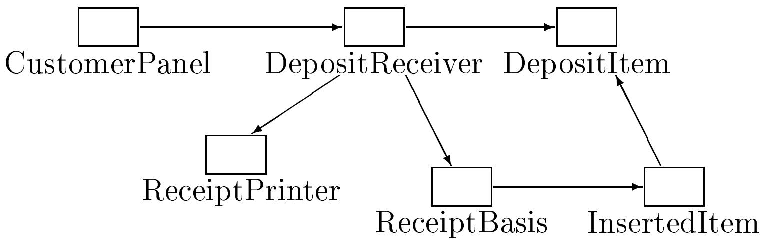

Figure 2.3: Block diagram of objects in the recycling machine design.

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 DepositReceiver represents the overall machine and is the main control object for the system's interactions. It maintains the list of DepositItems for each type of container that may be deposited and coordinates both of the main use case activities. The ReceiptPrinter encapsulates the interface to the printer for printing receipts.

2.3 Changes to the Container Recycling Machine

For our example of software evolution we present two changes that users might request based on their experiences with the existing product. The first change requested is to validate containers to help stores near the border cope with non-deposit containers from the neighboring state. The second change requested is to check for a stuck container as an influx of unwashed containers can make things get sticky. The two corresponding use cases, called Validate Item and Item Stuck, are listed in Fig. 2.4. In our example, the second change, Item Stuck, is requested after the first change, Validate Item, has already been applied. Figure 2.5 illustrates what the machine might look after the two changes have been applied.

The ideal result of applying change is a system with the changes included that is indistinguishable from the system that would have resulted if the new requirements

When a container is inserted, the system measures its dimensions and reads its bar code. The measurements and bar code 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. The user must then remove the container before inserting another. If the container is valid, the system collects the container and continues as per the Adding Item use case.

(a) Validate Item

After attempting to collect the container but before incrementing any counts, the machine checks to see if the item has become stuck. If the item is not stuck, the machine continues as before. If the item is stuck, the machine sounds an alarm. No totals are incremented. The machine then waits for an operator to clear the jam, after which it resumes as before.

(b) Item Stuck

Figure 2.4: Use cases for two changes to the container recycling machine.

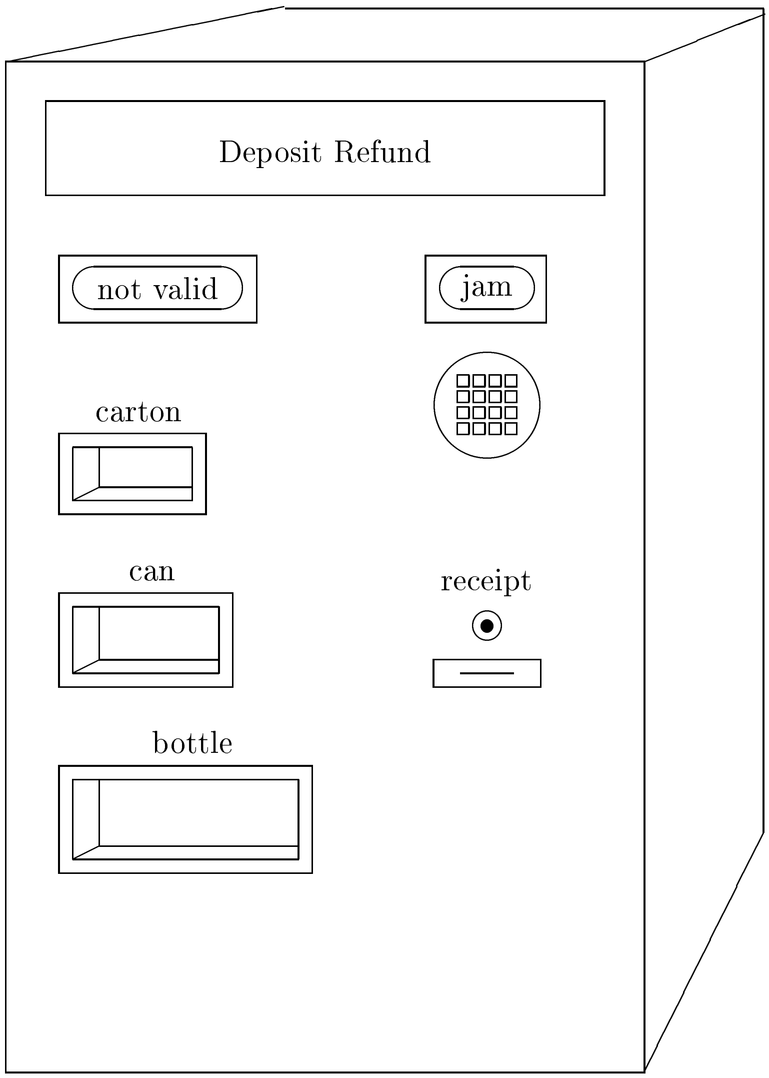

Figure 2.5: The container recycling machine with additions for container validation and jam alarm.

had been part of the original analysis. Thus it is worth describing how the new requirements would be handled if they were included in the original design. Figure 2.6 lists the refined form of the Validate Item and Item Stuck use cases stated in terms of the objects in Fig. 2.3. The refined use cases assume that their behavior can be integrated together with that of the original use cases within the same objects.

The challenge in the next several chapters will be to apply the Validate Item and Item Stuck changes, one after the other. In many cases it may be sufficient to discuss only one change, applying the Stuck Item requirement, in order to highlight the issue in question.

When a customer inserts an empty can, bottle, or carton, the CustomerPanel measures the container and reads its bar code. The CustomerPanel passes the measurements and bar code, along with the slot type, to the DepositReceiver. The DepositReceiver identifies the corresponding DepositItem and asks it if the dimensions and bar code are valid. If the DepositItem responds that the container is valid, the DepositReceiver signals the CustomerPanel to collect the container and then continues to increment the counts as in the Adding Item use case. If the DepositItem responds that the dimensions are not valid, the DepositReceiver signals the CustomerPanel and takes no further action. The CustomerPanel then lights the NOT VALID sign and waits for the customer to remove the container. When the customer removes the container, the CustomerPanel turns off the NOT VALID sign and is ready for the next event.

(a) Validate Item

After signaling the CustomerPanel to collect the container but before incrementing any counts, the DepositReceiver asks the CustomerPanel if the item has become stuck. If the item is not stuck, the DepositReceiver continues incrementing the counts, as before. If the item is stuck, the DepositReceiver signals the CustomerPanel to sound an alarm. No totals are incremented. The CustomerPanel then waits for an operator to clear the jam, after which it resumes as before, ready for the next event.

(b) Item Stuck

Figure 2.6: Two extension use cases refined and restated based on the original design.

Chapter 3

PROBLEMS OF CHANGE IN OBJECT ORIENTED PROGRAMMING

Object oriented programming is promoted for its ability to support change. Almost every book on object oriented development describes the importance of change in the introduction. For example, Meyer writes that, "one of the key benefits of the techniques studied in this book will be to make software easier to modify"[44, p.8]. Cox tells us that the key features of object oriented programming address the problem of "making software malleable enough to avoid destruction by changing requirements" [20, p.v].

Oddly enough, while change is given so much attention in the introduction, few books have an entry for change in the index. If change appears at all, it is typically only a brief mention of changing a data type. In Meyer's book, the one mention of change is immediately followed by a discussion of changing polygons to rectangles [44, p.217]. In Cox, the sole mention appears in the line, "The system as a whole is changing as new data types get implemented and put at the user's disposal" [20, p.62]. The reference is to a container that holds pens as well as pencils.

Changing data types is important. Word processors manipulate text with different fonts, and banks are constantly creating new types of accounts. But data type change is neither the only kind of change, nor, in the case of software evolution, even the dominant kind of change.

In software evolution, programs change to satisfy a user's new requirements. Often these requirements involve new or changed behavior. Meyer cited a 1979 study by

Leintz and Swanson. "More than two fifths of maintenance activities, according to this study, are extensions and modifications requested by the users"[44, p.8]. By comparison, less than one fifth of the changes involved changes in data format. (In descending order, the other change activities were emergency fixes, routine debugging, hardware changes, and efficiency improvements.) Yet, Meyer focused discussion on the cost of the data format change when the post office changed to 5+4 zip codes.

In this chapter we describe problems of adding or changing behaviors in object oriented programs. The discussion considers design models used in popular approaches to object oriented development and also the common mechanisms used in object oriented implementation. The purpose of this chapter is the dispel the myth that current object oriented approaches have solved the problem of change and convince the reader that a significant problem still exists. Additional discussion of specific languages and specific design approaches can be found in the chapter on related work. The problems described in this chapter are illustrated by the problem of applying the two requirements use cases described in the previous chapter.

Experienced programmers may already have found ways around some or many of the problems described below. But alternative solutions are not commonly found in language manuals or taught in programming classes. In our own implementation of roles we have had to address all of the problems described below. Our method of implementation is described in Chapter 5.

Before discussing the problems of object oriented programming, we first describe what it is. In object oriented programming, objects are implemented as abstract data types, capturing both data and operations on that data. In a typical structure, objects model entities from the problem domain—a word or paragraph in an editor or an account or transaction in a bank program. The entities can be concrete, such as a file in a file system, or abstract, such as a scheduling policy in a multiprocessing operating system [62]. Objects interact with other objects by passing messages. A message requests an object to perform one of its operations. Although similar to a function call, a message may invoke different functions, with different semantics, in different invocations, depending on context specific to the receiving object.

3.1 Object Oriented Design

It is commonly accepted that designs should support change by localizing the parts that change in separate components to isolate them from the rest of the application. In object oriented programming this usually means isolating changes in objects, since objects are the components in the design. Object oriented design methodologies take one of two views on isolating change in objects. One view is an implicit assumption that changes will naturally occur within objects if one follows a good design practice. The other view is that special measures should be taken to isolate expected change in objects—possibly with additional structure to make it easier to replace or interchange parts that would change.

3.1.1 Entities From the Problem Domain

For many approaches to object oriented design, change does not appear as a separate concern anywhere in the design process. Individual changes, or even types of changes, do not need to be anticipated. In this view object oriented design simply models the entities in the problem space and the relationships among them. There is an underlying assumption that the nature of object oriented design, as such, provides a natural support for change. Objects in the design map to entities in the problem domain. Object oriented designs are robust to change because the identities of entities and the relationships between them do not change. Only the behaviors of individual entities change—the kind of change supported by inheritance and encapsulation in object oriented programming.

The focus on change within individual entities can be traced back to Parnas' 1972 paper, "On the Criteria to be Used in Decomposing Systems into Modules" [49].

The paper compared a design based on functional decomposition with one based on essentially abstract data types—entities having public interfaces and hidden implementations. In its analysis of changeability, the paper considered five changes—three changes of a list's storage format, one change in a list's physical storage location, and one change to the time when to compute the contents of a list. In each case, only one entity was affected. Noticeably missing from consideration was any change with an externally observable behavior.

Use cases almost always involve observable behavior. As a rule, they are expected to involve more than one object—use cases are commonly modeled as interactions among several objects. Thus if evolution includes adding or replacing a requirement use case, the assumption that change naturally involves the behavior of a single entity cannot be considered valid.

Consider the Item Stuck change in the recycling machine example. To satisfy the Item Stuck requirement, the DepositReceiver's behavior must change. After signaling the CustomerPanel to collect the container, it needs a new behavior to control whether or not it continues. The CustomerPanel may have to change in order to suppress any new deposit events until the stuck container has been cleared. Behaviors for detecting the stuck container and sounding an alarm could also involve the CustomerPanel, which already has other container feed operations and the NOT VALID warning. On the other hand, it might be time to refactor the design to create separate container feed and alarm objects.

The entity modeling approach supports some simple changes, but they must fall within a single entity object and be made without changing the object's interface. No design features are provided to support changes that do not meet these two criteria—changes common in program evolution.

3.1.2 Objects of Change

Jacobson and others propose deviating from the entity model in order to capture changes in separate objects. Jacobson argues that since changes to functionality are common, in order to better isolate change some objects should be allowed to model functionality. In OOSE, such objects are called control objects.

> We do not believe that the best (most stable) systems are built by only using objects that correspond to real-life entities, something that many other object-oriented analysis and design techniques claim. When comparing a model developed with one of those techniques and an analysis model using OOSE, great similarities will be found between the entity objects in the analysis model and the objects yielded in other methods. ... However, behavior that we place in control objects will, in other methods, be distributed over several other objects, making it harder to change this behavior.[32, p.133]

To add the Item Stuck use case to the container recycling machine, Jacobson's design added a special control object called the Alarmist. The Alarmist was supposed to completely encapsulate all the changes needed to add the behavior of detecting stuck containers and sounding an alarm. But such a design could not be implemented. "Unfortunately, we cannot accomplish this with today's programming languages" [32, p.250]. The Alarmist encapsulates most of the code needed for the Item Stuck use case. But, diverting control flow to the new behavior required modifying the DepositReceiver object.

Separate control objects, like the Alarmist, bring entropy to the system. In the original recycling machine design, the CustomerPanel encapsulates the system's interface to the feed mechanism. The Alarmist represents a second object in the design that communicates with the feed apparatus. If we then want to change the feed mechanism, that change is no longer local. Although control objects are supposed to reduce fragmentation in the handling of the current changes, they add complexity to the design and make it harder to avoid fragmentation in later changes.

Using separate objects to add the Validate Item use case adds even more complication. Logically, the validation criteria should be grouped, with other information specific to container type, in the DepositItem objects. To isolate the Validate Item change, a duplicate data structure with DepositItem-like objects would be needed. But the new structure defeats an important feature of the original DepositItem list—the ability to add a new deposit item type by adding a single object to a list.

In summary, designs based on the assumption that entities capture changes handle a very limited range of possible change and are particularly vulnerable to requirements changes of the type discussed here. Allowing objects that capture new behavior supports a wider range of change, initially, but is more likely to degrade the quality of support for all changes, including data type changes, over time.

3.2 Object Oriented Implementation

The mechanisms in object oriented programming most often cited for supporting change are encapsulation and inheritance. With encapsulation, an object's clients depend only on the abstraction of its interface—not its implementation. Parnas referred to the inability of clients to become dependent on details of implementation as information hiding [49]. Inheritance allows more specific functions to replace other functions in the object's interface. In this way, an object's implementation can be changed without needing to change the syntax of its clients. The inheritance is further supported by dynamic binding. When a client sends a message to an object's interface, declared with only the base type, dynamic binding directs the call at runtime to the most specific function implementation for that message. Dynamic binding allows the client to interact with different implementations of the same object during a single execution of the program—a property called polymorphism.

Encapsulation and inheritance support changes to data types. An object of a particular data type can be replaced by a new object of different data type. If the two objects have the same interface, it should not be necessary to change the implementations of any of the object's clients.

To see how encapsulation and inheritance might be used for program evolution, consider applying the Item Stuck use case as the only change to the original container recycling machine. The original DepositReceiver object defined an addItem() method to increment the counts. For the Item Stuck use case, we want to add an additional step of checking for the stuck container before incrementing counts. We can define a new addItem() method in a subclass of the original DepositReceiver class. The new method performs the check and then calls the original method. When the CustomerPanel object sends the addItem message to a DepositReceiver object of the new type, dynamic binding invokes the newer addItem() method.

3.2.1 Encapsulation and Inheritance

In the above example of subclassing the DepostReceiver object, we implemented only part of one change. Unfortunately, things go downhill from there. When applied to behavior change, such as the two changes in the recycling machine example, the support for change provided by encapsulation and inheritance exhibits significant problems with interfaces, chronology, unzipping, library blowup, and aggregation.

In the example of adding the Item Stuck use case, begun above, the new addItem() method needs to find out if the container feed mechanism has a stuck container. In the original design, the feed mechanism was encapsulated in the CustomerPanel object. Introducing a separate object to check the feed status violates that encapsulation. To preserve the encapsulation, we would like to add the new checking operation to the CustomerPanel object. But there is no method in the original CustomerPanel interface that can be called to invoke that operation and get back an answer. If we define an isStuck() method in the CustomerPanel, the the DepositReceiver will need to apply a potentially unsafe type cast to its existing CustomerPanel pointer in to access it. If we introduce a new pointer with the correct type, the previous code could not access it. We would also have to change code elsewhere in the application to initialize the new pointer.

We described adding the Item Stuck use case as if it were the only change. But it is supposed to be the second change. This chronology of change reveals a problem for applying several changes in succession. Assume we had applied the Validate Item use case first and that the Validate Item use case defined its own addItem() method for the DepositReceiver object. The Validate Item version of addItem() checks the validity of the inserted item and then calls the original addItem() method. In the Validate Item change, containers aren't fed to the crusher until after they have been validated. Thus the Item Stuck version of addItem() should be invoked after the Validate Item version, and before the original. But as a subclass of the Validate Item subclass, there is no way to override just the original method. The Item Stuck implementation of addItem() will be called first. But it cannot then call the Validate Item addItem() method without having the original addItem() method being called right away, as well. Had we implemented the Item Stuck change first, and then the Validate Item change, the inheritance order would have matched the correct order of execution.

The Validate Item change also produces the unzipping problem—a chain reaction breakdown of modularity, propagated from a single non-subclass change to an object's type. For the Validate Item change, we would like to add some of the validation criteria to the DepositItem objects. We can do this by subclassing the original DepositItem class. However, since the DepositItem objects are created by the original DepositReceiver, we are forced to change its implementation to create DepositItems of the new subclass. But the change doesn't stop there. The new DepositReceiver is not a subclass of the original DepositReceiver, so its clients must also be edited. The clients are then also not subclasses of their original selves. "Propagating a change in a low-level class can require rebuilding large amounts of code" [23]. Keeping the same name for both old and new classes after an edit, to evade unzipping, is undesirable from a maintenance point of view.

A related modularity problem, called library blowup, occurs when we try to support multiple variants of a program with different combinations of features. Suppose we designed our recycling machine with base classes that measure and count containers and subclasses to print the receipt. If some stores want machines that give 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 need to change the validation criteria to only measuring cans or only reading their bar codes. Because of the hierarchical ordering, we would have to copy and edit the receipt printing subclasses for use with each of the validation base classes. To support alternative validation choices, receipt printing changes have to be applied in multiple places. It is not difficult to imagine combinations of many features. Batory noted that there are more than 400 classes in the Booch library, representing different combinations of less than 30 features [7].

Parnas discussed the hierarchical ordering of decisions in a paper on program families [51]. To avoid the problems associated with library blowup, the differences among members in a family of applications should be concentrated in the leaves of inheritance hierarchies, while those things that don't change should be nearer the root.[1] But choosing a good hierarchy is not always easy. Developers of window system libraries assume that the window system is the least likely thing to change and build their libraries accordingly. But for the developer of multi-platform applications, the window system might be the most likely thing to change, while the arrangement of popups, sliders, and menus is common to all versions.

The unzipping problem is also related to a problem of depth due to aggregation.

[1] Frameworks are based on the concept of providing stable foundations [33].

Encapsulation and inheritance only support change at one level of aggregation. Complicated objects should be hierarchically composed of smaller objects that address more narrowly defined pieces of the problem. Details of the aggregation are part of the outer object's implementation that should then be hidden from its clients. But, in taking full responsibility for its component objects, the outer object is made dependent on the final types of its components. Subclasses cannot replace components of the parent class—most languages allow methods, but not attributes, to be overridden. Details contained within an object's component objects cannot be changed without changing the type of the outer object. We saw this problem when we tried to change the class of the DepositItem's managed by the DepositReceiver. The encapsulation hides details immediately below the interface, but not deeper.

3.2.2 Multiple Inheritance

Some languages address the library blowup problem with multiple inheritance. Multiple inheritance allows a subclass to have more than one superclass. For example, the developers of the container recycling machine could use multiple inheritance to support combinations of receipt printing versus change giving, and measurement validation versus bar code validation. They would implement a separate class for each property, and then inherit from different combinations of two, e.g. receipt printing and bar code validation.

The usefulness of multiple inheritance is restricted by language specific limitations. In C++ and Eiffel, a multiply inherited class cannot call another of the multiply inherited classes. Suppose the developers wanted to offer a third receipt option that used information from the validation option to print brand or size information on the receipt. The new receipt class could not call the also multiply inherited validation class to get that information.

CLOS and Dylan linearize multiple inheritance [3, 21]. One multiply inherited class, implemented as a mixin class, can inherit from another multiply inherited class without naming the other in its implementation. The multiple inheritance in CLOS and Dylan, however, has limitations affecting design and reuse. If two inherited classes have a function of the same name, the language will merge the two functions. But existing classes, especially related classes, may accidentally use the same name for similar, but distinct functions. There are also cases where it would be desirable to inherit from the same class more than once. An example of this last situation occurs where the class contains context specific state, and you want an object that functions with the same type in two different contexts (e.g. as a node in two separate data structures).

3.2.3 Interface Types

The unzipping problem was created by objects depending on the name of a class for client objects [34]. Interface types allow an object to communicate with another object without naming its class. This can reduce some of the paths for the unzipping chain reaction. But objects that copy or create other objects, and classes that inherit from other classes, must still know the name of the other class. In addition, interface type names have dependency hierarchies of their own.

3.3 Patterns

Design patterns for dealing with change are presented in a popular book by Gamma, et al. [25]. A design pattern is a fragment of design that addresses a frequently encountered problem with an implementable solution. Several patterns, most notably the decorator and visitor patterns, present ways to introduce objects with new implementations at runtime without having to change the implementations of certain clients. The decorator pattern allows a series of features and behaviors (decorations) to be added to an existing object, while the visitor pattern allows a new function to be applied to an existing traversal (visits). These patterns use the same mechanisms of encapsulation and inheritance, with the same limitations discussed earlier.

The decorator pattern might be applied to the problem of adding new steps in succession to the DepositReceiver's addItem method. Each of the method's steps would be implemented as a separate decorator object. The DepositReceiver object and each of its decorator objects would be connected at runtime to implement the addItem method's steps in the appropriate order. New decorators can be added so long as they keep the same interface. In this situation, it is not unreasonable to assume that the original developer could predict that the addItem behavior would be augmented over the life of the application. It is less reasonable to assume that the developer could design an interface that would be sufficient for all of the additions. But decorators fall short of subclasses in other significant ways. Like a subclass, a decorator can locally define its own attributes and methods. But, unlike a subclass, those methods and attributes cannot be used by new clients or any other decorators.

Patterns often add structure in the original application in order to support future changes, at runtime, to specific objects in the design. Such design patterns use the model of isolating change in single objects, discussed earlier. They also assume that the changes can be anticipated, since the support mechanisms must be part of the original design.

> Good designers are aware of the changes that can prompt refactorings. Good designers also know class and object structures that can help avoid refactorings—their designs are robust in the face of requirements changes. A thorough requirements analysis will highlight those requirements that are likely to change during the life of the software, and a good design will be robust to them [25, p.354].

Even if there was a pattern for every change in the life of a system, it seems unlikely that every one of them could be anticipated when the system was first built. For those changes that can be anticipated, the supporting structure added by patterns like the Decorator add considerable complexity to the design.

> A design that uses Decorator often results in systems composed of lots of little objects that all look alike. The objects differ only in the way they are connected, not in their class or in the value of their variables. Although these systems are easy to customize by those who understand them, they can be hard to learn and debug. [25, p.178]

3.4 Summary and Conclusion

Change is driven by external requirements. Such requirements commonly concern externally observable behavior describable by use cases. In this chapter, we considered the maintenance problem of change in object oriented programs by looking at the problem of adding use cases to existing programs.

Traditional object oriented design addresses change with a design model that either implicitly assumes that change will be isolated in a single object or explicitly seeks to localize change in separate objects. Current approaches use encapsulation and inheritance to isolate the implementations of other parts of the application from the objects that change. But both the design model of isolating change in separate objects and the mechanisms of encapsulation and inheritance are inadequate for adding use cases to an existing application.

The assumption that change will naturally be isolated in a single object in a design based on entities in the problem space is unfounded, and in the case of adding a use case, simply false. Deliberately encapsulating changes in separate objects complicates designs in ways that make further changes progressively more difficult. With polymorphism, there is no support for adding to existing interfaces to support a new behavior. New steps can be added to existing behavior using polymophism and a carefully constructed design, but after each new step is added, the design becomes progressively less able to support change while reusing the existing code. Developers may eventually choose to refactor parts of an application's implementation. Rather than isolating this kind of change, subclass and client relationships propagate the effects of refactored implementation, requiring code duplication along entire networks of name dependencies. Duplication of this type (or failing to perform the duplication) further complicates the task of code maintenance. Factoring of features for families of application variants is also not well supported, contributing yet more code duplication.

None of the strategies for change discussed could produce the ideal implementation of the recycling machine after applying both changes without editing large parts of the original implementation. All of the common approaches to change leave artifacts of having been applied as a change both through factorings uniquely due to the ordering of the change, and additional structuring needed to perform the change.

Neither design approach of encapsulating complete entities nor encapsulating complete behaviors seemed especially suited to handling the problem of program evolution. In our search for an alternative approach to design, we found an approach, based on roles, that seems more promising.

Chapter 4

ROLES AND COLLABORATIONS: A BETTER APPROACH TO DESIGN

Our approach to supporting change is based on a design model using roles and collaborations. We originally described roles and collaborations in the introduction. In this chapter we present a more extensive discussion of how roles and collaborations are used in design, and an example of how they support change.

In the first section, we describe roles and collaborations at several levels and from different points of view. We also discuss how the same concepts, and related concepts, are used by others. In the second section we describe the issues posed by our goal of turning roles into implementable and composable components and the refinement process that we use to address those issues. The third section reinforces the earlier sections with a discussion of using role decomposition and refinement to manage complexity and facilitate reuse. The fourth, and final section describes how change is applied in the role and collaboration context.

This chapter is intended only to provide a theoretical understanding of the issues and concepts. The actual steps in the design process and the process of applying change are presented in Chapters 7 and 8. Chapter 6 presents specific idioms for handling common issues that arise in role composition.

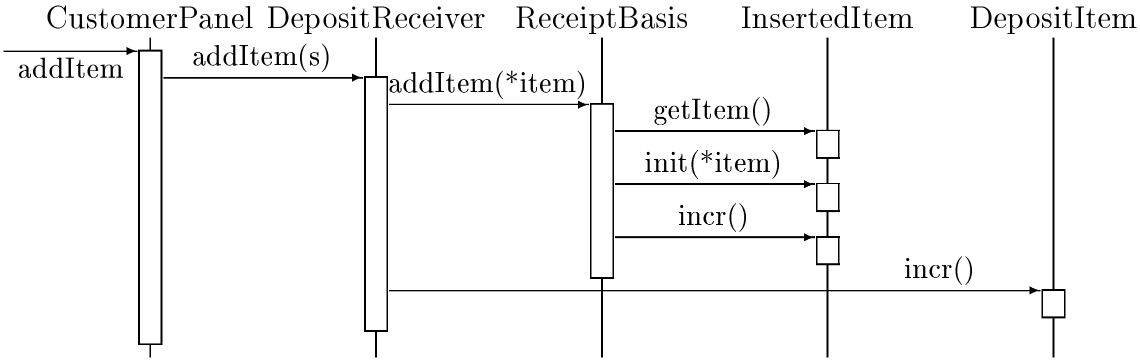

Figure 4.1: Diagram of interactions between objects for the Adding Item use case.

4.1 Collaborations and Roles

4.1.1 Definitions

Abstractly, a collaboration is a slice through an object oriented application from the point of view of a single concern. More concretely, a collaboration describes a set of objects together with obligations on them to address a particular concern, such as performing a task or maintaining an invariant. In our approach, we use collaborations to model the sequences of state changes and message passing in use case-like scenarios. In the container recycling machine, for example, the Adding Item use case is modeled as a collaboration involving the CustomerPanel, DepositReceiver, ReceiptBasis, InsertedItem, and DepositItem objects. The objects, and the messages they exchange to perform the adding item task, are shown graphically as an interaction diagram in Fig. 4.1.

We define a role as a part of a single object that addresses a particular concern. Using our earlier definition of collaborations, abstractly, a role is a slice through a single object from the point of view of a single concern. From the point of view of a collaboration, roles describe the parts of each object that address the concern of that collaboration. Collaborations, then, may be seen more as collections of roles than as collections of objects.

In the analysis, roles specify abstract responsibilities or obligations that an object must satisfy when it participates in a collaboration. In the design a role may specify the functions and values that an object must provide. In the implementation, a role may correspond to a single method, or a group of methods together with attributes. In the collaboration for the Adding Item use case, for example, the DepositReceiver's role requires that it have an addItem() method, a list of DepositItem objects, and access to the ReceiptBasis object. When the DepositReceiver's addItem() method is invoked by the CustomerPanel, it responds by selecting the DepositItem object for the type of the new container from its list of DepositItems, or adding a new DepositItem to the list if one for that container type does not exist. It then signals both the ReceiptBasis and the DepositItem that an item has been inserted before returning control to the CustomerPanel.

4.1.2 Properties

Roles provide a separation of concerns within objects. Objects are often involved in more than one collaboration. As roles, the different concerns from different collaborations are treated separately. The DepositReceiver object, for example, participates in both the Adding Item collaboration and the Print Receipt collaboration. In the Print Receipt collaboration, the DepositReceiver's role is to pass the receipt information from the ReceiptBasis to the ReceiptPrinter, and to tell the ReceiptBasis to reset the customer status.

Roles allow the creation of pieces of design addressing specific concerns without being tied to other details of the object, present in the final design. For example, the Adding Item use case defines DepositItem-like objects to hold information specific to each item type, such as its deposit value. The Print Item use case defines a similar object that holds deposit values and textual descriptions specific to each item type. The Validate Item use case also uses objects specific to item type to hold validation criteria. It seems logical to define these objects as being the same (and to use only one instance of the deposit value attribute). But in some designs the validation criteria could be separate, or part of some other object. Separating the activity of identifying roles within use cases from the activity of identifying common objects across use cases allows collaborations to be combined in different ways in different designs. This flexibility in assigning roles to objects gives collaborations special value in design reuse.

Roles provide a common link between models of functional behavior and models of static structure. Collaborations and their interaction scenarios model behaviors. Collaborations can be mapped directly to use cases in the requirements analysis. Objects and their classes model static structure based on entities. Objects in the design map directly to implementation classes. Roles are units of design common to both views. Figure 4.2 shows the two views of the container recycling machine superimposed on a single diagram.[1] The use cases are indicated by horizontal ovals and labeled on the left side. Implementation objects are shown as vertical rectangles and labeled at the top. The labels that appear where ovals and rectangles intersect indicate roles (e.g. `s1`). Roles can be grouped by collaboration (e.g. `{a1, a2, a3, a4}`) or by object (e.g. `{a3, p3, v3}`). Some objects do not participate in some collaborations and thus do not have roles for those collaborations. Viewed another way, not all collaborations use every object.

4.1.3 Related Uses and Concepts

In analysis, collaborations are often used informally, without being named or documented, to analyze how objects work together to address a concern [10, 12, 32, 78]. Collaborations have also been formalized in contracts [31], documented as patterns [24, 25], and associated with framework implementations [33, 55]. Our use

[1] The InsertedItem class was left out to save space.

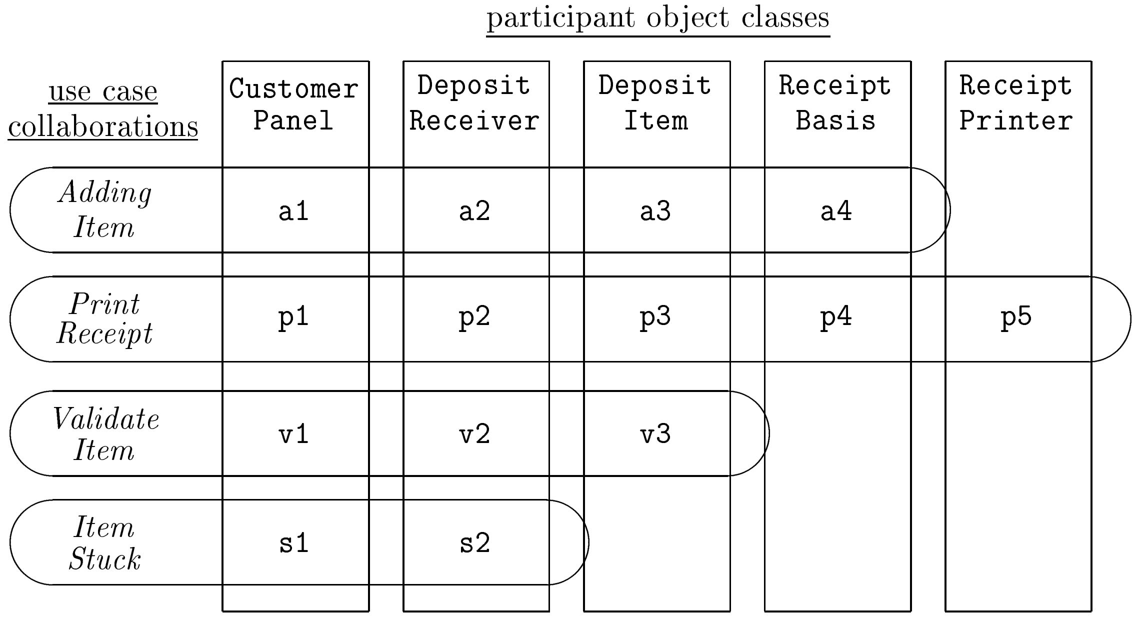

Figure 4.2: Conceptual view showing relationships among objects (vertical rectangles), collaborations (horizontal ovals), and roles (intersections).

of collaborations to reify use cases comes closest to that of Jacobson's OOSE and Reenskaug's OORAM approaches [32, 55, 56]. Neither the OORAM nor the OOSE approach carries the concept of collaborations into implementation. We compare design approaches in OORAM and OOSE to our own in more detail in Chapter 10.

Riehle has created a notation for describing the structural relationships in collaborations [58, 59]. Behavioral relationships in collaborations can be documented as interaction diagrams from Jacobson's OOSE, as shown in Fig. 4.1 [32]. Similar diagrams exist in other design methods, notably scenario diagrams in OORAM [55], timeline diagrams for use scenarios in Fusion [19, 41], and sequence diagrams in the Unified Modeling Language (UML) [61].

In Reenskaug's notion of a role, an object participates in a collaboration by playing a role in that collaboration [57]. The term comes from a theater metaphor. Objects are actors, a collaboration is an ensemble, and the concern is performing a play. In this sense, a role is an abstraction separate from any given actor—any actor can play a given role if he or she knows the part. In the OORAM approach, a role is something an actor can do, not something from which actors are constructed [55].

The notion of role has a counterpart in object-oriented databases [28, 76]. 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 and can be queried for specific roles. Roles, in this sense, correspond to the extension objects in Gamma's extension object pattern [42]. For database roles, the main issue is the ability of objects to dynamically change roles.

4.2 Issues and Role Refinement

In our approach we transform roles into implementable and composable design components in a process of refinement. Initially, the roles are considered in isolation, largely without regard for other collaborations. In composition, roles must not only interact with the other roles in their collaboration, but they must also coexist, and in some cases interact, with other roles in the same object. Aside from details of implementation, addressed in the next chapter, there are also conceptual issues concerning issues of overlap and dependency.

Roles are not always mutually exclusive. The concern of one collaboration may overlap with the concerns of others. Different collaborations may duplicate pieces of each other's behavior, and often operate on the same variables. The Print Receipt use case, for example, prints the container count that was computed in the Adding Item use case. Overlaps between roles must be addressed if roles are to exist and function as composable entities.

In the OORAM methodology, issues of overlap and dependence are ignored. The behavioral responsibilities of each object's roles are simply grouped together in a step called synthesis, prior to both implementation and the detailed design. It is left to each object's implementor to decide how to combine the roles. In the Responsibility Driven approach, where roles don't even exist in analysis, responsibilities are taken directly from the requirements scenarios and combined with others for each of an application's objects [77, 78]. Our goal is to defer any such mixing as long as possible. Once responsibilities from different concerns become mixed, it becomes much more difficult to change individual concerns.

In our approach, we isolate the shared and duplicated parts of roles in the process of role refinement. An important tool in this transformation is the ability to decompose collaborations and roles into smaller components addressing even narrower concerns. We decompose roles into overlapping and non-overlapping parts, each existing then as a component in its own right. Between two collaborations with a shared overlap, the overlap may be removed from one, creating a dependence on the other, or the overlap may be removed from both to form its own collaboration to specifically address the shared concern. In a few cases, the overlap may be harmless and the duplication is simply left as incidental duplication.

In the container recycling machine example, the Print Receipt role of the Deposit-Receiver object uses the same count information that was accumulated by the object's Adding Item role. In this sense, the two use cases overlap. Logically, the two roles should access the same variables. One way to address the overlap is to define a use case for maintaining counts that is separate from the activities of accepting containers or printing the count values. This separation is shown graphically in Fig. 4.3. The remaining Adding Item' and Print Receipt' use cases are then dependent on the new Maintain Count use case for the actual handling of the values. Abstracting out the common part, we maintain only one definition of each activity.

The Maintain Count use case corresponds to Jacobson's notion of an abstract use

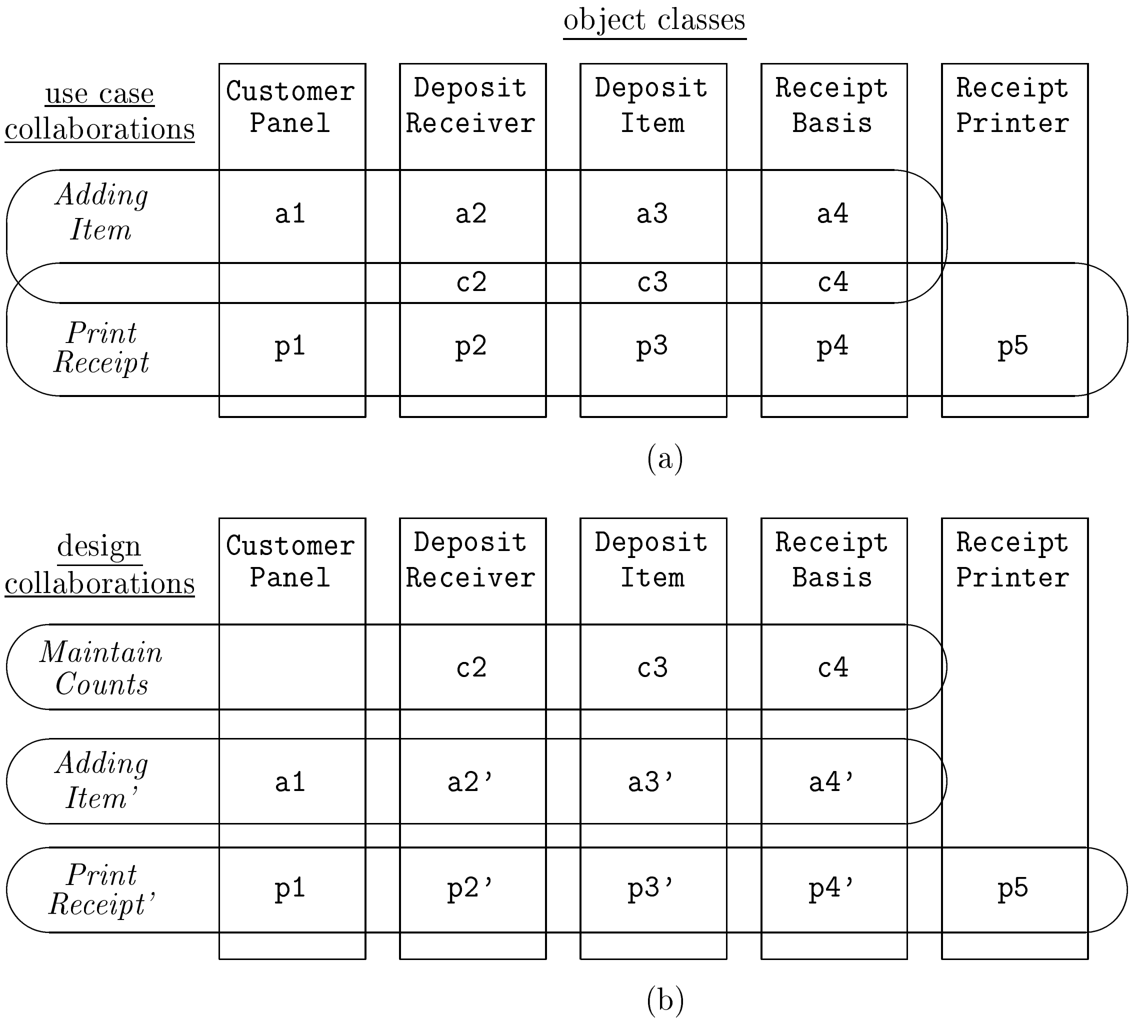

Figure 4.3: A graphical representation of overlap (a) and its removal as a separate collaboration (b).

cases—a use case with inputs and outputs not visible to the user, but, rather, tied to other use cases [32]. Rumbaugh referred to a use case providing services for other use cases as an embedded use case [60]. But the important property is not the user visibility or lack thereof. Rather, it is the uses relationship that exists between the Maintain Count use case and both the redefined Adding Item' and Print Receipt' use cases.

The alternative to isolating the overlap is to make only one of the two use cases responsible for the common part. The other use case is then defined as being dependent on the first—creating a uses relationship between the two. Since the count values in the above example are simple variables, it is also reasonable to leave the counts in the Adding Item use case and augment its responsibilities to include giving out their value and providing a reset function.

Roles can also be decomposed to reduce complexity or to isolate reusable parts. The DepositReceiver role of the Adding Item use case maintains a list of DepositItem objects. In the final design, the list might be implemented as a linked list, with the nodes being DepositItem objects. We can separate out the basic linked list data structure with list and node roles to be treated as a separate collaboration in the design. The list role defines a head pointer and setNext() and getNext() methods (used in the DepositReceiver object), while the next pointers are defined in the node role belonging to each DepositItem object. Figure 4.4 shows a separate linked list data structure collaboration being used for both the DepositReceiver's list of DepositItem objects and the ReceiptBasis' list of InsertedItem objects from the Adding Item collaboration.

When removing part of the concern of a collaboration, to be addressed in a separate collaboration, the integrity of each collaboration must be maintained. Conceptually, the collaboration must still describe a set of objects interacting to address a concern. The new concern may, however, be less than the original concern before the decomposition. Stated concretely, if a role calls another object as part of its responsibilities,

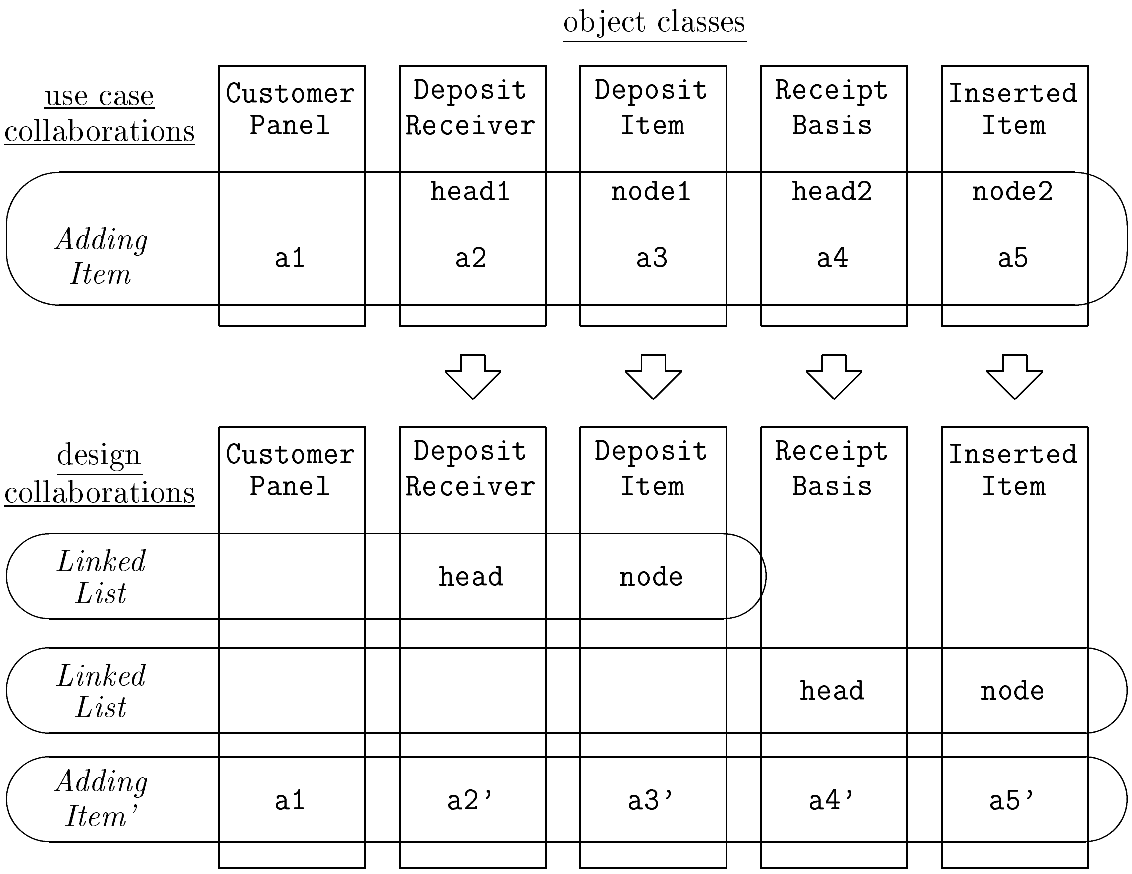

Figure 4.4: A decomposition to separate the common concerns of a linked list data structure into generally (and repeatedly) useful components.

as illustrated in Fig. 4.1, that other object must have a role from the same collaboration to field the call. On the other hand, within an object, the details of how a role carries out its responsibilities may involve a call to another role in the same object. The uses relationship between collaborations should be seen only within objects, so as not to violate the model.

4.3 Managing Complexity and Supporting Reuse

Roles represent an intermediate level of abstraction between an object and methods or attributes. Objects can be described in terms of the roles they play in one or more use case scenarios. The purposes of methods and attributes can then be understood in the contexts of the use case collaborations in which they serve. Conversely, roles allow a use case's requirements to be described in terms of the participation of the design objects and their roles, making it easier to trace the requirements into the design.

In the earlier examples of dependencies between collaborations, the actual dependencies were restricted to occurring between roles within the same object. This highlights another important aspect of role based design—dependencies are associated with a design context of manageable size. Dependencies between collaborations occur within objects. Dependencies between objects occur within collaborations. In the design process, dependencies between collaborations are addressed through the objects that they share. In maintenance, an object's dependency relationships can be found and understood through the collaborations in which it participates. Like the Law of Demeter, this channeling of dependencies aids in understanding by reducing the potential scope of what must be understood [40].

Roles represent potential units of reuse. A role specifies behavior and invariants specific to its concern. Any object that satisfies the specification can play that role. For example, the ReceiptPrinter object's role in the Print Receipt use case (labeled `p5` in Fig. 4.2) handles the interface to the printer and tells it what to print. In a different design, the printer interface might be included as part of the CustomerPanel object. In that case, that `p5` role in the Print Receipt collaboration would be performed by the CustomerPanel object. The role itself is not changed by the fact that it is being performed by a different object. A role is also not changed if the object playing it changes in unrelated ways. The addition of Print Receipt's `p5` role to the CustomerPanel does not change that object's roles in the Adding Item or Validate Item collaborations, or even its other `p1` role in the Print Receipt collaboration.

Collaborations can describe general concerns that are usable in many contexts. The collaboration for the list data structure, described above for holding DepositItem objects, is general enough to be used in many contexts. The collaboration's list manager role could be played by an object contained by the DepositReceiver, or by the DepositReceiver object itself. In the design of the container recycling machine presented in Jacobson's book, the ReceiptBasis object had a list of InsertedItems to count the customer's inserted items by item type [32]. The list data structure collaboration could be reused to handle the second list. In this case the node roles would be played by objects of the InsertedItem type. The same collaboration could be reused again in other applications to describe yet other lists. The container recycling machine is a small example with little repetition. In larger programs, much more repetition is likely.

An object can play more than one role in the same collaboration. In the previous example of moving the Print Receipt's `p5` role to the CustomerPanel object, the customer panel object played two distinct roles in Print Receipt collaboration. The `p1` role has an interface to the receipt button and initiates actions when the button is pressed. The `p5` role has the interface to the printer and tells the printer when and how to print the receipt. In refinement, roles can be decomposed without creating new collaborations. If a role contains more than one decision, separating it into component roles within the same object may allow decisions in one role component to change without affecting the other components of the original role. In the Item Stuck collaboration, the `s1` role in Fig. 4.2 interfaces to the container feed mechanism to check if it is stuck, and to an alarm to signal the stuck condition. Creating separate roles for the two functions would allow the alarm response to be replaced by using e-mail, for example, while leaving the feed interface untouched. The alarm part of the CustomerPanel could also be reused in another collaboration in connection with a different problem needing operator attention.

When decomposing collaborations into roles, we prefer to err on the side of creating too many roles, rather than too few roles. Being too conservative in decomposition may result in defining a role that spans two objects in the final design. In the next chapter, we describe a method of implementation for which there is little or no cost for decomposition within the same object—the compiler inlines it out.

Our approach to design analysis stresses both top down decomposition and stepwise refinement. In top down decomposition, models of concerns are decomposed into compositions of smaller models until a manageable level of complexity is attained for each model. In stepwise refinement, general models of concerns are elaborated with structure and details in iterative steps until the models are specific enough to correctly model the solution as implementation code. In our approach, a list of requirements use cases is both decomposed and refined to achieve the final design.

4.4 Supporting change

In our design process, new requirements appear in the form of new use cases. Requirements that add deviations or exceptions to an existing use case appear as extension use cases. As we explained in Chapter 2, a behavior in an extension use case may start or end within the behavior of the base use case, possibly overriding part of the original behavior [32]. At the point of departure from the base collaboration, the role from the extension collaboration overrides the original sender or recipient of a message from the base collaboration. The ability of one use case to override behavior or insert behavior in the sequence of behavior defined by another use case is an important factor in our approach's ability to support change.

In the container recycling machine, both the Validate Item and Stuck Item use case replace or insert new behaviors in the existing Adding Item use case. The Item Stuck collaboration's role in the DepositReceiver overrides the addItem() method from the base role to intercept the addItem message. After the Item Stuck behavior has been performed, the Item Stuck role then calls the original addItem() method in the base role. The Validate Item collaboration overrides the addItem() method in the CustomerPanel object which calls a different addItem method (with more arguments in its signature) in the Validate Item role for the DepositReceiver object. After the Validate Item behavior has been completed, its role in the DepositReceiver object calls the original addItem() method from the DepositReceiver role of the base collaboration.

When changes are applied, existing objects and use cases can be further decomposed to accommodate the change within the existing structure. In the above example, the new behaviors from the Validate Item and Stuck Item use cases could be added within the existing DepositReceiver object while still retaining the original role from the Adding Item use case and also without affecting the role from Print Receipt use case. However, suppose the validation behavior had been included as part of the original Adding Item use case. We'll call this single use case Adding Valid Item. We might then want to insert the check for a stuck container within the behavior of the DepositReceiver's combined Adding Valid Item role. At the time of adding the Stuck Item change, we could decompose the Adding Valid Item role into Validate Item and Adding Item parts to support the new Stuck Item behavior. The Adding Valid Item collaboration would then have two smaller roles in the DepositReceiver object, but would otherwise be unchanged.

In this chapter we described an approach to design that created opportunities for change, and reuse, by decomposing objects into smaller role components, and, occasionally, decomposing roles again into even smaller components. The role approach allows the application to be viewed both in terms of entities and in terms of behaviors, and supports a separation of concerns along both axes, simultaneously. Behavioral changes that, in other designs, would cut across objects are captured in a role design by roles in each of the affected objects.

In existing role based approaches, like the OORAM approach, the benefits of roles extends only as far as the synthesis step. To get the full benefit of a role based design, the same roles must also exist in the implementation. Roles that exist only in the analysis will indicate which objects are directly affected by a change. But they will not tell us how much of each object to change, nor what affect the change will have on concerns in other roles.

> The synthesis mechanism is not without its formal problems. This is particularly apparent if we later modify one of the basic models, i.e. we later revise our understanding of a subarea and the corresponding role model. In general, we then have to reconsider all synthesis operations to ascertain that they are still valid as descriptions of parts of the larger area of concern. It is an area of further research to put the concept of synthesis on a more solid formal foundation [57].

Chapter 5

ROLE ORIENTED IMPLEMENTATION

Building a standard object oriented implementation from a role oriented design is a straightforward task. A role oriented design is an object based design with additional intermediate details about the partitioning of concerns into roles. But, without preserving the partitioning of objects into roles, isolated changes to roles in the design will lead to costly rework of larger parts of the implementation.

In this chapter we extend roles from the design into source code components. The method of implementing roles must not only preserve role identities, but also support many of the same properties as roles in the design. These properties include the ability to be reassigned to different objects, composed in different groupings and different orderings, and replaced by a decomposition into smaller pieces fulfilling the same obligations. To support our method of change, they must also be able to extend and override the behaviors of other roles within the same object.

For choosing a method of implementation, three specific properties of roles and role composition stand out.

1. When added to the implementation of an object, a role extends the interface of that object.

2. Roles can be composed in different combinations in different objects, even within the same application.

3. Roles place no restrictions on the objects of the other roles with which they collaborate other than that they perform the appropriate role for that collabor-

ation.

Based on above three properties, we derived the following three properties of the implementation.

1. Roles should be composed using some form of inheritance or delegation.

2. The implementation of a role should defer the choice of class or object from which it inherits or to which it delegates for a later, separate specification of composition.

3. The implementation of roles should also defer any specification of the class or object of any other roles with which it collaborates, although it may specify the part of its interface (type) used in the collaboration.

A more detailed discussion and analysis of properties needed for role implementation is included in Appendix A.

In this chapter we present a method of implementation for role based designs. In our approach, roles are implemented as source code components—specifically, class templates defined in a stylized way—and composed into classes using separate specification statements—classes defined in terms of instantiations of those class templates. Our approach requires no special development environment and uses only standard features available in the C++ language. In our earlier papers, we compared our template approach to more traditional subclassing for frameworks [74] and demonstrated its use for decomposition in general [73].

Our intent is not to imply that this method of implementation is the only available approach to supporting roles in the implementation. There is other work on object factoring, some of which may provide equivalent support. In Chapter 10, we mention two other approaches, one using a Smalltalk metaclass and another using a preprocessor that we believe would work as well, and perhaps exhibit other useful properties.

```
template <class CustomerPanelType,
          class DepositItemType,
          class ReceiptBasisType,
          class SuperType>
class AddingItemDRRole : public SuperType {
    CustomerPanelType *customerPanel;
    ReceiptBasisType *receiptBasis;
    LinkedList<DepositItemType> depositItemList;
    ...
};
```

Figure 5.1: Partial implementation of the DepositReceiver role in the Adding Item collaboration.

Our goal in this thesis is to present a development approach that uses roles to support software evolution. To that end, we need a method of implementation to round out the process. The method presented here not only fulfills the basic requirement of supporting roles, but also provides static type checking and an especially efficient runtime implementation. In this chapter, and in the subsequent discussion of our development process, we will assume only the one method of implementation. But most of the discussion of issues unique to C++ templates are confined in this chapter.

5.1 Roles as Class Templates in C++

The six properties needed for role implementation, described above, can be satisfied using type parameterization if it also includes parameterized inheritance or delegation. In our approach, we implement roles as class templates in C++ and compose roles into classes by instantiating their templates with the appropriate bindings.

For each role, we define a separate class template that is parameterized by each of the collaborators. For example, the DepositReceiver role in the collaboration for the Adding Item use case might be defined in part as in Fig. 5.1.

The CustomerPanelType, DepositItemType, and ReceiptBasisType parameters

```
template <class CustomerPanelType,
          class ReceiptBasisType,
          class ReceiptPrinterType,
          class SuperType>
class PrintReceiptDRRole : public SuperType {
    CustomerPanelType *customerPanel;
    ReceiptBasisType *receiptBasis;
    ReceiptPrinterType *receiptPrinter;
    ...
};
```

Figure 5.2: Partial implementation of the DepositReceiver role in the Print Receipt collaboration.

indicate that the AddingItemDRRole will collaborate with one or more objects with an as yet unknown class playing each of the following roles: the CustomerPanel role, a DepositItem role and the ReceiptBasis role. The SuperType parameter is used in every role definition in our approach, since every role is itself part of some as yet unknown class. The AddingItemDRRole template corresponds to the `a2` role in Fig. 4.2. The Adding Item collaboration also includes an InsertedItem role, but the DepositReceiver role does not interact with it directly.

The DepositReceiver role in the PrintReceipt collaboration might be similarly defined, in this case collaborating with CustomerPanel, ReceiptBasis, and ReceiptPrinter roles, as shown in Fig. 5.2. In the two templates in Figs. 5.1 and 5.2 we used the same names for corresponding classes (e.g. CustomerPanelType) in the different collaborations. However, we could as well have used different names—the CustomerPanel in one collaboration may correspond to the same object as the SlotExaminer in a different collaboration.

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 PrintReceiptDRRole might, for example, appear as in Fig. 5.3. The declaration

```
class DepositReceiver02Class
 : public PrintReceiptDRRole < CustomerPanelClass,
                               ReceiptBasisClass,
                               ReceiptPrinterClass,
                               DepositReceiver01Class > {};
```

Figure 5.3: Template instantiation of the Print Receipt role in the DepositReceiver object's class.

```
class DepositReceiver01Class
 : public AddingItemDRRole < CustomerPanelClass,
                             DepositItemClass,
                             ReceiptBasisClass,
                             emptyClass > {};
```

Figure 5.4: Template instantiation of the Print Receipt role in the DepositReceiver object's class.

in Fig. 5.3 says that the DepositReceiver02Class includes the PrintReceiptDRRole. It also says that specific classes, CustomerPanelClass, ReceiptBasisClass and DepositItemClass, play the CustomerPanel, DepositItem, and ReceiptBasis roles in the collaboration with the DepositReceiverPRRole. DepositReceiver01Class is the class that we extended to get DepositReceiver02Class.

The DepositReceiver01Class might be defined in terms of the AddingItemDRRole. Its instantiation appears in Fig. 5.4. This instantiation statement defines DepositReceiver01Class as a collaborator with objects of the CustomerPanelClass, DepositItemClass and ReceiptBasisClass playing CustomerPanel, DepositItem and ReceiptBasis roles, respectively, in an Adding Item collaboration. The emptyClass (essentially a default base class) parameter simply indicates that the DepositReceiver01Class is a base class.

The ReceiptPrinterClass, which has only one role and no references to other collaborators, might be composed as in Fig. 5.5. Class names like ReceiptPrinter01Class

```
class ReceiptPrinter01Class
 : public PrintReceiptRPRole < emptyClass > {};
typedef ReceiptPrinterClass ReceiptPrinter01Class;
```

Figure 5.5: Template instantiation for the one role ReceiptPrint class.

and DepositItem02Class refer to classes at intermediate stages of composition. The final typedef statement aliases the externally used class name to the class at the appropriate level of composition. Our choice of using class declarations and typedefs is dictated by C++ considerations for supporting forward reference (class declaration) and permitting the addition of a constructor in the final declaration (typedef).

5.2 C++ Issues

The next chapter presents a collection of implementation idioms that address issues of composition and control flow. Some of those idioms encounter issues specific to C++. In this section, we describe language specific issues and their resolution.

5.2.1 Typedef Aliasing Versus Class Definition

Forward reference, as mentioned earlier, is one of the reasons we don't often use typedef statements to instantiate template components. In a forward reference, we wish to refer to a class that has not yet been defined. We need forward reference to allow circular references, e.g. two classes that each have a pointer to instances of each other's class.

A typedef creates an alias name, not a class name. In the first set of declarations in Fig. 5.6, `foo1` is an alias, but the real class name is `boo<empty>`. The name `foo2` is an alias for `moo<boo<empty>>`. We cannot forward declare an incomplete `foo2`, using `typedef foo2`. If we forward declare `foo2` with `class foo2;`, we cannot redefine the `foo2` name in a typedef later on; it must be a class definition.

```
typedef boo<empty> foo1;
typedef moo<foo1> foo2;

class goo1 : public boo<empty> {};
class goo2 : public moo<goo1> {};
```

Figure 5.6: Corresponding but not equivalent uses of typedef and class definition.

Many compilers name template instantiations by concatenating the template name with the names in the bindings. Lists of typedef instantiations of nested templates can lead to long names that are ugly to look at in the debugger and that quickly overflow name buffers. The `moo<boo<empty>>` name in the above example above illustrates how the names can grow with nesting. By placing each template instantiation within the declaration of a concrete class, the name nesting never occurs. The concrete class, though otherwise empty, provides a short concrete class name.

The class declaration form of composition is not without its disadvantages. A minor disadvantage is that each definition creates two classes in the hierarchy. The definition of `goo2` in Fig. 5.6 produces a class called `moo<goo1>` and an empty subclass, called `goo2`. A bigger disadvantage is that the moo template cannot use a constructor with arguments, because the `goo2` subclass does not define a constructor. If we wish to define a component with a class constructor, we compose it in the most derived position in the class, and use a typedef to instantiate it. An example of using constructors appears in the discussion of proxies and handle initialization in Chapter 6.

5.2.2 Callbacks to Methods

In C++, callbacks to methods are complicated by the fact that methods can't be called with function pointers—method calls need an additional "this" pointer to the object instance that is not part of a normal function call. Instead, we must define a non-member function for each class whose member we wish to call. The non-member function takes an object pointer, in addition to the arguments of the call, and uses that object pointer to call the method.

The resizeCallback template in Fig. 5.7 shows the callback function for a resize() method. By using a template function and parameterizing the object type, we define only one template for all registered resize() callbacks. The variable declaration for the rfunc pointer in the resizeRegisterMethod instantiates a function specific for the class of the listener's SuperType. The rfunc pointer and object pointer are then cast to the more general forms used by the announcer. When resize() is called by the function, it will be specifically targeted at the component before the ResizeListener component in the composition.

5.3 Discussion

Templates are not new to software developers. But parameterized inheritance, while available, is not as well known nor often used. Stroustrup's C++ manual includes one obscure example involving private inheritance[68]. Among those who use it, it has always been treated as a technique to be applied in special case situations, e.g. composing a family of components from layered features. Using parameterized inheritance for every class involves a leap to a different style of programming.

Only a few additional lines of code are needed to turn a code fragment into a template. The deferred decisions about the types of the collaborators are assigned parameter names which are then used in the source code to implement the role. The code is written as if the other types are known. The process is thus straightforward. Certain issues may arise in implementing two templates that each use a parameter that will be bound to the type of the other. We describe ways of addressing the mutual dependence issue in the section on forward reference in the next chapter.

By using inheritance for composition, the interfaces of all the roles in a composition

```
// Type declaration for announcer's general callback function
typedef void (*ResizeCallback)(int width, int height, void* obj);
// Struct to store callbacks with their object pointers
struct ResizeCallbackNode {
   ResizeCallbackCell() : callback(NULL), obj(NULL) {};
   ResizeCallback callback;
   void* obj;
};
// Announcer component
template<int MAXCALLS, class SuperType>
class ResizeAnnouncer : public SuperType {
   ResizeCallbackCell[MAXCALLS] resize_callbacks;
public:
   void registerResize(ResizeCallback callback, void* obj) { ... }
   void unregisterResize(ResizeCallback callback, void* obj) { ... }
   void resize(int width, int height) {
      // Call each registered callback in turn
      for(int i=0; i<MAXCALLS && resize_callbacks[i].obj != NULL; i++)
         resize_callbacks[i].callback(width, height,
                                      resize_callbacks[i].obj); }
};
// Template declaration for listener-specific callback function
template <class ObjectType>
void resizeCallback(int width, int height, ObjectType* obj) {
   obj->resize(width,height); }
// Listener component
template<class SuperType>
class ResizeListener : public SuperType {
public:
   void registerResizeListener() {
      // Declaration instantiates function template for SuperType
      void (*rfunc)(int,int,SuperType*) = &resizeCallback;
      registerResize(ResizeCallback(rfunc),(void*)this); }
   void unregisterResizeListener() {
      void (*rfunc)(int,int,SuperType*) = &resizeCallback;
      registerResize(ResizeCallback(rfunc),(void*)this); }
};
```

Figure 5.7: Code for callback collaboration to register and call an object's resize method as a callback.

are composed into a single interface. Components that call methods in other roles do not need to know if two calls are to the same role or separate roles. This allows one role to be replaced by two smaller roles, or two roles to be combined, without affecting clients. Also, when roles call other components within the same object, they simply call the method on their inherited superclass interface. There is no composition syntax involved.

Template instantiations to form the implementation of application objects is performed by sequences of class declarations, as described above. For large applications, the lists of template instantiations can be quite long, making the task of writing out the instantiations somewhat tedious. Although the statements describe the structure of the application, it is difficult to perceive that structure from a quick glance at the code. In Chapter 7 we present a visual representation of the application structure. All of the information needed to produce the template instantiations is available from the visual representation.

The composition is of roles into classes is performed statically at compile time. All of the interfaces between roles within an object and between objects are checked at compile time when the compiler composes the application. Method calls that cannot be bound to methods are reported as compile time errors, when the template instantiations are compiled, or link time errors, if they involve calls between objects instantiated in different files.

Because non-virtual methods are used, the compiler is free to inline method calls within objects. Inlining saves not only the indirection of dynamic binding, but also the context switch overhead of the call itself. In a comparison with a traditional framework implementation for a graph traversal example, our approach yielded a factor of two improvement in speed, in addition to the added compositional flexibility, that was our real objective [74]. Dynamic binding is still available for things that really do vary at runtime. But because of our style of composition by template instantiation, dynamic binding is not needed to support overriding in evolution or specialization.

The template method of implementation separates the implementation of roles—written as templates—from the composition of roles—specified as template instantiations. The method of implementation itself is fairly easy to describe and use. Its use within our design approach, however, has many facets, as described in the next three chapters.

Chapter 6

IDIOMS FOR ROLE ORIENTED IMPLEMENTATION

In this chapter, we present a set of idioms that address structural and compositional concerns. They address semantic and syntactic dependencies between roles, and wider issues of control flow. The use of these idioms allows the roles defined in collaborations to be implemented in a way that is independent of the context of their use.

The idioms presented here are consistent with, and enhance, our compositional model of programming. The developer controls bindings and control flow within objects, and also between objects, by inserting components in the composition. Problems that may result from certain compositions can be resolved by the addition of other components.

Several of the idioms described here have been used in one form or another in other contexts. In each case, however, the idiom acquires a unique use, a unique implementation, or both, when applied to our approach of template component composition. We have attempted to cite significant prior work, where we are aware of it, and to retain the original terminology. The role decomposition idiom, which is central to our approach, and the data structure idiom are new to this work.

6.1 Idioms for Semantic Issues of Composition

In the design, the semantics of role composition can take many forms. Roles can extend the behavior of other roles, replace the behavior of other roles, or augment the behavior of other roles. The idioms in this section describe how to achieve particular semantic effects in the physical composition of role components.

6.1.1 Method Call Interception

Calls between two components can be intercepted by a third component that defines a method with the same name and signature as the call to be intercepted. We simply place the third component between the other two. Call interception is an important mechanism for applying change. It allows us to replace existing behavior on a permanent or conditional basis, without editing either of the original components. By including code in the new method to also call the original method, though the inherited SuperType interface, we can add behavior within an existing sequence without actually replacing anything.

Method intercepting components can also be used to temporarily monitor the traffic in certain method calls. In a monitor component, the intercepting methods might store or report the arguments in their calls and then call the same method prefixed by the SuperType.

6.1.2 Role Decomposition

Component templates allow role implementations to be decomposed into two or more smaller components, while hiding this decomposition from the other parts of the application. Suppose we have a role template, called BigRole, as shown in Fig. 6.1. BigRole can be decomposed and implemented by two components, Part1Role and Part2Role, and shown in Fig. 6.2.

The definition of the BigRole template in Fig. 6.2 is interchangeable with the definition in Fig. 6.1. Because of the inheritance order between the two parts, no code in the Part1Role should try to access item_b or methodD() defined in the Part2Role component.

An application class could be composed from EarlierUndisturbedRole, BigRole and LaterUndisturbedRole role templates as shown in the upper part of Fig. 6.3.

```
template <class SuperType>
class BigRole : public SuperType {
public:
   int item_a, item_b;
   void methodC() {...};
   void methodD() {...};
}
```

Figure 6.1: A single role component called BigRole.

```
template <class SuperType>
class Part1Role : public SuperType {
public:
   int item_a;
   void methodC() {...};
}

template <class SuperType>
class Part2Role : public SuperType {
public:
   int item_b;
   void methodD() {...};
}

template <class SuperType>
class BigRole : public Part2Role<Part1Role<SuperType> > {};
```

Figure 6.2: BigRole decomposed and implemented by two separate components.

```
class appClass1  : public EarlierUndisturbedRole<empty> {};
class appClass2  : public BigRole<appClass1> {};
class appClass   : public LaterUndisturbedRole<appClass2> {};

class appClass1  : public EarlierUndisturbedRole<empty> {};
class appClass2a : public Part1Role<appClass1> {};
class appClass2  : public Part2Role<appClass2a> {};
class appClass   : public LaterUndisturbedRole<appClass2> {};
```

Figure 6.3: Two alternative compositions using either the BigRole component or its two smaller part components.

The same appClass could be composed using the decomposed parts, Part1Role and Part2Role, as shown in the lower part of Fig. 6.3.

There are many reasons for splitting a role. The original role might encode two or more separate decisions that are independently changed, or omitted, in different versions of the application. The original role might contain a sequence of behaviors to which we want to add an additional processing step somewhere in the middle. Splitting the sequence into two parts in separate roles, with a method call in between, allows us to interpose the additional, or alternative, processing where needed in some versions, but not others.

The importance of the ability to decompose a role and use a composition of the new parts in place of the original role is that it is not necessary to choose the right decomposition for every future change when the application is first designed. Splitting a role in an existing design only affects the role that is split.

If, when applying a change, it is necessary to split, or upgrade, an existing component to accommodate the change, the split, or upgraded, components should be applied in the context of the original application first. This allows the subdivision to be tested and verified in comparison with the original, before being placed in the new application.

```
template<class SuperType>
class Ignition : public SuperType {
public:
   start() { ... }
}

template<class SuperType>
class Interlock : public SuperType {
   int locked;
public:
   int isLocked() { return locked; }
   void lock() { lock = 1; }
   void unlock() { lock = 0; }
}

template<class SuperType>
class IgnitionLiftsInterlock : public SuperType {
public:
   void start() { if(!isLocked()) SuperType::start(); }
}
```

Figure 6.4: The implementation of separate Ignition and Interlock components and a lifter to add the interlock feature to the Ignition interface.

6.1.3 Lifters

A lifter coordinates the behavior of two components by lifting the semantics of one into the interface of the other [54]. Each of the two components addresses its own concern in isolation, while lifters address their interaction. For example, consider a car's ignition system and a separate interlock feature. The ignition has only a start() method. The interlock maintains a lock state and provides isLocked(), lock(), and unlock() methods. A lifter to add the lock feature to the ignition semantics would implement a start() method to override the start() method of the ignition. The implementation of both components and the lifter is shown in Fig. 6.4.

We then add a transmission neutral sensor to allow the car to start when the

```
template<class SuperType>
class Neutral : public SuperType {
   int neutral;
public:
   int inNeutral() { return neutral; }
   void shiftToNeutral() { neutral = 1; }
   void shiftToGear() { neutral = 0; }
}

template<class SuperType>
class NeutralLiftsInterlock : public SuperType {
public:
   void shiftToNeutral() { SuperType::shiftToNeutral(); unlock(); }
   void shiftToGear() { SuperType::shiftToGear(); lock(); }
}
template<class SuperType>
class InterlockLiftsNeutral : public SuperType {
public:
   void lock() { if(!inNeutral()) SuperType::lock(); }
}
```

Figure 6.5: The implementation of a neutral sensor and two lifters to coordinate it with the Interlock feature of Fig. 6.4.

transmission is in neutral, and prevent the car from being started, under normal circumstances, if it is in gear. Figure 6.5 shows the implementation of a neutral sensor and the two lifters to coordinate its behavior with the interlock feature.

In a car with a manual transmission, we can add a clutch sensor that allows the car to be started if the clutch is depressed, but prevents the car from being started, under normal circumstances, if the clutch is engaged. Figure 6.6 shows the implementation of a clutch sensor and the two lifters to coordinate its behavior with the interlock feature.

We can compose these components and their lifters in a number of ways to handle cars that require the transmission to be in neutral to start, {Ignition, Interlock, Neut-

```
template<class SuperType>
class Clutch : public SuperType {
   int engaged;
public:
   int isEngaged() { return engaged; }
   void engage() { engaged = 1; }
   void disengage() { disengaged = 0; }
}

template<class SuperType>
class ClutchLiftsInterlock : public SuperType {
public:
   void engage() { SuperType::engage(); lock(); }
   void disengage() { SuperType::disengage(); unlock(); }
}
template<class SuperType>
class InterlockLiftsClutch : public SuperType {
public:
   void lock() { if(isEngaged()) SuperType::lock(); }
}
```

Figure 6.6: The implementation of a clutch sensor and two lifters to coordinate it with the Interlock feature of Fig. 6.4. interface.

ral}, cars that require the require the clutch to be disengaged to start, {Ignition, Interlock, Clutch}, and cars that can start if either the transmission is in neutral or the clutch is disengaged, {Ignition, Interlock, Clutch, Neutral}. The reason for separating the two directions of lifting is evident in the last composition, where the InterlockLiftsClutch lifter must appear in the base class of the NeutralLiftsInterlock lifter, while the corresponding InterlockLiftsNeutral lifter must appear in the base class of the ClutchLiftsInterlock lifter.

The lifter idiom is another illustration of our building block style in which the choice of blocks and order of composition control the semantics. The main benefit of using lifters is that they allow the core components to address a specific concern without the confusion of how it should interact with other concerns. They then allow the core components to be semantically combined with a range of other components. In this way, they are related to Sullivan's mediators [69]. By using the semantics of inheritance to detect and coordinate responses to events, lifters avoid the initialization and runtime overhead of implicit invocation. Because lifters are pair-wise and directional, it may require several lifters to duplicate the coordination of a more general mediator.

6.1.4 Composable Data Structures

Traditional data structure libraries encapsulate data structures as single monolithic components. Often such libraries must either restrict the choice of features, or provide large numbers of similar data structures to cover different combinations of features [7]. Monolithic data structures also have difficulty interacting with the data they hold, the clients that use them, and other data structures. For example, clients must pass in separate components to perform data comparisons for sorting. The result of a search returns yet another component, e.g. an iterator, which then must be queried to get the actual data. By providing a set of subcomponents with which different data structures can be composed, we support user-customizable choices of features,

```
template<class NodeType, class SuperType>
class ListNode : public SuperType {
  NodeType *next;
public:
  ListNode() : next(NULL) {}
  NodeType* getNext() { return next; }
  void setNext(NodeType* n) { next = n; }
};

template<class NodeType, class SuperType>
class LinkedList : public SuperType {
  NodeType* head;
public:
  LinkedList() : head(NULL) {}
  NodeType* getHead() { return head; }
  void listAppend(NodeType* here, NodeType* n) { ... }
  void listRemove(NodeType* n) { ... }
};
```

Figure 6.7: Two components for a simple linked list implementation.

and closer integration between data structures and other code.

As with other objects, we subdivide data structures into smaller components addressing separate concerns. Application developers can then choose components for the allocation strategy (e.g. heap or pooled), the sorting algorithm (e.g. optimized for nearly sorted or randomly sorted lists), and insert and find behaviors (e.g. eager or lazy sorting). The allocation component can be reused for both trees and lists.

But decomposing data structures allows us to do other things as well. By providing separate node components, nodes can not only hold client data, they can also be client data. Figure 6.7 shows an implementation of two simple templates for a linked list and its nodes. The NodeType in instantiations of both the ListNode and the LinkedList must include a ListNode in its composition.

Figure 6.8 shows class declarations for a client of the simple linked list. The class

```
// Class declaration for forward reference.
class ClientData;
// Data is the class that client would have used for data
// ClientData is the specialized class that client actually uses
class ClientData : public ListNode<ClientData,Data> {};
// List of ClientData is a base class
class List : public LinkedList<ClientData,empty> {};
// ClientOfList inherits from List (instead of containing one)
class ClientOfList : public Client<List> {};
```

Figure 6.8: Class declarations for client that uses the LinkedList.

called Data is the class that the client would normally have used for data. The new class, ClientData, is a subclass that specializes Data by adding the ability to function as a node in a LinkedList. The LinkedList template adds the property of having a list head, and operations on that list, to any object into which it is composed. Here it is instantiated as a base class called List. Because the client's code derives from List in the declaration of ClientOfList, client code can call list methods locally (e.g. listRemove(node) instead of list.listRemove(node)). While it seems subtle, the advantage of list inheritance over list containment is that there is no instance name (e.g. "list") on which list-using code has to agree. Reducing sources of context dependence makes components, like the Client component, easier to reuse.

When the client's code calls getHead() or getNext() on a node, the pointer it gets back is a pointer to its data. Client code that is written to operate on Data does not have to be changed for operating on ClientData. In particular, there are no extra dereference or access operations. Yet, the data is also a node, and, as such, a place holder in the list. The client finds the next element in the list by calling getNext() on its data. There is no reason for using a separate iterator.

Any data that the client appends to the list must be an instance of ClientData. In our style of deferring bindings to type, all objects know the full types of all other

```
template<class NodeType, class SuperType>
class TreeNode : public SuperType {
   NodeType *right, *left;
public:
   TreeNode() : right(NULL), left(NULL) {}
   NodeType* getLeft() { return left; }
   NodeType* getRight() { return right; }
   void append(NodeType* n) { ... }
};

template<class NodeType, class SuperType>
class Tree : public SuperType {
   NodeType* root;
public:
   Tree() : root(NULL) {}
   void treeInsert(NodeType* n) { ... }
   void treeRemove(NodeType* n) { ... }
   NodeType* treeFind(NodeType& n) { ... }
};
```

Figure 6.9: Two templates for a simple tree data structure.

objects. Thus a requirement that Data be instantiated as ClientData does not pose a special problem.

With data structures found in libraries, like the STL, we can create a tree of lists or a list of trees, but we can't merge a tree with a list to create a data structure that is both a tree and a list. With our data structure subcomponents, we can.

Figure 6.9 shows the templates for a simple tree implementation. To use the tree, a client would declare classes as for the list in Fig. 6.8.

The Tree template declares treeInsert() and treeFind() operations. Because data and nodes share the same class in our implementation, the find operation takes an instance of the same class as the nodes. The data/node instance that is passed, however, is not expected to be a node in the tree. It is only used for testing greater than and equality on its data value in the search.

```
template<class NodeType, class SuperType>
class ListLiftsTree : public <class SuperType> {
public:
   void listAppend(NodeType* here, NodeType* n) {
      treeInsert(n); SuperType::listAppend(here,n); }
   void listRemove(NodeType* here, NodeType* n) {
      treeRemove(n); SuperType::listRemove(here,n); }
};
```

Figure 6.10: Lifter for merging tree semantics with list interface.

Our implementation of the tree data structure has no parameter or argument for the comparison operator. It uses the comparison operator of the data, directly. By composing the TreeNode part or the data node in a more derived position than the data, the tree code can call the data's comparison operator directly. To use a special comparison operator, we insert a translation component, as described later in Section 6.2.2., between the TreeNode and the Data.

By merging the tree and list data structures, we can get a structure that is both a list to arrange our data in an arbitrary order, and a tree that allows us to find list nodes in log n time. In order to make sure that nodes inserted into the list part also get inserted in the tree part, we use a lifter component, as described above, to add the tree insert to the calls of the list interface. The ListLiftsTree lifter is shown in Fig. 6.10. The declarations for the fully merged composition is shown in Fig. 6.11.

In the TreeList structure, data, list nodes, and tree nodes all share the same class. A pointer returned from either getNext() or find() operations is client data, a list node, and a tree node.

If we added a second linked list in our composition, we could have two arbitrary orderings of the data. Techniques for addressing the name ambiguities with repeated inheritance, as would be the case with two linked lists, is discussed as a separate idiom, later in Section 6.2.1.

```
// Class declaration for forward reference.
class ClientData;
// Data is the class that client would have used for data
// ClientData is the specialized class that client actually uses
class ClientData1 : public ListNode<ClientData,Data> {};
class ClientData  : public TreeNode<ClientData,ClientData1> {};
// List of ClientData is a base class
class List      : public LinkedList   <ClientData,empty> {};
class TreeList2 : public TreeList     <ClientData,List> {};
class TreeList  : public ListLiftsTree<ClientData,TreeList2> {};
// ClientOfList inherits from List (instead of containing one)
class ClientOfTreeList : public Client<TreeList> {};
```

Figure 6.11: Class declarations for a merged Tree and LinkedList.

6.2 Idioms for Syntactic Issues of Composition

Roles interact through their interfaces. Sometimes two interfaces don't support the intended connection, or other components interfere. The direction or scope of a connection relative to the order of composition may also pose a problem. The idioms in this section address problems of interface mismatch, visibility, and interference.

6.2.1 Disjoint Name Spaces and Repeated Inheritance

Name clashes can play havoc with two goals of software development, independent development and code reuse. Mechanisms for composing classes, such as dynamic binding and multiple inheritance, often don't allow the same name to mean different things in different parts of the composed class. When different fragments are developed in different places, or at different times, the same name may inadvertently be repeated in different fragments. This problem most likely occurs with simple names for common concerns, such as sum, count, total, or next. Even if the code is developed by the same developer at the same time, some components may be so generally useful that we want to use them twice in the same class. As examples of this last situation,

```
template<class NodeType, class SuperType>
class NodeRole : public SuperType {
   NodeType* next;
public:
   void setNext(NodeType* n) { next = n; }
   NodeType* getNext() { return next; }
};
```

Figure 6.12: Linked list node implementation.

```
class DataClass;
class Data1Class  : public Data<empty> {};
class Data2Class  : public NodeRole<DataClass,Data1Class> {};
class DataClass   : public NodeRole<DataClass,Data2Class> {};
```

Figure 6.13: Class declarations for a node in two separate linked lists.

we may want to include, or be part of, more than one linked list, or provide access controls for more than one server or queue.

Except where the composition really does change at runtime, our approach to composition uses single inheritance with static binding. The meaning of a name is specific to its position in the composition and can coexist with other uses of the same name in other parts of the composed class. For example, consider the simple linked list node role shown in Fig. 6.12. If we want our objects to be usable as nodes in two linked lists, we simply include the NodeRole twice in the composition, as shown Fig. 6.13.

If we have an object of class DataClass, called x, `x->getNext()` will return the second next pointer, while `((Data2Class)x)->getNext()` will return the value of the first next pointer. When using type coercion in the implementation of a role, the name of the type to which the pointer is coerced would be implemented as a parameter. The parameter would then be bound to the actual intermediate class name in the

```
template<SuperType>
class TranslateFunFoo : public SuperType {
public:
   void fun() { foo(); }
};
```

Figure 6.14: A template component to translate calls to fun() into calls to foo().

specification for template instantiation.

If roles that need to use the first NodeRole's methods are included between the two NodeRoles in the composition, their calls will bind to the methods of the first NodeRole. Placing roles that use methods of the second NodeRole after the second NodeRole in the composition will bind their setNext() and getNext() calls to the methods of the second NodeRole. Where such placement is not sufficient, such as a role that needs to access methods in both NodeRoles, type prefixing or name translation, as described below, can also be used to discriminate between the duplicated names.

6.2.2 Name and Signature Translation

When components are developed separately, they may end up using different names for the same method semantics or the same name for different method semantics. Where a method in one component calls a method defined in another but with a different name, how can the compiler know which method to bind to? We can often resolve the difference by providing an intermediate translation method in a separate component, and composing that component between the caller and the callee. The translation method could define a method name fun() and forward them as a call to foo(). A good compiler will remove the extra context switch of a translation method and simply establish the correct bindings. The implement of the fun() to foo() translation is shown in Fig. 6.14.

A translation can do more than change a method call's name. It can also rearrange the order of arguments, provide default values for missing arguments, and perform simple operations to compute new argument values. The changes to method calls that can be made in a translation component correspond to the changes described in Chow's thesis on incompatibility in library upgrades [17].

Translation can also address the problem of components using the same name with different meanings. Consider the case of a method in one component calling the fun() method in another component. Suppose a third component, appearing between the first two, also defines a method named fun(). We can route the fun() call around the interloper by placing a translation on either side, translating it first to foo(), and then, after the interloper, translating foo() back to fun().

6.2.3 Type Prefixing

Type prefixing is another way of routing the binding of a method call, like the fun() call in the previous example, around an interloping usage of the same name. When we compose components to form application classes, each component is added, in turn, by instantiating its template in a class definition. The resulting list of declarations defines a series of class names at intermediate stages of composition. We can use these intermediate class names as type prefixes to direct method calls to specific components in the composition. Figure 6.15 shows the implementation of a template to redirect the fun() method call.

Continuing the interloping fun() method definition, suppose the initial composition of the Stooges class has three components, Larry, Mo, and Curly. The components are composed in the above order in class definition statements for classes named Stooges1, Stooges2, and Stooges, respectively. If both Larry and Mo define a method fun() and Curly wants to call the Larry implementation of fun(), we would add the PrefixFun component between Mo and Curly. Class declarations with template compositions for the complete composition are shown Fig. 6.16.

```
template<class PrefixType, class SuperType>
class prefixFun : public SuperType {
public:
   void fun() { PrefixType::fun(); }
};
```

Figure 6.15: Adding a type prefix to direct a the fun() method call specifically to or around another component.

```
class Stooges1 : public Larry    <empty> {};
class Stooges2 : public Mo       <Stooges1> {};
class Stooges3 : public PrefixFun<Stooges1,Stooges2> {};
class Stooges  : public Curly    <Stooges3> {};
```

Figure 6.16: Composition of PrefixFun in the Stooges class to direct the fun() call around Mo.

Type prefixing need not be limited to special translation components. Any component that uses a method or variable name that is likely to encounter name clashes can include a type prefix in its own implementation. In our own experience, because of our use of proxies as described below, we often repeat the use of the handle name. Type prefixing might, therefore, be useful in the proxy components that try to access a particular handle.

6.2.4 Forward Reference

In Figs. 6.8 and 6.11, we used forward reference to declare the ClientData class before it was defined. We used the incomplete declaration of ClientData to declare pointers to an instance of the ClientData class within components that, when instantiated, became ancestors of the ClientData class.

While the rules of forward class reference allow the declared class name to be used for declaring pointers, they do not allow a forward declaration be used to declare an

```
class Stooges3;
class Stooges1 : public PrefixFun<Stooges3,empty> {};
class Stooges2 : public Curly    <Stooges1> {};
class Stooges3 : public Larry    <Stooges2> {};
class Stooges  : public Mo       <Stooges3> {};
```

Figure 6.17: Illegal composition with PrefixFun trying to direct the fun() call downward from Curly to Larry.

instance of the incomplete class. Including a forward declared class requires knowing how much space it needs. In the use of ClientData, above, creating an instance of ClientData instead of a pointer to one, evaluating its size would produce an infinite recursion.

At least in C++, we cannot use the incomplete class for type prefixing, either. This is unfortunate, since it would be convenient if we could use type prefixing to direct a call to a more derived class. Consider again the example of Larry, Mo, and Curly, but this time composed in the order, Curly, Larry, and Mo. Now the call from Curly to Larry has to move in the opposite direction of inheritance, to its immediate subclass. If we could use type prefixing with a forward declaration to make the fun() call simply use an offset into its subclass, we would use a composition as shown in Fig. 6.17.

6.2.5 Dynamically Bound Methods

Dynamic binding is usually used to bind method calls to methods defined in subclasses (down calls). For the Stooges example, in Fig. 6.17, we could replace the PrefixFun component with a VirtualFun component that declares fun() as a virtual method. But in the Curly, Larry, and Mo composition where both Larry and Mo define a method called fun, the dynamic binding results in the call from Curly being bound to the most derived definition of fun, which is the method defined in Mo. We can get to the method defined in Larry by putting a PrefixFun component in the most derived position, after Mo, with its type prefix bound to the Larry (stooges3) class. But that composition then makes every call intended for the fun() defined in Mo need to use a type prefix at the initial calling site—strategically placing additional copies of PrefixFun won't help. With a preprocessor that allows easy renaming, such accidental name clashes would not be a problem. We would just rename one of the uses. But, for our style of programming with C++ templates, we prefer to avoid this problem by implementing the down calls with implicit invocation, described later in Section 6.3.1.

6.3 Idioms for Control Flow

The semantics of an application depends on the runtime flow of control. Depending on the concern, the flow may or may not follow an application's static structure. The idioms in this section allow the control flow to be addressed independently of the application structure. Through the placement of components, setting of propagation parameters, and initialization of pointers, the developer can exert a wide degree of control over behavior within objects, and of the application as whole. The use of these idioms provides a compositional alternative to writing "wiring" code to connect the behaviors in an application.

Our approach offers an alternative to the tradeoff between distributed and centralized control over flow of execution. Distributed control allows the decision to be made using local information. Normally, distributed control is encoded in each component responsible for a behavior. Changing the execution order would require editing multiple components. Centralized control allows the developer to take an overall view in determining the order of execution. Centralized control is normally implemented as a separate dispatcher that calls each component in turn. Local information is not available to the dispatcher and each execution step potentially adds overhead by making a complete round trip through the dispatcher. Our approach guides the flow of control by ordering components in the composition, and setting parameters that determine whether components respond before or after propagating an event. There is no tradeoff between global view and local knowledge. Using the idioms described in this section, the order can be set when specifying the composition, when the global picture is available. The components that propagate control are local and can include local decision making.

6.3.1 Implicit Invocation

Implicit invocation is a general approach to managing control flow at runtime [47]. In implicit invocation, an announcer component offers to forward a particular method call to any listener component that expresses an interest. A listener component expresses its interest by registering a callback function with the announcer component. When the method call is made on the announcer component, the announcer calls each of the registered callback functions, in turn. We use implicit invocation to handle calls that otherwise violate compile time visibility rules, like the down call described above, and also for flexible inter-object event propagation. The C++ callback implementation of announcer and listener components for a resize event was shown in Fig. 5.7 in the previous chapter.

The listener can register itself with an announcer in the same object by using an anonymous constructor to invoke the register method. The announcer component, appearing earlier in the inheritance hierarchy, will be called first. When its constructor is called, the announcer initializes its callback list and is ready to take callback registrations. When the listener's constructor is called, the listener calls the announcer's register method in its inherited interface. The announcer then registers that listener. No additional code is needed to establish the connection for the down call.

A listener can also register itself with an announcer in another object if it performs the registration through a handle to that object. The use of handles is described with

```
template<class RealType, class SuperType>
class ResizeAnnouncerProxy : public SuperType {
   RealType* handle;
public:
   void init(RealType* hndl) { handle = hndl; }
   void registerResize(ResizeCallback callback, void* obj) {
      handle->registerResize(callback,obj); }
   void unregisterResize(ResizeCallback callback, void* obj) {
      handle->unregisterResize(callback,obj); }
};
```

Figure 6.18: Proxy component for the ResizeAnnouncer component defined in Fig. 5.7.

proxies, below. In this case, the registration cannot occur until after the handle has been initialized with the pointer to the other object. The correct sequence of initialization involves an init() function to set up the pointers, followed by an initialize() function for initializations that use the pointers. Control flow in initialization is also discussed in a separate section below.

6.3.2 Proxies and Handles

A proxy is a component that stands in for another component by providing the same interface as the other component and forwarding calls to that other component [24]. To a client, the proxy looks like the real component. In our compositional style, proxies are most often used as local stand-ins for components of remote objects.

Figure 6.18 shows an implementation of a proxy for the ResizeAnnouncer defined in Fig. 5.7. The handle variable is a pointer to the object that contains the real announcer. The ResizeAnnouncerProxy allows clients to make local calls within its object to registerResize() and unregisterResize() while the actual callback registration occurs in another object.

Proxies hide the details of the application's structure from the main implementation components. In the implementation of a role, all calls are local. In the composition, some of the calls are bound to methods in the same object and some of the calls are bound to proxy methods for forwarding to other objects. To the client, it makes no difference. The application could be a single object implemented by stacking all the roles in one long list, or widely distributed with one role per object. Externalizing the details of the structure allows role components to be used in different structures without modification.

Proxies transform structure into a matter of composition. The developer creates much of the application's structure by adding handles and proxies to the compositions of objects. Handles define the structure, while proxies define the interaction.

Proxies hide the method of access between objects. A proxy can call the real method through the name of a contained object, through a pointer to a separate object, or through a socket connection to an object in a separate process. Access mechanisms can be implemented in a component separate from the proxy, as shown in Fig. 6.19. By separating the handle from the proxy, several proxies that all forward to the same object can share the same handle. The template for the Handle component is extremely general and is used repeatedly for all handles to all objects.

Proxies can control access to the original object by revealing only part of the real component's interface. The ResizeAnnounceProxy component in Fig. 6.18, for example, did not include the resize() method defined by the real ResizeAnnounce component shown in Fig. 5.7. The resize() method triggers the announcement and is not meant to be called by the listener. If a component accesses another object only through a proxy, it can only call the methods forwarded by the proxy.

Proxies can forward calls through other proxies. Just as role components don't need to know whether they are calling proxies or real components, proxies don't need to know, either. As an example of where this might be useful, consider an image display application that draws graphical annotations on the images in the display. The image renderer accesses the display window through a window proxy. The annotation component also access the display window through a proxy. By

```
template<class HandleType, class SuperType>
class Handle : public SuperType {
   HandleType* handle;
protected:
   Handle() : handle(NULL) {}
   void setHandle(HandleType* h) { handle = h; }
   HandleType* getHandle() { return handle; }
};

template<class SuperType>
class ResizeAnnouncerProxy : public SuperType {
public:
   void registerResize(ResizeCallback callback, void* obj) {
      getHandle()->registerResize(callback,obj); }
   void unregisterResize(ResizeCallback callback, void* obj) {
      getHandle()->unregisterResize(callback,obj); }
};
```

Figure 6.19: ResizeAnnouncer proxy component with the handle management implemented in a separate component.

pointing the annotations' window proxy at the image renderer's proxy instead of directly at the window object, we can guarantee that the annotations will always be drawn in the same window with the image, even when the image moves to another window. We don't need code in the annotation components to keep track if the image has moved. This version of the proxy idiom might be described as the I'll-have-whatever-that-guy's-having idiom.

Yet another advantage of using proxies is that it allows calls between objects to be intercepted, as described above, in the context of the caller or in the context of the callee. To intercept the call on the calling side, we simply place the intercepting component between the calling component and the proxy for that method.

6.3.3 Pre- and Post- Event Propagation

When breaking large concerns into smaller subconcerns, we often end up with groups of behaviors that all must respond to the same event. For example, in a graphical user interface the response to a window being resized may be broken into separate pieces addressing window geometry, the sizes of buffers, coordinate transformations, and various pieces of window decoration. Some responses may be constrained to occur before or after other responses. For example a decoration cannot be regenerated until the buffer that holds it has been resized and the coordinate transform that it uses has been recomputed. Rather than write central control methods that find and call each of the pieces in turn, we can propagate the event through the composition as a method call and allow different components to respond as the event passes by.

Propagating a method requires each responding component to pass the same method call on to its inherited interface. If the resize event generates a resize(w,h) call, then each responding component must also include a SuperType::resize(w,h) call in its resize method. In most cases the resize method will call its SuperType either before or after performing its own response, although in some cases it may perform part of the response before and part of it afterwards. The problem with implementing

```
template<int resizeSEQ, class SuperType>
class ResizeBuffer : public SuperType {
public:
   void resize(int w, int h) {
      // propagate first for a post-resize response
      if( resizeSEQ < 0 ) SuperType::resize(w,h);
      // perform local resize response
      resizeBuffer(w,h);
      // propagate last for a pre-resize response
      if( resizeSEQ > 0 ) SuperType::resize(w,h);
   }
};
```

Figure 6.20: Template with parameterized conditional code to control before, after or no propagation of resize event.

the resize() function when a component is written is that it is hard to know whether it should respond before or after propagating the event, without knowing the rest of the composition.

Ordering the composition to accommodate the resize function is not reasonable, since there may be other propagated events among the same components, such as mouseMoved() or init(), with different orders of response. Instead we let the developer set the flavor of propagation for each method (pre or post) at composition time, using conditional code and a template constant parameter. A simplified resize component with a constant expression parameter and conditional propagation is shown in Fig. 6.20.

The constant conditional is also not ideal. The SuperType class must implement a resize() method, even if no propagation (constant = 0) is selected. We put empty methods with appropriate signatures for commonly propagated methods in the default base component to resolve the issue. A separate preprocessor with conditional code, such as Bassett's Frames [5], would avoid this problem. A good compiler should optimize away the branch on constant overhead, in any case. If the language's template mechanism did not support conditional code, we might alternatively have implemented duplicate components for each of the three flavors of propagation.

6.3.4 Initialization

In our approach to development, roles can use any of three mechanisms for handling initialization: a constructor with no arguments, an initialization method with no arguments, and an initialization method with arguments. Each mechanism is appropriate to specific situations.

Initialization that does not require any values to be passed in, and has only limited interaction with components of the same object, may define a constructor that takes no arguments (an anonymous constructor). Anonymous constructors are commonly used to NULL the initial values of pointers. We used NULLing constructors for the Handle component in Fig. 6.19, and the Node components in several of the earlier examples. In the discussion of implicit invocation, we also described using anonymous constructors to establish the connection between the listener component and an announcer in the same object.

Initialization that requires a value to be passed in is implemented with an init() method. If an object has more than one such method, then we may create separate initialization component to provide a single init() method for the group of them. The initialization component's init() method takes arguments for all of the other init() methods, and then calls them in an acceptable order with the right arguments.

Handles, as used in the proxy idiom, are the most common components needing initialization with arguments. We generalized handle initialization by creating a family of constructors for different numbers of handles. The code for the two handle example is shown in Fig. 6.21.

Initializations that need to be sequenced with other initializations within the same object can be handled in either of two ways. They can define an init() function, as above, and leave it to a central init() function to call it at the right time. Alternat-

```
template<class Pointer1Type, class Handle1Type,
         class Pointer2Type, class Handle2Type,
         class SuperType>
class TwoHandleConstruct : public SuperType {
public:
   void TwoHandleConstruct(Pointer1Type& ptr1, Pointer2Type& ptr2) {
      Handle1Type::setHandle(&ptr1);
      Handle2Type::setHandle(&ptr2);
   }
};
```

Figure 6.21: Template for an initializer component to initialize two handle components.

ively, they can define an initialize() function, and handle their own sequencing. An initialize() function takes no arguments, and calls initialize() on its inherited interface, as we described for the pre and post execution idiom above. With the initialize() method, a component performs its own local initialization behavior before calling the inherited initialize() or afterwards, or may perform parts of its initialization both before and after. For example, a component that implements a buffer to store the contents of a window may call initialize() on its base class and then call the base class to find out what size window it created. In another use of initialize(), a component can define both init() and initialize() methods. After calling the init() methods with arguments, the application calls initialize(), giving components a chance to perform parts of the initialization in a second pass.

In the Implicit Invocation idiom, we described using anonymous constructors to establish the connection between the listener and announcer in the same object. However, if the listener and announcer are not in the same object, and the listener uses an announcer proxy to forward the register call to the announcer object, the proxy's handle would not yet have been initialized when the listener's constructor is invoked. The listener should, instead, use an initialize() function, since we call the initialize() functions after passing values, like the handle pointer, to init() functions.

Because of the dependencies it creates on an object's composition, we do not use argument passing among constructors. We prefer, rather, to address initialization as an issue in composition. We can then manually apply various mechanisms and components, as described above, to tailor the initialization to the semantics of the situation.

6.4 Discussion

The idioms presented in this chapter address the separation of semantic concerns from structural and compositional concerns. They are an important part of our role approach to the separation of concerns, and the corresponding compositional approach to application construction. They provide necessary tools for the development process presented in the next chapter.

Chapter 7

THE PROCESS OF ROLE ORIENTED DEVELOPMENT

This chapter describes the steps in a development process for applying our approach in practice. The process was originally presented in an earlier paper [75]. However, due to space limitations, only a superficial description was given. The description here, by comparison, is fairly complete and provides considerably more detail.

The process is divided into eighteen steps, grouped in four distinct phases. Starting with a set of use case requirements, the first phase models the use cases as collaborations of roles and addresses reuse. The second phase addresses composition issues, such as overlap and dependency. The third phase addresses issues of structure by adding components to make appropriate connections. The fourth and final phase produces an implementation. Each of the eighteen steps addresses a single well defined issue in the overall process of transforming the set of requirements use cases into an executable program.

Figure 7.1 shows the abstract overview diagram for the container recycling machine. Each object is shown as an ordered composition of roles. The arrow annotations indicate relationships between roles within a collaboration. Solid arrows represent pointers managed by a collaboration. Dotted arrows are calls that use those pointers. Dashed arrows are calls between roles that do not use a known pointer. A dashed arrow represents an unresolved issue—the missing connections are addressed by step 12 in the third phase.

Our two main concerns in the structure of the development process are traceability and flexibility. The contribution of any step in the process must be traceable in any

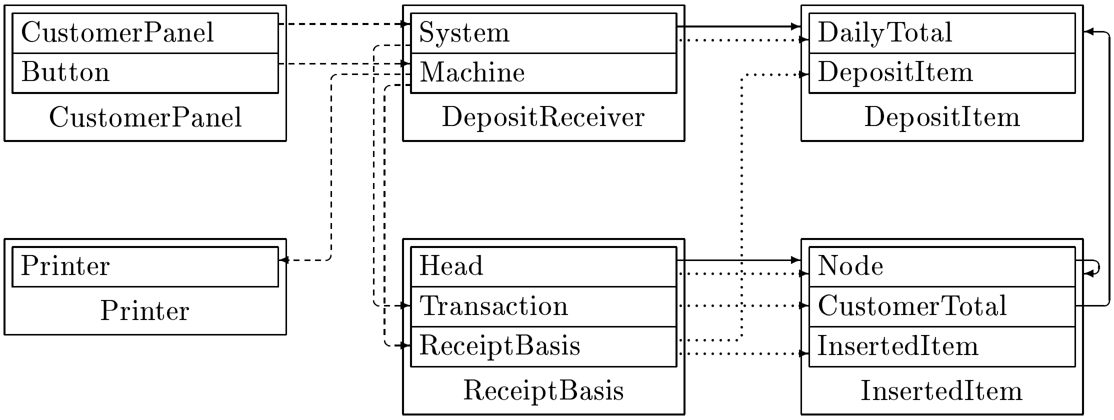

Figure 7.1: Overview diagram showing the role composition of each object with annotations showing relationships between roles defined for each collaboration.

subsequent step, and ultimately in the final artifact. Using this traceability, steps can be revisited to improve their contribution as the design evolves, and again later, as changes are applied. Even in the implementation, existing components can be further subdivided and/or reshaped as the need arises. The steps are intentionally small and well defined to facilitate this iterative and incremental process.

By comparison with other processes, the process here is fairly complete. Other approaches commonly leave several issues for one or two steps that are left intentionally vague. The synthesis step in the OORAM approach, for example, tells the developer to take the descriptions in a set of roles and produce a single implementation that satisfies all of them. Issues of interactions between roles, traceability, code reuse, and support for change in the implementation are left to the programmer to figure out. Each of out steps address a single issue with explicit methods for doing so.

The process, as presented here, is structured in terms of a progression of overlapping steps. The sequence corresponds roughly to the order in which each step is first performed. But the process is not entirely linear. Any step can be revisited at any time to improve upon its contribution to the design. Feedback is continuous (as

When a customer inserts a can, bottle, or carton into the appropriate slot, the system collects and crushes the container, and then increments both a customer total and a daily total for that container type.

(a) Adding Item

When the customer presses the receipt button, the following information for each container type is printed on a receipt: name, number deposited, unit deposit value, and total deposit value. Then the sum of the deposit values is printed, and the receipt is issued through the slot. Finally, the customer totals are cleared, and the machine is ready for a new customer.

(b) Print Receipt

Figure 7.2: Initial use case requirements for the container recycling machine

opposed to discrete), coming from all subsequent steps in the process and applied whenever needed. Thus it is not necessary to "get it right the first time" when any step is first performed.

7.1 Phase 1: Defining Collaborations and Roles

Step 1. Collect requirements use cases.

The design process begins with the use case descriptions from a requirements analysis. The use cases can either be complete descriptions of system behaviors, or extension use cases describing variations on, or extensions to, behavior described in other use cases. In the container recycling machine example, as described in earlier chapters, the initial requirements consist of two use cases, Adding Item and Print Receipt. The use cases, originally presented in Chapter 2, are shown again in Fig. 7.2.

Step 2. Identify likely objects based on entities in the problem domain.

To identify objects, we begin with the common approach of looking for nouns in the problem description. The nouns can either hold state or be the subjects that perform actions. The intent is to start with a structure based on the entities in a description of the problem domain that is both logical and understandable to the client.

In the description of the Adding Item use case, there is a customer, an empty container, a set of slots, a set of customer totals, a set of daily totals, and something called the system. The block diagram for this decomposition is shown in Fig.7.3(a). The lines in the diagram represent structural concerns such as a call from one component to another. A circle on the end of a line indicates that there may be more than one instance of the adjacent component. There are separate customer totals and daily totals for each container type. There is only one CustomerPanel, however, for the three front panel slots. The customer and the containers themselves are external to the system, and are thus not represented.

The description of the Print Receipt use case includes a customer, a button, a customer total, a receipt, several pieces of information for each container type including their customer totals, a printer slot, something called the machine, and a new customer. Passively voiced expressions, such as "is issued," often imply the system as the subject of the action. However, the passively voiced expression, "is printed," could imply a printer.

In choosing objects, small items can be grouped together to yield a simpler design. The primary criterion for selecting items to be grouped is that they can all be created or destroyed at the same time. In a more general sense, they should all be handled in a similar manner and address a similar concern. In our decomposition of the Print Receipt use case, the DepositItem groups the three items, name, value, and customer total. These items are all specific to the container type and are used to create one line on the receipt. A block diagram of the object decomposition we chose in the Print Receipt use case is shown in Fig. 7.3(b).

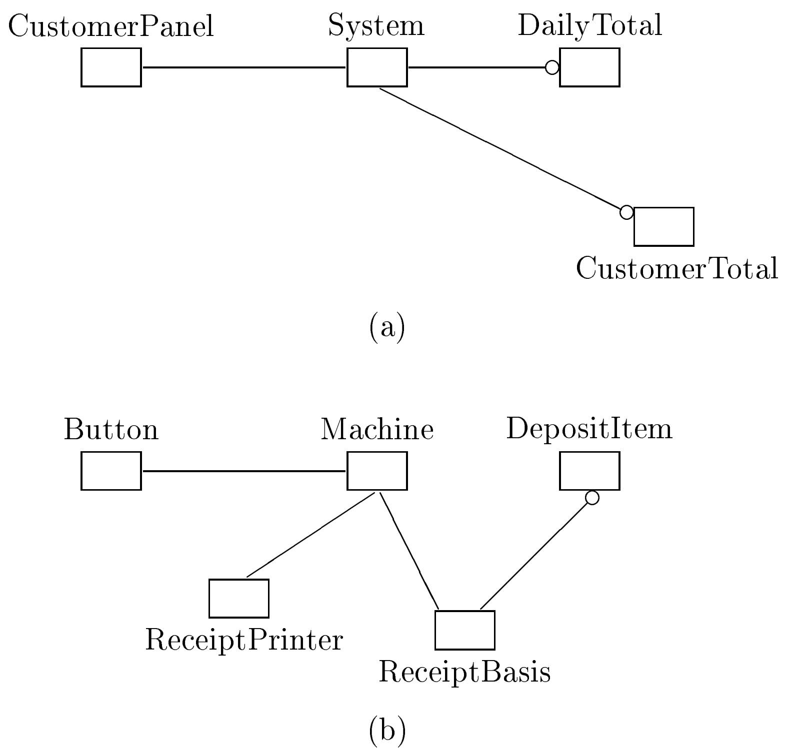

Figure 7.3: Block diagrams od initial object decomposition of Adding Item use case (a), and Print Receipt use case (b).

In this first step, no special attempt is made to define structures that uniquely address problems of the implementation language or its environment. The initial decomposition should not be cluttered with extra objects that have no meaning in the problem domain and may or may not be used depending on the implementation. "Is a" relationships between object types, as used in inheritance, are also not considered.

Step 3. Refine the choice of objects, taking into account commonalities across different use cases and likely sources of reuse.

In this step we want to choose objects with corresponding boundaries of granularity across use cases. Later, when we compose roles, we will be matching an object as defined in the collaboration of one use case with the same object as defined in another. Thus we must compare the objects in different use cases to make sure we have divided the problem domain in similar ways. Many entities are well defined in a given problem domain and are likely to appear in any use case decomposition. Such objects might include account, vendor, machine, order, transaction, receipt, date, employee, etc. But internal details of an implementation may be divided in different ways, especially if different designers work on different aspects. In the previous step, we deliberately chose different decompositions for the two use cases to highlight this type of structural conflict.

In the Adding Item use case, we defined two separate object types for holding information about the deposited containers. Both CustomerTotals and DailyTotals are specific to container types. But CustomerTotals hold values valid only for the current customer transaction, while DailyTotals hold values that are used in every customer transaction. An entity, called the System, manages both sets of objects. The decomposition of Print Receipt use case, on the other hand, has only one object type for container information, called DepositItem. The DepositItem, here, includes both the customer total, which is specific to the current customer transaction, and name and value, which are common across transactions. The Machine object correspond to Adding Item's System object. But the DepositItem objects are managed by a separate entity, called the ReceiptBasis.

Where two items are grouped in one use case but separated in another, our tendency is to favor the structure that is more decomposed. Thus we separate out the part of Print Receipt's DepositItem object type that has the customer total, calling it InsertedItem. For the Adding Item use case, we separate the System role into two roles, with a new role, called Transaction, that corresponds to the ReceiptBasis role in the Print Receipt use case. The refined Adding Item and Print Receipt use case decompositions are shown in Figs. 7.4(a) and (b), respectively. The use cases are refined and restated to reflect the object decompositions, as shown in Fig. 7.5.

In this step, reuse can also be a factor. Suppose our company already made a container crushing machine that did not offer receipts. We could match the new machine's object structure with parts of the earlier implementation to facilitate reuse. Reuse can come from other applications, application frameworks, large scale components, and smaller components such as data structure implementations. We will have more to say on reuse in the Step 5, where we add the linked list collaboration shown in Fig. 7.4(c).

Step 4. Describe the detailed responsibilities and behaviors of each role in each use case collaboration.

By this point in the process, the designer should already have some ideas of how the system will carry out the various functions for which it is responsible, and, to some extent, how responsibilities in different use cases might fit together. Now it is time to make responsibilities and behavioral relationships explicit. Following accepted practices, we use a CRC card-like approach to documenting responsibilities in each object's roles and scenario diagrams to document the interactions between object roles in a collaboration [10, 9].

In the CRC (class responsibility and collaboration) card approach, each class is represented by an index card. As each use case scenario is considered, the responsibilities of each participating class are written on the card for that class. In our case,

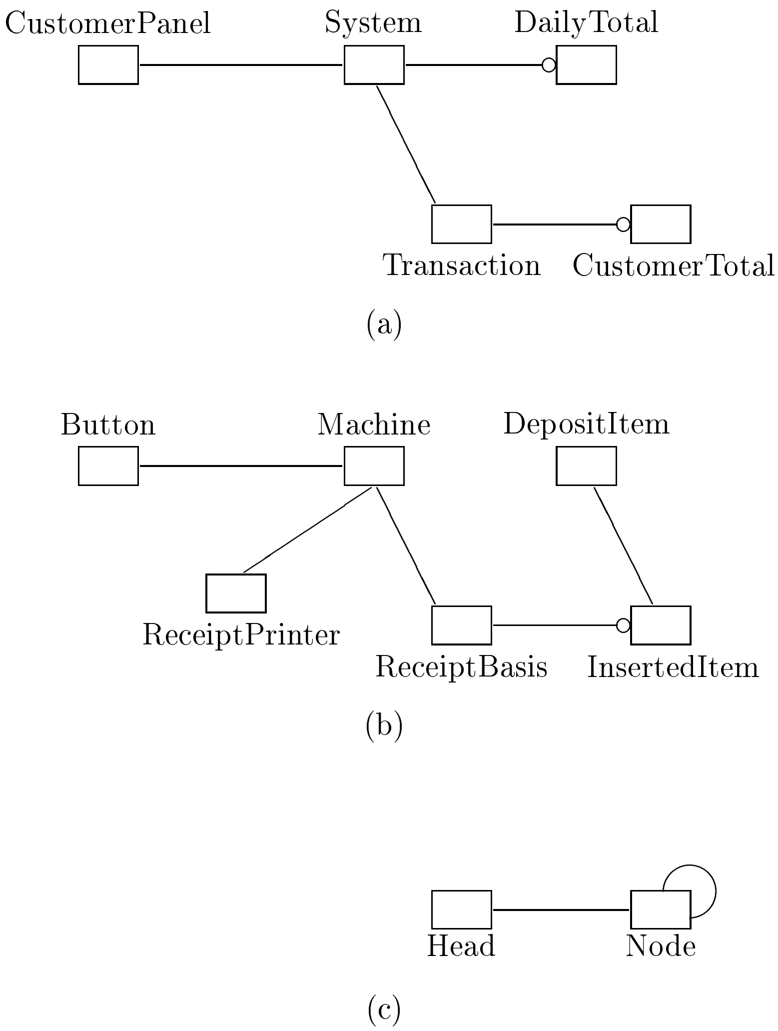

Figure 7.4: Block diagrams of objects in (a) Adding Item and (b) Print Receipt use cases after refinement to coordinate decompositions between use cases, and (c) objects in separate Linked List collaboration.

When a customer inserts an empty beverage container into one of the CustomerPanel slots, the CustomerPanel signals the System that a customer has inserted a container of a particular type. The System tells the Transaction, which increments a CustomerTotal of the appropriate type. The system then increments it's own DailyTotal for that container type.

(a) Adding Item

When the customer presses the receipt button, the Machine requests the ReceiptBasis to package up the information needed to print the receipt. The ReceiptBasis in turn requests the information for each line from its InsertedItem objects. Each InsertedItem object gets the container type name and unit deposit value from its associated DepositItem object, then adds the customer total, and computes the total deposit value. The RecieptBasis formats each line and adds the sum of the deposit values at the end. The Machine passes the formatted text to a ReceiptPrinter which prints the receipt and issues it through the slot. The Machine then instructs the ReceiptBasis to clear the customer totals, and the machine is ready for a new customer.

(b) Print Receipt

Figure 7.5: Restatement of the use cases of Fig. 7.2, refined for the object decompositions shown in Figs. 7.4(a) and (b).

we are not dealing with classes that span multiple use cases, but only with roles for a single use case.

When looking for responsibilities, a good way to start is to look for the verbs in the use case description. In the Adding Item use case, for example, the System *increments* a DailyTotal count. We must also look for implied or missing behaviors needed to fill gaps in the scenario's logic. How does the system know which DailyTotal to increment? Who manages the lists of DailyTotals and CustomerTotals? We might write on the CRC card for the System role in the Adding Item use case that it manages the list of DailyTotals and that when signaled by the CustomerPanel, it selects the appropriate DailyTotal, notifies the Transaction of the event, and tells the chosen DailyTotal to increment its count.

Scenario diagrams show the messages or method calls that pass between roles when a use case scenario is being carried out.[1] Communication between roles, earlier denoted by terms like "signal" or "command," become method calls that appear in the diagram as arrows with distinguishing names. Some responsibilities, like "manage," are vague and must be further elaborated before being expressed in terms of method calls.

The outputs of this step include both more refined descriptions of the use case collaborations and the scenario diagrams. The new Adding Item and Print Receipt use cases are stated in Fig. 7.8, while the corresponding scenario diagrams appear in Figs. 7.6 and 7.7.

Step 5. Decompose use case collaborations into smaller collaborations to manage complexity and facilitate reuse. Decompose compound roles into smaller roles, taking into account the possibility of change.

Use cases that can be logically defined as a single sequence of behavior at the

---

[1] The name, scenario diagram, comes for the terminology used in the OORAM method [55]. In OOSE, scenario diagrams are called interaction diagrams [32]. The Fusion method has a similar diagram, called a timeline diagram, to represent what it calls "use scenarios" [19]. In the Unified Modeling Language (UML) the corresponding diagram is called a sequence diagram.

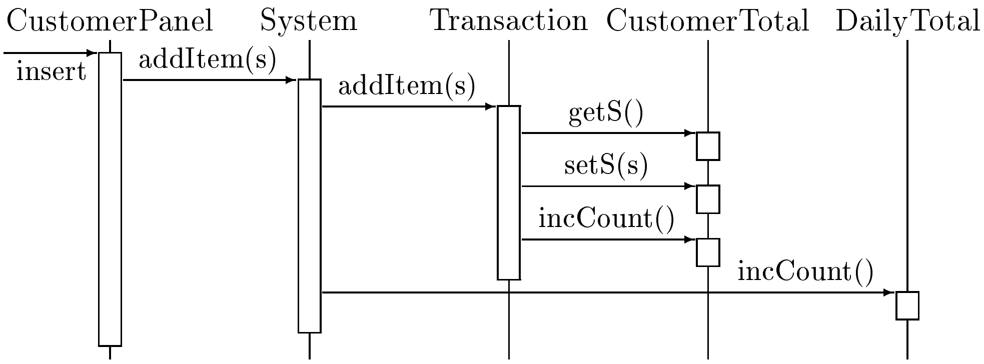

Figure 7.6: Interaction diagram for Adding Item collaboration.

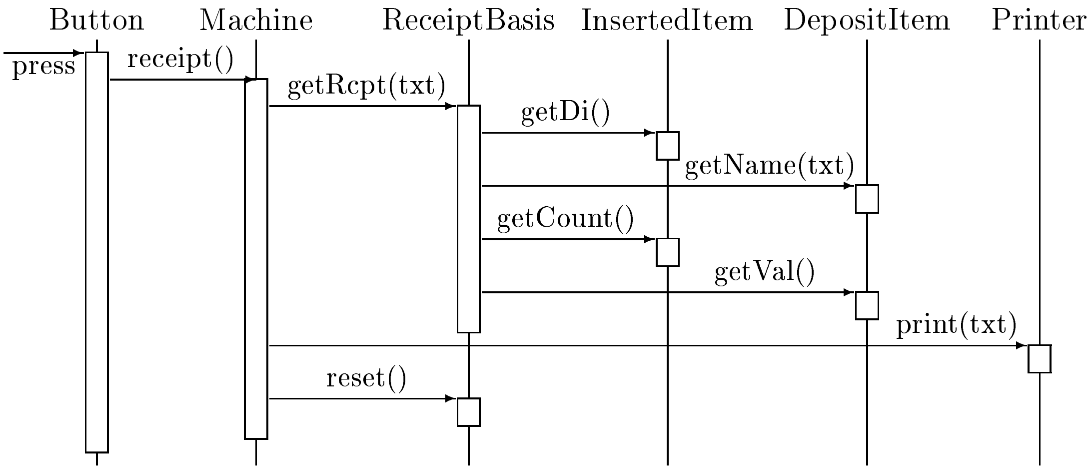

Figure 7.7: Interaction diagram for Print Receipt collaboration.

When a customer inserts an empty beverage container into a slot, the CustomerPanel sends an addItem message with the slot type to the System. The system, in turn, sends an addItem message, with the container type, to the Transaction object. The Transaction queries the CustomerTotals in its list to find the CustomerTotal for that container type. If it does not find one, it adds a new CustomerTotal object to its list. The Transaction then commands the appropriate CustomerTotal to increment its count. Finally, the System tells the DailyTotal, for the deposited container type, to increment its count.

(a) Adding Item

When the customer presses the receipt button, the Button object sends a receipt message to the Machine object. The Machine sends a request to the ReceiptBasis to prepare formatted text for the receipt. For each line, the ReceiptBasis, gets the total count and a pointer to the corresponding container type's DepositItem object. The ReceiptBasis basis then gets the container type name and unit deposit value from the DepositItem object. The ReceiptBasis creates the formatted text and returns it to the Machine. The Machine passes the text to a the Printer object which prints the receipt and issues it through the receipt slot. The Machine then instructs the ReceiptBasis to clear the InsertedItems, and the machine is ready for a new customer.

(b) Print Receipt

Figure 7.8: Restatement of the use cases of Fig. 7.2, refined with the responsibilities shown in Figs. 7.6 and 7.7.

requirements level are often composed of two or more distinctly definable processing behaviors at the level of implementation. In this step, we separate such compound use case collaborations into two or more smaller collaborations. There are three main reasons for decomposing use cases in this way. First, decomposition allows complicated behaviors to be defined in terms of smaller, more manageable pieces. Second, parts of the processing needed to support a requirements use case may be duplicated in other use cases, either within the same application or in different applications. Examples of potential duplication include basic domain frameworks, data base and GUI support, design patterns, and data structure implementations. Separating out common collaborations supports reuse and allows each collaboration to better focus either on what is common, or on what is unique. Finally where the design of a use case collaboration results from combining several decisions, decomposing the use case into parts specific to each decision facilitates later changing some of those decisions without affecting others [73].

In the description of the Adding Item use case from the previous step, the Transaction object manages a list of CustomerTotal objects. Manage, in this case, means performing traversal and insert operations on the list.[2] The corresponding ReceiptBasis object in the Print Receipt use case traverses the corresponding list of InsertedItem objects, and resets the list back to an empty state. We can separate both sets of behaviors out to a separate Linked List data structure collaboration. The Link List's block structure is shown in Fig. 7.4(c). The implementation is handled as in the data structure idiom described in Chapter 6.

In the description of the Adding Item use case, the System object manages a list of DailyTotal objects. The DailyTotal list appears to be of fixed length and could be implemented as an array. Alternatively, we might want to allow new container types to be added and thus implement the list with another linked list data structure. In

[2]Neither traversal nor insert operations appear in Fig. 7.6 because no messages are passed between objects. The getNext and insert calls, if any, occur within the Transaction object.

the latter case, the Linked List data structure collaboration could be reused for both lists.

Conceptually, collaborations should define continuous sequences of behavior among groups of objects. Responsibility for each part of a sequence is allocated to one of the roles in the collaboration. Together, they are responsible for the entire sequence. A role may forward some of the details of carrying out the work to other methods, but it is still responsible. In this step we must be careful not to violate a chain of responsibility when separating concerns in smaller collaborations. To see how a violation might occur, consider the reset() call from the Machine role to the ReceiptBasis role in the Print Receipt collaboration as shown in Fig. 7.7. The purpose of this call is to signal the ReceiptBasis to prepare for a new customer by clearing its list of InsertedItems. When we decomposed the Print Receipt collaboration, the details of how to clear the list were moved to the new Linked List collaboration. In the new Print Receipt collaboration, the Machine role could just call the clear() method in the Linked List collaboration's Head role to perform the reset function. But reset is the responsibility of the ReceiptBasis role in the Print Receipt collaboration. The forwarding decision should be made in the ReceiptBasis role.

In concrete terms, a role may call other methods in its inherited interface to handle the details of some operation, but it may not call another object unless that call is to a role in the same collaboration. Viewed at the level of collaborations, control may flow from one collaboration to another, and back again, within a single object, but not in a call between different objects in the application. In the case of the reset operation, the ReceiptBasis must still define a reset() method to be called by the Machine role, even though its implementation may simply be to call clear() on its inherited interface.

The rule on calls between collaborations is essential both for understandability and for flexibility. For purposes of documenting the design, all aspects of a concern are defined, at some level, within a single collaboration. When trying to understand a single role, the restriction maintains the invariant that all relationships between roles occur either within a collaboration or within an object. For purposes of flexibility, our model of composing collaborations assures that the join points where collaborations interact occur only within objects. Explicit source code references to details of an application's structure make designs hard to change. Keeping the join points within objects enables us to use only the implicit connection mechanisms of inheritance for composing collaborations.

After the decompositions in this step, we no longer use the term use case when referring to collaborations in the design. Use cases in our discussion are specifically the sequences of behavior defined in the requirements analysis. There is still a mapping between collaborations and use cases, but a use case may correspond to two or more collaborations in the design.

Step 6. Define coordination, initialization and similar behaviors needed to support the behaviors already defined. Refine existing collaborations or add new ones to include the needed additions.

The focus in previous steps has been on system behavior corresponding directly to the requirements. Many of these behaviors assume that the machine is in a certain state or that certain information is available. The machine may have to take actions not described in the original collaboration to get into that state or to make the information available where it is needed. In this step we identify those situations and make the needed additions to our design.

In the Print Receipt collaboration, each DepositItem role contributes the name and unit price for its type of beverage container. We might define a behavior to read this information from a file at startup time.

In the Print Receipt collaboration each InsertedItem role has a pointer to a corresponding DepositItem object. How does this pointer get set? When instances of the CustomerTotal role in the Adding Item collaboration, the DepositReceiver knows the identity of the corresponding DailyTotal. Assuming certain roles will be composed in the same objects, we can modify the Adding Item collaboration to accommodate the pointer expectation in the Print Receipt collaboration. In the original collaboration, as shown in Fig. 7.6, the slot id, s, was used as a discriminator in calls to the Transaction and CustomerTotal roles. In the change, we use a DailyTotal pointer, dt, as the discriminator.

The relationship between the application and its environment must also be considered in this step. System resources may need to be allocated and/or configured. Inputs to the application, in the form of event callbacks from library components or hardware interrupts, may require runtime registration. Some activities related to system startup, especially those requiring operator interaction, will appear in the use case addressing operator interactions.

Many use case behaviors are responses to events. But the use case itself may not describe how the system knows when an event has occurred, or if it does know, how it propagates that news to the appropriate roles. Event detection may pose a special problem if waiting for one event precludes seeing other events. In such systems, where each use case assumes it waits for its particular event, we may need to define an independent event detector to wait for all events. Chapter 6 discusses simple collaborations that can be used to propagate events among objects, and also within objects. The issue of coordinating event responses will visited again in Step 15.

Step 7. Define and name the objects of the final application and assign each role from each instance of a collaboration to an application object.

In this step we group roles by application object and choose a single name for each object or object class. The discussion of join points and the shared dt pointer in the previous two steps implied that we had some knowledge of which roles in different collaborations belong to the same object. The process of grouping roles began in Step 3 with the effort to coordinate decompositions. Here we make the groupings explicit.

Roles should combine to form reasonable data type abstractions. Using normal object oriented principles, all the roles grouped in an object should logically associate with the same problem domain entity. The use of common attributes is a strong indicator of logical association. The CustomerTotal and InsertedItem roles, for example, both refer to the same count of items inserted by the customer. Similar lifetime behaviors, such as the time of creation and destruction, should also be considered. Some objects, like the Printer in the Print receipt collaboration, may be unique to one collaboration, and thus only have single role.

As with every step, this decision can be visited again later. In fact, as shown in an earlier paper, our approach provides substantial flexibility for grouping role affiliations to try out different structures of implementation [73].

7.2 Phase 2: Composing Roles Within Objects

Step 8. Create a Roles/Responsibilities matrix (or matrices).

We still need more information to construct our application, especially when trying to compose several roles to form the class of a single object. 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 further subdivided. Second, where one collaboration extends another, we need to identify the calls between roles within the same object. These may be implied, but not shown, in scenario diagrams. 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 [16], to be a useful tool. Figure 7.9 shows a roles/responsibilities matrix for the three collaborations as discussed so far. In the matrix, rows represent collaborations, while columns represent objects or their classes. The internal cells of the matrix represent roles. Names within a cell, in *italics*, are role names from the collaboration named in the left column. Names

	DepositReceiver	ReceiptBasis	InsertedItem	DepositItem	Printer
Linked List		*Head* list prepend() clear() getHead() getNext(n)	*Node* next setNext(next) getNext()		
Adding Item	*System* item[N] addItem(s)	*Transaction* addItem(dt)	*CustomerTotal* count dt setDt(dt) getDt() incCount()	*DailyTotal* count incCount()	
Print Receipt	*Machine* receipt()	*ReceiptBasis* getRcpt(txt) reset()	*InsertedItem* count di getDi() getCount()	*DepositItem* name val getName(txt) getVal()	*Printer* print(txt)

Figure 7.9: Roles/responsibilities matrix for part of the recycling machine design.

followed by parentheses are method names. Names without parenthesis are attribute variable names.

Each role in the matrix is labeled with its name and contains both methods and attributes needed by the role in its collaborations. For each role we collect its methods from the scenario diagrams, as in Figs. 7.6 and 7.7, and its attributes by determining which attributes the methods use.

Where one collaboration depends on another, we must take into account any calls or accesses within an object that might not appear in a scenario diagram. The Transaction role from the Adding Item collaboration, for example, uses methods of the Head role in the Linked List collaboration to find and add CustomerTotals. These methods are included in the Head role description in the ReceiptBasis column.

Step 9. Identify and resolve duplication among roles assigned to the same object.

Different collaborations often refer to the same state variables, and may also share certain sub-operations. In Step 5 we addressed a shared sub-concern, for a Linked List, that could be expressed as its own collaboration. In this step we address overlap, or sharing, that occurs within the context of a single object.

Using the roles/responsibilities matrix, for each column we look at each variable or method to determine whether it might be duplicated in another role of the same object. Consider, for example, the InsertedItem column in Fig. 7.9. Both the CustomerTotal and InsertedItem roles define a count variable. In their respective collaborations, both count variables represent the number the beverage containers inserted during a customer transaction—they are the same. Similarly, consider the di and dt variables. These variables both point at a DepositItem object. We added the dt variable to the DailyTotal role in step 6 to initialize the pointer value used in Print Receipt's InsertedItem role. Between the CustomerTotal and Inserted Item roles, both the count and di/dt values are shared.

One approach to overlap would be to create a new role to handle the shared part. For the overlap in the InsertedItem object, we might create a role called IiData to manage the two shared variables, as shown in Fig. 7.10(a). The new role would be part of both collaborations. With the addition of an IiData role, both the Adding Item and Print Receipt collaborations would have two roles assigned to the InsertedItem object—IiData just happens to be the same role in both collaborations.

Another approach to the overlap is to eliminate the duplicated variables in one of the roles and make it dependent on the other role to provide those values. Our preference is to assign variables to roles where they are set, and remove them from roles that only read them. Semantically, the Print Receipt's use of the count and di variables requires them to have been set by the Adding Item collaboration. So it is logical to make them syntactically dependent as well. We eliminate the two variables from the InsertedItem role and make the implementations of the getDi() and

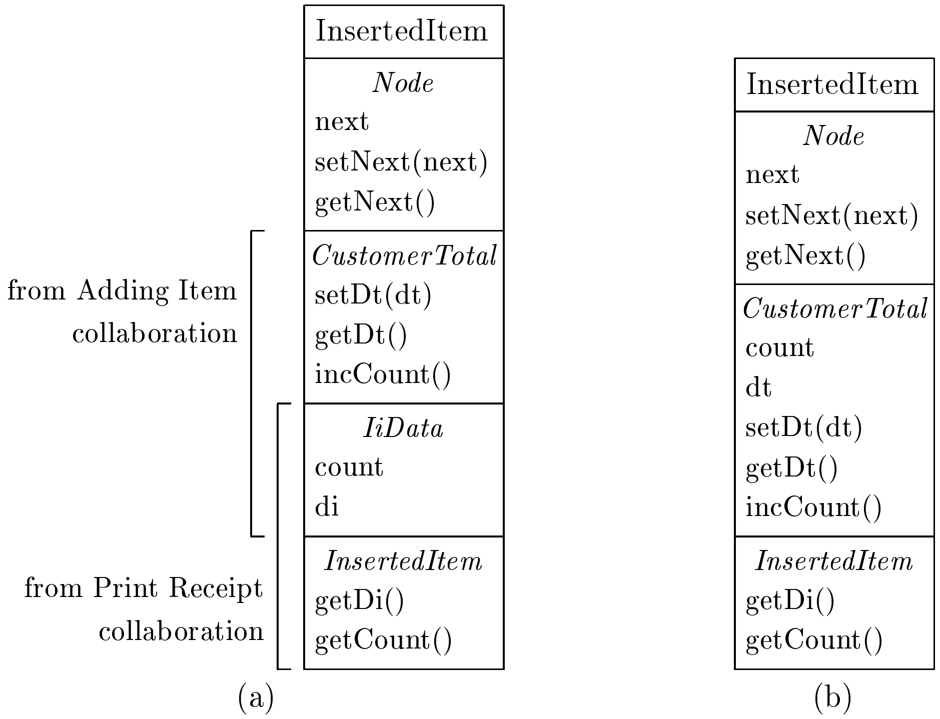

Figure 7.10: Two alternative strategies to address the overlap between the CustomerTotal and InsertedItem roles.

getCount() methods refer to variables (or other accessor functions) defined elsewhere in the same object. This second approach is shown in Fig. 7.10(b).

Why did we create a getDi() method when we could just call getDt()? Consider the two methods. Each is an accessor function for the dt/di variable—both methods perform the same operation. However, unlike the shared count and di variables in the previous discussion, the two methods are called from other objects in their collaborations. They are also called using different names. In addressing this sharing, we must be careful to maintain the consistency of the original collaborations—we cannot just remove one of the duplicate methods. If we put a common method in a shared role, such as the IiData role, we must also change one of the names used to call either the getDi() or getDt() method. We chose, instead, leave the two methods in place and address the sharing in the implementation of one or both of the methods.

Step 10. Identify dependencies among roles within objects and establish partial orderings for composing roles.

In the implementation of a role, calls to methods in other roles within the same object appear as calls on the inherited interface. When roles are composed, the methods being called will be part of the inherited interface only if the roles that implement those methods are included earlier in the composition. Interface dependencies impose constraints on the order of composition. In this step we determine which orderings can satisfy the dependency constraints by annotating the columns from the roles/-responsibilities matrix with information about the dependencies among roles. Later, we will also consider additional information relevant to the grouping of roles with acceptable orderings.

Figure 7.11 shows the annotated column for the ReceiptBasis object. The intra-object dependencies are represented by a modified version of the scenario diagram. The standard scenario diagram uses one line per object and labels the arrows with names of the methods being called. In our modified version, we turn the diagram 90 degrees, use one line per method or attribute being accessed and don't label the

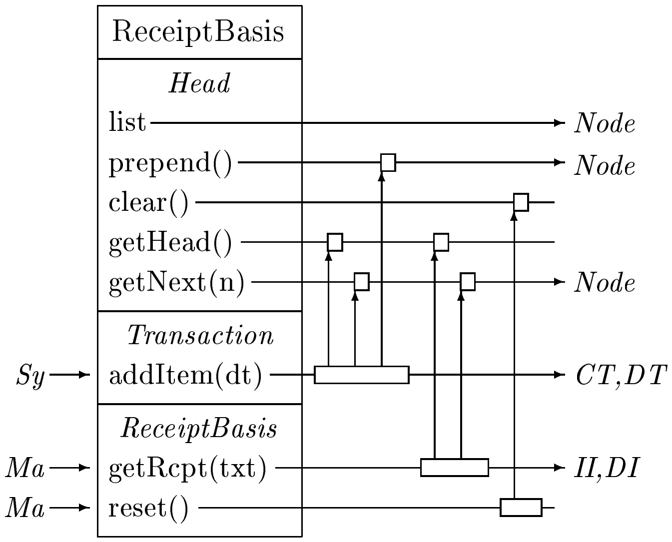

Figure 7.11: Annotated column for the ReceiptBasis object.

arrows.[3] The additional annotations on the left and right in the figure list the calls to and from roles in other objects of the application. This information will be used later to put the objects together and to define proxy components to handle some of the calls.

Assuming that the roles in the column appear in the top to bottom order in which they will be composed, all arrows in the figure should be pointed upward. As it happens in Fig. 7.11, this is already the case. Thus a composition with first Head, then Transaction, then ReceiptBasis would work for resolving the dependencies between roles. Since there are no dependencies between the Transaction and ReceiptBasis roles, a composition ordering of Head, then ReceiptBasis, then Transaction would also work.

Accesses from a method in one role to an attribute in another role are treated the

[3]For purposes of finding orderings, the standard form of the scenario diagram, with one line per role, would also work. We chose the alternative form to fit the details of the matrix, save space, and avoid writing vertically.

same as method calls between roles, with a box for the attribute and an arrow leading to it from the box of the method. An example of this situation appears with the count and di variables in Fig. 7.12(a). It is, however, generally good practice to provide protected access methods for attributes that may be accessed from other roles. With access methods, all inter-role dependencies appear as method calls, allowing separate name translation (e.g. getDt() versus getDi()) and access forwarding. In the common case of simple accesses, the extra overhead can be inlined away by the compiler.

In the annotated column diagram, virtual (dynamically bound) calls require special handling. The dependency arrow should be drawn from the specialized method to the method that it overrides. Even though the call actually travels in the opposite direction (from the base class to the subclass), the direction of the arrow indicates that the specialization is required to appear later in the hierarchy than the method it overrides.

Circular dependencies pose a special problem. When two roles depend on each other, either directly or transitively through other roles, no simple ordering can resolve all inter-role dependencies. For this situation, additional components can be used to forward calls in the opposite direction through either implicit invocation or dynamic binding. As described in Chapter 6, when a call appears earlier in the hierarchy than the method it calls, an announcer for that call is placed earlier in the hierarchy than the role with the call (above, in top to bottom order) with a corresponding listener placed later than the role that defines the method (below, in top to bottom order). The implicit invocation components are treated as additional roles in the composition.

As in the case of the ReceiptBasis object, it is often the case that several different orderings can achieve the desired bindings. This retained flexibility can be used to address issues considered in the remaining steps.

Step 11. Resolve name mismatches by coordinating names or adding translation components.

Up to this point we have either ignored naming issues or assumed that we could change names as needed to make references within and between collaborations match up. But it is not always possible or desirable to unify the names of every method and attribute. We may wish to reuse components from other existing applications for which the code has already been implemented. Where different collaborations represent different views, each collaborations may be easier to understand if expressed in terminology specific to the concern of its particular view. By using translation components, we can allow calls made with a particular name in one role to be bound to methods defined with a different name in another role. Argument translation is also possible.

In the previous discussion of the di/dt pointer used by both the CustomerTotal and InsertedItem roles, the CustomerTotal role defined a getDt() method while the InsertedItem role had a getDi() method. We could implement the getDi() function to access something called dt, as indicated in Fig. 7.12(a). Alternatively, without changing or mixing names in the implementations of either collaboration, we could implement getDi() as forwarding a getDi() call to its inherited interface. We then put a translation layer between the InsertedItem and DailyTotal roles that takes the getDi() call and forwards it to a getDt() method. This solution is shown in Fig. 7.12(b). Figure 7.12(b) also shows the access from getCount() in InsertedItem to count in DailyTotal changed to use an access method in DailyTotal. As mentioned above, a good compiler can optimize away the intermediate method calls in this solution.

When composing components with independent naming schemes, there is also a risk that two components use the same name to mean different things. What if the Transaction role in Fig. 7.11 defined a method called clear()? In a composition of first Head, then Transaction, then ReceiptBasis, the call from reset() to the clear() method in the Head role would be bound to the wrong method. The problem can be resolved with the alternative Head, then ReceiptBasis, then Transaction ordering. In more difficult cases, type-prefixed calls or intermediate translation layers, as described

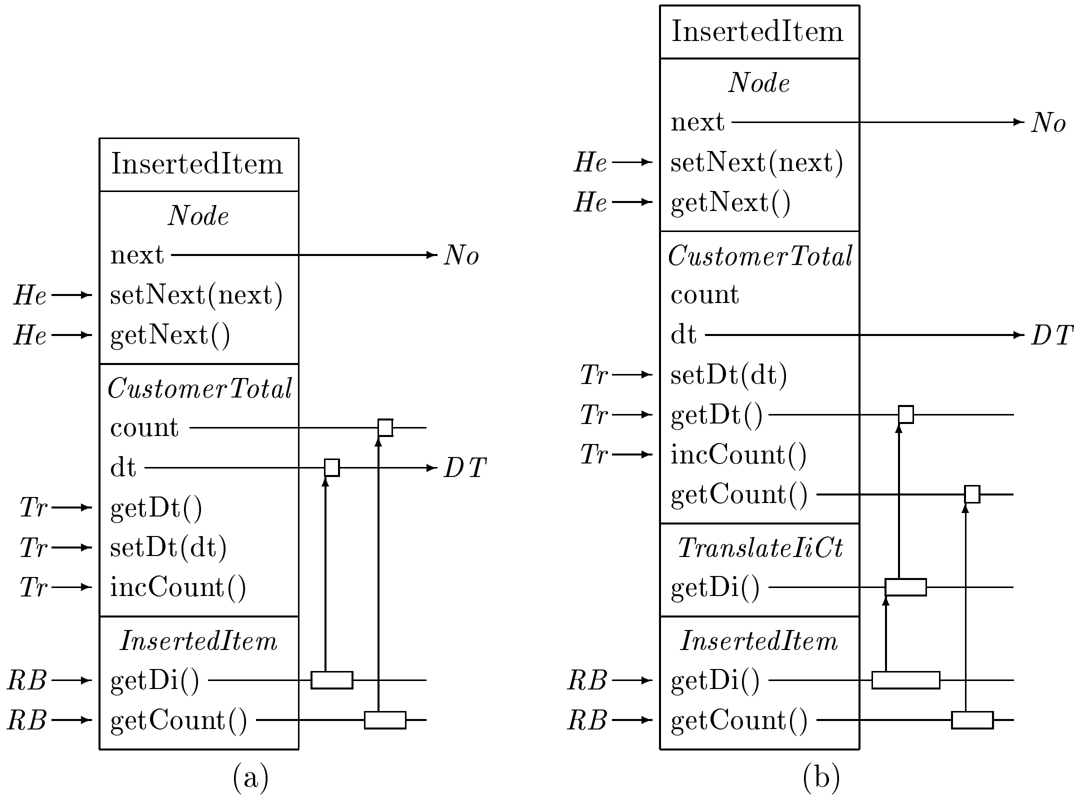

Figure 7.12: Two alternative approaches to addressing name differences between the CustomerTotal and InsertedItem role components.

in Chapter 6, can be used to jump over the interfering usage.

In the container recycling machine example there is both a CustomerTotal count and a DailyTotal count. In our design, we placed the two roles in separate objects. Thus, in this design, there is no issue in using the same name for both attributes. But in some other design, we might try to compose the CustomerTotal and DailyTotal roles in one object. This could create a name clash between the two components. Because we use static binding, the two uses of the same name remain distinct.[4] Methods within each role would see their own version. However, calls or accesses from outside the two roles could see either. In the final implementation, we would have to make sure the correct calls and instances are bound.

7.3 Phase 3: Connection Between Objects and Other Structural Issues.

Step 12. Add proxies and handles.

Now that we have addressed the connections within objects, we must address the connections between objects. To visualize the broader issues involved, we will start with a diagram that combines much of our earlier understanding in a single view. In Fig. 7.1, the role composition of each of the objects is presented as an ordered stack of boxes. Relationships between objects appear as arrows of various types connecting specific roles.

In our earlier discussion, certain roles manage associations with other roles. Where one role manages a pointer to, or contains the object of, another role, we draw a solid arrow from the managing role to the object of the role being referenced. The Machine role in the Adding Item collaboration has an array or list of DailyTotals (called item in Fig. 7.9). Each CustomerTotal role has a dt pointer that points to a corresponding DailyTotal. In the Linked List collaboration, the Head role points to a Node role, while each Node has a pointer to another Node.

---

[4]Allowing the same name to mean different things within the same object requires name scope control. Some languages do not support such control in conjunction with dynamic binding.

Where one role calls methods in a role of another object we draw a dotted or dashed arrow. Calls between objects occur strictly within collaborations and can be found by looking at the scenario diagram for each collaboration. Calls that follow one of the solid lines can be implemented by calling the method on that pointer or object and are drawn as a dotted arrow. The getVal() and getName() calls between the ReceiptBasis and DepositItem in Fig. 7.7 use the InsertedItem's pointer returned by the getDi() call and are thus also represented by a dotted arrow.

Calls that have no corresponding solid arrow are shown by a dashed arrow. These calls typically involve static relationships between objects that are not addressed in the functional description of any use case or collaboration. These are the calls that must be addressed in this step. In a normal application, we would simply add a suitable pointer or containment relationship to the client role. But such references make the implementation of the client dependent on structural details of the surrounding application. Instead, we introduce proxy components to manage the relationships independent of the roles in the collaboration.

Figure 7.13 shows the composition of the DepositReceiver object without proxies, (a), and with proxies added, (b). A proxy is a component that provides in its interface the calls of another role, while internally forwarding each call to that other role. To other role components, a proxy looks just like the role for which it is a proxy. Proxies are easy to implement and can, in fact, be generated from the interface description of a role. As discussed in Chapter 6, proxies offer other benefits such as access control and access through distributed object protocols. The underlined annotations on the right in Fig. 7.13 represent pointers to roles or other reference, while the non-underlined annotations represent calls to methods in other roles.

Proxies commonly use a pointer to the object of the other role. Often, several proxies forward their calls to the same object and can use the same pointer. We separate out the pointer parts of proxies and implement them in a separate Handle component. Since all handles have the same interface, modulo the pointer type, we

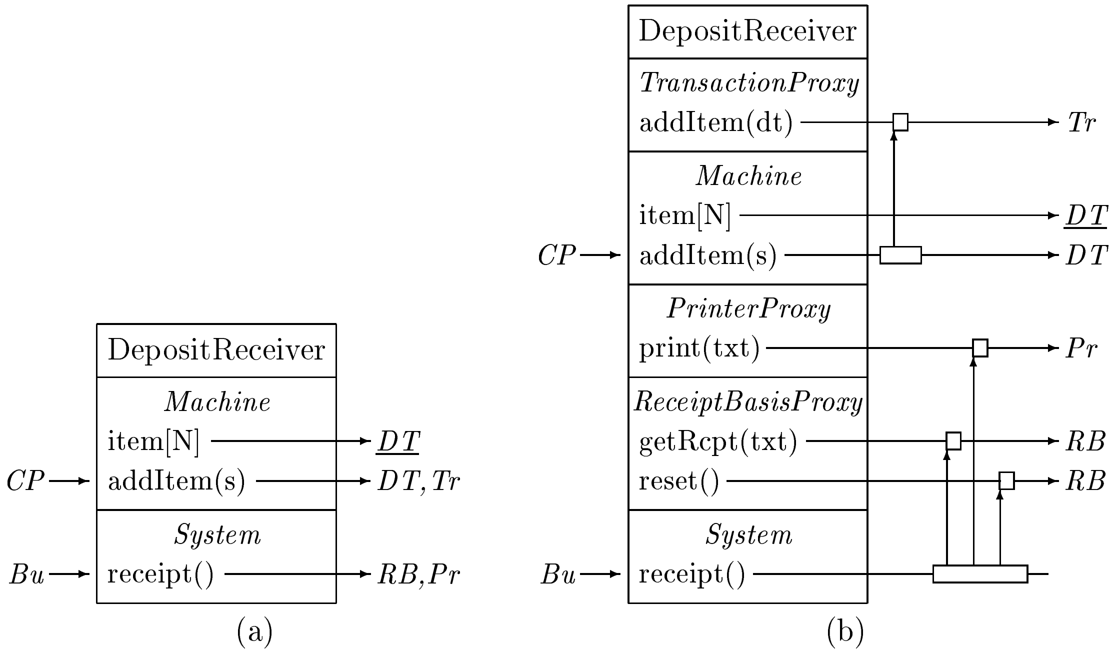

Figure 7.13: (a) The initial configuration of the DepositReceiver object with roles from the original collaborations. (b) The DepositReceiver object after the addition of proxy components.

need only implement one handle template for the entire application.

Figure 7.14 shows the completed structure of the DepositReceiver object including proxies and their corresponding handles, and a constructor component to initialize the pointers in the two handles when the object is created. The position of the Printer and ReceiptBasis proxies was reversed from Fig. 7.13(b) to allow the ReceiptBasis proxy to share the handle of the Transaction proxy.

The overview diagram in Fig. 7.15 shows the complete structures of all of the object types. In this view, all inter-object calls have managed pointers to provide the corresponding connections, and can thus be drawn as dotted arrows. Note also that the solid arrows have been moved down to indicate that the pointers will be declared with the complete types of the objects they point to.

Step 13. Identify common classes and define class hierarchies.

Through most of the discussion, little attention has been paid to the issue of class or class structure. In our source code implementation, the units of commonality are roles and not classes. But in the compiled image of the implementation, classes are the units of sharing, and not roles. To reduce the size of the compiled image, we would like, as much as possible, to identify common classes among the objects and to structure the application as a hierarchy of base classes with specializing subclasses.

An object's class is synonymous with its implementation. The implementation of objects in our application is defined by the composition of its roles. We can identify objects with the potential for sharing the same class by looking for objects that have the same roles. If two objects have the same roles composed in the same order, then they have the same class. If they have the same roles, but not in the same order, then they do not have the same class.

Before implementation, we look for objects that have the same role prefixes composed in the same order. These common prefixes can be shared as common classes. We also look for objects that have the same role prefixes but not in the same order. Going back to the ordering activity of Step 10, we try to reorder either or both of the

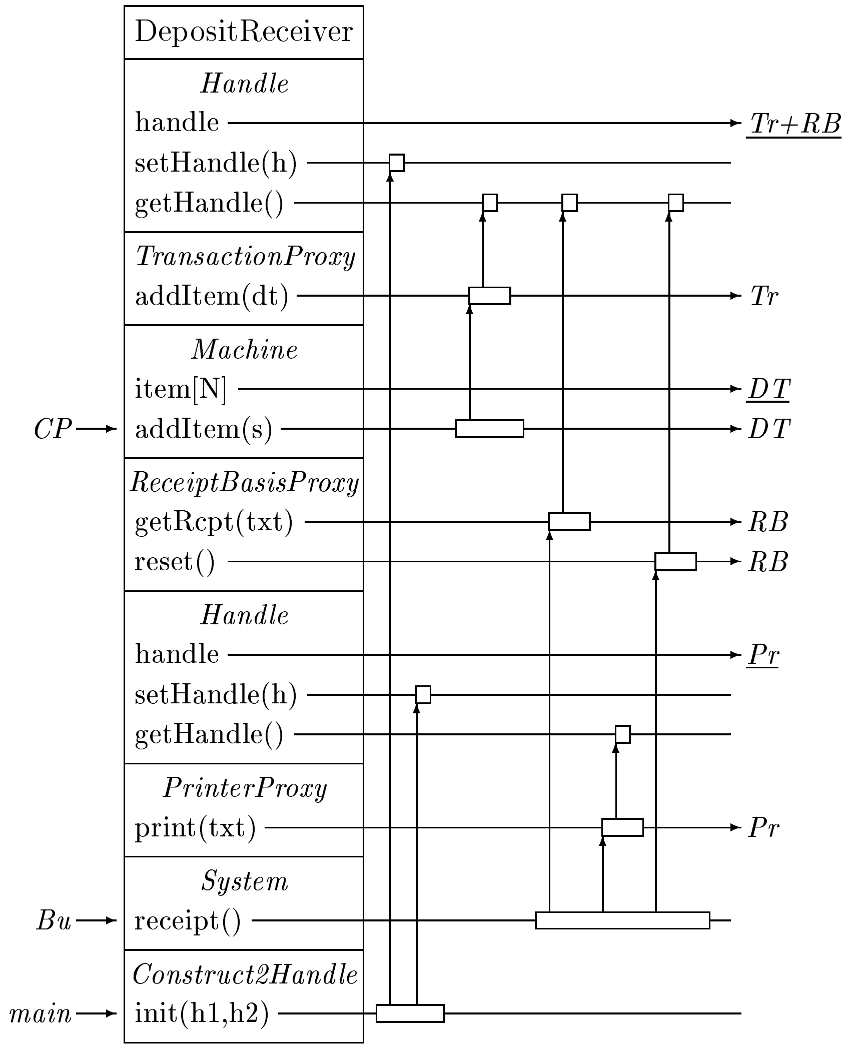

Figure 7.14: The completed form of the DepositReceiver object with three proxies for the inter-object calls between roles, two handles to connect the proxies to other objects, and a constructor to initialize the two handles.

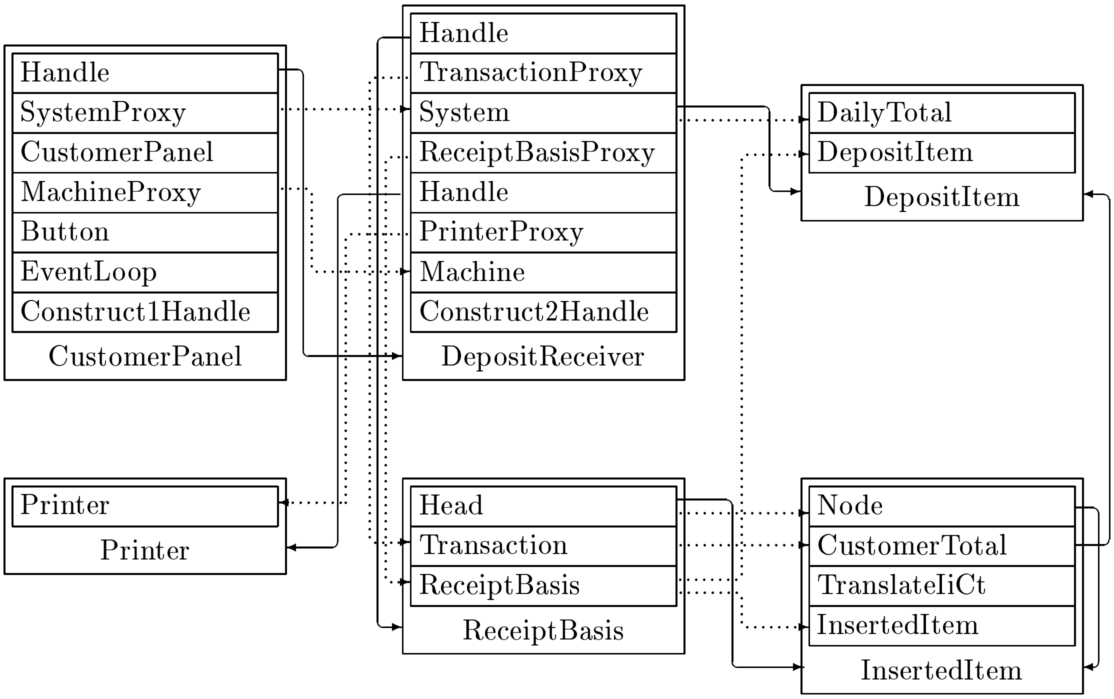

Figure 7.15: Overview diagram showing the complete role composition of each object and the relationships as implemented with the addition of proxy, handle, and translate components.

objects to share a common class. Often we may be able to match the compositions up to a point, say, the first 4 roles. We make that portion of each object a common base class, and, from there, create separate subclasses for the different objects.

Finally, we look for objects that have a similar composition, but where one object has additional roles not found in the other. Here, we look to see if we can add the missing roles to the smaller object without affecting its behavior–in other words, if the additional role can be present but not used.

In the design of the container recycling machine, there is no commonality among the six objects. Had we created different objects for each of the three container types, either for the InsertedItems or the DepositItems, the commonalities would have been readily apparent. We could then have structured the container specific objects with their unique aspects last. We may yet make distinct subclasses to address future requirements.

Step 14. Disambiguate names and note issues requiring special attention to types.

Because types are bound at compile time, every role can use the complete object type of every object, other than its own, with which it interacts. This was shown graphically in Fig. 7.15, where each solid arrow points at the bottommost role of the object to which it points.

Where the same method or attribute name is used in more than one role in the same object, we may not wish to access the most derived use of that name. In this case, may want a particular pointer to use the type of an intermediate class in the object composition. This must be noted so that the pointer's type can be bound to the correct type name when its template is instantiated.

Type names may also be useful within objects. Although we tried to address name ambiguities within objects by adjusting the order of composition in Step 10, we may still have cases where we need to use type prefixing to disambiguate among names. In the Construct2Handle component shown in Fig. 7.14, two different setHandle() methods are called. Ordering is not of use here. Cases of this type must be noted so that prefixes are included when the role template is written and correctly bound whenever the template is instantiated.

Step 15. Add object and role initialization, and adjust other response sequences.

When performing a number of initializations, it is often the case that certain things must be initialized before others. In the application discussed in Chapter 9, for example, a file could not be loaded until the buffer where it was stored had been initialized. Other events, besides initialization, may also involve interdependent sequences of response. In systems with graphical user interfaces where mouse move, key press, and window resize events are common, responses in different collaborations may have to be coordinated.

The sequencing of behavior with pre- and post-response parameters was discussed in Chapter 6. The use of those parameters, and the appropriate bindings for each role instance, should be addressed in this step. Our three-phase protocol for initialization, also described in Chapter 6, should be tailored in this step, as well.

For simple event responses, the flow of control can be drawn by hand on an overview diagram, or, within objects, on the annotated object columns. More complicated sequences within and among objects may use formal representations like finite state machines or petri nets, as shown by Aliee and Warboys [2].

7.4 Phase 4. Implementation

Step 16. Implement roles.

The design is now complete. We can proceed to implement any roles that have not already been implemented. Role implementation, using C++ templates, was described in Chapter 5. Each role is implemented as a class template with a SuperType parameter for its base class. Each reference to the type of any other object is also replaced by a parameter. Calls to other roles are implemented simply as calls on the SuperType interface, where they will be bound either to methods defined in other roles in the same object, or, through proxies, to roles defined in other objects.

Step 17. Compose roles, objects and classes.

Once the roles have been implemented, or even before, we elaborate the lists of template bindings and instantiations to create the classes of the application. Composition specification was also discussed in Chapter 5. The first role forms a base class by binding its SuperType parameter to a default empty class and is instantiated in a class with the suffix number 1. Each role appearing next in the composition is bound to the prior class name by binding its template's SuperType parameter to that name, and is then instantiated in a class with a suffix number that is one greater. Common classes, forming base classes in the class hierarchy as discussed in Step 13, may be given more distinguishing names and their specializations may continue with different names and a suffix numbering again starting with one.

Many role templates will have more type parameters than just the SuperType. In most cases these will refer to the type of another object in the application. However, in some cases, as noted in Step 14, these must be bound to an intermediate type names of its own, or some other object. Forward reference of type names can be used, but within the same constraints applying to normal uses of forward reference (e.g. non-recursive space computation).

Step 18. Write initialization calls and main.

If all the specification of use cases and collaborations are complete, and initialization is correctly handled, the only task remaining should be to implement a main() function for the application.[5] The main() function defines a series of object declarations to instantiate the top level objects, calls init() with appropriate arguments, and initialize() for objects needing initialization, and then calls a starting method in one of the objects. The main subroutine for the Container Recycling Machine example is shown in Fig.7.16.

---

[5]In the case of distributed applications, there may be multiple main() functions for each autonomous part.

```
int main() {
   // instantiate top level objects
   Printer         printer;
   ReceiptBasis    receipt_basis;
   DepositReceiver deposit_receiver;
   CustomerPanel   customer_panel;
   // call initialization methods
   deposit_receiver.init(printer, receipt_basis);
   customer_panel.init(deposit_receiver);
   // call method with outermost event loop
   customer_panel.run();
}
```

Figure 7.16: The main subroutine for the Container Recycling Machine application.

7.5 Analysis

Compared to other development processes, the process presented here may seem to have many steps. In part this may be attributed to the steps being narrowly focused on one issue per step. But, in comparison with other similar processes, such as OORAM or OOSE, our process also has steps for many concerns that aren't addressed in the other methods. The description of the development process in the book on OOSE, for example, covers steps one, two, four, and seven [32]. In the book on the OORAM approach, the development process covers steps one through five and step seven. The remaining issues are left unaddressed, to be resolved in implementation. Other methods which jump from requirements to classes typically cover only steps one, two and thirteen.

The process provides traceability in all artifacts. The requirements map to collaboration to their roles, to collaborations and roles, with auxiliary composition components, and finally to components in the implementation. All five of our diagramming models related directly to both the implementation and the requirements.

The final concrete overview diagram, shown in Fig. 7.15, contains many extra components compared with the original collaborations for the two use cases. But the diagram can be viewed as an extension of the abstract overview diagram shown in Fig. 7.1. With tool support, the extra detail could be elided and made visible only when needed, or perhaps grouped by use case with color coding.

The focus throughout most of the process is on objects, and more specifically, on roles within objects. The issue of class and class hierarchy does not appear until late in the process. The roles themselves are implemented in a way that does not commit them to a particular class structure. Thus the design of a class structure is almost an orthogonal activity.

The design and implementation in our approach can still be viewed in terms of classes and class structure. Our overview diagrams, minus some detail, and with inheritance relationships explicitly shown, correspond to class diagrams in other methodologies.

Reuse was introduced in the third step of our process, long before any decisions about implementation or class structure had been made. This early concern for reuse allows the design to be adapted to available artifacts more easily than other methods that try to match class definitions late in the design with code available in libraries. Also our decomposition around smaller concerns, and the direct implementation in components, makes it easier to reuse the resulting collaborations in other applications. Generalization is an integral part of our process rather than an afterthought to support reuse.

Chapter 8

THE PROCESS OF ROLE ORIENTED CHANGE

To change an application we go through the same steps that we used in development. For many of the steps we only consider the parts that are new. However, in some of the steps we may reconsider decisions made during the initial development. In this chapter we describe the process of change. To illustrate the discussion we apply the Validate Item use case, as described in Chapter 2. Later, we will also apply the Item Stuck use case.

Our compositional approach stresses adding new behavior by adding new components. We want to maximize the reuse of existing components which have already been verified and validated in the existing application. When it becomes necessary to upgrade or subdivide existing components, wherever possible, changes to original components should be tested in the context of the original application before using them together with new collaborations in the new application.

8.1 Adding the Validate Item Use Case

Step 1. Collect requirements use cases.

The Validate Item use case is an extension use case that provides a deviation from the behavior described in the earlier Adding Item use case to address the problem of verifying that the inserted container was valid for returning a deposit. We restate the use case from Chapter 2 in Fig. 8.1.

Step 2. Identify likely objects based on entities in the problem domain.

The block structure of our initial decomposition is shown in Fig. 8.2. The In-

When a container is inserted, the system measures its dimensions and reads its bar code. The measurements and bar code are used to determine if the container should be accepted for a deposit refund. If it is not accepted, no totals are incremented, and a NOT VALID sign is highlighted. The user must then remove the container before inserting another. If the container is valid, the system collects the container and continues as per the Adding Item use case.

Figure 8.1: The Validate Item use case.

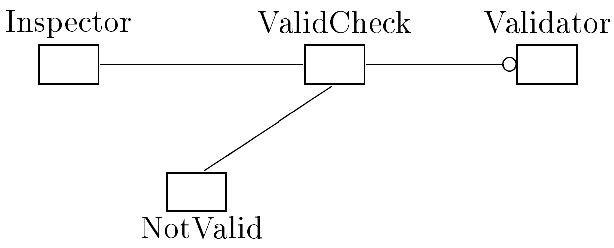

Figure 8.2: Initial role decomposition of the Validate Item use case.

spector measures the containers and reads their bar codes, the Validator checks the measurements against those of valid containers, and the NotValid signals the customer that a container is not valid.

Step 3. Refine the choice of objects, taking into account commonalities across different use cases and likely sources of reuse.

Since the Validate Item use case is an extension to the Adding Item use case, we need to coordinate its role decomposition with the existing structure used by the Adding Item collaboration. The Adding Item collaboration had a System role to coordinate activities. Input comes from the CustomerPanel role, while the DailyTotal role keeps information specific each container type. We can structure the Validate Item roles in the same way. The CustomerPanel object, where the slots are located in Adding Item, can take on the responsibility for measurements made by the slots. Since

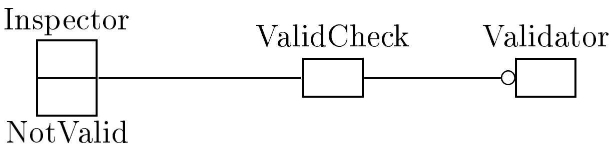

Figure 8.3: Roles in the Validate Item use case arranged to fit objects of existing application.

the CustomerPanel is the only entity with a customer interaction, it can also take on the new customer interaction by providing the NOT VALID warning. Because measuring containers and displaying a warning are logically different activities, we could have added a separate object for the NotValid role. Since Adding Item's System role already manages the array of DepositItems, we let the ValidCheck role share that part of its coordination responsibility with the System role. By placing it in the DepositReceiver object, the ValidCheck role can also pass control on to the addItem() method within the same object. The refined block diagram, arranged to fit the CustomerPanel, DepositReceiver, and DepositItem objects, is shown in Fig. 8.3.

Step 4. Describe the detailed responsibilities and behaviors of each object role in each use case collaboration.

In this step we create a description of the use case collaboration using specific role and method names, and draw the corresponding interaction diagram. For the Validate Item use case, the description of its collaboration is shown in Fig. 8.4.

The interaction diagram for both sequence alternatives is shown in Fig. 8.5. We use arrows that loop back on the same object to show the beginning and end of the extension use case's deviation from its base use case. In this case, the addItem(s) call from the CustomerPanel to the System in the Adding Item use case is intercepted by the Inspector role and resumed in the ValidCheck role.

Step 6. Define initialization and similar behaviors needed to support the behaviors already defined. Refine existing collaborations or add new ones to include the needed

When a customer inserts an empty beverage container into a slot, the Inspector measures the container and reads its bar code. The Inspector calls the ValidCheck's addItem method with the measurements, bar code, and slot type as arguments. The ValidCheck calls the corresponding Validator object's isValid method and passes the dimensions and bar code to check if they are valid. If the Validator object responds that the container is valid, the ValidCheck signals the Inspector's feed method to collect the container and then passes control to the System addItem method from the Adding Item base case. If the Validator object responds that the dimensions are not valid, the ValidCheck calls the NotValid's notValid method and takes no further action. The NotValid then lights the NOT VALID sign and waits for the customer to remove the container by polling the Inspector's isClear method. When the customer removes the container, the NotValid object turns off the NOT VALID sign, and processing resumes.

Figure 8.4: The refined Validate Item use case.

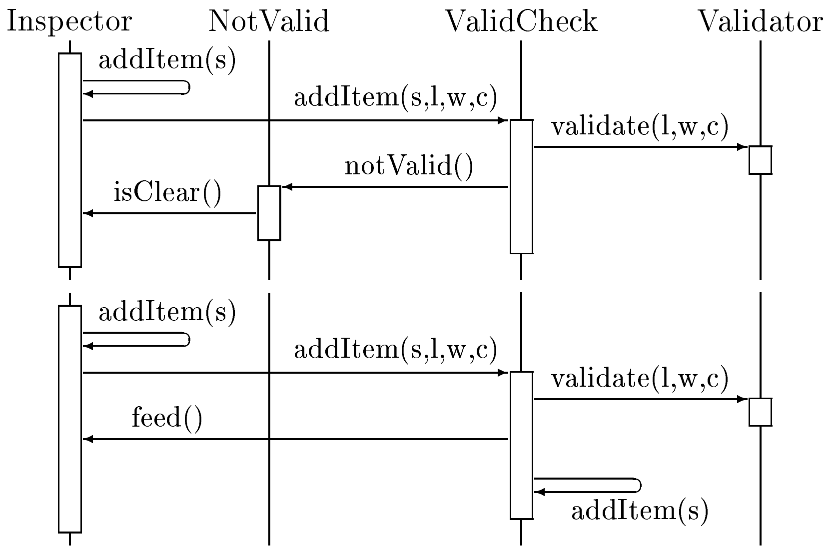

Figure 8.5: Interaction diagram for two alternative sequences of the Validate Item collaboration. Looped arrows indicate intra-object calls to or from another collaboration.

additions.

In the earlier discussion of this step, the Print Receipt's DepositItem roles needed to be initialized with the names and values of each container type. In the Validate Item collaboration, a similar issue exists for the Validator role where we need to initialize the criteria for validating items, e.g. the dimensions of valid items of that type. Again to simplify discussion, we ignore the initialization components needed for this task.

Step 8. Create a Roles/Responsibilities matrix (or matrices).

As in development, work on defining compositions for a change starts with a roles/responsibilities matrix. Here we reuse the roles/responsibilities matrix created in development and simply add, or replace, rows for the new collaborations, and add or remove columns to new or omitted objects.

Figure 8.6 shows the matrix for the container recycling machine as implemented in the previous chapter, with an added row for the Validate Item collaboration. The Printer column has been omitted, while the CustomerPanel column, not shown in Fig. 7.9, is included. The EventLoop role was added in Step 6 of development to address the system's interaction with it's environment. Some other roles differ slightly from the matrix presented in Fig. 7.9 of the previous chapter due to refinements applied in later steps of the development process.

Step 9. Identify and resolve duplication among roles assigned to the same object.

New roles may introduce new duplications that must be addressed as in the development process. The Validate Item collaboration begins with the detection of an inserted container in the CustomerPanel object and ends either with the container being removed in the notValid() method or with a continuation of the addItem() process in the DepositReceiver object. None of the roles, in this case, overlap with responsibilities in roles of other collaborations.

Step 10. Identify dependencies among roles within classes and establish partial orderings for composing roles.

	Customer-Panel	DepositReceiver	ReceiptBasis	InsertedItem	DepositItem
Event Loop	*EventLoop* run()				
Linked List			*Head* list prepend() clear() getHead() getNext(n)	*Node* next setNext(next) getNext()	
Adding Item	*Cust'Panel* insert()	*System* item[N] addItem(s)	*Transaction* addItem(dt)	*CustomerTotal* count,dt setDt(dt) getDt() incCount() getCount()	*DailyTotal* count incCount()
Print Receipt	*Button* press()	*Machine* receipt()	*ReceiptBasis* getRcpt(txt) reset()	*InsertedItem* getDi() getCount()	*DepositItem* name,val getName(txt) getVal()
Validate Item	*Inspector* addItem(s) feed() isClear() *NotValid* notValid()	*ValidCheck* addItem(s,w,h,c)			*Validator* l,w validate(l,w,c)

Figure 8.6: Roles/responsibilities matrix for the recycling machine with the Validate Item collaboration added.

Roles to apply change are very likely to use and interact with roles from the existing application. Most of the work in applying a change involves the problem of incorporating these new relationships into the existing application. Our strategy for inserting a new behavior is to intercept a call between roles in the original application. After the new behavior has been performed, the original sequence can be resumed by calling the originally intercepted method, partially overridden by calling a method later in the sequence, or aborted by simply returning from the intercepted call.

To add the Validate Item use case to the container recycling machine, we want to intercept the addItem() call from the Adding Item collaboration's CustomerPanel role to its System role. We could intercept it by adding a new addItem() method to the DepositReceiver object in a more derived position than the System role. However, taking advantage of the fact that calls to other objects go first to a proxy component, we will instead intercept the call within the CustomerPanel object. This will allow us to read the measurements from the container slot before passing control to the DepositReceiver object. The composition of the original CustomerPanel object, before applying any changes, is shown in Fig. 8.7. The method intercept is performed by defining an addItem(s) method in the Inspector role and placing it between the CustomerPanel role and the SystemProxy component. The completed composition of the CustomerPanel object, with the two role components for the Validate Item collaboration and the proxy defined in Step 12, below, is shown in Fig. 8.8.

Step 11. Resolve name mismatches by coordinating names or adding translation components. Examine name clashes and decide what action, if any, is needed.

As in the development process, there may be name clashes or mismatches between the new roles and the existing roles with which they are composed. In the case of the Validate Item collaboration, the implementation is written specifically for the existing container recycling machine implementation, so we can choose names that fit without translation. We defined two addItem() methods, using the same name as the addItem() method in the System role. In the case of the Inspector role, as

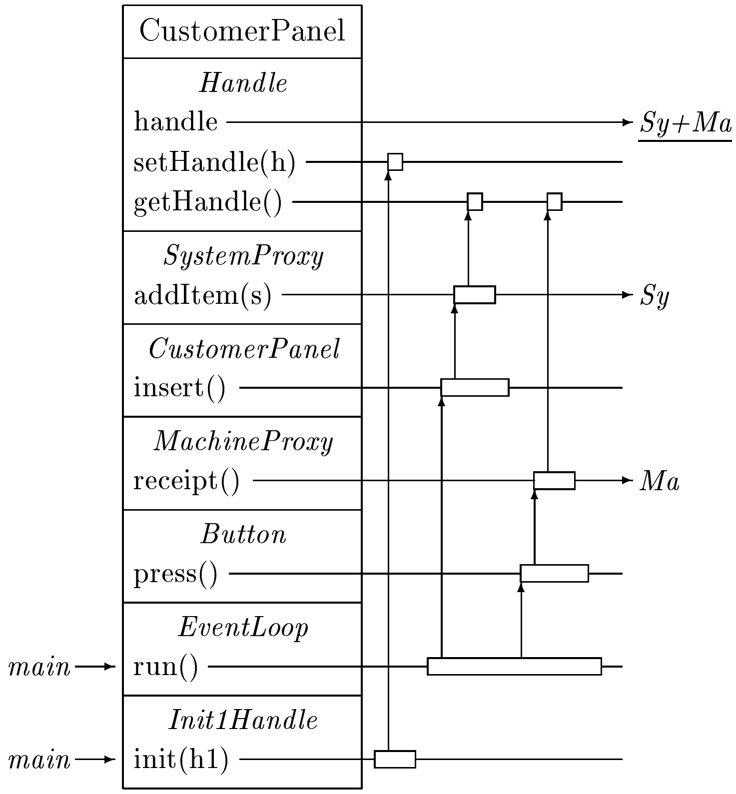

Figure 8.7: The original form of the CustomerPanel object before inserting roles for the Validate Item use case.

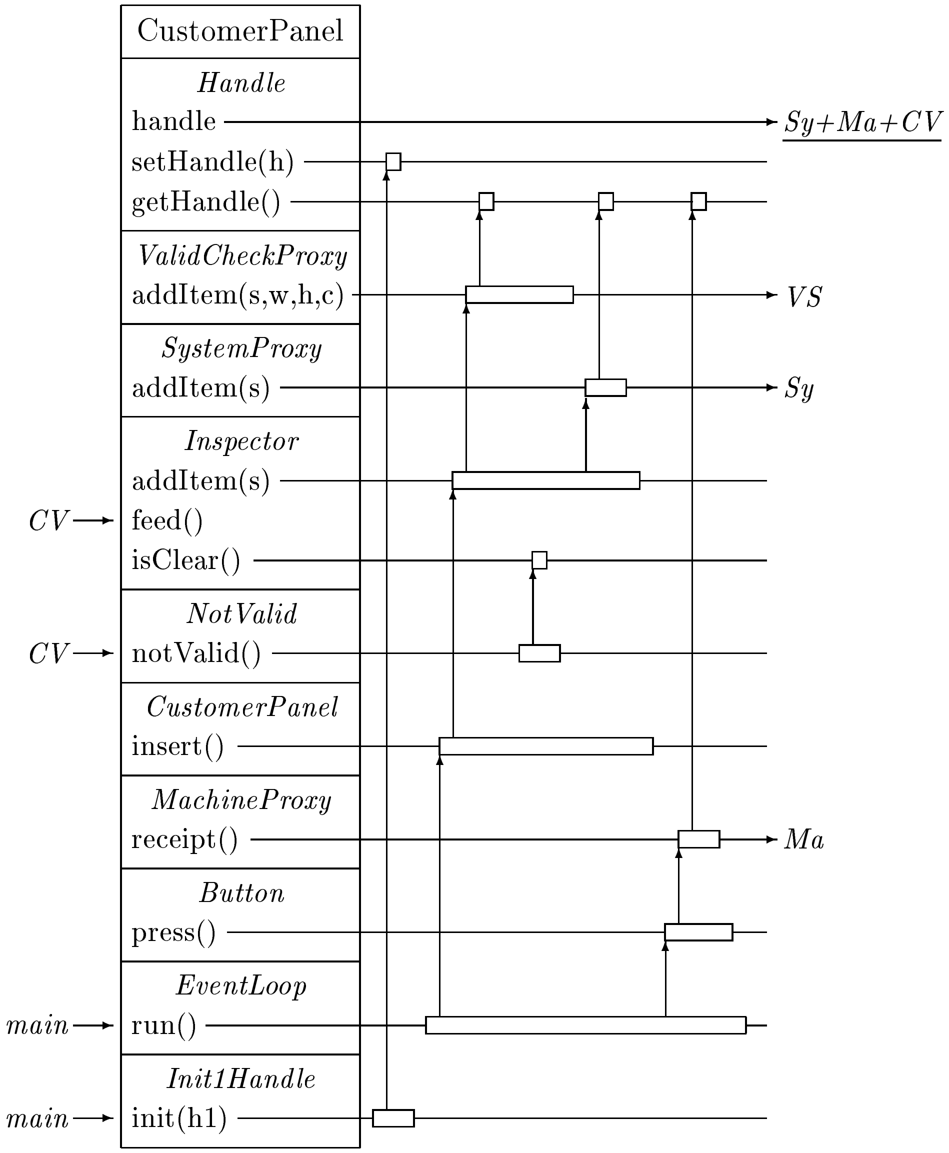

Figure 8.8: The complete composition of the CustomerPanel object after adding three new components for the Validate Item collaboration.

described in the previous step, this was intentional. The addItem method in the ValidCheck role, with additional arguments, was given the same name to indicate its close association with the original. The language should disambiguate the two based on their signature. If this was not the case, the situation would be noted for special attention in Step 14.

Step 12. Add proxies and handles.

We apply the same process used in development to the roles in any new collaborations to define and insert proxy components. Figure 8.8 showed the use of a ValidCheck proxy in the CustomerPanel object.

Step 13. Identify common classes and define class hierarchies

In this step we must revisit our earlier choices of class sharing to see if they are still valid in light of the new object compositions. Specializations may have been applied to some of the objects that shared class implementations, and not the others. We can again apply the same types of permutations as in development to try and maximize class sharing. In of our example of applying the Validate Item change, the level of available sharing is unchanged from before.

Step 14. Disambiguate names and note issues requiring special attention to types.

In this step we look at the repeated use of names and apply any adaptive measures that might be needed. In the case of change, we look at names that are used in new components, but also names that are used in existing components whose position has changed relative to each other. Usage includes both names that are defined, and names that are called or accessed.

Step 15. Add object and role initialization.

As discussed in development, initialization requires special attention. Because we have added new components to the mix, and may have rearranged the order of existing components, we must revisit the initialization design to make sure it all works correctly.

The original recycling machine design did not have ordering issues for the little initialization that is done. With the addition of the Validate Item components there are still no ordering issues to be addressed.

Step 16. Implement roles.

If all the arrangements work out, we will only have to implement components for roles in the new collaborations and new helpers for translation or initialization. For the Validate Item collaboration, we implement only the code specific to the four Validate Item roles and their proxies. All of the existing components are reused as is.

Step 17. Compose roles, objects and classes.

The original specification of composition is edited to add in the new components and apply any rearrangements that may have been made. For the Validate Item addition, we added a number of components to various positions in the composition of three classes. However, the order of composition among the existing components did not change.

Step 18. Write initialization calls and main.

We add declarations for new top level objects to our earlier version of the main() function. We also must add or change code for new or altered initialization calls.

For the new version of the container recycling machine with item validation, no new objects were created. We did, however, add a handle pointer in the DepositReceiver to point at the CustomerPanel object. The only change needed for the main() function is to add one argument to the deposit_receiver.init() call.

8.2 The Item Stuck extension

Part of the original challenge was to apply two changes in succession. In this section we discuss applying the Item Stuck use case. Since we have already discussed the steps in the process, we will just describe the highlights of the change.

The initial decomposition of the Item Stuck use case has three roles, a CheckStuck role that decides when to check for stuck containers, a FeedStatus role that can tell

> After signaling the CustomerPanel to collect the container but before incrementing any counts, the CheckStuck asks the FeedStatus if the item has become stuck. If the item is not stuck, the CheckStuck continues as before. If the item is stuck, the CheckStuck signals the Alarm to sound its alarm. The Alarm then waits, polling the FeedStatus until the jammed item has been removed. The CheckStuck then returns without incrementing any counts, and awaits new input.

Figure 8.9: The Item Stuck use case, as refined.

whether or not the container is stuck, and an Alarm role that signals a problem and waits for it to be cleared. When refined to fit the existing object structure, the FeedStatus and Alarm roles are both assigned to the CustomerPanel object. The refined description is shown in Fig. 8.9.

The roles/responsibilities matrix, with Item Stuck's roles and methods added, appears in Fig. 8.10. In adding the Item Stuck behavior, we will intercept the call within the DepositReceiver, from the ValidCheck component to the addItem(s) method in the System component. The overview of the entire composition with all changes and all proxies is shown in Fig. 8.11.

8.3 Discussion

The OOSE book emphasized choosing new objects when adding a new use case to an existing application. Because our unit of reuse is the role, we do not hesitate to assign new responsibilities to existing objects. In fact, we encourage it.

The process of change follows the same steps as the process of development. In most of the steps we were only concerned with new parts. Adding the two new use cases to the container recycling machine posed no special problems. The design that resulted from both changes had the same structure as the original application. It had no features that would indicate that it was the product of an initial application with

	Customer-Panel	DepositReceiver	ReceiptBasis	InsertedItem	DepositItem
Event Loop	*EventLoop* run()				
Linked List			*Head* list prepend() clear() getHead() getNext(n)	*Node* next setNext(next) getNext()	
Adding Item	*Cust'Panel* insert()	*System* item[N] addItem(s)	*Transaction* addItem(dt)	*CustomerTotal* count,dt setDt(dt) getDt() incCount() getCount()	*DailyTotal* count incCount()
Print Receipt	*Button* press()	*Machine* receipt()	*ReceiptBasis* getRcpt(txt) reset()	*InsertedItem* getDi() getCount()	*DepositItem* name,val getName(txt) getVal()
Validate Item	*Inspector* addItem(s) feed() isClear() *NotValid* notValid()	*ValidCheck* addItem(s,w,h,c)			*Validator* l,w validate(l,w,c)
Item Stuck	*FeedStatus* isStuck() *Alarm* alarm()	*CheckStuck* addItem(s)			

Figure 8.10: Roles/responsibilities matrix the recycling machine with the Item Stuck collaboration added.

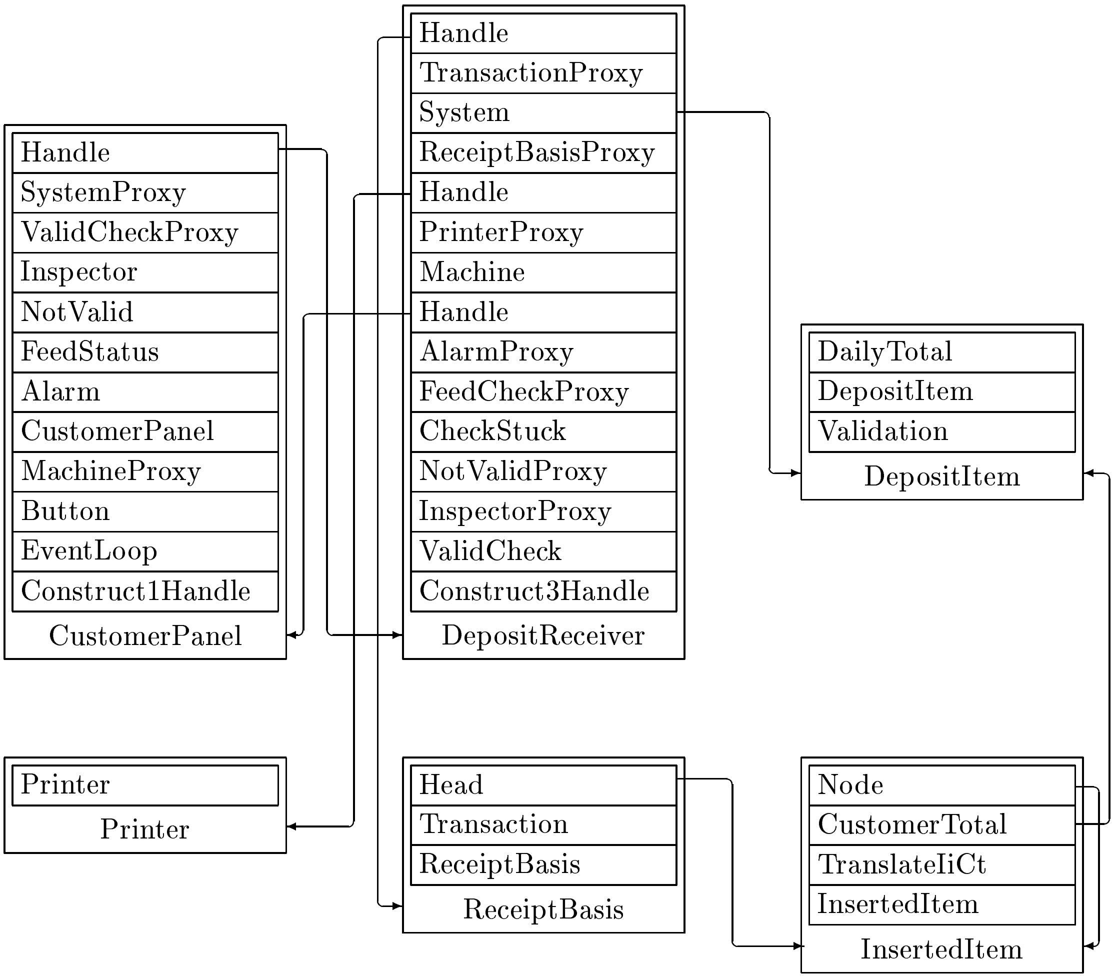

Figure 8.11: Overview diagram of the complete application with both the Validate Item and Item Stuck changes added.

subsequent changes applied.

Chapter 9

EXPERIENCE DEVELOPING A MEDIUM SIZED APPLICATION

To explore the feasibility of using our approach on a non-toy example, we developed a new implementation of an existing, medium sized application with which the author was already familiar. In this chapter we describe some of our experiences in carrying out that development.

The application discussed in this chapter motivated the initial investigations that lead to this work. It is a good example of the need for this kind of approach. Thus we describe a little more of its history and context than otherwise might be necessary. We discuss our experiences and some analysis of that experience in the latter part of the chapter. We conclude with an analysis of a few small experiments involving change.

The hypothesis to be tested was that the development approach could be used to develop a program of significant size, in its entirety. We were curious to know if there would be bottlenecks or show stoppers in the evolutionary approach to development, or the compositional approach to implementation. Could we handle composition and control flow in objects with tens of role components in an object rather than five or six? What kind of growth in complexity would we experience in an application upwards of a hundred components? We also wanted to get a sense of the kinds of idioms that would be needed and the relative importance among idioms. Finally, we were interested in the properties of the resulting artifacts—the design, the source code, and the executable program.

The development was carried out by the author over the course of ten months, interspersed with other activities. As the approach is anything but mature, no effort was made to compare the time and effort involved with other development methods. Much of the time was, in fact, spent testing different idioms and refining details of the method. We were, however, interested in the quality of the implementation and how it differed from other implementations of the same application. The author had developed two earlier implementations and is familiar with several more.

9.1 Description

9.1.1 Scalar Images and Pseudocolor Display

The application that we implemented displays scalar images on a pseudocolor display. Scalar images are images for which the pixel information of interest is intensity rather than color. With only one the intensity value for each per pixel, scalar images typically contain less information than color images. The issue for the pseudocolor display is to maximize our ability to perceive what little information the image contains. When looking at a monochrome display, like a black and white television, the human eye can reliably distinguish only six or seven levels of intensity. If we displayed our scalar image in monochrome, we would only see a small amount of the information available in the pixel intensities. By mapping pixel intensities to colors—say low values to blue, middle values to red, and high values to yellow or white—the eye can discern a far greater number of distinct levels.

9.1.2 Astronomy

The particular images that our application is intended to display come from astronomy. At night, when we look up at the sky, most stars look pretty much the same. But their intensities are actually very different, and many of them are not even stars. With powerful telescopes and sensitive detectors, astronomers take pictures of the night sky, collecting light from stars and galaxies, and also the background radiation—stray photons coming from seemingly random directions. There is a lot of information in those images, but it is hard to extract.

One of the problems astronomers face in looking at their images is discerning faint objects from the background radiation. Another is comparing how a star's brightness compares with that of its neighbors. Of particular interest is finding objects whose brightness, relative to their neighbors, changes over time. In nebula, light is given off or absorbed by particles of dust forming massive clouds. There is a lot of structure in the dust, but at a much lower level of intensity than the stars that shine through it.

Our application helps astronomers bring out the detail and subtle "shading" information in their images. It reads in digitized scalar data from an image file, rescales it to the number of colors in the display hardware color lookup table, and displays it on the screen. The user then plays with the colors in the lookup table to bring out the details of interest.

9.1.3 Intensity Scaling

There are two levels of intensity manipulation in the process of rendering an image. The first manipulation rescales the range of intensity values present in the data down to the range of values in the color lookup table. This process is sometimes called binning because we assign intensity values from the data to one of the two hundred or so bins corresponding to a cell in the color lookup table. The scaling is not always linear. An astronomer may want to highlight differences within the closely grouped values of the cosmic dust rather than among the much more spread out values of the stars.

The second level of intensity manipulation is the assignment of colors to cells in the color lookup table. In this case the astronomer chooses a sequence of colors to maximize conceptual distances around the features of interest without cluttering the image with a lot of color noise for features of less interest.

9.1.4 Auxiliary Displays

The user may want to pinpoint a specific pixel and take precise measurements from the image. But at the same time, they may need to view a large enough area to provide meaningful context. To enhance spatial acuity, the application provides a second window showing a magnified image of the area immediately around the cursor. It can also print the current cursor coordinates off in a corner.

If a user is not displaying the entire image in the main display window, they may want to know what part of the overall image they are looking at. A third window displays a small picture of the entire image and indicates the area of the main display by outlining it with a rectangle drawn as a graphic.

Because colors can be arbitrarily assigned in the color lookup table, the user needs to know the relative levels of intensity represented by different colors. To provide this information, the application provides a colorbar in which an artificial image of uniformly increasing value is colored by the same lookup table as the other images.

9.1.5 Overall Description

From the descriptions above, we see that the application includes a main display window, three smaller display windows for the magnifier and navigator, and a narrow colorbar. It may also have an area for printing the cursor coordinates, although this could be printed on the image itself.

The implementation has a number of challenges for coordinating different activities. The magnifier needs to track the mouse. The navigator needs to know the coordinates of the image being displayed in the main window. All three renderings from the image data should use the same scaling information. The application needs to behave reasonably when a window is resized. Different parts need to respond in unison, although they may view relationships through different coordinate systems.

The relationships themselves may apply to only parts of images or parts of display windows, e.g. when several different images are "mosaic'd" into a single window.

In a full application, there are numerous other features that would be needed to enhance its usability—including menus, popup windows, and additional controls to select new images and manipulate things like the color lookup table. However, once we had implemented the main functionality we did not wish to spend additional time polishing an application that is not intended for release.

9.2 History

The author first encountered the scalar image display application in 1982 while working at the Smithsonian Astrophysical Observatory. At the time he was assigned to upgrade a heavily used implementation written in assembly and a derivative of Forth. The application was specific to a particular piece of display hardware attached to a time-shared Data General mini-computer. Forth has an elegant stack semantics, but the syntax required far more comments than code to be rendered understandable. The program was small enough to be attacked, en mass, and by the time the upgrade was complete, eight months later, very few lines of code were untouched.

In 1987, the author was lured back to the Smithsonian with the assignment to write "the best image display program in astronomy." By that time, many astronomers were moving to workstations. Many of them were using a display program called Imtool that had just been written by the IRAF group at the National Optical Astrophysical Observatories (NOAO). But Imtool was specific to the IRAF environment and not general enough for widespread use.

We began by trying to extend Imtool. But we soon discovered that it was too dependent on the IRAF environment. There were IRAF specific assumptions distributed throughout the code. At the same time, the Suntools library was inefficient and made it difficult to customize parts of the user interaction for our application.

The new implementation, called SAOimage, was written in C for the then new X window system and used the Xlib library of user interface calls. SAOimage was based on a display program written by Bill Wyatt and included on the first X10 distribution tape. The user interface for SAOimage was developed by a collaboration of Eric Mandel, Richard Berg, and the author.

Largely due to its innovative user interface, SAOimage achieved widespread acceptance almost immediately upon its release [72]. It was used as a model in the development of the X11 toolkit, and is now part of the GNU distribution. Although written ten years ago, it is still in use by hundreds of astronomers around the world and a smattering of researchers in other fields, such as microbiology and physics. It was used by researchers at the Space Telescope Science Institute to view the first images returned from the Hubble Space Telescope after its initial deployment and after the optical corrections were installed. Even though several other good display programs with other features have been written since the design of SAOimage, many users are reluctant to switch because of their attachment to the SAOimage interface.

The success of SAOimage depended on our being able to develop in an incremental fashion, and to test alternatives in working configurations. Although our immediate users requested features from systems they were already using, we wanted to develop an application that took advantage of new technology to do things in ways that had never been done before. We also had become believers in user testing of interfaces. For earlier systems, we had written what we thought were ideal interfaces. But many users found them quirky. Imtool, also, has what is obviously Doug Tody's ideal interface.

Building alternative configurations proved to be a lot of work and progress was always slow. As features were added, SAOimage required several major restructurings to support further evolution. Design documentation, such as it was, fell way behind. The last major restructuring accompanied the port to X11, when a new window manager standard required the various menus and display windows to coordinate their placement and geometry in a different way.

By the time of the X11 port, the author had discovered that several groups in other locations were building their own extensions on top of SAOimage and were unwilling to adopt new versions after the restructuring.[1] Years later, some groups were still running X10 on their workstations in order to support their version of SAOimage. Simple renamings, intended to improve clarity, brought numerous complaints. The menu preprocessor, which had been created to support extension by others, was never used by anyone other than the author because its documentation was never up to date.

Over the years we have received a lot of mail requesting changes to support new uses and changing technology. Some changes are small, others not. Most requested changes seem reasonable, even desirable. But in almost all cases, the answer is the same. No, at this point it's too hard to change.

The design of SAOimage is based on a functional decomposition. The source code files are organized by concerns (e.g. file reading, window management, color). Within the files the concerns are subdivided into numerous functions. The functions pass or cache intermediate state in large data structures that are passed from one function to the next, but ultimately belong to main(). As predicted by Parnas, changing any of those data structures affects a large number of functions [49].

But it is not clear that an object oriented design would fare that much better. NOAO replaced Imtool with a new implementation, called ximtool, that has a nicely written object based design (though certainly not optimal). ximtool is designed to function as a widget in Tcl and has interchangeable shape objects for drawing graphics. But the main display object in ximtool is implemented in a file with 13000 lines of code. The high level design is quite logical, but the code within the main object is hard to understand and not much easier to change than the earlier implementation.

The problem with creating a nice object design for the image display application

[1] Two groups added a feature by replacing an existing one rather than trying to add something new, while Fermi Lab, on a big budget, has done their own major restructuring.

is that there is no one hierarchy that can logically structure all the concerns down to small objects. An object composition based on display windows is logical, and virtually required by widget oriented high level languages like Tcl/Tk or Visual Basic. But the relationships for image rendering, the sharing of coordinate systems, mouse tracking, and the response to window resizing, all follow different structures. The ximtool implementation hides the crossed couplings in a single 13000 line object module.

The approach we wanted to test with the new implementation is to address each concern in a separate collaboration, and then compose roles from different collaborations to create objects.

9.3 The Work

The development of the new application was not meant as a controlled experiment. Observations were to be qualitative, rather than quantitative. The developer had already the experience of developing earlier forms of the application, using common approaches, and was thus familiar with the conceptual problems that it posed. The new approach was still being developed, and thus much of the effort was directed at exploring and improving the method of development. Many of the idioms described in Chapter 6 were developed while working with this application.

The work was carried out, on a part time basis, over the course of ten months. The one developer was a programmer with ten years of professional experience in a procedural languages and five years of experience in object oriented development in an academic setting.

The requirements were made up of fifteen use cases, most of which were known at the outset. Examples of some of them include display window, create colorbar, render image, and propagate resize events. The requirements use cases were refined, and in some cases subdivided, as the work progressed.

The final application required about 3500 lines of code. The final design had eleven objects, implemented with 214 instances of 105 different role components. The components were grouped to form 10 classes in 6 separate hierarchies. The final collection of components includes 35 additional components, and 900 lines of code, for features that were tested but aren't being used and components used for debugging purposes.

An overview of the application is shown in Fig. 9.1. A listing of the main file with the composition lists and initialization code appears in the appendix. The navigator window and its renderer is omitted in Fig. 9.1 for lack of space. 120 component instances appear in the figure. Because many of the components are hierarchically defined in terms of smaller components, only 70 of the 105 different component types appear in the top level figure.

9.4 The Experience

Our experience with developing the application contrasted sharply with our earlier experiences in application development. At the surface, the main difference was that we were doing role analyses and writing little templates in the code. But at a deeper level, we found that the new approach changed the sequence of development and the way we thought about the design. Although, ultimately, the design is an object oriented design with 10 classes in 6 hierarchies, we would not have arrived at this particular design had we used a more conventional approach, going from the analysis to the design of classes.

The work proceeded in an incremental fashion, building the application by adding one concern at a time. At each step, the concern was refined in terms of its collaboration and a source code template for each role was implemented. Because of the autonomous nature of collaborations and their roles, there was no reason not to implement the roles once they had been specified. This allowed us to explore the handling

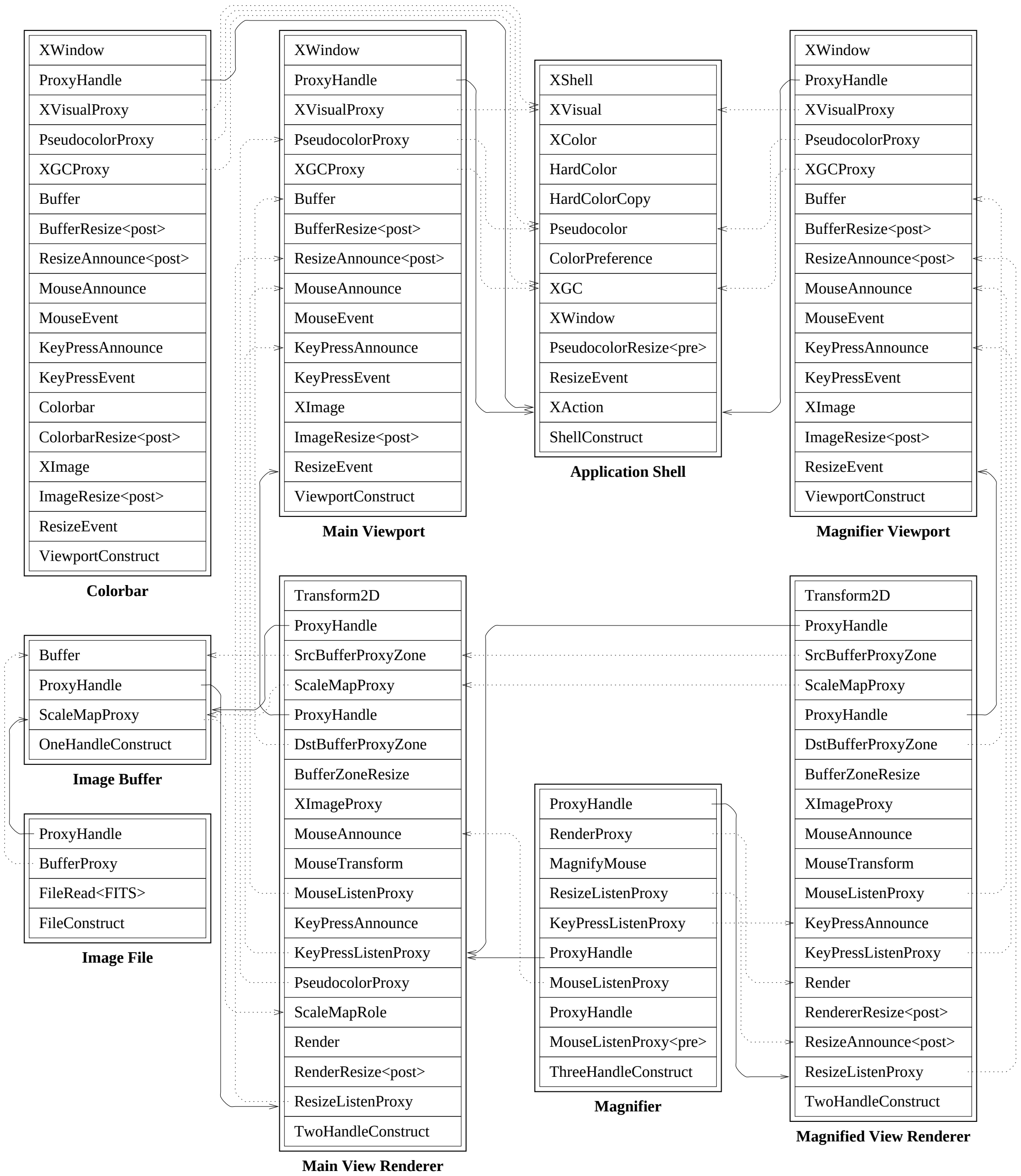

Figure 9.1: Overview of one configuration

of each concern and test it with running code, before committing to a particular strategy in the design. This horizontal approach contrasts with the vertical waterfall approach proceeding from requirements to a complete design, before implementing the code.

As we had expected, the order of evolution of the image display application did not at all follow the order of inheritance in the class compositions. Added features tended to go in the middle of the class hierarchy where they interact with existing features in a natural order. The more derived roles in each composition tended to be roles for event detection and object initialization. The code could not have been implemented in this way had we been forced to order the inheritance to support evolution.

The relationships between collaborations allowed us to work at intermediate levels of abstraction. The difference can be seen in how we implemented the magnifier. The magnifier tracks the mouse in an image display window and displays the area of the image around the mouse, magnified. In earlier versions, the magnifier addressed many low level concerns from interpreting low level mouse events to packaging the magnified image and sending it to the screen.

In the new abstraction, the magnifier knows nothing about windows, image handling, or being a magnifier. It simply tracks the mouse across the image and requests a rendering at a particular scale centered on that position. To provide the abstraction of the mouse moving across the image instead of a window, we inserted mouse tracking components in the image Renderer object, including a MouseListener to get the events from MouseAnnouce component in the window, and an adjacent MouseTranslate component to translate the window coordinates to image coordinates. Another MouseAnnounce component makes the mouse move events available as events from the Renderer.

The abstraction of the magnifier collaboration has three roles: MouseEvent, MagnifyMouse, and Display. We added a keyboard interaction, with a KeyPressEvent role, to allow the user to change magnifications. Instead of implementing a whole new magnifier module (600+ lines in the earlier implementation), we wrote only a 44 line magnifier role, most of which is code for the keyboard menu. Components for mouse tracking and rendering already existed, having been developed in other collaborations. Because the existing instances of rendering and tracking are integral parts of dissimilar objects, we would not have been able to reuse the code in a more traditional approach.

The Magnifier implementation also demonstrates the flexibility inherent in our approach. We could place the Magnifier in either the source or destination Renderer object—Main View Renderer and Magnified View Renderer, respectively, in Fig.9.1. We chose, in this case, to create a separate Magnifier object to allowing us to track mouse events in any renderer (two are supported here) and ask any renderer to display the magnified image. The Magnifier is connected to roles implemented in other components through proxy components—its implementation does not depend on which of the three alternative designs is chosen. The earlier implementations could not offer this degree of flexibility.

Using collaboration views and role components led us to maintain an entity structure, even when implementation concerns might have dictated otherwise. If we look at a calls graph for this application, we will probably find more calls between the Renderer and its Viewport, than among roles within the Viewport. But the abstraction is better served by their separation. Within each collaboration, the separation is logically associated with window and image entities. The Viewport object uses the coordinate system of the window, while the Renderer uses the coordinates of the image file.

We found that the development process emphasized objects rather than classes. The design of classes was almost an afterthought. By inspecting the role hierarchies, we could easily identify common base classes. In some cases, by changing the order of role composition, we were able to reduce the number of classes needed by the application. For example, we were able to reorder the Colorbar to share a common base class with other Viewports based on the first twelve roles. This commonality had not been recognized in the original SAOimage where the implementations appeared to be very different. The Main Renderer differs significantly from other renderers in that it computes its own scale map. Yet, by adding an unused ScaleMapProxy and reordering the roles, we created a shared base class among all Renderers for the first fourteen roles. We were later able to take advantage of the common proxy to make the Magnifier more general.

As discussed in Chapters 6 and 7, we managed control flow with propagation components, and selectable pre- and post-processing options. We found this approach very useful. It allowed us to coordinate initialization and the response to resize events without writing extra code. The use of separate resize, key press, and mouse tracking components allowed us to design and extend the propagation of those events, as illustrated in the Magnifier example described earlier. Combining implicit invocation with listener "proxies" allowed arbitrary paths to connect up at initialization without external code. Listener proxies register an event callback with an announcer in the object pointed at by a ProxyHandle.

Reuse was common within the application. Sometimes the reuse coincided with the possibility of a shared base class. But often we reused components in objects that are otherwise different. The ProxyHandle component is used in almost every object, often several times in the same object. Propagation components for mouse, key, and resize events, and components for data buffers, also appear in a variety of objects. This kind of reuse would not be possible with C++ multiple inheritance. The repeated uses of ProxyHandle would not be possible with CLOS or Dylan mixins, either.

9.5 Analysis

As we explained earlier, the exercise of building a new image display application was not, in any sense, a controlled experiment. Our interest was in gaining initial anecdotal experience, and in having a larger test bed for refining our approach. Any comparisons with other methods can be only vague and qualitative, and must be prefaced with the understanding that, in a scientific sense, they may only apply to a particular application being developed by a particular programmer with an obvious bias. However, that being said, we do feel that we gained some valuable insights.

9.5.1 Complexity

Complexity, for any large application, is an issue. We were able to follow the code by looking at the instantiation lists and the file names in the build directory, but not without difficulty. The experience was similar to looking at a large Smalltalk program—where the code is broken into small pieces that must be browsed.

The design and code was much easier to understand once we started using an overview diagram, similar to the one in Fig. 9.1. Whenever we returned to the application source code after a long period of being away, we always started by looking at the component overview. The overview diagram gives more detail than class or object diagrams while providing better abstraction than looking at source code. With one overview diagram we can trace each concern through the design and identify the components that address it. We also get a sense of how different concerns fit together. After studying this diagram for a while we could usually discover where the last build left off and where the next build should begin. Between builds we referred to the diagram often to find out where new concerns had not yet been addressed.

Working with the overview diagram and small components was a lot easier than working with large source code files. The largest component in the application has 300 lines of code. While a few other components have around 200 lines, the average component has fewer than 40 lines. The overview diagram also saved us from having to traverse levels of hierarchy to find where something was implemented.

Understanding the design well enough to find code, is one issue, but understanding it well enough to change is another. Rearranging composition order requires understanding the dependencies among roles. This was not as much of a problem as it might seem. With the exception of proxies, which must obviously appear before the components that use them, only a few concerns involved multiple roles. We separated out the resize concern into its own components to make it more visible. That left initialization as the one issue that had us looking at code.

The intra-object dependency annotations were sometimes helpful in development, but rarely essential. For maintenance, they created tremendous leverage for understanding the makeup of an object in one quick glance.

Drawing the diagrams without special tool support was extremely tedious. We often thought in terms of the the roles/responsibilities matrix, but we rarely drew it. We suspect that the development experience would be different with tools to aid in producing the various diagrams.

Our inheritance hierarchies tended to be much deeper than those found in other object-oriented applications. Each of the major classes is composed of 10 to 20 roles. In other approaches, deep structures spell serious trouble because of hierarchical ordering and the unzipping problem. In our experience the deep structures were not an issue.

9.5.2 Compiling and Debugging

We had problems controlling template instantiation with separate compilation. For some files, the template instantiation must be analyzed to produce information needed by the linker. But the instantiation should generate an executable image only once. At the time we did most of our work, we were unaware of a standard syntax for controlling this distinction. Because we used several different compilers, we avoided the problem by keeping the instantiation lists in a single file and compiling everything at once. The problem is not unique to our work. We assume there will be a solution, if there is not one already.

Debugging differed from our earlier experience both conceptually and physically. It differed conceptually because we viewed concerns in terms of collaborations, while the debugger only provided an object and class view. It differed physically because the location of code could depend on both the object and the role component's subclass. This added a little difficulty when setting break points. We also occasionally encountered template instantiation names, which are always ugly.

For many components, we created instrumentation wrappers, which could be applied in place of the naked components in a debugging version of main.C. This made it easy to swap instrumentation in or out at any level to get runtime feedback, without editing the actual code. The wrapper made it easier to locate the source of a problem without the tedium of setting up a lot of debugger watch points.

9.5.3 Change

In our approach, the process of development and the process of change are very similar. Our development process could be characterized as applying one change after another as each new concern was added on. The major unknowns, concerning complexity and other issues faced by larger applications, were observed and tested during the course of development. However, we also made some attempt to simulate a more characteristic scenario for software evolution.

Eleven months after the application was last touched, we conducted a limited number of experiments trying to apply changes that were not thought of earlier in development. The process of adding a requirement, e.g., a new key press interaction, is no different than adding any other concern in our incremental approach to development. Changing a concern that was already included required a little more thought. For some concerns we could remove the original roles and treat the change as an addition. An example here might be changing the way in which file names or image data are passed in.

Many changes could be handled by modifying an identifiable set of roles and substituting the new versions. Adding a new keyboard interaction that required richer detail than initially assumed for key press events involved changing every component that handled a key press event. While key press events are passed in many places, we only needed to consider five components, totaling 140 lines of code including comments. Four of the components were identifiable by their participation in the Propagate Key Press collaboration. They could also be tested in that limited context once the change was applied. The remaining component appeared in the keyboard interaction of the Magnify collaboration. To be fair, the same components could be identified syntactically by searching for variations of the word "key," but no semantic meaning accompanies this alternative. Viewed in terms of a more traditional object oriented design, the key press change occurred in the middle of three distinct inheritance hierarchies affecting seven different application classes, and it required extending the interface in each case. In our approach, that poses no special problem. Extending interfaces without concern for its effect on the type system is a major strength of our approach.

Even for the biggest change, creating a different application with a completely different architecture, we found that we could reuse many of the existing components that address similar concerns. We created another window application for a key press interaction, but no image display. It was as if, in the course of developing our application, we had created a library of generally useful parts.

Compared to our earlier SAOimage implementation, the outlook for change is very different. Whether for simple changes that affect a few role components, or a huge change that reuses components as if from a library, the new implementation offers many alternatives that were not available in the context of the earlier implementation.

Chapter 10

RELATED WORK

Our approach to software development and software evolution touches on related work in many aspects of software development. We divide our discussion of related work into three sections. The first section looks at discussions of the software evolution from a theoretical or high level point of view. The second section discusses design methods that are similar to our own, and in fact served as precursors for our own ideas. The third section discusses work on program implementation techniques that seeks to provide a similar level of flexibility as that discussed here.

10.1 Theory

In studying the nature of systems, Simon declared that all natural and artificial systems evolve [64]. He also pointed out that the evolution of complex systems can proceed much faster if stable intermediate parts exist.

Lehman and Belady studied software evolution extensively [37]. They describe many of the qualities of support that we list in the introduction. They do not provide a list for issues of support, as such, which is why we felt a need to do so. However, they list the characteristics of program evolution, presented as a set of laws. Our approach does not change the validity of any of their observations. We provide a quantitative improvement in process within the framework they describe. As an example, one of Lehman and Belady's observations is that as the rate of change increases, more of the effort must be diverted from enhancement to redesign, restructure, and reimplement. Our approach allows more changes to be applied without needing a cleanup, and lowers the cost of both applying change and performing cleanups. In our approach, cleanups can be as major as regrouping components in a different architecture or as minor as subdividing an existing component into two smaller components.

Parnas discussed criteria for choosing the best modularization for supporting evolution [49, 50, 53]. We presented a response to Parnas' early papers on modularization in one of our conference papers [73].

In addition to choosing a good modularization, Parnas argued that should hide the details of their implementation [49, 53]. On the one hand, we feel that the level of information hiding should be appropriate to the context of its use. At finer levels of granularity, more detail must be revealed to allow tighter integration. The need for hiding can be moderated by the use of adaptor components. On the other hand, the reasons for information hiding work both ways. Not enough attention has been focused on shielding the surrounding context, especially details of an application's structure, from the implementation of modules.

At the time Parnas wrote his seminal papers, the dominant paradigm for modularization was functional decomposition. Parnas presented his case for information hiding together with an argument for data abstraction [49, 50]. We feel that in carrying out Parnas's ideas, the scales have been tipped a little too far in the direction of data abstraction. Further improvement in modularization requires that we give sufficient attention to behavior abstraction as well.

Parnas also presented a case for the hierarchical structuring of decisions [51, 52]. He observed that while hierarchy is a valuable model for composition, it also works against change. Parnas discussed the problem of working within the constraints of a hierarchical structure, but did not present an alternative. Our work preserves the hierarchical structuring of composition. But we provide a non-hierarchical mechanism for change.

10.2 Design Methods

Beck and Cunningham pioneered the method of transforming user requirements into object design by mapping role-like concerns onto objects. In the CRC (Class, Responsibilities and Collaboration) approach to analysis, each object in the design is represented by an index or CRC card [9, 10]. The analysts decompose usage scenarios around objects to identify the responsibilities of each class, writing them down on the CRC cards as they go. The information on each card is then used in designing the code for that class. The CRC card approach has been used for teaching object oriented methods [10] and is incorporated in at least one development method [77, 78].

The CRC card approach provides a good start for object oriented design. But it is generally used only as an informal exercise for eliciting design information. No attempt is made to relate responsibilities back to requirements or to retain responsibilities as identifiable components in the design.

The OORAM (Object Oriented Role Analysis and Modeling) method of Reenskaug, et al., uses role analysis to decompose problems into simpler concerns [55, 56, 57]. The OORAM approach is analogous to earlier object oriented methods like Coad and Yourdon's Object Oriented Analysis and Design (OOA/D) [18], in that the decomposition starts with a general notion of the problem domain [18]. But unlike OOA/D, it decomposes the domain into both entities (objects) and behaviors (collaborations). In the OORAM method collaborations are called role models. Larger complex role models are decomposed further into smaller role models until a manageable level of complexity is reached. Roles are abstractions used to talk about objects while considering only aspects and details relevant to a particular concern. There is more a sense that objects play roles than that they are composed of roles. Philip Dellaferra coined the hat tree analogy where an object puts on different hats to play different roles [55, p.11].

In the OORAM method (formerly called OORASS), role definitions are combined by the developer when designing each object's class in a step called synthesis. "When we subdivided our area of concern into smaller areas, creating role models for each subarea, we succeeded in reducing the modeling problem to manageable proportions, but created the new problem of integrating the smaller models into a model of the entire area of concern. The OORASS solution to the integration problem is called synthesis" [57, p.35]. As in the CRC approach to defining classes, the synthesis process is largely informal and unstructured, although, with available tool support, pieces of the resulting code can be annotated with role associations.

The dominant process of analysis in OORAM is top-down decomposition. Problem descriptions are decomposed into smaller pieces until a manageable level of complexity is attained for each piece. The decompositional approach addresses complexity as a normal part of the design process. No part of the process, however, corresponds to change, and there is no concept of overriding. In top-down decomposition, reuse and change are handled in the same way. During the process of decomposing the concerns, the designer looks ahead to anticipate where reusable components might fit in and guides the decomposition toward them. Reenskaug uses a yo-yo analogy to characterize this alternation between top-down and bottom-up views. To apply a change, the designer revisits the original process of decomposition, starting with a new problem description that includes the change. The designer looks ahead, this time, to try to reuse the original design wherever possible.

Decomposition allows the designer to isolate "generally useful" pieces of the design that either are being reused, or might be reusable in a future context. But only pieces that fit "as is" can be reused. If a piece doesn't quite fit, one decomposes further, to try to isolate the parts that do. The same process is used in separating the parts that changed from the parts that didn't.

The OORAM design process uses a separation of concerns in the design analysis, but that separation is not maintained for software evolution. Without a structured approach to synthesis, there is no way to separate parts of the synthesis process into parts that are affected by a change and parts that aren't. Making even a small change requires revisiting the entire synthesis process for each affected object.

In our approach, roles are treated as entities with identities of their own. Issues between roles within the same object, such as overlap and dependency, are addressed in a disciplined way as part of a design process that extends the role structure into source code components.

In Jacobson's OOSE (Object Oriented Software Engineering), the design analysis starts with a list of use cases [32]. Use cases describe behavioral requirements from the user's point of view. As defined by Jacobson, the behavior in a use case begins with a stimulus from an actor external to the system and may involve a sequence of transactions. In the analysis phase of design, design objects, called blocks,[1] are chosen by a combination of domain analysis, inspection of the use cases, and an assessment of future change. Each use case is then refined and expressed in terms of specific objects in the design. In the construction phase, pieces of an object's interface and behavior are collected from the refined description of each use case and used by the implementor in constructing the implementation.

OOSE has two special kinds of use cases. Abstract use cases capture the common part between two overlapping use cases. Extension use cases extend or override parts of the behavior of a normal use case. Extension use cases arise in the requirements analysis and allow a system's behavior to be described both in terms of the general (base) case and more specific (extension) cases. Our approach uses both abstract and extension use cases more aggressively than in OOSE, separating them from other use cases not just in the requirements analysis, but throughout the process of design. In our approach, abstract use cases are part of a more general model of decomposing use cases to smaller collaborations.

[1]In the design phase of OOSE, design objects are called blocks to allow more flexibility in implementation. Generally a block's interface is implemented by a single class, but in some situations it may divided among several separate classes.

In OOSE, a role corresponds to the interface and behavior of a block in one use case, but no name is given to that concept. The object or block structure of the design is assumed to be frozen early in the analysis, so no separate concept is needed. By contrast, our approach makes roles the central focus of analysis. OOSE carries the refinement of each use case's part of a design object quite far in the design process. However, like the synthesis step in OORAM, OOSE has a step in the construction phase that is informal and unstructured. Thus, like OORAM, requirements can be mapped to and from specific groups of objects, but not within the objects, although source code annotations can be used. As mentioned earlier, our intention is to extend the concept of a role all the way into implementation.

The dominant process of analysis in OOSE is stepwise refinement. Use case descriptions are refined at each step of the design process. After the requirements analysis phase, use cases are not decomposed into smaller use cases (although abstract use cases can be used to describe shared overlaps). Object choices too are made early in the design analysis and held constant for each use case throughout the remaining process of design. The lack of further decomposition pushes a significant amount of complexity into the less structured implementation parts of application construction where it must be revisited by changes that affect existing objects. As described in an earlier chapter, OOSE's tendency is to apply changes by adding new objects to the design.

Applying new requirements requires applying a different design process or repeating the original process with a new definition of the problem domain at the start. The danger in either case is that much of the existing implementation will be invalidated.

Our approach stresses both top-down decomposition and stepwise refinement. As in OOSE, new requirements are added as new and extension use cases. Overriding is an important mechanism in that process. Unlike OOSE, existing objects and use cases can be further decomposed to accommodate the change within the existing structure, while still isolating other parts of that same structure.

Although our focus is on change, the differences among OORAM, OOSE, and our own approach more clearly manifest themselves in a comparison of how each approach handles reuse.

Because of OOSE's early commitments to decompositions based on the problem domain, the resulting design's architecture is strongly tied to specific parts of that problem domain and thus unlikely to be suitable for reuse within or across applications. Reuse of prior artifacts must occur either at the beginning of the refinement process, when the application's block structure is chosen, or at the end of the refinement process, when each block is implemented. In the former case, portions of the block structure are chosen to correspond to the structure of a multiblock artifact (e.g., design pattern) to be reused. The artifact's description can then be incorporated in the use case description as part of refinement. In the latter case, code artifacts that can be used within a single block (e.g. simple data structures for handling attributes) are used by the implementer. In the latter case, the reuse appears only in the implementation— it does not appear as part of the design process. In reuse between successive versions of the same application, whole use cases are reused. Changes are applied in new extension use cases using the semantics of overriding where needed. As discussed in an earlier chapter OOSE attempts to leave the implementation of existing use cases untouched by choosing distinct objects for the new use cases. Where existing implementation is affected, the details of the change are addressed informally in the process' unstructured implementation step.

In the OORAM approach, decomposition is used to isolate "generally useful" pieces of design [57]. To reuse prior artifacts, the decomposition must be guided toward the structures being reused. The design process is modeled top-down, but artifacts of potential reuse can only be seen by taking a bottom-up compositional view. Thus the designer must actually take a dual view approach, described by Reenskaug using a yo-yo analogy [55, p.26]. In OORAM, patterns can be expressed as role models and data structures can be isolated with the appropriate decomposition.

Without the concept of overriding, only pieces that fit "as is" can be reused. Because of the problems of applying change within an existing design, as discussed earlier, reuse between successive versions of the same application is not well supported.

Our approach combines top-down decomposition and stepwise refinement. Generally useful pieces can be isolated to reuse existing parts, or for future reuse, as in the OORAM approach. Whole use cases, and abstractly useful use cases, (e.g., pieces that were used before applying a change) can also be included and refined for reuse, as in OOSE. Unlike OOSE, our approach allows further decomposition during the refinement process to reuse collaborations and roles that do not correspond to complete requirements use cases. Unlike OORAM, our approach supports adaptations to reuse parts that would otherwise need modifications. Unlike either OORAM or OOSE, reuse can occur at any point in our design process. Top-down decomposition favors the reuse of common components through aggregation, while stepwise refinement favors a reuse of common base classes through inheritance. Our method leverages the advantages of both approaches, as well as the symbiosis between the two.

10.3 Implementation Mechanisms

In this section we look at related work on mechanisms of implementation. A variety of research has addressed the decomposition of objects into factors and the composition of those parts of objects to form the objects of an application. Some related work also concerns the use of type parameters or flexible inheritance to achieve adaptable implementations.

We group the mechanisms under three categories: preprocessors and code generators, approaches that involve manipulating the dispatch mechanisms directly, including metaclass implementations, and various language level approaches to inheritance. The three categories are not mutually exclusive. Our own approach uses templates, which is like preprocessing. But we parameterize inheritance using a language feature that has the affect of allowing us to manipulate inheritance relationships.

No one of the three approaches is necessarily better than either of the others. In fact, for each of the three groups, we have found approaches to implementation that may be viable alternatives to the template approach presented here. In each group, we present these hopeful candidates last.

10.3.1 Generators and Preprocessing

In theory, code generators can produce any code. A good generator needs only a small set of parameters to produce the code for a desired application. For a role oriented design one could specify the roles and their relationships and let the generator work out the details of connecting up dependencies and perhaps even determine the control flow. Changing the implementation then involves editing the generator script (presumably a small task) and rerunning the generator.

Generators are usually built to produce domain specific components and applications within a fairly limited range. The programmer chooses from a fixed set of components to be arranged in narrowly defined relationships. A careful domain analysis by the builders of the generator is needed to assure that the components and their relationships cover common concerns in the particular domain. Adding new components with new interfaces requires writing a new generator.

Preprocessors are usually more general than generators. They define a syntax for special preprocessor commands that can be added either within the native code, or read separately. The preprocessor then reads code which is close to the final implementation, or a representation of it, and transforms it into compilable code.

Lieberherr's Demeter [38, 39, 40] is a preprocessor environment that 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 separates the specification of behavior from the specification of structure. In Demeter, components encode fragments of the program structure to allow their position in the finished application to be inferred. Our components do not encode any structural dependencies. Instead, we require explicit statements to specify the positions of components. The Demeter model is based on graphs and graph traversals rather than object modules, and requires the developer to think of the application in terms of a graph model. The Demeter system requires a special development environment and its own preprocessor.

The Predator and P2 systems, by Batory, et al., factor data structures and other domain specific structures into independent components [7, 8]. In the Predator approach, hand-crafted generators are used to compose modules from a limited set of choices. The fine grained factoring in this approach is similar to our own, but applicable only to a small set of well understood domain components. Our approach uses the compiler's own template class generating features, and is applicable to every module in an application. New components in our approach can be written by the application developer as they are needed.

Recent work by Kiczales, et al., addresses an idea called aspect oriented programming [35]. Aspects are issues that necessarily cut across multiple components. In normal component designs, aspects are mangled in the code that implements each component. Like Batory's generator approach, the AOP approach is to perform a domain analysis on families of applications and to then create small generators, called weavers. Weavers take code fragments that address the aspect's concern and weave them into the application code. The AOP weavers are intended to be simple enough that the cost of building them can be amortized over a relatively small number of applications.

At some level, the AOP approach is similar to ours—we both create code fragments that address specific concerns and compose them in various ways to produce applications. Unlike AOP, we currently do our weaving manually. (This is not to say, however, that, in the future, the composition process could not be automated.) In using an AOP weaver, there is no guarantee that the code fragments will remain distinct in the generated code—code from several fragments may, in fact, be merged on the same line. Not having to isolate code for different concerns gives the AOP approach more flexibility in addressing specific concerns. But it may also make it more difficult to understand the resulting compositions. The main difference between our two approaches is more one of emphasis. The work on AOP seeks to carve out those aspects for which a component approach would not be feasible. Aspects that seem likely to fall into that category include persistence, concurrency, security, and distribution. Our approach is to carve out those aspects for which a fine grained component approach is sufficient. However, there is plenty of overlap. Mendhekar, et al., wrote a paper on an aspect oriented approach to ordering processing steps in an image enhancement application—an example that also has a role oriented solution [43]. Similarly, the proxy components described in Chapter 6 can be used to address some issues of security and distribution.

Bassett uses a preprocessor to compose Cobol applications hierarchically from small code fragments. His system, called Frames, defines a context free preprocessor language that operates on text, and supports conditional string and section replacement [4, 5, 6]. The preprocessor makes multiple passes, allowing replacements to be propagated through hierarchical relationships and allowing preprocessor commands to be embedded along the propagation paths. The language includes arbitration to handle conflicting and overriding commands. With the power of this preprocessor, it should be possible to define composition operations on fragments similar to our role components. The ability to define nested and conditional operations might add capabilities beyond those of C++ templates. The Frame approach is independent of the target language.

10.3.2 Dispatching and Metaclasses

Harrison and Ossher [29, 30] have been working on composing fragments of object-oriented programs under the name subject-oriented programming. The intent of SOP, to allow different applications to be implemented with different concerns and then merged, is similar to our own. Their is a slight difference in emphasis—the SOP work has been geared toward merging large systems, while our approach aimed at decomposing to small concerns. The SOP approach uses a special runtime dispatcher that "merges" classes by rerouting method calls. Their system supports a larger variety of merging semantics than the inheritance and parameterization mechanisms used here. The initial implementation of the SOP dispatcher required two indirections per call, which, for their model of moderate decomposition, was not a heavy overhead. Recent work on compile time efficiency improvements may make the SOP approach to composition a viable implementation for the work presented here. The SOP approach offers many potential benefits. In addition to the richer merging semantics, their approach merges compiled components and components implemented in dissimilar languages.

The composition filters of Akşet, et al. [1] also address composition through runtime dispatching. Their Sina language offers additional capabilities such as dispatching based on runtime queries, and can encode a wide range of concerns, including persistence and transactions. The method call dispatchers in both these systems provide considerably more compositional flexibility than our approach to implementation. But custom dispatching requires special runtime support and adds extra levels of indirection. Our approach stays within the semantics of inheritance and aggregation. It can be handled by standard C++ compilers and does not require any special, extra-language mechanisms.

Mezini has developed a Smalltalk metaclass that allows mixin-like classes to be composed into the inheritance structure of a class [45, 46]. Her metaclass manages the dispatching and variable accesses, and also manages the name spaces. Although we have not used this implementation, we have talked with Mezini and it appears to support our approach. In addition, her mechanism is dynamic—compositions can be changed at runtime. With this mechanism, our approach might be usable for rapid prototyping.

10.3.3 Inheritance and Parameterization

It is interesting to note that in the first object language, Simula I, objects could be nodes of multiple data structures [48]. Simula I did not have inheritance. In Simula II, when inheritance was introduced, Nygaard, et al., had to sacrifice the multiple node feature due to the hierarchical nature of inheritance.

Bracha and Cook demonstrated delayed inheritance using type parameters, calling the resulting components mixins [14]. The term roughly corresponds to the use of multiply inherited classes in CLOS and Dylan. 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 method [13].

Multiple inheritance is treated as linear inheritance in CLOS and Dylan. Multiply inherited classes can call functions in other multiply inherited classes if those classes appear earlier in the order of linear inheritance. The semantics of composing mixins in linearized multiple inheritance is essentially the same as our lists of singly inherited template instantiations. The implementation of multiple inheritance in CLOS and Dylan, does however have one significant difference: variables or methods with the same name are merged. Our approach allows names, and even whole mixins, to be reused in the same object. In CLOS, the problem of merged name spaces might be resoled by changing the inheritance semantics in a metaclass, much like the Mezini metaclass in Smalltalk, discussed above.

Goguen did extensive work on reuse and parameterized programming in Ada and his own specification language [27]. This work has similarities with our own, including building applications from code fragments, and composing them using module expressions. Goguen developed his ideas on parameterized programming at a time when the dominant model of program structure was functional. In addressing the problem of how to compose the fragments, he chose a functional model. He developed an approach, using a higher order functional language, based on the theory of sorts, that works with semantic constraints on the components and their compositions. A major concerns appears to be capturing some of the semantic issues of composition in the components themselves. Our approach requires the developer to separately analyze composition semantics when building the composition. OBJ's model and syntax are difficult for non-mathematicians to understand. By contrast, we address composition as simple inheritance. Our components and specifications are written in a common programming language. The underlying model of inheritance and parameterization can be described using a modal logic where the semantic interpretation of a component depends on its position in a structure [22, 63].

Steyaert, et al., demonstrated a dynamic mixin mechanism in a prototype delegation language called Agora [67]. In Agora, an object includes a set of alternative mixins from which it inherits. The object accepts messages to dynamically switch among the mixins in the set, but new mixins cannot be added. The mechanism supports alternative roles, but not extension.

The ML language supports encapsulation and inheritance-like properties through functors [71]. Functors are operations that take one or more structures and produce another structure that combines them in some way. But the signature of the structures on which it operates cannot be a parameter or of variable type—it's implementation names the signatures on which it operates. In our approach a role may expect some features of the supertype's signature in a composition. But the resulting composition must include all the other features of the inherited supertype—they might be needed by the next role in the composition. In a private conversation, Peter Lee of CMU said that they had thought of providing functors with parameterized signatures, but, at the time, could not think of a compelling use.

Erik Ernst, at Aarhus University, is working on a new definition of the semantics of BETA that allows mixin style multiple inheritance without name ambiguity. Where a name is declared more than once, the semantics defines which name will be externally visible and the conditions under which each declaration is available for binding. This approach should support our technique, although it adds an overhead for each component. Some additional code may also be needed in the container that specifies an object's composition to address sharing of state between different components. No papers have yet been published on this work.

Chapter 11

CONCLUSION AND FUTURE WORK

11.1 Supporting Change

In the introduction, we presented a list of criteria for judging the nature and extent of support for change. In this section, we revisit that list and describe the ways in which our approach addresses each property.

Our support for change has two major themes. First is that we model change in terms of use cases, and then keep that same model through the design and in the implementation. Second, our approach to applying change is compositional—components specific to that change are inserted into compositions with existing components of the application. Other than the possible addition of new objects, the existing object structure does not change.

- Flexibility refers to allowing many kinds of changes with simple fixes.

  Our approach allows changes, modeled by use cases, to be added without major restructuring or having to edit a lot of existing code. Our approach retains the ability to support changes to data types inherent in object oriented implementations, while adding support for behavioral change. It has fewer limitations than other common approaches, such as using only existing interfaces. It does not require the changes to have been anticipated or otherwise limited to specific sites.

  Our approach assumes that the application, or at least the affected parts of the application, can be recompiled to effect the change. Changes that must be applied to running systems are not specifically addressed.

Our approach takes a compositional approach. Some kinds of changes are difficult to apply in this way. A new requirement to increase the speed of an application by a factor of two might be an an example of a concern that could not be isolated in its own components.

- Modularity refers to the ability to isolate concerns.

  Because the role compositions are normal classes, our approach keeps the modularity provided by standard object oriented approaches. The roles add another layer of modularity. Role components can be used to isolate behavior and other concerns that are either fine grained or cut across objects. In our idioms, we show how to separate concerns or composition from algorithmic concerns. In one of our papers we even demonstrated separating the architecture from other concerns [73].

  Not all concerns can be represented as modules. However, by using groups of fine grained components, we have considerably extended the range of concerns that can be addressed in a modular fashion. We have also removed the requirement that concerns addressed by modules be ordered in a hierarchy.

- Durability refers to continuing to support new changes even after many changes have already been applied.

  In our example of applying two successive changes to the container recycling machine, we demonstrated the unique ability of our approach to accept change in a way that does not qualitatively affect the ability to accept further change. In our experience with the image display program, we found that using a development style that applies concerns as a series of small changes actually enhances the ability of the application to support further change by reinforcing fine grained modularity around concerns.

- Wholeness refers to support for change across all phases and artifacts of development.

  Our approach addresses change as a complete process from requirements through implementation. Because our approach uses the same model for requirements, design, and implementation, it also makes it easier to maintain consistency among development artifacts. Many design approaches collect lists of object responsibilities or specify the functions in object interfaces, but do not address how the different responsibilities or functions interact. The result appears as a "here's the design, now implement it" step where the developer is left to apply his or her own expertise with little guidance from the development methodology. The issues in this gap between the conceptual design and the concrete implementation create a barrier between design reuse and implementation reuse. In this thesis we laid out a process with small well defined steps all the way from requirements to compilation.

  - Proportionality refers to the relative difficulty of applying some changes in the design versus. applying the corresponding changes in the implementation.

    Again, because we use the same models in every phase, we have a high likelihood that straightforward changes in the design have straightforward implementations. We also present a method of implementation that avoids chain reaction breakdowns of the type system that could otherwise turn seemingly simple changes into major projects.

  - Traceability refers to the ability to trace the connections between identifiable requirements and identifiable components of implementation.

    Again, because our use case models of requirements directly correspond to collaboration models in the design and implementation, our approach provides unique support for traceability.

- Clarity refers to the ability to understand the design for maintenance purposes.

  Our approach augments the high level models of architectural structure and low level listings of code with an intermediate level of representation that provides a road map based on concerns. At the high level, the same models exist, and are less likely to be cluttered by extra objects addressing non-algorithmic concerns. At the low level our code is broken into smaller pieces addressing well defined concerns.

  We believe that our approach represents a potential improvement in program understandability. However, we have not yet tested this hypothesis by giving one of our programs to a neophyte programmer to change. It is a task for future work.

- Breadth refers to the range of changes that are supported.

  As we explained above, by going to finer grained components and a less hierarchical mechanism of composition, we have extended the range of changes that could be handled by code modules over other object oriented approaches.

- Depth refers to the ability to make changes at many levels of granularity.

  Depth is another area where our approach stands out in comparison to other common approaches to flexibility. By editing the specification of composition, we can replace one little component, or large components made of many smaller components. Developers can choose a large component that gets them close to their goal, and still edit or replace small pieces of that component to make it better fit their needs.

- Balance refers to what has to be sacrificed in order to gain flexibility.

  In our design of the container recycling machine, and in an earlier paper on a design of graph traversals [74], we showed that our approach not only provides more flexibility than common approaches such as patterns and frameworks, but, when compared with the earlier published designs, it yields simpler architectures, and, at least in its C++ implementation, it yields more efficient implementations.

  It is certainly easier to build an initial application without worrying about separating concerns into smaller modules. Even though in our own experience it became second nature to do so, separating concerns requires more careful thinking. We do not know how much of a difference in effort our approach requires. That is also a subject needing future study. Approaches that build in flexibility, such as our own, expect to recoup the added initial cost by reduced prototyping and maintenance costs and higher value of products that better meet customer expectations.

Our approach takes a qualitatively different view of program construction and implementation than most programmers learn and practice. As such, it opens up many opportunities for new approaches to existing problems. We will not attempt to list all of the new ideas we have found so far—this thesis is already overly long. We discussed some of the alternatives, such as our approach to data structures and the handling of control flow, in Chapter 6.

11.2 Future Directions and Open Questions

11.2.1 Implementation

While we demonstrated one way of implementing a role design, there are other methods using different mechanisms and other languages. This work started with a set of real requirements from a common real-world activity. Researchers proposing other methods of producing flexible or adaptable software can demonstrate the utility of their tools in solving the same set of problems. The appendix includes additional guidelines for those attempting to provide an implementation specifically for the development process presented here.

11.2.2 Scalability

Questions remain about the scalability of our approach both in terms of program size and in terms of numbers and abilities of developers. In half a dozen small sample programs we had little difficulty applying the approach. In the one medium sized application, the approach seemed to help in managing the complexity. But this experience is very limited. More experience with larger applications is needed to understand the strengths and limitations of the approach.

All of experience, to date, has involved the same one programmer. We know it can be understood by other programmers—Krausmuth, a professor at the University of Vienna, taught our template approach as part of a tutorial on template programming at ECOOP'97, and Smaragdakis, a student at the University of Texas at Austin, has extended and improved one of our designs and its implementation [65]. However, we don't know how accessible it is for average programmers to understand or what difficulties they might find in applying it.

Finally, we have no experience with using the method on a multi-programmer team. There are many questions about the ability of several programmers to develop or share a common design, or the ability of one programmer to use or reuse components written by another.

11.2.3 Tool support

For our approach ever to become viable, we would need tools that support it. Our experience showed that program understanding and analysis was greatly enhanced by the use of visual representations such as role matrices (e.g. Fig. 7.9), annotated matrix columns (e.g. Fig. 7.11), and overview diagrams (e.g. Fig. 7.15). We produced all of these graphical representations by hand. Tools could reduce the work of producing such diagrams. Moreover, the tools could be smart—automatically identifying syntactic dependencies within compositions, modeling control flow and comparing interfaces. Tools could be used to play what-if games on various orderings and combinations of components, to reduce the mental effort of finding workable compositions. Many of the glue components that we added to the abstract model in Fig. 7.1 to produce the executable model in Fig. 7.15 could be automatically generated by a smart set of tools. In fact, an interesting question remains as to just how much of the process could be automated (see next item).

11.2.4 Visual Programming

Roles could form the basis of an approach to visual programming. Visual programming is a way to reduce the complexity of programming by providing rich visual abstractions that can be graphically manipulated to produce meaningful programs. Our approach has been presented as text based, with the developer writing template components and lists of template instantiations. However, overview diagrams, such as the one shown in Fig. 7.15, contain all of the information needed to produce the complete set of template instantiations for an application. Many of the components are common across applications or could be generated from information contained in other components. It would be interesting to compare the strengths and weaknesses of our approach with other component based visual application builders. The depth of our approach might make it easier to customize large grained components to produce more polished applications.

11.2.5 Reuse

This work focused on the process of change, but it also has implications for reuse. Common approaches to code reuse involve looking for components in a library that match the specifications of components in the design—by comparing facets or signatures. In 1986, Brad Cox predicted that the revolution in software productivity would come from reuse based on libraries of generic components, called software IC's [20].[1] But in this model of reuse, potential savings apply only to the final implementation steps of the development process. There is also overhead attached to each reusable component, both on the front end for making components available for reuse and at the back end for finding and applying foreign code in an application.

With our approach, reusable code can be sought at the beginning of the design process, based on concerns in the requirements, and the code doesn't have to exist in a library to be found or reused. In applications developed with our style, code fragments that address specific concerns exist as separate components and are already implemented in a reusable form—role templates. Reuse could involve looking at applications that address a similar concern, even if the application is otherwise quite different. Reusable code is then found and extracted by looking at how the existing application addresses that concern. The fact that the components can be run in an existing application should make it easier to understand what they do and how to use them, especially if they have not been carefully documented for purposes of reuse.

Historically, the reuse of large scale components has posed a difficult tradeoff between specificity and generality [11]. Components that are overly customized, and thus well suited, for a particular use are hard to adapt to other requirements and unlikely to be reused. Components that are overly general are either poorly adapted to any specific use, or, if loaded with features to accommodate a variety of uses, are

---

[1] To Cox, the most compelling argument for dynamic binding was to protect the library provider's source code. "It is hard to see how a commercial marketplace in generic packages could develop as long as suppliers must trust the compiler to protect their proprietary interests in source code" [20].

likely to be large and inefficient. Our approach offers an alternative to this tradeoff. By factoring components into small, easily interchangeable pieces, we allow low level customizability in large systems and an efficient implementation.

A recently emerging model for corporate reuse is based on large scale components, precisely tailored to the customer's specification, adapted to the customer's architecture, and delivered just in time. For potential consumers, the current problem with reusing large scale components is overcoming architectural mismatch [26, 70]. The issue for the suppliers is not so much reuse as change—adapting products to satisfy new requirements in a timely manner. The approach presented in this thesis could be applied to building and maintaining modest sized components for this model of reuse.

11.2.6 Hybrid approaches

Our approach has potential to be used in combination with other component development approaches. Our approach has strength in its ability to maintain flexibility at many levels of detail and for supporting families of related applications, and it can produce an efficient implementation. It could bring these strengths to a collaboration with approaches that address other issues such as dynamic change, realtime constraints, embedded systems, and distributed objects in client-server applications.

11.2.7 Legacy code

Roles could provide a migration path from structured legacy code. Much legacy code has a functional structure rather than an object oriented one. Functions can also be decomposed into roles—the collaborations in our approach capture functions. The same roles can be composed around functions just as they can be composed into objects. An interesting line of research might try to decompose the functions of an existing application into roles—duplicating the original application, and then rearranging those roles to form an object oriented architecture for the same application.

BIBLIOGRAPHY

[1] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 372–395, 1992.

[2] 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.

[3] K. Barrett, B. Cassels, P. Haahr, D.A. Moon, K. Playford, and P.T. Withington. A monotonic superclass linearization for Dylan. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 69–82, 1996.

[4] P.G. Bassett. Frame-based software engineering. *IEEE Software*, 4(4):9–16, July 1987.

[5] P.G. Bassett. *Framing Software Reuse: Lessons from the Real World*. Prentice Hall, 1996.

[6] P.G. Bassett. The theory and practice of adaptive reuse. In *Proceedings of the 1997 ACM Symposium on Software Reusability*, pages 2–9, 1997.

[7] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *Proceedings of SIGSOFT'93: Foundations of Software Engineering*, pages 191–199. ACM Press, 1993.

[8] 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.

[9] K. Beck. Think like an object. *UNIX Review*, 9(10):39–43, 1992.

[10] 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.

[11] T. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proceedings of the Third International Conference on Software Reuse*. IEEE Computer Society Press, 1994.

[12] G. Booch. *Object Oriented Design with Applications*. Benjamin/Cummings, 1991.

[13] G. Bracha. *The programming language JIGSAW: Mixins, modularity and inheritance*. PhD thesis, University of Utah, 1992.

[14] G. Bracha and W. Cook. Mixin-based inheritance. In *Proceedings of the 1990 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 303–311, 1990.

[15] F.P. Brooks. No silver bullet; essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[16] A.M. Burkett. Clarifying roles and responsibilities. *CMA: the Management Accounting Magazine*, 69(2):26–28, March 1995.

[17] K. Chow. *Supporting Library Interface Changes in Open Systems Software Evolution*. PhD thesis, University of Washington, 1996.

[18] P. Coad and E. Yourdon. *Object Oriented Design*. Prentice-Hall, 1991.

[19] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes. *Object-Oriented Development: The Fusion Method*. Prentice-Hall, 1994.

[20] B. Cox. *Object Oriented Programming: An Evolutionary Approach*. Addison-Wesley, 1986.

[21] R. Ducournau, M. Habib, M. Huchard, and M.L. Mugnier. Proposal for a monotonic multiple inheritance linearization. In *Proceedings of the 1994 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 164–175, 1994.

[22] J. Fiadeiro, T. Sernadas, T. Maibaum, and A. Sernadas. Describing and structuring objects for conceptual schema development. In *Conceptual Modeling, Databases, and Case: An Integrated View of Information Systems Development*, pages 117–138. John Wiley and Sons, Inc., 1992.

[23] S. Freeman. Partial revelation and Modula-3. *Dr. Dobb's*, 20(10):36–42, October 1995.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *Proceedings of the 1993 European Conference on Object-Oriented Programming*, pages 406–431, 1993.

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

[26] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, 12(6):17–26, November 1995.

[27] J.A. Goguen. Principles of parameterized programming. In *Software Reusability*, pages 159–226. Addison-Wesley, 1989.

[28] G. Gottlob, M. Schrefl, and B. Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3), July 1996.

[29] 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.

[30] W. Harrison, H. Ossher, R.B. Smith, and D. 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.

[31] 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.

[32] I. Jacobson, M. Christenson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 2nd edition, 1992.

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

[34] G. Kiczales. Traces (a cut at the "make isn't generic" problem). In *Object Technologies for Advanced Software: Proceedings of the First JSSST International Symposium*, pages 27–42, 1993.

[35] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J-M. Loingtier, and J. Irwin. Aspect oriented programming. In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[36] M.M. Lehman and L.A. Belady. A model of large program development. In *Program Evolution*, pages 165–200. Academic Press, 1985.

[37] M.M. Lehman and L.A. Belady. *Program Evolution*. Academic Press, 1985.

[38] K.J. Lieberherr and A.J. Riel. Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, 1(3):8–22, August/September 1988.

[39] K.J. Lieberherr and C. 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.

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

[41] R. Malan, R. Letsinger, and D. Coleman. *Object-Oriented Development at Work: Fusion in the Real World*. Prentice-Hall, 1995.

[42] R. Martin, D. Riehle, and D. Buschman. *Pattern Languages of Program Design 3*. Addison-Wesley, 1998.

[43] A. Mendhakar, G. Kiczales, and J. Lamping. RG: A case-study for aspect oriented programming. Technical Report SLP 97-008 P9710042, Xerox Palo Alto Research Center, 1997.

[44] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall International, 1988.

[45] M. Mezini. Dynamic object evolution without name collisions. In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, pages 190–219, 1997.

[46] M. Mezini. Maintaining the consistency and behavior of class libraries during their evolution. In *Proceedings of the 1997 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1997. to appear.

[47] D. Notkin, D. Garlan, W.G. Griswold, and K.J. Sullivan. Adding implicit invocation to languages: Three approaches. In *Proceedings of the 1st JSSST International Symposium on Object Technologies for Advanced Software*, pages 489–510. Springer-Verlag, 1993.

[48] K. Nygaard and O-J. Dahl. The development of the Simula languages. In *History of Programming Languages*, pages 439–480. Academic Press, 1980.

[49] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[50] D.L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, May 1972.

[51] D.L. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, SE-2(1):1–9, March 1976.

[52] D.L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–137, March 1979.

[53] D.L. Parnas, P.C. Clements, and D.M. Weiss. The modular structure of complex systems. *IEEE Transactions on Software Engineering*, SE-11(3):259–266, March 1985.

[54] C. Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, pages 419–443, 1997.

[55] T. Reenskaug. *Working With Objects: The OOram Software Engineering Method.* Manning, 1995.

[56] T. Reenskaug and E.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.

[57] 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.

[58] D. Riehle. Describing and composing patterns using role diagrams. In H. Steffen, editor, *WOON '96, Conference Proceedings*, St. Petersburg, Russia, June 1996.

[59] D. Riehle. Composite design patterns. In *Proceedings of the 1997 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, 1997. To appear.

[60] J. Rumbaugh. Getting started: Using use cases to capture requirements. *Journal of Object-Oriented Programming*, 7(5):8–23, September 1994.

[61] J. Rumbaugh. *UML Reference Guide.* Addison-Wesley, 1998.

[62] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[63] A. Sernadas, C. Sernadas, and J.F. Costa. Object specification logic. *Journal of Logic and Computation*, 5(5):603–630, October 1995.

[64] H.B. Simon. *The Science of the Artificial*. MIT Press, 1969.

[65] Y. Smaragdakis. A cleaner way to implement collaboration-based designs. Department of Computer Science, University of Texas at Austin, 1997.

[66] R. Stata and J.V. Guttag. Modular reasoning in the presence of subclassing. In *Proceedings of the 1995 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 200–214, 1995.

[67] P. Steyaert, W. Codenie, T. DeHondt, K. DeHondt, Lucas C., and M.V. Limberghen. Nested mixin-methods in Agora. In *Proceedings of the 1992 European Conference on Object-Oriented Programming*, pages 197–219, 1993.

[68] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, second edition, 1991.

[69] K.J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[70] K.J. Sullivan, J. Socha, and M. Marchukov. Using formal methods to reason about architectural standards. In *Proceedings of the 19th International Conference on Software Engineering*, pages 503–513. IEEE Computer Society Press, 1997.

[71] J.D. Ullman. *Elements of ML Programming*. Prentice Hall, 1994.

[72] M. VanHilst. SAOimage. *Bulletin of the American Astronomical Society*, 2(22), 1990.

[73] M. VanHilst and D. Notkin. Decoupling change from design. In *Proceedings of SIGSOFT'96 Foundations of Software Engineering*, pages 58–69, 1996.

[74] 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.

[75] M. VanHilst and D. Notkin. Using role components to implement collaboration-based designs. In *Proceedings of the 1996 ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 359–369, 1996.

[76] 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.

[77] 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.

[78] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

Appendix A

A LIST OF CRITERIA FOR ROLE IMPLEMENTATION

What properties of a role oriented design should extend into the implementation to maintain the ability to support change? Below, we enumerate a set of criteria for a role oriented implementation. They are not intended to be exhaustive or indicative of any particular method of implementation, but merely to provide points of discussion for others following in this path.

1. Objects: The implementation must support an object based design model.

   Role oriented design is an extension of object oriented design. At some level, role oriented designs are object based, if not object oriented. Thus the implementation must meet the same expectations as those of other object oriented methodologies.

2. Roles: The implementation must support units of encapsulation that correspond to roles.

   For roles to exist in the implementation, there must be a way to implement roles as components of the implementation. Since roles are composable parts of objects, role components must be composable parts of object implementations or classes.

   (a) Role composition: Since objects in the design are formed by composing roles, it must be possible to form object implementations by composing role components.

(b) Role reuse: Since roles are reusable in design, role components must be reusable in implementation. Roles should have an implementation that is separate from that of the object compositions in which they are used.

Raymie Stata described a way of grouping parts of classes to facilitate modular reasoning [66]. Smalltalk's Envy environment provides a similar grouping mechanism called class categories. In both cases the class groups have no meaning outside of the class where they are defined.

(c) Role state: Since roles in the design can specify both state and behavior, role components must be able to include both state variables and behavioral functions or methods.

The Smalltalk Envy version control mechanism called categories is sometimes used to support multiple views. The developer can specify which categories are included in each version. The class subcomponents in an Envy category can include methods, but no state variables. Thus Envy categories could not be used to implement roles in a role oriented design.

3. Recomposability: Roles can be recomposed in different combinations to form objects in the design. In other words, new objects can be created from existing pieces and used by other objects in the design.

In traditional object oriented programming, classes can theoretically be reused in different compositions in different applications, or different parts of the same application. But the implementation imposes significant limitations. The main problem is that the source code for each class typically encodes the type names or, worse, the class names of other objects in the application's structure to which the class refers, or which it includes as parts. Classes bound in such fixed relationships cannot be switched for other classes, or classes that do not have the named type, without causing the unzipping problem described earlier.

(a) Signature translation: Two roles developed in different contexts may use different names and signatures for the same functions. The implementation should allow translations to be applied to establish the intended bindings.

(b) Disjoint naming: Separate roles may use the same name to access different variables or functions. Confounding of names occurs when different roles are forced to use the same name space.

Two roles developed in different contexts may inadvertently use the same name for different things. The implementation should allow each role to make its own associations without being confounded by the use of names in other roles.

A single role may also be included twice in the same object, either because the same role occurs in different use cases, or because the object participates in different instantiations of the same use case collaboration. The latter might occur when an employee divides his or her time between two projects, or an object appears as a node in the orderings of more than one list data structure. In such cases, the implementation must be able to duplicate state rather than merging it.

4. Decomposability: Within an object, roles can be decomposed into two or more smaller roles, where each new role addresses a narrower part of the original concern.

   Decomposition is used, for example, where roles overlap—the shared part is separated out to be instantiated only once. The implementation can require that a concern is met, but it should not assume that the functions that address that concern are always provided by a single role or component. If different functions are accessed through a single handle or interface, they should continue being accessible through that handle or interface even after being separated in different roles.

5. Extensibility: Adding a new role to an existing object must be able to add new behavior to that object and also to change existing behavior.

    (a) Role augmentation: New roles must be able to add new functions to the interface of the existing object and access new functions in objects with which their own object already collaborates.

        In the example of adding the Validate Item use case to the container recycling machine, a new isValid function was defined for the DepositItem objects and called from a new addItem function in the DepositReceiver object.

    (b) Role access: New roles must be able to access functions and attributes in existing roles of the object.

        In the container recycling machine example, roles from the PrintReceipt use case access state variables belonging to roles from the Add Item use case and may also call functions defined in the Add Item roles.

    (c) Role overriding: New roles must also be able to override existing functions in calls from other objects. Overriding is an important part of the relationship between extension use cases and the use cases they extend.

    (d) Role insertion: A new role must be able to add a new step between existing steps of an object's behavior.

        Within the recycling machine's DepositReceiver object, the Stuck Item use case's role added an additional step between the Validate Item and Add Item steps of the addItem behavior. Taking an example from another domain, to add an annotation graphic to an image display object, the graphic must be inserted after the image buffer is updated and before it is sent to the display.

If control flow within an object is handled by the roles themselves, then adding a step implies overriding an existing function in a call between two roles, and allowing the new function to access the overridden function to resume the remaining steps.

6. Ceteris paribus: Compared to other methods of implementation, the role implementation should not impose a substantial penalty in development effort, design complexity, or runtime speed.

Appendix B

DISPLAY APPLICATION INSTANTIATION LISTS AND MAIN ROUTINE

```
// default empty base class
class empty {};
// shell class for display connection and initialization
class Shell01    : public XShellRole                          <empty> {};
class Shell02    : public XVisualRole                       <Shell01> {};
class Shell03    : public XColorRole                        <Shell02> {};
class Shell04    : public HardColorRole                     <Shell03> {};
class Shell05    : public HardColorCopyRole                 <Shell04> {};
class Shell06    : public PseudoColorGammaRole              <Shell05> {};
class Shell07    : public ColorPreferenceRole               <Shell06> {};
class Shell08    : public XGCRole                           <Shell07> {};
class Shell09    : public XWindowRole             <XFormClass,Shell08> {};
class Shell10    : public PseudocolorPResizeRole        <-1,Shell09> {};
class Shell11    : public ResizeEventRole                   <Shell10> {};
class ShellClass : public XActionRole                       <Shell11> {};
typedef ShellConstruct<ShellClass> ShellCClass;

// basic part of widget/window for all canvas widgets
class Canvas01   : public XWindowRole               <XFormClass,empty> {};
class Canvas02   : public ProxyHandleRole  <ShellClass,Canvas01> {};
class Canvas03   : public XVisualProxytRole                <Canvas02> {};
class Canvas04   : public PseudoColorProxyRole             <Canvas03> {};
class Canvas05   : public XGCProxyRole                     <Canvas04> {};
class Canvas06   : public BufferRole                  <char,Canvas05> {};
class Canvas07   : public BufferPResizeRole              <1,Canvas06> {};
class Canvas08   : public ResizePAnnounceRole          <1,4,Canvas07> {};
class Canvas09   : public MouseAnnounceRole              <4,Canvas08> {};
class Canvas10   : public MouseEventRole                   <Canvas09> {};
class Canvas11   : public KeyPressAnnounceRole           <4,Canvas10> {};
class CanvasClass : public KeyPressEventRole               <Canvas11> {};
```

```
// special Viewport widget class that draws colorbar
class Color1    : public ColorbarRole         <char,CanvasClass> {};
class Color2    : public ColorbarPResizeRole          <1,Color1> {};
class Color3    : public XImageRole                     <Color2> {};
class Color4    : public ImagePResizeRole             <1,Color3> {};
class ColorbarClass : public ResizeEventRole            <Color4> {};
typedef ViewportConstruct<Canvas02,ShellClass,
                          ColorbarClass> ColorbarCClass;

// general image display Viewport widget class
class Viewport1 : public XImageRole                <CanvasClass> {};
class Viewport2 : public ImagePResizeRole          <1,Viewport1> {};
class ViewportClass : public ResizeEventRole         <Viewport2> {};
typedef ViewportConstruct<Canvas02,ShellClass,
                          ViewportClass> ViewportCClass;

// forward declare Main renderer class
class MRendClass;
// class for holding image data
class ImgBuf1 : public BufferRole                  <short,empty> {};
class ImgBuf2 : public ProxyHandleRole      <MRendClass,ImgBuf1> {};
class ImgBufClass  : public ScaleMapProxyRole          <ImgBuf2> {};
typedef OneHandleConstruct<ImgBuf2,MRendClass,
                           ImgBufClass> ImgBufCClass;

// render class for transforming data from image to window
class Rend01  : public Transform2DRole                   <empty> {};
class Rend02  : public ProxyHandleRole      <ImgBufClass,Rend01> {};
class Rend03  : public SrcBufferProxyZoneRole     <short,Rend02> {};
class Rend04  : public ScaleMapProxyRole                <Rend03> {};
class Rend05  : public ProxyHandleRole    <ViewportClass,Rend04> {};
class Rend06  : public DstBufferProxyZoneRole      <char,Rend05> {};
class Rend07  : public BufferZoneResizeRole             <Rend06> {};
class Rend08  : public XImageProxyRole                  <Rend07> {};
class Rend09  : public MouseAnnounceRole              <4,Rend08> {};
class Rend10  : public MouseTransformRole               <Rend09> {};
class Rend11  : public MouseListenProxyRole             <Rend10> {};
class Rend12  : public KeyPressAnnounceRole           <4,Rend11> {};
class Rend13  : public KeyPressListenProxyRole          <Rend12> {};
class Rend14  : public RenderRole                       <Rend13> {};
class Rend15  : public RenderPResizeRole              <1,Rend14> {};
```

```
class Rend16  : public ResizePAnnounceRole         <1,4,Rend15> {};
class RendClass : public ResizeListenProxyRole          <Rend16> {};
typedef TwoHandleConstruct<Rend02,ImgBufClass,
                           Rend05,ViewportClass,
                           RendClass> RendCClass;

// special renderer for main display
class MRend1Class  : public PseudoColorProxyRole        <Rend13> {};
class MRend2Class  : public ScaleMapRole                <MRend1> {};
class MRend3Class  : public RenderRole                  <MRend2> {};
class MRend4Class  : public RenderPResizeRole         <1,MRend3> {};
class MRendClass   : public ResizeListenProxyRole       <MRend4> {};
typedef TwoHandleConstruct<Rend02,ImgBufClass,
                           Rend05,ViewportClass,
                           MRendClass> MRendCClass;

class Magni01 : public ProxyHandleRole         <RendClass,empty> {};
class Magni02 : public RenderProxyRole                <Magni01> {};
class Magni03 : public MouseMagnifyRole               <Magni02> {};
class Magni04 : public ResizeListenProxyRole          <Magni03> {};
class Magni05 : public KeyPressListenProxyRole        <Magni04> {};
class Magni06 : public ProxyHandleRole      <VRendClass,Magni05> {};
class Magni07 : public MouseListenProxyRole           <Magni06> {};
class Magni08 : public ProxyHandleRole       <RendClass,Magni07> {};
class MagniClass : public MousePListenProxyRole     <-1,Magni08> {};
typedef ThreeHandleConstruct<Magni01,RendClass,
                             Magni06,MRendClass,
                             Magni08,RendClass,
                             MagniClass> MagniCClass;

// class for reading image data
class File1   : public ProxyHandle1Role       <ImgBufClass,empty> {};
class File2   : public BufferProxyRole              <short,File1> {};
class FileClass : public FileReadRole   <ReadArray<short>,File2> {};
typedef FileConstructorRole<ImgBufClass,FileClass> FileCClass;

int main(int argc, char* argv[]) {
  ShellCClass form("windowtest","form",argc,argv);
  ViewportCClass magnifier("magnifier",&form);
  ViewportCClass navigator("navigator",&form);
  ViewportCClass viewport("viewport",&form);
```

```
  ColorbarCClass colorbar("colorbar",&form);
  ImgBufCClass imbuf(NULL);
  FileCClass readfile(argv[1],&imbuf);
  MRendCClass mainrender(&imbuf,&viewport);
  mainrender.setHistogram(2);
  mainrender.setScaleMap();
  imbuf.initializeHandle1(&mainrender);
  RendCClass navirender(&imbuf,&navigator);
  RendCClass magrender(&imbuf,&magnifier);
  MagniCClass magcontrol(&magrender,&mainrender,&navirender);
  form.realize();
  form.sync();
  form.start();
  return 0;
}
```

VITA

Michael VanHilst was born in Amsterdam, the Netherlands, in 1953. He graduated from Washburn Senior High School in Minneapolis, Minnesota, in 1971. From 1971 to 1978, he attended the Massachusetts Institute of Technology, earning a Bachelor of Arts degree in Art and Design from the Department of Architecture, and Bachelor of Urban Studies and Master of City Planning degrees from the Department of Urban Studies. In 1979 and 1980 he worked as the Director of Community Development for the city of Grinnell, Iowa. From 1981 until 1990 he worked as a hardware specialist and computer programmer for the Smithsonian Astrophysical Observatory, in Cambridge, Massachusetts. In 1985 he took a year off to work as a research associate at the University of Paris, at Jussieu. In 1990 he worked as a contractor at IBM's T.J. Watson Research Center. He received a Master's degree in Computer Science from the University of Washington in 1993, and the Ph.D. in 1997.