Role Model Designs and Implementations with Aspect-oriented Programming

Elizabeth A. Kendall

Departments of Computer Science and Computer Systems Engineering

Royal Melbourne Institute of Technology, GPO Box 2476V, Melbourne, VIC 3001, AUSTRALIA

email: kendall@rmit.edu.au

ABSTRACT AND OUTLINE

This paper describes research in applications of aspect-oriented programming (AOP) as captured in the AspectJ^{TM} language. In particular, it compares object-oriented and aspect-oriented designs and implementations of role models.

Sections 1, 2, and 3 provide background information on role models, object-oriented role model implementations, and aspect-oriented programming, respectively. New aspect-oriented designs for role models are explored in sections 4, 5, and 6.

The base reference for this exploration is the Role Object pattern. Although useful for role models, this pattern introduces some problems at the implementation level, namely object schizophrenia, significant interface maintenance, and no support for role composition. Our research has resulted in alternative aspect-oriented designs that alleviate some of these problems.

Section 7 discusses how an agent framework that implements role models has been partially reengineered with aspects. The reengineering addressed concerns that are orthogonal or cross cut both the core and the role behavior. The aspect oriented redesign significantly reduced code tangling, overall method and module count, and total lines of code. These results and other conclusions are presented in section 8.

Keywords

Aspect-oriented Programming, Role Modelling

1. ROLES AND ROLE MODELS

1.1 Background

Roles and role models [1, 8, 16- 19, 23- 24, 25- 27] are abstraction and decomposition mechanisms. Classes stipulate the capabilities of individual objects, while the notion of a role focuses on the position and responsibilities of an element within an overall system or subsystem. A role model identifies an archetypal structure of elements (objects) and describes it as a corresponding and reoccurring structure of roles. Role models capture how objects interact with each other in collaborations; role models have been proven to be useful during conceptualization, analysis, and design.

1.2 Example

A sample role model is provided in [26, 27] for the Bureaucracy pattern. This pattern is often found in software systems [27], but it also captures the structure of human bureaucracies. In a bureaucracy, there is a long chain of responsibility, a multilevel hierarchical organization, and centralized control. The Bureaucracy role model features five roles: Director, Manager, Subordinate, Clerk, and Client. A Director manages the entire organization. Managers report to the Director, supervising the activities of their Subordinates. Due to the multiple levels in the Bureaucracy, intermediate level Managers have other, lower level, Managers reporting to them as Subordinates. The lowest level role in the Bureaucracy is a Clerk; these entities perform the actual work or service for a Client.

The role diagram is provided in Figure 1, with notation that extends [1] and [26]. A rounded box represents a role, and an arrow depicts a collaboration path between two roles. Role specialization is indicated by a triangle, and a filled circle means that more than one entity can play a given role at the same time. As can be seen in Figure 1, Manager, Subordinate, and Director all refine the Clerk role. This means that even a Director must be able to act as a Clerk for certain Clients.

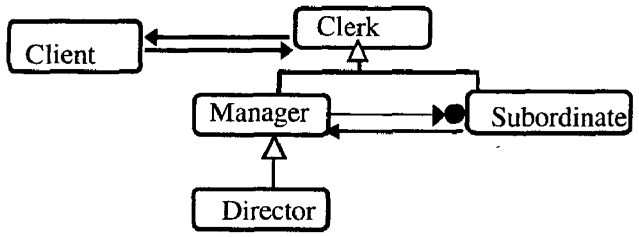


Figure 1: Role Diagram for the Bureaucracy Pattern [26]

During design, the roles in a role model are assigned to objects in an application; an object plays or carries out the roles that are assigned to it. In a given Bureaucracy, the same object may play many roles, such as Manager, Subordinate, and Clerk. An employee in a Bureaucracy can be a Manager to their own underlings. However, that same employee can be a Subordinate to their immediate supervisor, and a Clerk to an outside customer. The roles played by an object are determined by other objects' perspectives or views.

1.3 Roles as Perspectives

B. Kristensen [16 - 19] provides a conceptual model of an object and its roles. The object to which a role is allocated is the intrinsic object; it has intrinsic members (data and methods). Roles add extrinsic members (data and methods), and they provide perspectives that can be used by other objects as a selective way of knowing and accessing the object. In terms of the Bureaucracy role model in section 1.2, every employee has some core (intrinsic) behavior. However, depending on the perspective (a supervisor, an underling, or an outside customer), each employee can also exhibit extrinsic, non-core behavior.

These concepts are depicted in Figure 2, which uses the notation found in [16]. In Figure 2, Object2 (boss) knows Object1 (worker) according to RoleA (Subordinate). From the view of Object2, Object1 has three intrinsic members and three extrinsic members. In Figure 2, the perspective of Object3 (customer) composes RoleA and RoleB (Provider). That is, from the view of Object3, Object1 is playing both roles A and B, simultaneously. This means that Object3 views Object1 to have three intrinsic members and five extrinsic members. (The Provider role and role composition are discussed further in section 7.)

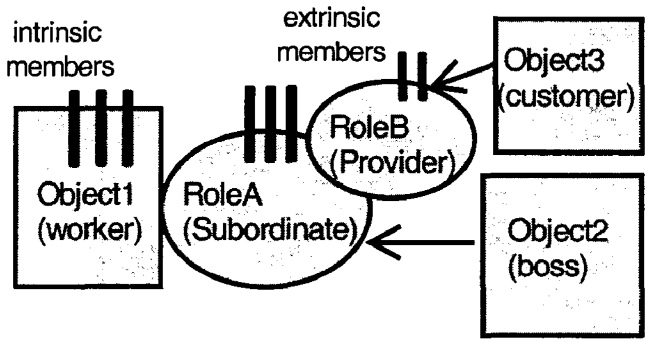


Figure 2: An Object and its Roles: Intrinsic and Extrinsic Members [16]

1.4 Role Properties

In addition to the conceptual model depicted in Figure 2, Kristensen [16] provides properties of roles. These properties are provided in the following list and form the basis of our research into role model design and implementation with object-oriented and aspect-oriented techniques.

- ***Abstractivity:*** roles can be organized in hierarchies
- ***Aggregation/ Composition:*** roles can be composed of other roles, with varying visibility
- ***Dependency:*** a role can not exist without the object. According to [16], the methods of the role can be defined in terms of the methods of the object, but not vice versa.
- ***Dynamicity:*** a role can be added or removed during the lifetime of an object. This occurs at the instance level; different instances of the same class can have roles added or removed during their lifetime.
- ***Identity:*** the role and the object have the same identity. The object and its roles are seen and can be manipulated as one entity.
- ***Inheritance:*** a role for a class is also a role for any subclass, and a super-role is a role for a class if its sub-role is a role for the class. However, an alternative criteria [1, 8] states that a role should be able to be assigned to any class.
- ***Locality:*** a role only has meaning in a role model
- ***Multiplicity:*** several instances of a role may exist for a given object at one time. An object may play several roles at once, including multiple instances of the same role.
- ***Visibility:*** access to the object is restricted by a role. The visibility of an object can be restricted to the methods of a role. This may include the intrinsic methods of the object, but it will exclude the extrinsic methods of other roles.

2. OBJECT-ORIENTED DESIGNS FOR ROLE MODELS

2.1 The Role Object Pattern

Several approaches have been used for implementing roles in object-oriented languages [2, 6, 8, 17, 28]. In [6], M. Fowler evaluates the various approaches; the most common is the Role Object pattern [2]. This pattern provides an individual class for every role. The roles are organized in a hierarchy, with subclasses for more specialized role behavior. A core object (an instance of Core class) contains the roles that it plays as a set of role instances; roles do not exist on their own. Dynamic role assignment is supported because the instances that represent the current roles can be changed at runtime.

The Role Object design for the Bureaucracy pattern is provided in Figure 3. The Role Object pattern stipulates that a class be provided for every role, but there are three major options for the design of the interface to the role: State pattern [7], Role Object pattern per M. Fowler [6], and the Decorator pattern [7]. The Decorator pattern version is in Figure 3; both Role and AgentCore implement the same interface. An object using an instance of AgentCore only has knowledge of one object; however, at runtime, the roles transparently add behavior. In the figure, the Director role has been omitted; it is a further subclass of Manager.

2.2 Problems with the Object-oriented Design

The Role Object with the Decorator pattern is proposed by Kristensen and Osterbye [17] as the best support for role models in standard object-oriented languages. However, they and other authors [6], [11] point out the following major drawbacks:

- ***Object schizophrenia:*** The agent's behavior is distributed over the AgentCore and its roles. The agent is intended to be one object; but, instead, it is comprised of multiple objects, each with its own identity. This violates the identity property in section 1.4, and it can lead to many symptoms, including broken delegation, broken assumptions, and dopplegangers [11].
- ***Interface bloat or, alternatively, downcasting:*** The interface for all roles must be provided in AgentInterface. If this is not done, objects must be downcast at runtime to invoke role specific behavior.
- ***Role composition:*** The Decorator version of the Role Object pattern does not support references to different, but overlapping subsets of decorators. That is, it does not support the role property of aggregation/ composition (section 1.4).

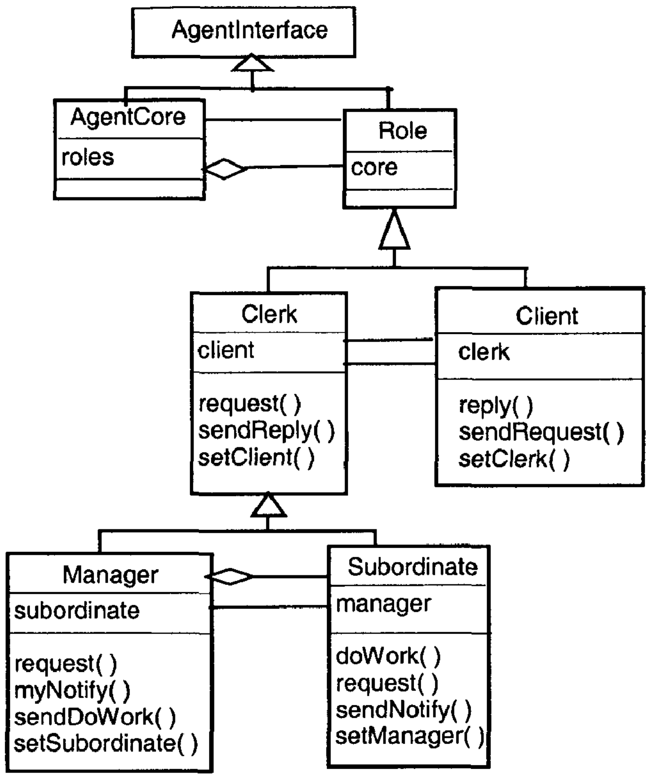


Figure 3: Role Object Design for Bureaucracy Pattern

2.3 Related Designs and Implementations

Gottlob et al [8] present a variation on the Role Object pattern that is available in Smalltalk. As Smalltalk is a dynamically typed language, a role hierarchy can be developed that is independent of the core class' hierarchy. This approach avoids interface bloat and downcasting. The Smalltalk design also allows an object to play any role, and it supports role dynamicity.

VanHilst and Notkin [28] implement and compose roles with templates in C++. However, templates compose roles at compile time; this approach is valid in applications where roles are not dynamically changing at runtime.

3. EXTRA LANGUAGE FEATURES

As stated in section 2.2, object-oriented designs do not adequately support role models. This has led us to consider extra language features, such as aspect-oriented programming and subject-oriented programming.

3.1 Aspect-oriented Programming in AspectJ

Aspects cut across or *cross-cut* the units of a system's functional decomposition (objects). Examples provided in the literature are synchronization, exception handling, monitoring and auditing, quality of service, and many others. Research presented in [14, 15] has produced extra language constructs and language processors (called Aspect Weavers) that can interleave or *weave* component and aspect definitions (programs) appropriately to formulate a unified and executable program. The Java^{TM} based AOP language AspectJ (version 0.2) from Xerox PARC has been used in this research.

In AspectJ, each file of Java source code can contain a class or an aspect. During the first phase of compilation, aspects are woven into the class definitions that they cross-cut. When an aspect is woven into a class, it either *introduces* behavior in the form of new methods, or it adds or *advises* behavior into the signature of a method that already exists. Advise weaves (also called advise cross-cuts) alter the members found in a class by adding cross-cutting code that runs *before* or *after* existing methods and constructors. *Catch* and *finally* constructs are also supported.

In AspectJ, aspects can be static and impact all instances of a given class; alternatively, aspect instances can be used to dynamically advise behavior to a given instance of a class. Introduce weaves are always static, but advise weaves can be static or can be applied at an instance level. Aspects can have their own members (data and methods), and aspects can be abstracted and specialized.

Figure 4 depicts some of the notation and capabilities of aspects. In Figure 4, a static aspect (shown with brackets) introduces new members (methods or data) to a class. An aspect instance (shown with a diamond) advises or modifies members that already exist in an instance. Therefore, in Figure 4, the class has two of its own (intrinsic) members. (As in Figure 2, these are shown with vertical bars.) A new member (extrinsic) is introduced by the static aspect. The instance has three members (two intrinsic and one extrinsic), and one of these is advised or modified by the aspect instance. The advise weave is shown with a white, rounded bar, and the modification is indicated schematically by shadowing the relevant member.

Figure 5 translates Figure 4 into AspectJ code. Class Agent has an intrinsic data member and an intrinsic method Aspect Client introduces a new extrinsic member to class Agent. Aspect Client also has an advise weave; this can enhance the definition of the intrinsic and/or the extrinsic members of class Agent. In Figure 5, the advise weave adds to the definition of the extrinsic member `Agent.send()`.

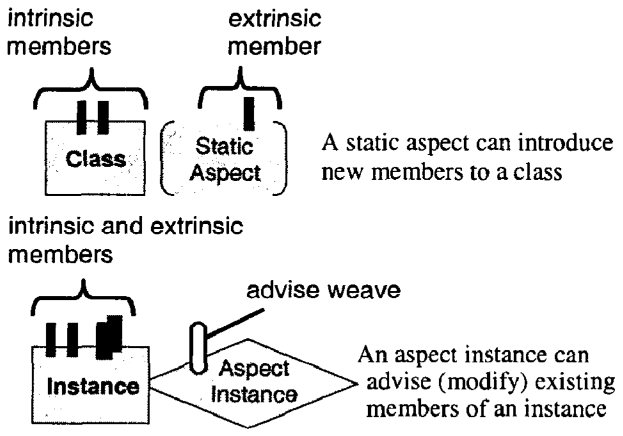

intrinsic members
extrinsic member
Class
Static Aspect
A static aspect can introduce new members to a class
intrinsic and extrinsic members
advise weave
Instance
Aspect Instance
An aspect instance can advise (modify) existing members of an instance

Figure 4: Notation and Capabilities of Aspects

The code at the bottom of Figure 5 is required to instantiate the Agent class and the Client aspect. The Shopper instance is attached to the Client aspect through the `addObject()` method; after this step, Shopper has the extrinsic behavior implementation from the advise weave.

```
class Agent
{  // intrinsic members of the class
   protected String name;
   public String getName() {return name;}
   Agent(String n){name = n;}}
```

```
aspect Client
{  // introduce extrinsic member to Agent
   introduce public void Agent.send ()
   { // do nothing  }
   // advise weaves can impact intrinsic
   // or extrinsic members
   advise public void Agent.send ()
   {  before
      {  System.out.println("sending");}}}
```

```
// Java code to instantiate Agent and
// Client and to attach Shopper to the
//  aspect
Agent Shopper = new Agent("Shopper");
// Shopper does nothing
Shopper.send();
Client clientAspect = new Client();
clientAspect.addObject(Shopper);
// Shopper prints out the words sending
Shopper.send();
```

Figure 5: AspectJ Implementation of Figure 4

3.2 Role Models and Cross-cutting

Kristensen and Osterbye [17] originally discounted AOP for role model implementation. However, the view of role models provided in [1] and in Figure 6 demonstrates that role models are a form of cross-cutting. Figure 6 has five objects, and they are involved in three activities or role models: [A],[B], and [C]. If the objects are instances of different classes (a role property in section 1.4), the behavior required to carry out the activities *cross-cuts* method definitions in five separate classes.

For example, the five objects can be involved in three different Bureaucracies (three different Bureaucracy role models, [A], [B], and [C]). (Alternatively, they can appear in one Bureaucracy [A], one Supply Chain [B] (section 7), and one Auction [C] (section 8).) Object 1 can be a Manager in Bureaucracy [A], but a Subordinate in [B], and a Client in [C]. Object 2 can be a Subordinate in Bureaucracy [A], but a Client in [B], and a Manager in [C]. Additionally, Objects 3 through 5 can play various roles in the three different role models.

Therefore, roles can dynamically cross-cut several objects. From this argument, it is obvious that both AOP and role models deal with cross-cutting behavior.

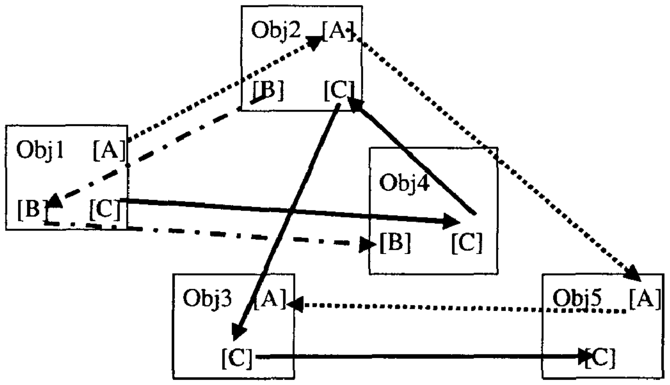


Figure 6: Role Models Cross-cut Object Models [1]

3.3 Subject-oriented Programming

We considered subject-oriented programming (SOP) as another option for role model designs. In fact, Kristensen's description of roles [16, 17, 19] is closely related to Harrison and Ossher's subjects [9]. A subject is defined as a collection of classes or class fragments that model a domain in a unique, subjective way; subjects address the fact that different entities view the same class from different perspectives. These perspectives are not filtered views; some properties and behavior only exist because of the perspective.

Subject-oriented programming supports many of the key requirements of role model implementation (section 1.4); in particular, it provides excellent support for role composition. However, we decided to employ aspect-oriented programming in this research for the following reasons:

- In this research, we use many of the SOP capabilities found in AOP and also attempt to determine if AOP should be expanded to incorporate more of SOP.
- At this point in time, SOP does not easily support role dynamicity (dynamic role assignment at the instance level), but AOP does.

- A Java based SOP environment is not currently available, while a Java based aspect-oriented programming (AOP) environment (AspectJ) is.

4. OPTIONS FOR ASPECT-ORIENTED DESIGNS

The role concepts and properties of sections 1.3 and 1.4, and the discussion in section 3.1, provide a foundation for our aspect-oriented designs and implementations. First we need to determine what aspect-oriented designs are appropriate for representing roles. Five options have been considered, as depicted in Figure 7.

The notation in Figure 7 is based on that found in Figure 4. A vertical bar represents a member (data or method). Intrinsic members are part of the core (class) behavior, while extrinsic members belong to a role (aspect). However, to simplify the diagram, Figure 7 does not depict intrinsic members. Advise weaves are shown by white, rounded bars. The advise weaves shown in the figure impact members found in the Core Instance; this is indicated by shadowing the appropriate member.

All five of the options depicted have been investigated, and no one solution is complete.

- *Option 1* places behavior at a class, rather than at an instance, level. This would mean that all instances of a given class play the same roles, violating role dynamicity (section 1.4).

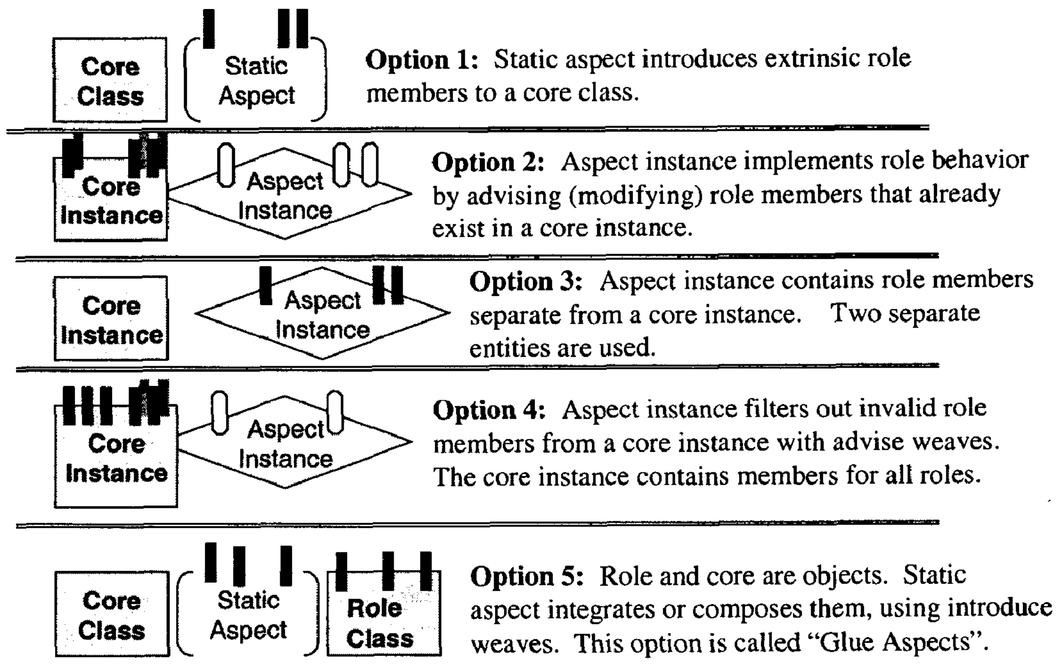
- *Option 2* requires that the Core class' interface supports the extrinsic behavior for *all* of the roles that an instance might play. This is because only existing members can be modified. Figure 7 depicts three extrinsic members in the Core Instance for Option 2, but in fact there would be many more.

- *Option 3* places the role behavior in an entity that is separate to the object, and this is not desirable because it violates role identity (section 1.4) and leads to object schizophrenia.

- *Option 4* requires that the Core class provides the interface and the implementation for the extrinsic behavior in all of the roles that an instance might play. This option should be revisited when roles are very similar, with only slight differences between them. As in Option 2, many members would actually be required in the class, and only five are depicted in Figure 7.

- *Option 5 (Glue Aspects)* represents the most extensible approach. However, it requires three levels of components (core objects, roles, and aspects), and it becomes complex when there are many dependencies between core objects and roles.

A hybrid approach to role aspect design is discussed in sections 5.1 and 5.2, and section 5.3 details Glue Aspects. Often an object plays more than one role at a time. These roles may be independent (role multiplicity), or they may be aggregated (role composition). Role multiplicity can be implemented by indexing roles by the context in which they appear (section 5.2). Role composition is more complicated and is discussed in section 6.



Option 1: Static aspect introduces extrinsic role members to a core class.

Option 2: Aspect instance implements role behavior by advising (modifying) role members that already exist in a core instance.

Option 3: Aspect instance contains role members separate from a core instance. Two separate entities are used.

Option 4: Aspect instance filters out invalid role members from a core instance with advise weaves. The core instance contains members for all roles.

Option 5: Role and core are objects. Static aspect integrates or composes them, using introduce weaves. This option is called "Glue Aspects".

Figure 7: Options for Aspect-oriented Design of Role Models (refer to Fig. 2 and 4)

5. ASPECT-ORIENTED DESIGNS FOR ROLE MODELS

5.1 Hybrid Approach

A hybrid approach was suggested by G. Kiczales. Role behavior is placed in a combination of introduce weaves (option 1) that are added to the core class and advise weaves (option 2) that are added to the core instances. That is, a static aspect introduces the interface for the role specific behavior to the Core class, and then an aspect instance adds or advises the implementation of that behavior to instances of the Core class, dynamically and as needed. Further, role relationships and role context reside in the aspect instance (option 3) to easily support role multiplicity.

The hybrid approach is detailed further in Figures 8, 9 and 10; the notation and the concepts are based on those found in Figures 2, 4, 5, and 7.

Figure 8 demonstrates that, with the hybrid approach, all role specific (extrinsic) behavior is localized in the aspect source code. In this way, a role aspect can be used to effectively separate each role concern.

Figure 9 illustrates how a static aspect introduces the interface for the extrinsic behavior to the class during the first phase (the weaving phase) of compilation. After compilation, all instances of the Core class will recognize the interface, but exception handling can be placed in the introduce weave to address invalid messages.

The implementation for the role specific behavior is not found in the class; it resides in the advise weaves. As discussed in section 4.1 and depicted in Figures 4 and 5, these advise weaves are only activated when an aspect instance is created and when an instance of the Core class is attached to the aspect.

Figure 10 shows what happens at run time. CoreInst, which is an instance of the Core class, is attached to AspectA, an aspect instance. At this point, the advise weaves give CoreInst the implementation for the extrinsic and role specific behavior. As in Figure 4, this is shown by shadowing the relevant members. At runtime, only the role relationships and context remain in the aspect instance.

To support role dynamicity, CoreInst can be subsequently removed from AspectA and assigned to another aspect instance, AspectB. (Static AspectB must provide the appropriate interface for role B during compilation.)

This approach to role aspects provides excellent support for all of the role properties of section 1.4. The role aspect restricts the visibility of the object, but yet the role is dependent on the object for its existence. Role locality, composition, and multiplicity can also be easily addressed (see sections 5.2 and 6).

This approach also has the following benefits over the Role Object pattern:

- *Interface maintenance:* The class' own intrinsic interface is not bloated with every potential role. However, the extrinsic behavior is also accessible without downcasting.
- *Object schizophrenia:* Most of the role specific behavior resides in the object; only role relationships and role context reside in the aspect.

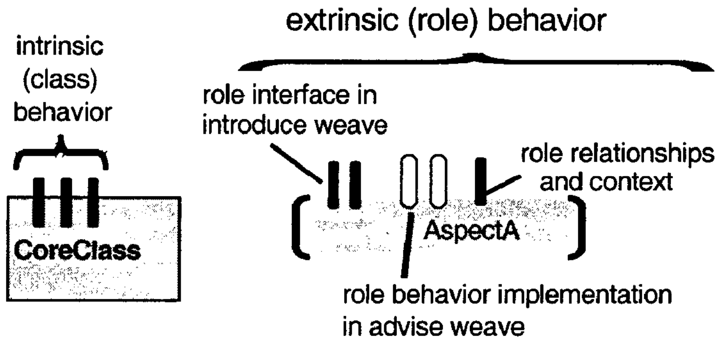

extrinsic (role) behavior
intrinsic (class) behavior
role interface in introduce weave
role relationships and context
CoreClass
AspectA
role behavior implementation in advise weave

Figure 8: Intrinsic and Extrinsic Members in Source Code

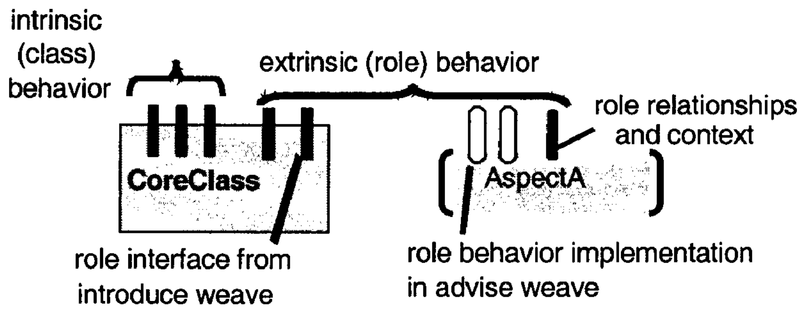

intrinsic (class) behavior
extrinsic (role) behavior
role relationships and context
CoreClass
AspectA
role interface from introduce weave
role behavior implementation in advise weave

Figure 9: Intrinsic and Extrinsic Members after Compilation - Hybrid Approach

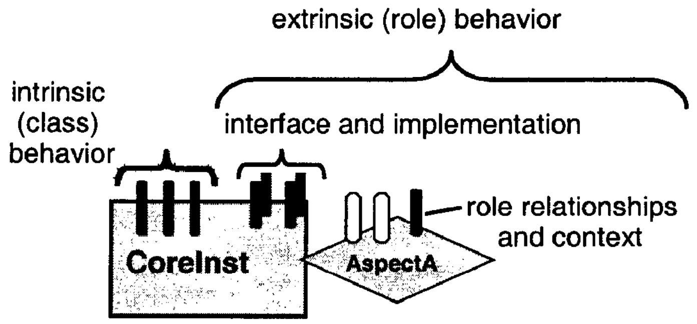

extrinsic (role) behavior
intrinsic (class) behavior
interface and implementation
role relationships and context
CoreInst
AspectA

Figure 10: Intrinsic and Extrinsic Members at Runtime - Hybrid Approach

5.2 AspectJ Implementation of the Hybrid Approach

The example is an aspect-oriented implementation for the Bureaucracy role model. The role classes and methods in the Role Object implementation (Figure 3) are mapped to corresponding aspects and weaves, as depicted in Figures 11 and 12. The AgentInterface class is no longer needed, and the AgentCore class (renamed Agent) only holds intrinsic behavior because the interface and the extrinsic behavior will be built up incrementally with the aspects.

In Figure 11, each aspect holds the introduce and advise weaves that are appropriate to the given role. For example, the Client aspect has the interface (introduce weave) and the implementation (advise weave) for making a request (method `sendRequest()`in Figure 3) and receiving a reply (method `reply()`). The Clerk holds the complementary behavior; it responds to a request and sends a reply.

The Client aspect holds the clerk role relationship, while the Clerk aspect holds the client role relationship. The aspects are specialized according to the inheritance relationships in Figure 3. The Manager and Subordinate aspects add behavior to the Clerk role; they also override or redefine methods. In particular, a Manager redefines `request()` so that it can send work to a subordinate. A Subordinate, in turn, redefines `reply()` so it can notify its manager of progress.

Sample code is provided in Figure 12 in four parts. In many ways, it represents an expanded version of the short example given in Figure 4.

Figure 12, Part A encodes the aspect Client that introduces and advises two extrinsic capabilities to class Agent: `sendRequest()` and `reply()`. The interface is in the introduce weaves; the role specific behavior implementation is in the advise weaves.

Figure 12, Part B shows that the aspect Clerk introduces and advises two complementary capabilities: `request()` and `sendReply()`.

Manager (Figure 12, Part C) extends Clerk, so it already has the introduce and advise weaves from Part B. It also introduces and adds its own extrinsic "managerial" behavior that delegates a request to a subordinate (new definition of `request()`), and receives notification of progress.

In Parts A through C, role relationships are stored in the aspects themselves. This means that `getPlayer()` messages are required to access the agents from the roles.

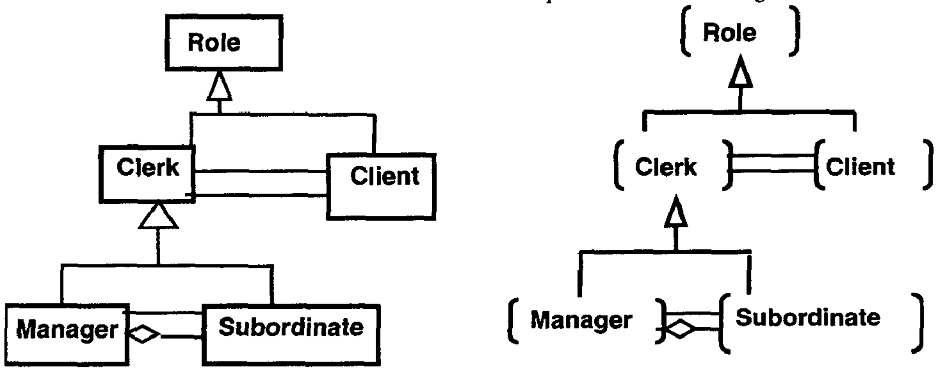


Figure 11: Bureaucracy Role Aspects in Hybrid Approach

FIGURE 12 Part A. Role Aspect Client

```
aspect Client extends Role
{  protected Clerk server; // role relationships in aspect
   public void setClerk(Clerk a){server = a; }
   // introduce default (empty) behavior to class Agent
   introduce public void Agent.sendRequest()  {}
   introduce public void Agent.reply()  {}
   // advise weaves for aspect instances that will be attached to
   // an instance of class Agent
   advise public void Agent.sendRequest()
   {  before
      {  System.out.println(name + " sending request as Client");
         (server.getPlayer()).request();
         return; // required to avoid additive weaves during aspect extension   }    }
   advise public void Agent.reply()
   {  before
      {  System.out.println(name + " received reply as Client");
         return; // required to avoid additive weaves during aspect extension    }   }}
```

FIGURE 12 Part B. Role Aspect Clerk (Basic Server)

```
aspect Clerk extends Role
{  protected Client client; // role relationships are in the aspect
   public void setClient(Client a)    {   client = a;}
   // introduce interface for extrinsic behavior to class Agent
   introduce public void Agent.request() {}
   introduce protected void Agent.sendReply() {}
   // advise weaves for an aspect instance
   advise public void Agent.request()
   {  before
      {  System.out.println(" Clerk/ Server processing request");
         sendReply();
         return;  // required to avoid additive weaves during aspect extension }    }
   advise protected void Agent.sendReply()
   {  before
      {  System.out.println("Clerk/ Server sending reply");
         // this gets the client stored at the aspect
         (client.getPlayer()).reply();
         return;        }    }}
```

FIGURE 12 Part C. Role Aspect Manager

```
aspect Manager extends Clerk
{  protected Subordinate subordinate;  // role relationship
   public void setSubordinate(Subordinate a)   {  subordinate = a;}
   // introduce weaves for static aspect
   introduce public void Agent.myNotify()    {}
   introduce protect void Agent.sendDoWork() {}
   // advise weaves for aspect instance
   advise public void Agent.myNotify()
   {  before
      {  System.out.println("Manager receiving notification");
         return;        }    }
   advise public void Agent.request()
   {  before
      {  System.out.println("Manager receiving request");
         sendDoWork();
         return;        }    }
   advise protected void Agent.sendDoWork()
   {  before
      {  System.out.println("Manager sending do work");
         (subordinate.getPlayer()).doWork();
         return;        }    }}
```

FIGURE 12 Part D: Playing the Roles

```
// Agent objects are added to the three aspects
// clientRole, managerRole, and subordinate roles are Client,
// Manager, and Subordinate aspect instances
// Shopper, Boss, and Worker are instances of class Agent
// addObject()adds an object instance to an aspect instance so the
// advise weaves take effect
clientRole.addObject(Shopper);
managerRole.addObject(Boss);
subordinateRole.addObject(Worker);
// role relationships are between aspects
clientRole.setClerk(managerRole);
managerRole.setClient(clientRole);
managerRole.setSubordinate(subordinateRole);
subordinateRole.setClient(clientRole);
subordinateRole.setManager(managerRole);
// nothing happens because a Worker can not send a request
Worker.sendRequest();
// client messages manager, who messages subordinate,
// who does the work, replies to the client, and notifies its manager
Shopper.sendRequest();
```

Figure 12: Sample Code for Hybrid Approach

Figure 12, Part D, and Figure 13 show what happens when Agent objects play roles. Three agents have been instantiated and assigned roles: Shopper, Boss, and Worker. In Figure 12, Part D, Shopper is placed in a Client aspect instance, Boss is a Manager, and Worker is a Subordinate. As shown in Figure 13, this means that Shopper plays the role of Client; Boss plays the role of Manager; and Worker is the Subordinate. Role relationships are shown in Figure 13 and implemented in Figure 12, Part D. Boss is a Clerk to the Shopper, but a Manager to the Worker. Shopper is the Boss' Client, and the Boss is the Shopper's Clerk. As the Boss delegates all tasks, Shopper is also the Worker's Client.

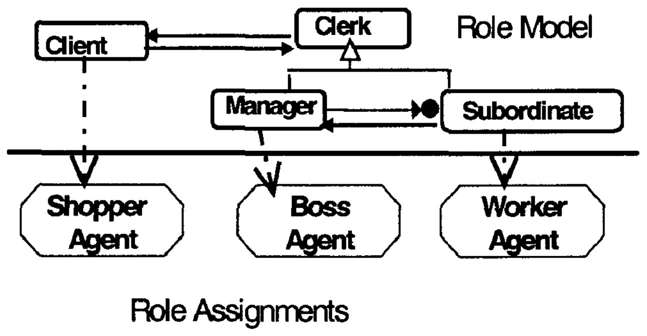


Figure 13: Role Assignments in the Application

In the actual implementation, role aspect creation and assignment is done by the Agent that is to play the role. This is because of role dependency; a role can not exist without an object (section 1.4). The actual implementation follows the use of Creator objects described in [2].

Only Shopper can act as a Client; when a Worker tries to make a request (Figure 12, Part D), nothing happens. The interface exists, but the behavior is not implemented. In a real application, exception handling would be placed in the introduce weaves. When the Shopper makes a request (the last line in Figure 12), the Boss acts as a Manager, and the Worker does the actual work, in the role of Subordinate.

If an Agent can play multiple roles in different contexts, the code in Figure 12 needs to be modified. First, every message between agents must pass a context parameter. Second, a role aspect must store a context. In this case, an advise weave executes only if the role is relevant in the given context, as shown in Figure 14. This is the simplest way to support role multiplicity with the hybrid approach; however, alternatives have to be considered to minimize object schizophrenia.

```
aspect Role
{  // Role Relationship pattern
   protected String context;
   Role(String c) { context = c;}  ...}
aspect Clerk extends Role
{...
  advise public void Agent.request(String c)
  {  before
     {  if(context.equals(c))
        { System.out.println("Clerk/Server
                           processing request");
          sendReply(c);
          return;}         } } }
```

Figure 14: Hybrid Approach Revised for Role Multiplicity

5.3 Option 5: Glue Aspects that Integrate Core and Role Objects

The hybrid approach requires that role aspects specify what type of object is going to play a given role, because the current version (0.2) of AspectJ does not support aspect parameterization. This reduces role reusability and contradicts the criteria (section 1.4) that a role can be played by any object. In option 5 (Glue Aspects), roles are represented by objects, and aspects integrate a Core object to the role(s) that it plays. This is a style of programming proposed in [14]; *in this style of programming, all the pieces of state and behavior are captured by regular objects. The aspects glue the pieces together.*

The Glue Aspects design for the Bureaucracy role model is shown in Figure 15. There are three categories of components: Core, Glue Aspects, and Roles. The role hierarchy is the same as that found in Figures 3 and the left side of Figure 11, except that the Role class does not have a `core` attribute and the Agent class does not have a `roles` attribute. The Role classes have no data or behavior that pertains to the Core object, and the Agent class has no data or behavior that pertains to the Role objects. All integration or "glue" is achieved by the aspects.

In Figure 15, all of the aspects are static aspects; they introduce behavior to the Agent and Role classes. The RoleAspect introduces the data and behavior that establishes the `core` and `role` relationships between the Agent and Role classes. The other four aspects introduce new public methods to the Agent class.

The methods in the introduce weaves found in aspects ClientAspect, ClerkAspect, ManagerAspect, and SubordinateAspect are specific to the relevant roles (Client, Clerk, Manager, and Subordinate, respectively). In each case, the message or behavior is delegated to the role. This means that, for example, the method `Agent.request()` that is introduced in the aspect ClerkAspect only contains the following line of code:

```
        role.request();
```

With glue aspects, the approach to dynamic role assignment is the same as that found in the Role Object pattern; the role instances can be varied at runtime and dynamic binding can be employed. This means that if the role data member of an Agent holds a reference to an instance of class Clerk, the definition for `role.request()` will be taken from the Clerk class. However, if the role data member instead holds a reference to an instance of class Manager, the definition will come from the Manager class.

The design shown in Figure 15 has many variations; the RoleAspect can introduce the entire interface and delegate the behavior, eliminating the need for the other aspects.

The benefits and drawbacks of the glue aspect design are the following:

- ***Independent Core and Role hierarchies:*** Any Core object can play a given Role if the appropriate Glue Aspect is provided. This is the major advantage of this design.
- ***Interface maintenance:*** The role specific interfaces are introduced to the Core objects in a modular fashion. This is also true in the hybrid approach.
- ***Object schizophrenia:*** The Role and Core objects are independent, so the Glue Aspects have to encode and manage all integration. The hybrid approach is superior in this area, and glue aspects should only be employed when there are only minimal dependencies between Role and Core objects.
- ***Additional level of components:*** The major drawback of this design is that it requires three levels of components

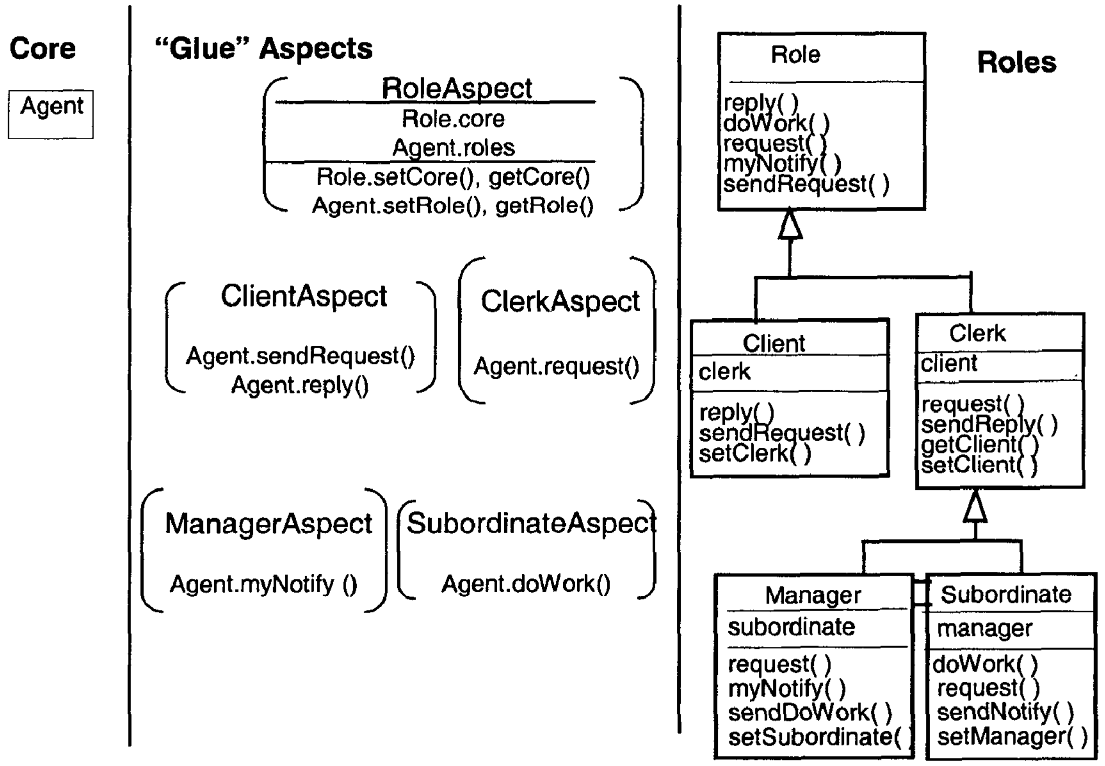


Figure 15: Option 5 (Glue Aspect) Design for the Bureaucracy Role Model

6. Role Composition

6.1 Explanation and Illustration

Role multiplicity means that a Core object plays various independent roles in different contexts. However, a Core object can also play more than one role in the *same* context. In this case, different roles need to be composed (Figure 2). The semantics of role composition have to be carefully established. Overlap and dependencies can occur if objects are playing a combination of roles in a composed role model, and the roles have extrinsic behavior that is not independent.

For example, the Bureaucracy role model can be composed with the Supply Chain [13]. The Supply Chain role model captures the structure of organizational supply chains, such as that found in supermarkets, telecommunications, and manufacturing. If a Supply Chain has only two elements there are four roles: Customer, User, Provider, and Operator. The Customer makes the original request to the Provider, and they negotiate regarding terms. Once agreement has been reached, the User role takes over from the Customer, and it interacts with the Operator role. The Operator produces the supplies and passes them back to the User; the User pays the Operator.

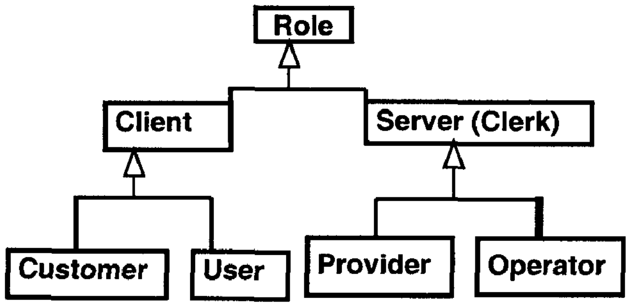


Figure 16: Supply Chain Roles

If the Supply Chain were implemented separately, seven roles would be required, as shown in Figure 16. Customer and User refine Client, and Provider and Operator refine Server (Clerk). Role composition is depicted in Figure 17, where the Bureaucracy and Supply Chain role models are merged. Here, there are two subjects; Bureaucracy addresses accountability, while the Supply Chain role model represents another, separate concern.

Two role models are composed when they both hold in a given context. A simple example was depicted in Figure 2; according to Object3 (customer), Object1 (worker) is playing both role A (Subordinate) and role B (Provider). This example is expanded upon in Figure 18. Figure 18 uses the same notation as Figure 13, and this notation is based on that found in [1]. Role models that are relevant in the given context are depicted in the top half of the figure, and both the Bureaucracy and the Supply Chain role models appear in Figure 18. Role assignments are shown in the bottom of Figure 18. ShopperAgent is a Client and a Customer; BossAgent is a Manager and a Provider, and WorkerAgent is a Subordinate and a Provider.

When more than one role model is relevant, conflicts and/ or overlaps are going to occur if the roles involve the same methods and/ or data members (the same extrinsic behavior). There are several possible ways that the roles should be composed, including the following:

- *Role Merge:* two roles correspond and should be merged with no duplicate behavior.
- *Role Override:* one role overrides another. A Manager-Provider should delegate any work. This means that the Manager role should override the Provider role.
- *Role Add:* two or more roles should be added together. A Subordinate-Provider should act as a Subordinate and a Provider.

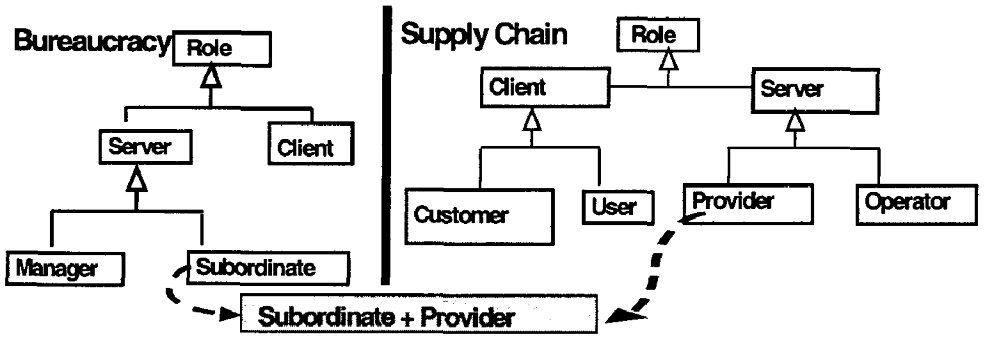


Figure 17: Role Composition

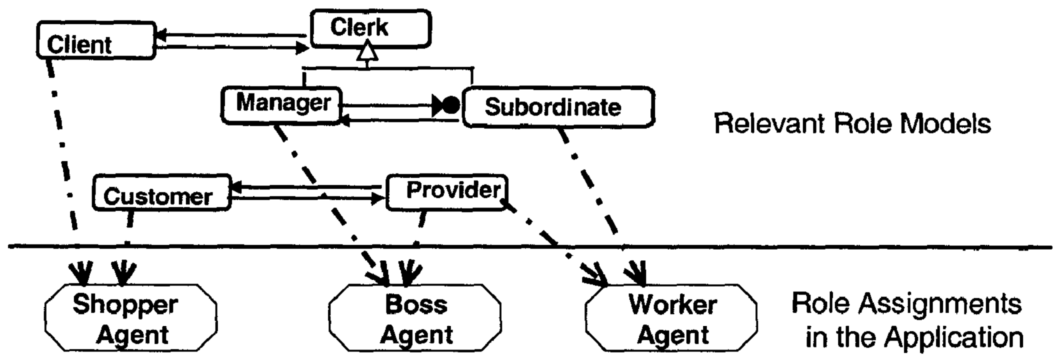


Figure 18: Role Model Composition: Both Role Models are Relevant in the Context

6.2 Hybrid Approach to Role Composition

The role composition in Figures 17 and 18 was implemented utilizing the hybrid approach discussed in sections 5.1 and 5.2. As in sections 5.1 and 5.2, each role introduces and advises its own role specific behavior to the Agent class and Agent instances.

We integrated the Bureaucracy and Supply Chain aspect implementations together by weaving (compiling) the twelve aspects (Figure 17) with the Agent class. To effect this composition in a way that preserved the semantics of the Bureaucracy and Supply Chain role models, we needed to be able to accomplish the following:

- ***merge*** the multiple definitions of Role, Client, and Server found in Figure 17
- ***override*** the behavior of a Provider with the behavior of a Manager in instances that play both roles in the same context. This is because the Manager role overrides all other roles by delegating tasks to its Subordinates.
- ***add*** the behavior of a Provider to the behavior of a Subordinate in instances that play both roles in the same context

However, we found that the AspectJ compiler did not support merge and override mechanisms during weaving. Additive weaves were supported with some restrictions. Therefore, two types of manual adjustments had to be made to bring about the desired results in the code produced by the compiler.

First, AspectJ does not allow duplicate introduce weaves. Therefore, the Role, Client, and Server aspects had to be merged manually.

Second, the precedence of overlapping advise weaves in AspectJ is undefined when the aspects are not related through extension. This means that, with the current version of AspectJ (0.2), we can not predict what the AspectJ compiler will yield from source code when two (or more) advise weaves (in unrelated aspects) impact the same member. We still wanted to attempt role composition with AspectJ, so we used trial and error with the order of the aspect and class files in the makefile until an override effect (Manager overrides Provider) could be produced. This would of course not be practical in a real role model application, as discussed in section 8.

Trial and error would not have been necessary if Manager extended Provider. However, as can be seen in Figure 17, Manager and Provider both extend Server, but Manager is not a refinement or extension of Provider. The key difference between a Provider and a Server is that a Provider negotiates with a Customer before providing services. Provider extends Server with behavior that pertains to negotiation. Meanwhile, Manager extends Server with behavior to delegate a task to a Subordinate. A Manager does not negotiate with a Client, so Manager does not extend Provider.

If multiple inheritance is allowed, it would be possible to perform role composition with a role hierarchy so that a new role can descend from Manager and Provider. However, AspectJ and Java do not allow multiple inheritance. Therefore, it is not practical to expect that two roles that impact the same extrinsic behavior will always be related by an inheritance or specialization relationship. As such, we have concluded that aspect override without extension is required for role composition. This conclusion is discussed further in section 8..

Once the resulting nine aspects (twelve originally and three eliminated during merging) and the Agent class were appropriately woven, execution was straightforward. The three agents were instantiated. ShopperAgent was placed in Customer and Client aspects; BossAgent was placed in Manager and Provider aspects; and, WorkerAgent was placed in Subordinate and Provider aspects. Role relationships were then set up between the aspects.

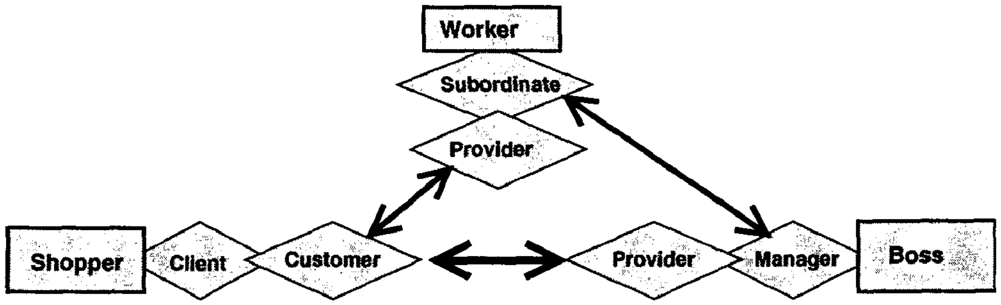


Figure 19: Aspect Attachments for Role Composition in Figure 18

The role assignments and role relationships are the ones depicted in Figure 18. A simplified view of the aspect attachments is provided in Figure 19; all intrinsic and extrinsic members have been omitted. The figure depicts the override effect; Worker and Boss see each other only through a Manager-Subordinate relationship. The result printed out during execution is shown in Figure 20; in the printout, the agents give their name and role.

```
ShopperAgent sending request as Client
BossAgent receiving request as Manager
BossAgent sending do work as Manager
WorkerAgent doing work as Subordinate
WorkerAgent received request as Subordinate
WorkerAgent sending negotiate as Provider
ShopperAgent negotiate as Customer
ShopperAgent sending complete negotiation as Customer
WorkerAgent complete negotiation as Provider
WorkerAgent sending contract as Provider
ShopperAgent got contract as Customer
ShopperAgent sending accept contract as Customer
WorkerAgent accept contract as Provider
WorkerAgent sending notification as Subordinate
BossAgent receiving notification as Manager
```

Figure 20: Sample Output from Role Composition with Hybrid Approach

6.3 Role Composition with Glue Aspects

In the glue aspect design, roles are represented by conventional classes and objects, so role composition must be accomplished in the glue aspects. This provides flexibility. If an agent plays more than one role in a given context, the glue aspect can introduce or advise behavior that does any of the following: i) iterates through all of the roles, ii) allows the behavior of one kind of role to override all other roles, iii) merges the behavior of certain roles, or iv) other variations.

However, the code is hard to design and implement because the Role and Core classes are independent and links are only available through the glue aspects. If role methods overlap and/ or conflict with each other during role composition, this in effect causes new dependencies between Role and Core objects. This is because the roles themselves are still independent; the dependencies arise because the roles are composed in a Core object. As stated in section 5.3, the glue aspect approach is useful only when there are minimal dependencies between Role and Core objects. Therefore, the hybrid design in section 6.2 is superior for role composition.

7. AOP and Code Tangling in Role Model Applications

7.1 Original Object-oriented Design

Another investigation of aspect-oriented programming involved determining the extent that AOP can be used to reduce code tangling in role model applications. Here, an existing object-oriented design based on the Role Object pattern was used as the starting point, and it was reengineered with aspect-oriented techniques.

This role model application involved five role models: Supply Chain, Negotiate for Services, Contract Net, Iterated Contract Net, and Auction. The latter four role models expand on the Supply Chain role model of section 6 in the area of negotiation. In particular, the Customer and Provider roles interact with each other in more detailed, aggregated role models. This occurs because the Customers and Providers enter into lengthy negotiations with each other regarding supplies, delivery schedule, and price. Further, competitors (other Providers) may be involved in the negotiations.

The original design featured an AgentCore class, a NegotiateRole superclass, and 24 subclasses for the individual roles. These classes contained 115 methods.

7.2 Code Tangling

The role objects and the role transitions were not considered in this study, as this research was complementary to that discussed in sections 4 through 6. Beyond the behavior of the role models, the 115 original methods were seen to involve six concerns. Each concern can be thought of as a Separation of Concern (SOC). They are listed below in Figure 21a with the number of methods involved.

Some of the code tangling is depicted in Figure 21b. The numbers on the figure correspond to the number for the separation of concern in Figure 21a. Only ten subclasses and a subset of the methods are shown in Figure 21b.

SOC 1.	**Interagent communication**. Sending messages to another agent. *17 methods.*
SOC 2.	**Exception handling.** Incorrect messages or sequences. *44 methods*
SOC 3.	**Failed conversations**. A conversation has ended due to failure. *12 methods*
SOC 4.	**Successful conversations**. A conversation has ended. *4 methods*
SOC 5.	**Negotiation strategies**. Behavior involved in competitive bidding. *8 methods.*
SOC 6.	**Iterative protocols**. Auctions and iterated contract nets. *6 methods.*

Figure 21a: Six Concerns in the Original Object-Oriented Design

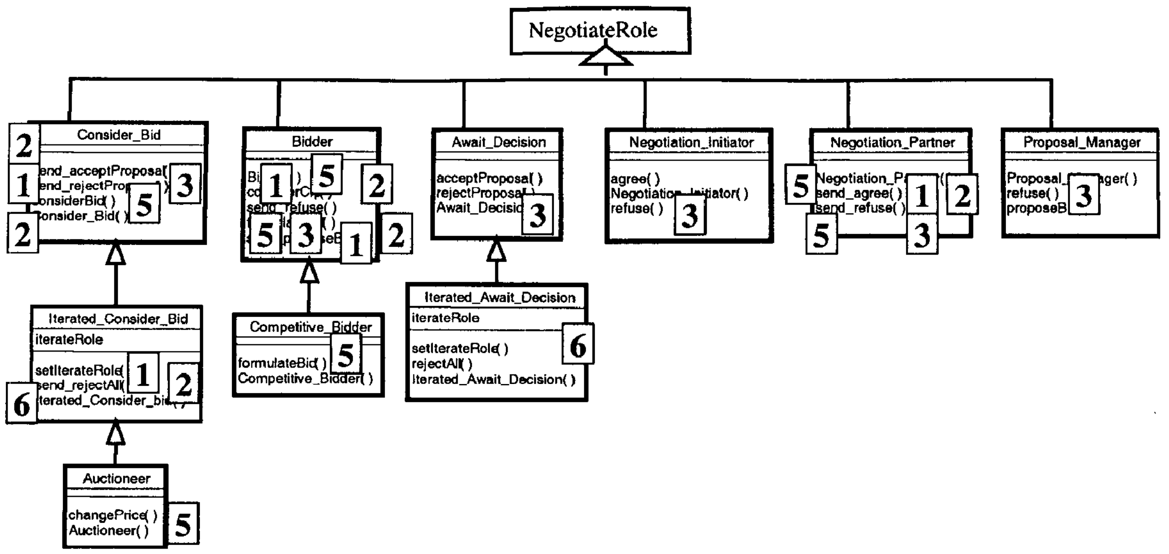


Figure 21b: Indication of the Code Tangling found in the Original Design

7.3 Object-oriented Techniques

Object-oriented techniques can alleviate some of the code tangling; behavior can be promoted to a superclass or delegated to a component. Promoted behavior decreases cohesion. Delegated behavior adds additional components, and it can lead to object schizophrenia. Further, interface maintenance is complicated unless the contained object and its interface are publicly accessible.

Two of the concerns were promoted to the NegotiateRole superclass: SOC 3 (Failed Conversations) and SOC 4 (Successful Conversations). These two types of behavior were encoded in two new methods within the NegotiateRole superclass. This was acceptable because all failed and successful negotiation protocols can be concluded in the same way.

Interagent communication was addressed via delegation. In the simplest design, the interagent communication was delegated by the role object back to the Agent. This normally would require that the interface for sending messages be duplicated in the Agent object, adding 17 methods to the overall design. (They were protected methods of the NegotiateRole class in the original design.) However, the Java JDK reflection package was used to provide general purpose transmission or *sending*. With reflection, only one new method had to be added for transmission and one for reception. However, additional runtime overheads were incurred.

7.4 Aspect-oriented Techniques for Exception Handling

Exception handling is an important concern for this application as invalid transmissions and receptions have to be caught. However, exception handling often leads to redundant and tangled code, because exceptions usually have to be thrown locally, at the methods where the errors occur. The behavior can not be promoted to a superclass, and it is not beneficial to delegate the behavior to another component.

The superclass NegotiateRole defines the default behavior for all 28 messages in the protocols, throwing an exception if the message is invalid. Each method that sends a message to another agent (SOC 1 - 17 methods) also incorporates exception handling.

The exception handling is redundant, but, as already stated, it can not be delegated or promoted. AOP is therefore appropriate, as shown in Figures 22 and 23. Static aspects are employed because the behavior is required for all instances of the NegotiateRole subclasses. Aspect Invalid State Message (Figure 22) introduces the interface and the behavior to the NegotiateRole class to throw the correct exception. Wildcard notation can not be used in introduce weaves, so each method has to be listed. Aspect SendCatcher (Figure 23) uses static advise weaves with the *catch* construct to add behavior to existing methods in some of the subclasses of NegotiateRole.

```
aspect InvalidStateMessage
{introduce public void NegotiateRole.agree(),  ...
  // all methods listed in an introduce weave
 { throw new InvalidStateMessage("Invalid Message");} }
```

Figure 22: Aspect Invalid State Message

```
aspect SendCatcher
{advise * Idle_Client.send_request(*),
 * Idle_Query_Client.send_query(*), ...
 // wildcard notation can be used
 {static catch (InvalidStateMessage e)
  {System.out.println("Invalid State Message " + "\n");}
  static catch (NoSuchMethodException e) {} }}
```

Figure 23: Aspect Send Catcher

7.5 Aspect-oriented Programming for Other Concerns

Separation of concern 5 involves different strategies for competitive bidding. For example, an agent may be negotiating for services with a monopoly; on the basis of a fixed price contract; or in a joint venture. Separation of concern 6 involves restarting or resetting a protocol that is iterative. For example, an auction protocol is carried out repeatedly, but the bidding does not start from scratch. A contract net can also be iterated.

These two concerns are classic "mix-ins". They can not be promoted to the superclass because all of the possibilities will occur. Because only single inheritance is allowed in Java, they would lead to duplicate and redundant hierarchies. That is, two subclasses would be required for each state in the contract net protocol: one for the non-iterative and one for the iterative version.

Additionally, delegation is not an attractive option because it just adds components and indirection. As in the case of interagent communication, interfaces would have to be duplicated. That is, the interface for each of the eight methods that deal with negotiation strategies would have to be duplicated in the strategy components.

Figure 24 depicts the aspect-oriented solution. Aspect instances are utilized, and an aspect instance is required for each negotiation strategy. In the figure, aspect instances are shown for fixed price contract, monopoly, and joint venture. With aspect instances, strategies are only added when needed; each of these aspects adds or advises behavior to the required methods. Seven of these methods are shown on the figures, along with a strand that connects them to the Fixed Price Contract aspect instance.

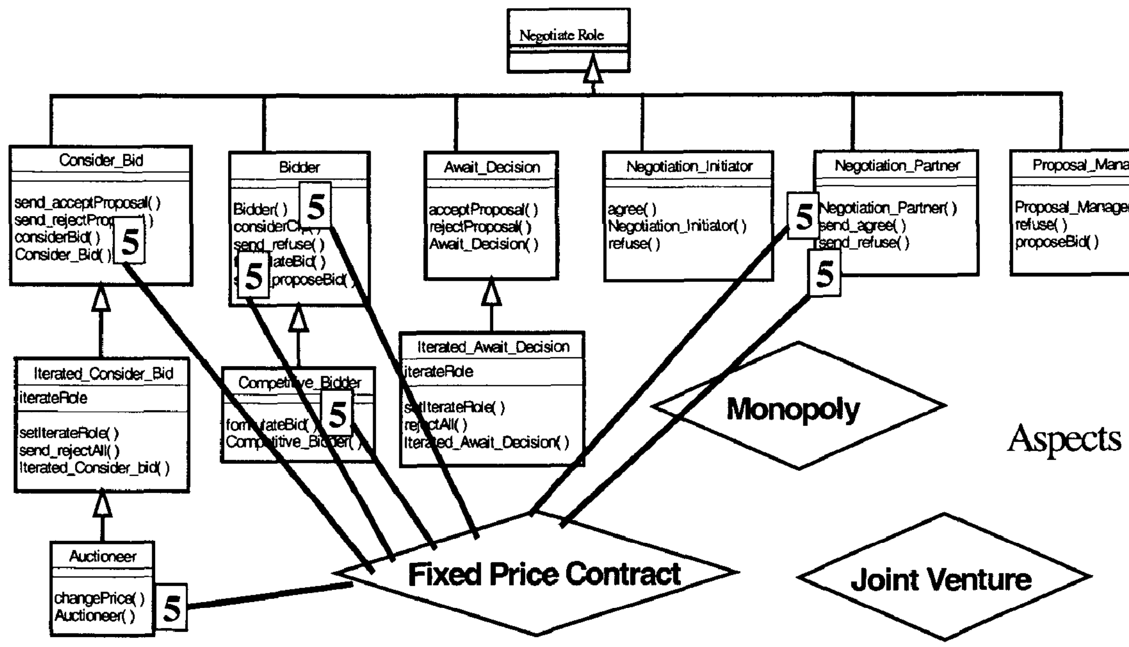


Figure 24: Negotiation Strategy Aspects

7.6 Code Tangling Summary

The use of AOP in this role model application reduced the overall module (class and method) and lines of code (LOC). In exception handling, the Invalid State Message aspect with one introduce weave (listing all relevant methods) replaced the 28 methods, reducing the module count by 26. The Send Catcher aspect replaced 51 (17 * 3) lines of code with five, saving 46 LOC. The Iterative protocol aspect replaced six methods with one aspect and one weave. Lastly, the Negotiation Strategies aspect reduced the code for strategies from 40 * 3 (8 methods * 5 SLOC * 3 strategies) to 21 (7 * 3). These results are summarized in Figure 25.

Aspect	Method Reduction	LOC Reduction (source)
Invalid State Message	- 26	
Send Catcher		- 46
Iterative Protocol	- 4	
Negotiation Strategies		- 99
Total AOP	**- 30**	**- 145**

Figure 25: Impact of Aspect-oriented Programming on the Application

8. Conclusions

This paper has presented our research in applications of aspect-oriented programming as captured in the AspectJ programming language; we have also documented our efforts in aspect-oriented design. We utilized the subject-oriented programming (SOP) features of AOP to produce new designs and implementations for role models. We then provided results regarding the impact of AOP on code tangling in object-oriented role model implementations.

Although our findings are preliminary, it appears that AOP is a promising approach to

- reducing object schizophrenia and interface maintenance in role model designs
- supporting dynamic role assignment at an instance level
- providing flexible integration of object hierarchies. This is based on our experience with the Glue Aspect approach (section 5.3).
- modelling, representing, and integrating inidividual concerns
- reducing module count and lines of source code for cross-cutting behavior

We were also able to implement role composition with AOP. However, AspectJ did not adequately support role merge or role override. We had to manually merge roles. We also had to utilize trial and error to give the desired results in role override. This is because the order of precedence in aspect weaving is undefined in AspectJ when two unrelated aspects overlap, impacting the same method. (As discussed in section 6.2, unrelated means that the aspects are not related through specialization or extension.) It is therefore our recommendation that AspectJ support the composition rules that are found in subject-oriented programming [20]. These composition rules include facilities for merging, overriding, and adding subjects.

As we discussed in section 3.3, role models and SOP have many of the same conceptual foundations [16]. Further, B. Kristensen [19] has documented research on the close relationships between subject and role composition. Therefore, our conclusion is in agreement with results presented elsewhere.

Many other questions remain, including AspectJ constructs for various aspect-oriented designs and implementations, and interfaces between aspects and objects. The proposed new aspect-oriented designs for role models also need to be further evaluated so that object schizophrenia is minimized.

Additional work is also required in appropriate metrics for comparisons between aspect-oriented and object-oriented designs and implementations. Lines of code and module count (section 7.6) are only two limited metrics.

9. Acknowledgements

The author wishes to thank E. Andersen, G. Kiczales, C. Lopes, H. Ossher, and D. Riehle for their valuable contributions to this research. A large part of this research was carried out while the author was employed as a Senior Research Fellow in Intelligent Business Systems Research at BT Research Labs in England.

References

[1] Andersen, E. (Egil), *Conceptual Modelling of Objects: A Role Modelling Approach*, PhD Thesis, University of Oslo, 1997.

[2] Baumer, D., D. Riehle, W. Siberski, M. Wolf, "Role Object," *Proceedings of the 4th Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, USA, September 2-5, 1997.

[3] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, *Pattern- Oriented Software Architecture: A System of Patterns*, Wiley, 1996

[4] Dickinson, I., "Agent Standards", Agent Technology Group, 1997. http://drogo.cselt.stet.it/fipa.

[5] Dyson, P., B. Anderson, "State Patterns," in *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, F. Buschmann, Ed., Addison Wesley, 1998.

[6] Fowler, M., "Dealing with Roles," *Proceedings of the 4th Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, USA, September 2-5, 1997.

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

[8] Gottlob, G., Schrefl, M., and Rock, B., "Extending Object-oriented Systems with Roles," *ACM Trans on Info. Sys.*, Vol. 14, No. 3, July, 1996, pp. 268 - 296.

[9] Harrison, W., H. Osher, "Subject- Oriented Programming (a critique of pure objects)," in *Proceedings of the Conference on Object-oriented Programming: Systems, Languages, and Applications*, Washington, D. C. September, 1993. pp. 411 - 428.

[10] Helm, R., I. M. Holland, D. Gangopadhyay, "Contracts: Specifying Behavioral Compositions in Object-oriented Systems," *Object-oriented Programming, Systems and Lanugages*, ECOOP/ OOPSLA '90 Proceedings, October, 1990, pp. 169 - 180.

[11] IBM Research: Subject- oriented Programming Group, "Subject-oriented Programming and Design Patterns," http://www.ibm.research/sop

[12] Kaplan, M., Harold Ossher, William Harrison, Vincent Kruskal, Subject-Oriented Design and the Watson Subject Compiler, Position paper for OOPSLA'96 Subjectivity Workshop, October, 1996

[13] Kendall, E. A., "Agent Roles and Role Models: New Abstractions for Multiagent System Analysis and Design," *International Workshop on Intelligent Agents in Information and Process Management*, Germany, September, 1998

[14] Kiczales, G., C. Lopes, "Aspect-oriented Programming w/AspectJ™, " Tutorial and Primer, Xerox PARC, www.parc.xerox.com/spl/projects/aop/

[15] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. - M. Loingtier, and J. Irwin, "Aspect-oriented Programming," *Proc. of the European Conference on Object- Oriented Programming (ECOOP)*, Finland, Springer- Verlag LNCS 1241, June, 1997.

[16] Kristensen, B. B., "Object-oriented Modelling with Roles", OOIS'95, *Proceedings of the 2nd International Conference on Object-oriented Information Systems*, Dublin, Ireland, 1996.

[17] Kristensen, B. B., Osterbye, K., "Roles: Conceptual Abstraction Theory and Practical Language Issues", *Special Issue of Theory and Practice of Object Systems (TAPOS) on Subjectivity in Object-oriented Systems*, 1996.

[18] Kristensen, B. B., D. C. M. May. "Activities: Abstractions for Collective Behavior". *Proceedings of the European Conference on Object-oriented Programming (ECOOP'96)*, Linz, Austria, 1996.

[19] Kristensen, B. B., "Subject Composition by Roles", *Proc. of the 4th Intl. Conf. on Object-oriented Information Systems*, Brisbane, Australia, 1997.

[20] Ossher, H., Matthew Kaplan, William Harrison, Alexander Katz and Vincent Kruskal, "Subject- Oriented Composition Rules," *Proceedings of 1995 Conference on Object-oriented Programming Systems, Languages, and Applications*, October 1995

[21] Ossher, H., M. Kaplan, A. Katz, W. Harrison, V. Kruskal, "Specifying Subject- Oriented Composition," *Theory and Practice of Object Systems (TAPOS)*, Vol 2, No 3, 1996.

[22] Rational Software, "UML Documentation: Behavioral Elements Package: Collaboration Overview," http://www.rational.com/uml/resources/docmentation/semantics/semant9a.jtmpl

[23] Reenskaug, T., Wold, P., Lehne, O. A., *Working with Objects*, The OOram Software Engineering Method, Manning Publications Co, Greenwich, 1996.

[24] Reenskaug, T., "Role Modelling Enters the Main Stream," *Object EXPERT*, January, 1997.

[25] Riehle, D., T. Gross, "Role Model Based Framework Design and Integration," OOPSLA'98, *Proceedings of the 1998 Conference on Object-oriented Programming Systems, Languages and Applications*, ACM Press, 1998.

[26] Riehle, D., "Composite Design Patterns", OOPSLA '97, *Proceedings of the 1997 Conference on Object-oriented Programming Systems, Languages and Applications*, ACM Press, Page 218-228, 1997. http://www.riehle.org.

[27] Riehle, D., "Bureaucracy", in *Pattern Languages of Program Design 3*, R. Martin, D. Riehle, F. Buschmann (Ed.), Addison Wesley, 1998, pp. 163 - 185.

[28] VanHilst, M., D. Notkin, "Using Role Components to Implement Collaboration- Based Designs," OOPSLA'96, *Proceedings of the 1996 Conference on Object-oriented Programming Systems, Languages, and Applications, ACM Press*, 1996, pp. 359 - 369.